BRICS

Basic Research in Computer Science

BRICS NS-94-6 Engberg et al. (eds.): 6th Nordic Workshop on Programming Theory

Proceedings of the 6th Nordic Workshop on Programming Theory

17–19 October 1994, Aarhus, Denmark

Uffe H. Engberg
Kim G. Larsen
Peter D. Mosses (editors)

BRICS Notes Series **NS-94-6**

ISSN 0909-3206 **December 1994**

See back inner page for a list of recent publications in the BRICS Notes Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone:	**+45 8942 3360**
Telefax:	**+45 8942 3255**
Internet:	**BRICS@daimi.aau.dk**

Proceedings of the

6th Nordic Workshop on Programming Theory

17–19 October 1994 — Aarhus, Denmark

Uffe H. Engberg
Kim G. Larsen
Peter D. Mosses
(editors)

Foreword

The basic aim of the Nordic Workshop on Programming Theory series is to offer an opportunity for researchers from the Nordic and Baltic countries to meet each other and to discuss the current state of their research, and also to serve as an easy introduction to the research community for young researchers wishing to enter this field.

This 6th Nordic Workshop on Programming Theory took place in Aarhus, Denmark, 17–19 October 1994, and was arranged by BRICS[1] at the Department of Computer Science at University of Aarhus. The workshop attracted some 63 participants coming from: Norway (3), Sweden (11), Finland (5), Latvia (1), Lithuania (2), Estonia (1), England (3), Germany (4) and Denmark (33).

Presentations:
There were 47 regular presentations at the workshop, arranged in two parallel sessions. In addition the following invited speakers gave 60-minute presentations in plenary sessions: Matthew Hennessy (University of Sussex), Bernhard Steffen (Universität Passau), and Ib Holm Sørensen (B-Core (UK) Limited). Moreover, four systems was demonstrated, all closely related to talks given at the workshop.

Acknowledgement:
The 6th Nordic Workshop was financially supported by grants from BRICS and the Danish Science Research Council. Technical and administrative support was provided by the Department of Computer Science, University of Aarhus.

Programme Committee

Kim G. Larsen	Aalborg Univ., Denmark
Peter D. Mosses	Univ. of Aarhus, Denmark
Ralph-Johan Back	Åbo Akademi, Finland
Reino Kurki-Suoni	Tampere Univ. of Tech., Finland
Sigurd Meldal	Univ. of Bergen, Norway
Olaf Owe	Univ. of Oslo, Norway
Bengt Jonsson	SICS/Uppsala Univ., Sweden
Bengt Nordström	Univ. of Göteborg/Chalmers Univ. of Tech., Sweden

Local Organization

Peter D. Mosses, Janne K. Damgaard, Karen K. Møller.

[1]Basic Research in Computer Science, a centre established in cooperation between the Danish National Research Foundation and the Universities of Aarhus and Aalborg.

Addresses

For a hardcopy of the proceedings, please contact:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark,

or, by e-mail: `<BRICS@daimi.aau.dk>`.

Copies of the Proceedings and individual papers in A4 format are electronically accessible through WWW and anonymous FTP:

`http://www.daimi.aau.dk/BRICS/NS/94/6/BRICS-NS-94-6/`

`ftp ftp.daimi.aau.dk (cd pub/BRICS/NS/94/6/NWPT6)`

Contents

Demonstrations

Higher-Order Processes and Their Models

Matthew Hennessy

University of Sussex
School of Math. & Phys. Sciences
England
Email: matthewh@cogs.susx.ac.uk

A higher-order process algebra in which processes can be sent and received as data along channels is investigated. Using a simple operational semantics two behavioural preorders are defined. The first, based on may testing, is in terms of the ability of processes to offer communications on channels while the second, based on must testing, depends on the communications which processes can guarantee.

The first behavioural preorder can be modelled by a denotational semantics which uses a notion of higher-order traces while for the second we develop a denotational model using higher-order Acceptance Trees.

Finite Model Checking and Beyond

Bernhard Steffen*
Universität Passau
Gemany

Abstract

This paper reviews finite and infinite-state model checking form two pragmatic perspectives: the application to the static analysis of imperative programs or data flow analysis, and the implementational aspect. Data flow analysis provides an interesting area of application for model checking, which is particularly challenging in the interprocedural case, since infinite state spaces need to be considered. Moreover, as it is a typical case of a structurally unrestricted problem, one should use global iterative methods for solution. This can be exploited for a uniform and efficient treatment based on a *fixpoint analysis machine* that covers all kinds of logics and model structures important for the considered application.

1 Introduction and Motivation

Over the past decade model-checking has emerged as a powerful tool for the automatic analysis of concurrent systems. Whereas model-checking for finite-state systems has already a long history (cf. e.g. [EmLe86, ClES86, Lars88, StWa89, Wins89, BCMD90, Clea90, ClSt91, EnFT91, ClGL92, ClPS93, Lars92, AnSW94]), the theory for infinite systems is still in its initial phase. Bradfield and Stirling [BrSt91] observed that tableaux-based model-checking covers general infinite-state systems. However, their method is not effective. Therefore much work has focussed on *context-free processes*, a subclass of the infinite-state systems. In particular, in [BuSt92] an iterative model-checking algorithm is developed that decides the *alternation-free* part of the modal mu-calculus for *context-free* processes. Moreover, in [HuSt93] it is shown how this can be done using tableaux-based techniques, and in [BuSt94] a model checker for pushdown processes that are a strict generalization of context-free processes has been proposed. More general but also much more expensive are the decision algorithms proposed by Muller and Schupp [MuSc85] for monadic second order logic on pushdown transition graphs implying the decidability of the full mu-calculus [Koze83] for pushdown processes, by Hungar [Hung94] for model checking of macro processes, which are a 'higher order variant' of context-free processes,

*Lehrstuhl für Informatik, Universität Passau, Innstr. 33, D-94032 Passau (Germany), tel: +49 851 509.3090, fax: +49 851 509.3092, steffen@fmi.uni-passau.de

and by Bouajjani et al. who verified non-regular properties for context-free processes [BoER94]. Still, the limits of model checking infinite-state processes are by far not reached, even though it seems to be clear that, except for some few unifying generalizations, practically relevant efficient algorithms will require certain application specific profiles.

In this paper we review (infinite-state) model checking from two pragmatic perspectives, 1) the application to the static analysis of imperative programs (or data flow analysis), which, in the interprocedural case, provides a challenging but still tractable scenario for infinite state model checking, and 2) the implementational aspect, where we propose a uniform platform for efficient implementation.

This choice of application concentrates our attention to *global, iterative* model techniques, as in static analysis properties must be determined *globally*, i.e., for all program points, in order to determine the *full potential* for optimizing program transformations. Moreover, for structurally unrestricted problems of 'tractable' size, standard iterative methods are most efficient. Data flow analysis provides a typical example of this class of problems. Thus, we do not consider local [StWa89, Wins89, Clea90, Lars92, HuSt93, AnSW94] or BBD-based [BCMD90, EnFT91] model checking techniques in this paper, simply because they do not fit the considered application profile.

Concentrating on global iterative model checking techniques allows a *uniform* treatment based on a *fixpoint analysis machine*, which, despite its uniformity, outperforms most of the 'iteration based' special purpose tools. This goal is achieved by translating the specific fixpoint problems into unstructured, very fine-grained but computationally advantageous representations, which allow us to eliminate as much redundancy as possible. Currently we are uniformly covering the kinds of fixpoint computations summarized in Table 1,[1] which are specific to the structure of the underlying model and formula (cf. Section 5). In fact, the differences between these kinds of problems and their corresponding fixpoint computations only require a change in the data domain the fixpoint is computed over and in the program controlling the order of the fixpoint computation. The architecture and all the other data structures of the machine coincide in all these applications.

Fixpoint class	Model class		
	regular	CFR/PDA	macro
homogeneous	(chaotic, $I\!\!B$)	(chaotic, $\mathcal{F}_{I\!\!B}$)	(chaotic, high.ord. $\mathcal{F}_{I\!\!B}$)
hierarchical	(layered, $I\!\!B$)	(layered, $\mathcal{F}_{I\!\!B}$)	(layered, high.ord. $\mathcal{F}_{I\!\!B}$)
alternated nesting	(backtrack, $I\!\!B$)	(backtrack, $\mathcal{F}_{I\!\!B}$)	(backtrack, high.ord. $\mathcal{F}_{I\!\!B}$)

Table 1: Classification of the Fixpoint Applications

The next section sketches the link between data flow analysis and model check-

[1] Here $I\!\!B$ denotes the set of Booleans and $\mathcal{F}_{I\!\!B}$ the set of Boolean functions.

ing, while Section 3 gives an overview over the scenario addressed in this paper. Subsequently, Section 4 presents the internals of the fixpoint analysis machine. Finally, Section 5 discusses the various forms of fixpoint computations considered, and Section 6 our conclusions and directions to future research.

2 Data Flow Analysis as Model Checking

Data flow analysis (DFA) is concerned with the automatic identification of program points enjoying specific properties, like for example, liveness of variables or equivalence of program terms. Typically, DFA algorithms are constructed for a given program property and therefore have the following functionality:

$$\text{programs} \quad \rightarrow \quad \text{program points enjoying the considered property}$$

Model checking is concerned with the automatic identification of those states of a finite system satisfying a specific modal (or temporal) formula that expresses, for example, properties like deadlock, divergence or liveness. Typically, model checkers are parameterized on the formula of interest and therefore have the following functionality:

$$\text{modal formulas} \times \text{model} \quad \rightarrow \quad \text{states satisfying the argument formula}$$

Identifying programs with models, program points with states and program properties with modal formulas, model checkers can be seen as DFA algorithms that have the program property of interest as a parameter.

This observation can be exploited for the development of an algorithm that automatically generates efficient implementations of DFA-algorithms from specifications written in a modal logic.

In essence, this DFA-*generator* works by *partially evaluating* an appropriate model checker with respect to its modal formula parameter; the result is a standard iterative DFA-algorithm [Hech77], which directly runs on the machine the model checker is implemented on. Using our fixpoint analysis machine, this partial evaluation is part of an optimizing compilation into the corresponding fixpoint analysis machine code.

Currently, our implementation covers standard bit-vector DFA algorithms[2] in an efficient manner: it allows concise high level specifications, and the generated algorithms are guaranteed to be linear in the size of the program being analysed. Moreover it supports the specification of and reasoning about DFA-algorithms. Both can be done structurally by reasoning within the modal logic serving as specification language.

This approach is not limited to intraprocedural analysis via model checking of regular structures, it can also be extended to cover interprocedural data flow analysis via context-free model checking. Having a model checker for context-free structures at hand, this is straightforward for a setting with (recursive) parameterless procedures. However, the clean treatment of parameters is quite

[2]Typical such algorithms determine e.g. live variables, (very) busy expressions or use-definition chains[Hech77].

hard. Surprisingly, the extension to formal procedure parameters, i.e. to higher order, is very complicated but still straightforward [Hung94]. It complicates the domain of the fixpoint computation, leads to an unacceptable runtime complexity, but the framework of specification remains completely unchanged. In contrast, the general treatment of value parameters, which intuitively are much simpler, requires some modification of the whole setting. Fortunately, such modifications are not required for large classes of practically relevant applications [KnRS92, KnRS94, Knoo93].

In [Stef91, Stef93], the framework is illustrated by means of an example of practical relevance: an improved version of Morel/Renvoise's algorithm for eliminating partial redundancies [MoRe79]. The algorithm generated there from a one line specification[3] is as efficient as the standard uni-directional algorithms, because the decomposition of the usual bi-directional structure into a hierarchy of uni-directional analyses guarantees the efficiency of the well-known bit-vector techniques. Moreover, the results of the generated algorithm are provably optimal. In [KnRS92, KnRS94], this optimality result has been strengthened by minimizing the lifetimes of the registers required for the transformation. These modifications could easily be implemented on top of the fixpoint analysis machine.

3 The Scenario

Figure 1 summarizes the scenario discussed in this paper. We will now first explain the top row of the figure addressing the considered application areas, and subsequently sketch the lower part discussing the implementation of the various kinds of model checkers.

- *Intraprocedural dataflow analysis*: algorithms of this kind can be realized efficiently and at almost no implementation cost on our analysis machine via a data flow analysis generator, which automatically produces fixpoint machine code from high level specifications [Stef91, Stef93].
- *Interprocedural data flow analysis*: in this setting we are able to cover a wide class of programs that contain recursive procedures with value parameters. The corresponding data flow analysis generator, which uses the same high level specifications as the intraprocedural version, is under implementation. It requires the combination of the methods presented in [BuSt92, Stef93, BuSt94].
- *Higher order data flow analysis*: this setting allows us to deal with further types of parameters, like reference parameters and procedure parameters. Whereas the case of reference parameters is still rather efficient, the optimal treatment of arbitrary (finite mode) procedures requires an unacceptable effort and should therefore be treated approximatively. The reader is referred to [Knoo93] for details.

[3] Usually, this type of algorithm is given by means of a complex equation system.

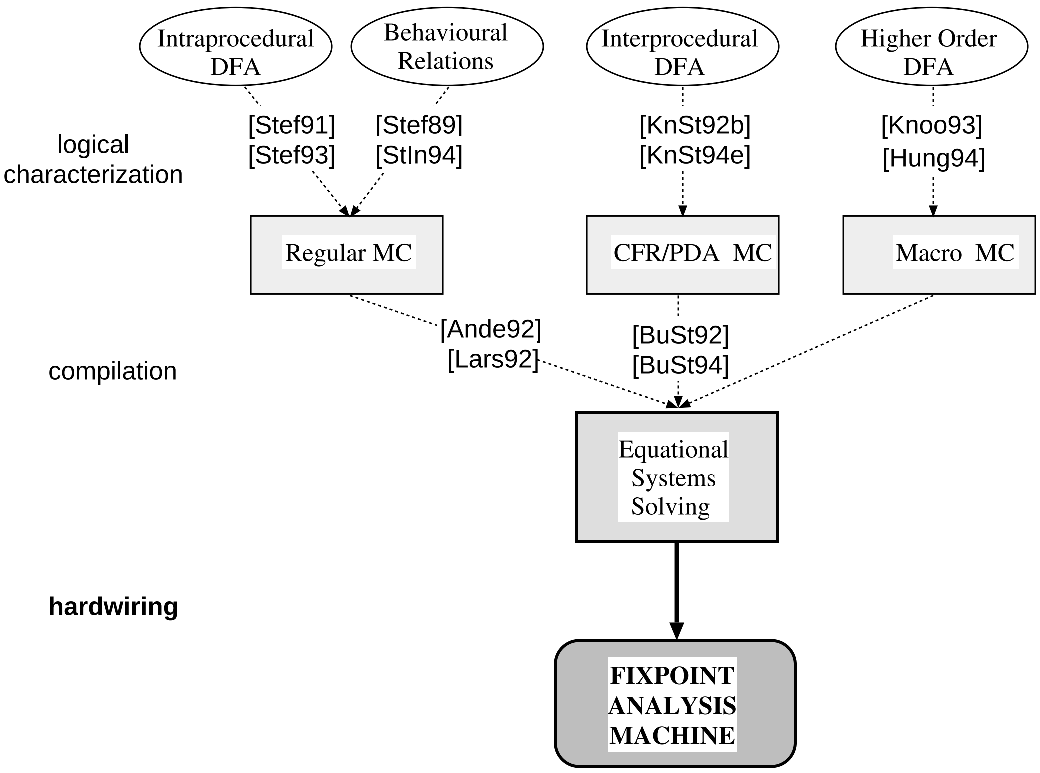


Figure 1: Setup of the Analysis Environment

The remaining top row application concerns the verification of behavioural relations between distributed systems. Like the three applications already mentioned, it can be transformed into a particular model checking problem via logical characterization [Stef89, StIn94].

The second row presents a hierarchy of model checking problems classified according to the structure of the underlying model. Additionally, the possibility and the extent of interference between minimal and maximal fixpoints – homogeneous, hierarchical or alternating fixpoints [EmLe86] – is an important classification criterion. Whereas all the top row applications require at most hierarchical fixpoints, alternating fixpoints are necessary when dealing with properties like fairness. Thus we face a potential of nine structurally different model checking problems, two of which (the alternating case for context-free and macro models, see Table 1), even though decidable, do not yet have a fixpoint characterization. The other seven cases can uniformly be represented by means of a finite equational system together with a parity vector specifying the particular solution of interest, leading to the third row. Actually, we proposed such transformations also for the two exceptional cases, but their correctness is still a conjecture. As most of the translation into the equational representation, which can be regarded as a common intermediate level, is more or less standard, we will concentrate on the analysis machine.

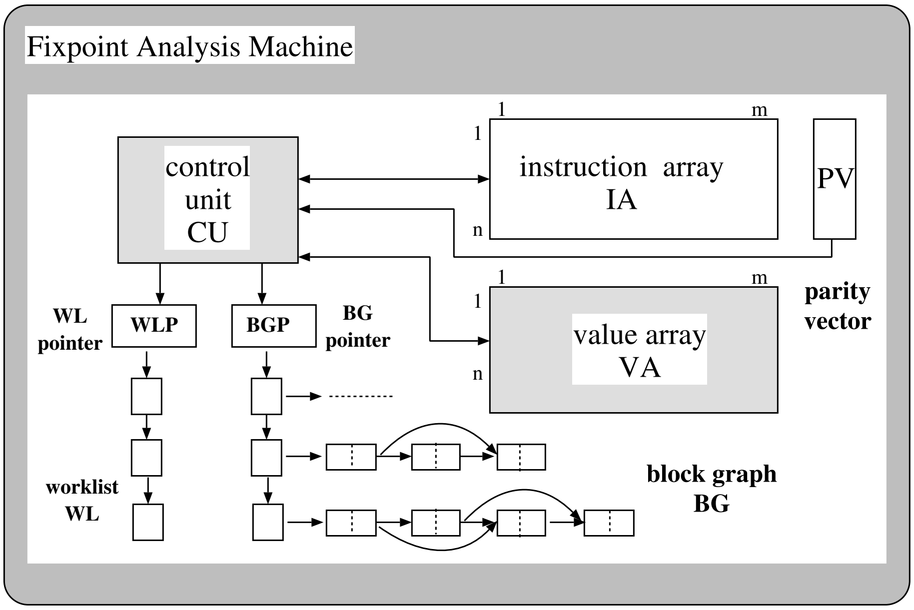


Figure 2: The Architecture of the Fixpoint Analysis Machine.

4 The Fixpoint Analysis Machine

The structure of the analysis machine is a consequence of the observation that, although each of the different kinds of fixpoint computations (cf. Table 1) requires special care, there is a large common core as soon as one breaks the problem down to the appropriate kind of granularity and allows a limited form of parameterization. In fact, the machine architecture we are going to present allows us to uniformly cover all the kinds of fixpoint problems mentioned without performance penalty, as our choice of granularity is tailored for runtime optimization.

The machine architecture is illustrated in Figure 2. Here, the white parts are common to all analyses. Only the value array and the control unit (shaded in the figure) are parameterized in the kind of the problem. The value array is parameterized in the value domain the fixpoint must be computed over, which depends on the kind of model under investigation, and the control unit steers the order of the fixpoint computation according to the considered kind of fixpoint problem.

4.1 The Instruction and Value Arrays

Fixpoint computation proceeds by successively updating the components of the *value array* by means of component-specific operations that are stored in the *instruction array*. Allowing distinct operations for each component of the value array, and being flexible in the choice of the granularity, leads to a very simple structure that must be modeled by the instruction array: *AND/OR/COMP graphs*. They represent (i) the kind of *operation* to perform (conjunction, disjunction, or functional composition), (ii) the *list of operands* for this operation, consisting of components of the value array, as well as (iii) the *influence list*, i.e. those components that are influenced by the value of the considered component. All this can easily and automatically be determined by means of a simple compiler.

Whereas the structure of the instruction array is completely independent of the considered fixpoint problem, the component type of the value array depends on the kind of model we are considering (see Table 1):

- In the *regular* case single bits are sufficient. They indicate whether a specific node of the model satisfies a particular formula.

- In the *CFR/PDA* case the elements correspond to property (predicate) transformers, and are therefore Boolean functions $\mathcal{F}_{\mathbb{B}} : \mathbb{B}^{n \times m} \rightarrow \mathbb{B}$ of an arity determined by the size of the considered system and formula. For reasons of efficiency, they are represented as Binary Decision Diagrams (BDDs) [Brya86].

- In the *macro* case, elements correspond to mappings between property transformers. Thus we are dealing with higher order Boolean functions here.

The instruction array is completely constructed during compile-time. Thus there is only read-access at runtime. In contrast, the value array must be read and written at runtime, and only its space allocation is a matter of compilation.

4.2 The Parity Vector

A fixpoint problem is completely specified by a system of mutually recursive equations, where each equation is classified as maximal or minimal according to the kind of fixpoint of interest. The instruction array does not contain this classification. It merely stands for a set of fixpoint problems. We therefore specify the classification separately in a vector, that stores for each row of the instruction array the desired kind of fixpoint, called its *parity*.

Like the instruction array, the parity vector is created at compile-time and only accessed at runtime.

4.3 The Block Graph

Homogeneous fixpoint problems can be solved by means of a totally chaotic iteration over the value array. However, each time both kinds of fixpoints are involved the order of fixpoint computation is essential. For hierarchical systems a layered approach is sufficient, while alternating fixpoint formulas require a very strict discipline involving backtracking. This observation motivates the structure of our block graphs, which are *lists of DAGs*[4] (directed acyclic graphs), where edges represent ordering constraints and nodes collect the set

[4]This choice is an elaboration of the block graphs presented in [ClSt91b, ClKS92]

of all equations which need not be distinguished according to these constraints. This graph structure is already sufficient to uniformly capture even strongly optimized organizations of the fixpoint computation for the alternating case.

Technically, the need for blocks, which are collections of equations whose fixpoints can be computed in an arbitrary order, arises as soon as there are depending minimal and maximal fixpoints. This dependency requires a strict organization of the order in which these fixpoints are computed. In the hierarchical case, a simple sequentialization is required, and in the more complicated alternating case a backtracking procedure must be organized following the strucure of the underlying block graph. It is convenient to additionally split blocks according to their parity, which leads to the notions of min-blocks and max-blocks. This additional separation is uncritical as a single block could anyhow only comprise completely independent minimal and maximal equations.

Blocks are important, as they allow an efficient fixpoint computation. The switching between different blocks according to ordering constraints is more expensive then the worklist-oriented chaotic iteration allowed within a block. Thus efficient fixpoint algorithms will completely determine the fixpoint of a block before taking a switch. This approach is only guaranteed to be optimal when using counters [ClSt91, ClSt91b].

- For *homogeneous fixpoints*, a one component list containing a one element DAG is sufficient, as we are dealing with a single block here where no ordering constraints need to be taken care of.
- *hierarchical fixpoint* computations postpone the evaluation of an equation until all equations of different parity which may influence it have reached their fixpoints. In terms of model checking, this corresponds to an 'inner-most' strategy for the evaluation of the formula. This requirement can be expressed sufficiently by means of a list of sets of formulas, i.e. by a list of one component DAGs.
- *alternating fixpoint* computations require backtracking, which leads to an exponential complexity in the alternation depth ([EmLe86]) of the considered formula. Thus the most important source of optimization is the reduction of backtracking steps. Block graphs support such a reduction by structuring the global dependence graph between the fixpoint equations in the following fashion:
  - The list structure reflects the dependence (ordering constraints) between the strongly connected components of the dependence graph. Of course, these constraints form in general a DAG structure. However, as in the hierarchical case, we can simply collapse this DAG to a list without runtime penalty.
  - The DAG structure reflects part of the ordering constraints within a strongly connected component: a constraint between two equations e_1 and e_2 is only kept if the row of e_1 precedes the row of e_2 in the value array. This DAG of equations is then collapsed by combining all equations that have a 'similar' dependence relation into a node.

Block graphs are constructed at compile-time, and there is only read-access at runtime.

4.4 The Worklist

Whereas the block graph is a mean to steer the fixpoint computation globally, i.e. between blocks, the worklist organizes the fixpoint computation inside a block. It contains the addresses of the elements of the current block whose values must be updated as a consequence of earlier changes in the value array. The list is dynamically initialized when entering a new block, and it is updated during the computation by appending the addresses of all the influenced value array components. The worklist is a pure runtime entity, initialized, updated, and read at runtime.

5 Computing Fixpoints

In this section we sketch the computation mechanism for the various kinds of fixpoint problems. We start by considering the three regular problems, which only require a fixpoint computation for bitvectors. Subsequently we discuss the extension to context-free structures. Even though the decidability of the model checking problem is implied by decidability results about monadic second order logic [MuSc85], the known efficient algorithms cover alternation free, i.e. hierarchical formulas, only. The best known algorithm for the general case is non-elementary. The further extension to push-down structures, which in contrast to classical automata theory do not coincide with context-free structures when the branching structure of the models are essential, is rather straightforward and still in the range of tractability. This is no longer the case for macro structures [Hung94], which require a very expensive higher-order treatment and will not be discussed here.

5.1 First-Order Fixpoints

Uniform fixpoints: Homogeneous and regular problems can be regarded as the common core of all the regular fixpoint computations. It consists of the determination of the fixpoint over a single block. This computation is performed by successively decrementing counters until the fixpoint is reached, a process which is steered by a worklist that contains references to all the components whose counters are currently known to require decrement. The worklist is updated by adding references to all influenced components whenever one counter becomes zero, which indicates a change of its corresponding Boolean value [ClSt91, ClSt91b].

Hierarchical Fixpoints: Here, blocks are sequentially computed in the order indicated by the block graph. The fixpoint computation within the blocks is identical to the one in the homogeneous case. It should be noted that the counters for a block must be constructed immediately before its fixpoint computation. This is necessary in order to capture the effects of the earlier fixpoint computations.

Alternating Fixpoints: Again blocks are treated exactly as in the homogeneous case. However, in contrast to the previous two cases, this computation must be repeated according to changes in blocks of different parity that are higher in the hierarchy but still in the same strongly connected component.

5.2 Second-Order Fixpoints

Structurally, these fixpoint computations follow exactly the same lines as in the first order case. Only the domain of the components of the value array is now *second order*, i.e., instead of determining properties for states of the considered systems, we must determine property transformers for certain classes of states. Moreover, in the alternating case it is still an open problem whether the straightforward extension to the second order domains computes the intended values. We hope that experimenting with our machine will help us clarifying this point.

We are now going to sketch the essence of an algorithm that decides the alternation-free modal mu-calculus for *context-free* processes, i.e. for processes that are given in terms of a context-free grammar, or equivalently, as mutually recursive systems of finite state-labelled transition systems. For the understanding of the key idea behind the model checking algorithm, the second perspective is advantageous, as it stresses the transformational character of the approach: processes are represented as sets of named standard finite-state labelled transition systems, where the label set additionally contains the names identifying the different transition systems. A transition labelled with such a name should be viewed as a call to the 'procedure' (transition system) identified by this name.[5] Operationally, these possibly recursive procedure calls are treated by *recursive action refinements*, i.e. by the copying process illustrated in Figure 3 for a single recursive named transition system CM.

CM defines a little encoding machine, which reads a word over the alphabet $\{a, b\}$ until it finds the word delimiter #. Then, in its second phase, it outputs the corresponding sequence of codes in reverse order.

The right hand side shows CM after a single expansion step, whose infinite repetition yields a unique infinite transition system containing no name-labelled transition.[6] It is exactly this infinite transition system, which is represented by CM, and which, as we will argue below, can be model checked by applying a second order model checker to its finite representation CM.

The heart of our algorithm is to iteratively compute for each node σ of the finite process representation the *property transformer* telling which properties are valid at this node depending on the properties considered to be valid after

[5] Considering names as nonterminals and the other labels as terminals establishes the match to context-free processes.

[6] Technically, we are restricting our attention to representations that are guarded, i.e. whose initial transitions are not labelled by names, and which have a unique start and end node.

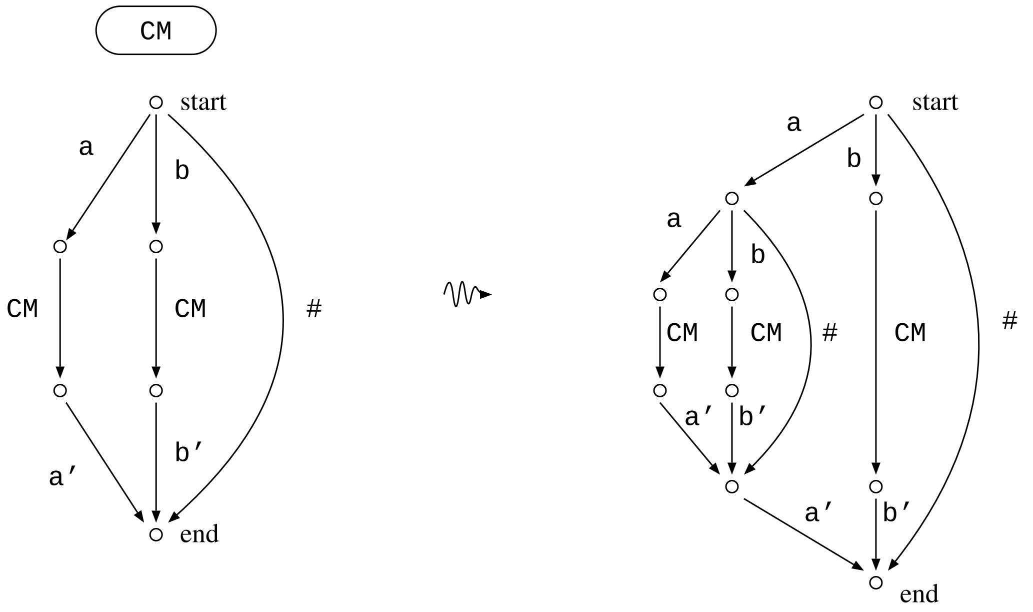


Figure 3: Expanding a Procedural Transition System

the 'termination' of the procedure containing σ. The subsequent decision step is more or less trivial. This results in the following two step structure:

1. iterative computation of a *property transformer* for each state class of the finite process representation. This computation works block-wise. In the case of a *min*-block it successively updates the initial "minimal" predicate transformers until the smallest fixpoint is reached. The handling of *max*-blocks is completely dual, and the global algorithm, which deals with a hierarchy of blocks proceeds hierarchically as described in [ClSt91, ClKS92].

   The iteration mechanism as such is essentially the same as for usual iterative algorithms, except for:

   - the handling of the procedural identifiers (non-terminals). They are dealt with by applying the currently valid approximation for their defining procedural transition systems.
   - the domain for the iteration, which is second order here, i.e. it consists of property transformers (Section 3).

   This computation is effective because we can restrict our attention to the finitely many subformulas of the property currently being investigated.

2. decision of the model-checking problem: having the predicate transformers at hand, the model-checking problem can be solved simply by checking whether the formula under consideration is a member of the set of formulas that results from applying the property transformer of the "main

procedure" to the set of formulas that are valid for a deadlocking or terminating state.

The complexity of the resulting algorithm is linear in the size of the system's representation and exponential in the size of the property. This is quite promising, as many practically relevant problems can be composed of very small properties: e.g. bitvector analyses, which are common in data flow analysis have exponent one ([KnSt93a])!

6 Concluding Remarks

We have reviewed finite and infinite-state model checking from two pragmatic perspectives: the application to the static analysis of imperative programs or data flow analysis, and the implementational aspect. This application driven review gives some further evidence for the general observation that practically relevant efficient verification algorithms will require certain application specific profiles.

Of course this observation is not new and there is already a wide range of application or structure specific approaches to fight the well-known state explosion problem, ranging from the most elegant and ambitious *compositional* methods [AnSW94, ClLM89, GrSt90], over *partial order* methods that suppress unnecessary interleavings of actions [GoWo91, GoPi91, Valm93] to *Binary Decision Diagram*-based codings that work for an interesting class of practically relevant systems [Brya86, BCMD90, EnFT91]. All these techniques can be accompanied by *abstraction*: depending on the particular property under investigation, systems may be dramatically reduced by suppressing details that are irrelevant for verification [CoCo77, ClGL92, ClRi93, DaGG93, GrLo93]. This list of references and approaches is by far not exhaustive and still increasing. Thus we are faced with an almost unmanageable variety of approaches and algorithms asking for meta-tools managing this diversity. The DaCapo environment applied to the organization of UNIX commands in [SFCM94] addresses this need.

Acknowledgement

The work presented in this paper was strongly influenced by collaboration and intensive discussion with Olaf Burkart, Andreas Claßen, Marion Klein, Jens Knoop, Gerald Lüttgen, Tiziana Margaria, and Oliver Rüthing.

References

[AhSU85] A. Aho, R. Sethi, J. Ullman: "*Compilers: Principles, Techniques, and Tools*," Addison Wesley, 1985.

[AnSW94] H. Andersen, C. Stirling, G. Winskel: "*A Compositional Proof System for the Modal Mu-Calculus*," Proceedings 5th IEEE LICS, 1994.

[BCMD90] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang: "*Symbolic Model Checking:* 10^{20} *States and Beyond*," Proc. of 5th IEEE LICS, pp. 428–439, Philadelphia, 1990.

[BoER94] A. Bouajjani, R. Echaed, R. Robbana: "*Verification of nonregular temporal properties for context-free processes*," Proc. CONCUR'94, Uppsala (Sweden), Aug. 1994, LNCS N.836, pp.81-97, Springer.

[BrSt91] J. Bradfield, C. Stirling: "*Local Model Checking for Infinite State Spaces*," Tech. Report ECS-LFCS-90-115, LFCS, Edinburgh, 1991.

[Brya86] R. Bryant: "*Graph-Based Algorithm for Boolean Function Manipulation*," IEEE Trans.on Computers, Vol. C-35, No. 8, pp. 677-691, 1986.

[BuSt92] O. Burkart, B. Steffen: "*Model Checking for Context–Free Processes*," Proc. of CONCUR '92, Stony Brook (NJ), August 1992, LNCS 630, pp. 123–137, Springer Verlag.

[BuSt94] O. Burkart, B. Steffen: "*Pushdown Processes: Parallel Composition and Model Checking*," Proc. of CONCUR '94, Stockholm (Sweden), August 1994, LNCS N. 836, pp. 98-113, Springer Verlag.

[Clea90] R. Cleaveland: "*Tableau-Based Model Checking in the Propositional Mu-Calculus*," Acta Informatica N. 27, pp.725–747, 1990.

[ClES86] E. Clarke, A. Emerson, A. Sistla: "*Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications*," ACM TOPLAS, V.8 N.2, pp. 244–263, 1986.

[ClGL92] E. Clarke, O. Grumberg, D. Long: "*Model Checking and Abstraction*," Proceedings POPL'92.

[ClKS92] R. Cleaveland, M. Klein, B. Steffen: "*Faster Model Checking for the Modal Mu-Calculus*," Proc. of CAV '92, Montreal (Canada) LNCS 663, pp. 410–422, Springer Verlag, 1992.

[ClLM89] E. Clarke, D. Long, K. McMillan: "*Compositional Model Checking*," Proceedings LICS'89.

[ClPS93] R. Cleaveland, J. G. Parrow, B. Steffen: "*The concurrency workbench: A semantics-based verification tool for finite state systems*," ACM TOPLAS, Jan. 1993.

[ClRi93] R. Cleaveland, J. Riely: "*Testing-Based Abstractions for Value Passing Systems*," Proc. of CONCUR '94, Stockholm (Sweden), August 1994, LNCS N. 836, Springer Verlag.

[ClSt91] R. Cleaveland, B.Steffen: '*A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus*," Proc. CAV '91, Aalborg (Denmark), July 1991, LNCS N.575, pp.48–58, Springer.

[ClSt91b] R. Cleaveland, B.Steffen: "*Computing Behavioural Relations, Logically*," Proc. ICALP'91, Segovia (Spain), Aug. 1991, LNCS N.510, Springer.

[CoCo77] P. Cousot, R. Cousot: "*Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction and Approximation of Fixpoints*," Proceedings POPL'77.

[DaGG93] D. Dams, O. Grumberg, R. Gerth: "*Generation of Reduced Models for Checking Fragments of CTL*," Proc. CAV'93, Crete (GR), June 1993, LNCS 697, Springer Verlag, 1993, pp.479–490.

[EnFT91] R. Enders, T. Filkorn, D. Taubner: "*Generating BDDs for Symbolic Model Checking in CCS*," Proc. CAV '92, Aalborg (Denmark), July 1991, LNCS N.575, Springer Verlag.

[EmLe86] A. Emerson, C.-L. Lei: "*Efficient Model Checking in Fragments of the Propositional Mu-Calculus*," Proc. of LICS'86, IEEE Computer Society Press, pp. 267–278, 1986.

[GoPi91] P. Godefroid, D. Pirottin: "*Refining Dependencies Improves Partial-Order Verification Methods*," Proc.CAV'93, Elounda (Greece) June/July 1993, LNCS N.697, pp.397–408.

[GoWo91] P.Godefroid, P.Wolper: "*Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties*," Proc. CAV'91, Aalborg (Denmark), Juli 1991, LNCS N.575, 1991, pp.332–342.

[GrLo93] S. Graf, C. Loiseaux: "*Program Verification using Compositional Abstraction*," Proceedings FASE/TAPSOFT'93.

[GrSt90] S. Graf, B. Steffen: "*Using Interface Specifications for Compositional Minimization of Finite State Systems*," Proceedings CAV'90, Rutgers (NJ), June 1990, LNCS N. 407, Springer Verlag, 1991.

[Hech77] M. Hecht: "*Flow analysis of computer programs*," Elsevier, North-Holland, 1977.

[Hung94] H. Hungar: "*Model Checking of macro processes*," Proc. of CAV'94, Palo Alto (CA), June 1994, LNCS N.818, Springer V., pp.169–181.

[HuSt93] H. Hungar, B. Steffen: "*Local Model–Checking for Context–Free Processes*," Proc. ICALP'93, LNCS N. 700, Springer Verlag, pp. 593-605. Accepted for the *Nordic Journal of Computer Science*, 1993.

[Knoo93] J. Knoop: "*Optimal interprocedural program optimization: A new framework and its application*," PhD thesis, Dep. of Computer Science, Univ. of Kiel, Germany, 1993. To appear as LNCS monograph, Springer V.

[KnRS92] J. Knoop, O. Rüthing, B. Steffen: "*Lazy Code Motion*," Proc. PLDI Conference'92, San Francisco, CA, June 1992, ACM SIGPLAN Notices, Vol.27, pp. 224–234.

[KnRS94] J. Knoop, O. Rüthing, B. Steffen: "*Optimal Code Motion: Theory and Practice*," ACM Transactions on Programming Languages and Systems, Vol. 16, N.4 (1994), pp.1117 - 1155.

[KnRS94e] J. Knoop, O. Rüthing, B. Steffen: "*A tool kit for constructing optimal interprocedural data flow analyses*," Fakultät für Mathematik und Informatik, Univ. Passau, Germany, MIP-Bericht Nr. 9413 (1994).

[KnSt92b] J. Knoop, B. Steffen: "*The interprocedural coincidence theorem*," Proc. CC'92, Paderborn (Germany), Springer-Verlag, LNCS 641, pp.125 - 140.

[KnSt93a] J. Knoop, B. Steffen: "*Efficient and optimal bit-vector data flow analyses: A uniform interprocedural framework*," Inst. für Informatik und Praktische Mathematik, Universität Kiel (Germany), Bericht Nr. 9309 (1993).

[Koze83] D. Kozen: "*Results on the Propositional μ-Calculus*," TCS Vol. 27, pp. 333–354, 1983.

[Lars88] K.G. Larsen: "*Proof Systems for Hennessy–Milner Logic with Recursion*," Proc. CAAP '88, LNCS N. 299, pp.215–230, Springer V.

[Lars92] K.G. Larsen: "*Efficient Local Correctness Checking*," Proc. of CAV'92, Montreal (CAN), LNCS N.663, pp.410-422, Springer V.

[MoRe79] E. Morel, C. Renvoise: "*Global optimization by supression of partial redundancies*," Comm. of the ACM N.22, Vol. 2, pp. 96–103, 1979.

[MuSc85] D. Muller, P.Schupp: "*The Theory of Ends, Pushdown Automata, and Second-Order Logic*," TCS N. 37, pp. 51–75, 1985.

[SFCM94] B. Steffen, B. Freitag, A. Claßen, T. Margaria, U. Zukowski: "*Intelligent Software synthesis in the* DaCapo *Environment*," Proc. 6th Nordic Workshop on Programming Theory, Aarhus (Denmark), October 1994, same volume.

[Stef89] B. Steffen: "*Characteristic Formulae*," Proc. of ICALP'89, Stresa (Italy), LNCS N.372, Springer Verlag, 1989.

[Stef91] B. Steffen: "*Data Flow Analysis as Model Checking*," Proc. TACS'91, Sendai (Japan), LNCS N.526, pp. 346–364, Springer V., 1991.

[Stef93] B. Steffen: "*Generating Data Flow Analysis Algorithms from Modal Specifications*," Science of Computer Programming N.21, 1993, pp.115-139.

[StIn94] B. Steffen, A. Ingólfsdóttir: "*Characteristic Formulae for Finite State Processes*," Information and Computation, Vol.110, No.1, 1994.

[StWa89] C. Stirling, D. Walker: "*Local Model Checking in the Modal Mu-Calculus*," Proc. TAPSOFT '89, LNCS N.351, pp.369–383, Springer.

[Valm93] A. Valmari: "*On-The-Fly Verification with Stubborn Sets*," Proc.CAV'93, Elounda (GR) June 1993, LNCS N.697, pp.397–408.

[Wins89] G. Winskel: "*A Note on Model Checking the Modal Mu-Calculus*," Proc. ICALP '89, Stresa (Italy), LNCS N.372, pp. 761–772, Springer.

The B-Technologies : A system for computer aided programming

D.S. Neilson and I.H.Sørensen
©**B-Core(UK) Limited**

October 1994

Abstract

The paper introduces the B-Technologies, a mathematically based formal method and a toolset for computer aided software engineering.

The *B-Technologies* (comprising three components - the *B-Method*, the *B-Tool* and the *B-Toolkit*) have been designed to scale up formal methods for practical application. The *B-Method* and the *B-Toolkit* are described in this paper.

The *B-Method* is designed to provide a notation and a methodology for the formal specification, design, implementation and maintenance of industrial-scale software systems. The features of incremental construction of layered software as well as its incremental verification have been guiding principles in the development of the B-Method. The method uses J-R Abrial's *Abstract Machine Notation* (AMN) as the language for specification, design and implementation within the software process. AMN is is based on an extension of Dijkstra's guarded command notation, with built-in structuring mechanisms for the construction of large systems.

The *B-Toolkit* supports the method over the entire spectrum of activities from specification through design and implementation into maintenance. The B-Toolkit comprises automatic and interactive theorem-proving assistants, a proof printer and a set of software development tools: an AMN syntax & type checker, a specification animator and generators promoting an object oriented approach at all stages of development, and the re-use of specification models/software modules. All tools are integrated with the proof assistants into a window-based development environment.

0 Introduction

Formal methods of software engineering offer a number of benefits. Firstly, they enable the production of reliable software systems for which all possible behaviour can be accurately predicted and verified to meet their functional specifications. Secondly, they have the excellent potential for automation, which will reduce software production costs, and encourage standardisation. Thirdly, they encourage accountability, uniformity of approach and reusability from specification through to coding, all easing the task of software maintenance.

Existing formal methods have failed to make provision for all of these benefits, and existing applications of formal methods have thus failed to demonstrate the advantage of the approach convincingly. So far, the emphasis has been on formal specification, rather than on the entire software cycle of specification, coding and maintenance.

The B-Method [3],[5],[2],[4],[6] is a formal software development process which has been designed as a complete methodology for production of highly reliable, portable and maintainable software on an industrial scale; extremely powerful integrated tool support is provided at all stages of the development process in the form of the B-Toolkit.

This paper is based on an earlier paper written in collaboration with J-R Abrial ([1]).

1 The B-Method and the B-Toolkit

Several important principles have guided the design of the B-Method:

1. The B-Method is a mathematical method that belongs to the "model oriented" approach to software construction (VDM [7] [8] and Z [9] employ a similar approach).
2. Great attention has been paid in ensuring that the notational aspect of the method is as simple as possible. In particular, there is no real distinction between the specification notation and the programming notation; the latter appears as a restricted subset of the former.
3. The method is founded on set theory in a way which is made as solid as possible by reconstructing the original Zermelo set theory [10],[11] within the method itself. In particular, each construct of the specification (and programming) notation has a set-theoretic counterpart.
4. The method has been designed in parallel with the B-Toolkit that supports it, so that every aspect of the method is supported by providing significant help to the user.
5. Separate techniques are proposed within the method to support the design of (large) specification construction and that of (large) program construction.
6. The method is based on a series of embedded notations: the logical notation, the basic set notation, the relational notation, the mathematical object notation, the generalised substitution notation, and, at the highest level, that of the Abstract Machine.

The notation used by the B-Method relies on an extension to Dijkstra's calculus [12]. The extension allow specification of operations in terms of preconditions and postconditions, and permits object-oriented designs. Similar extensions are common within university research, eg. Morgan's Specification Statement [13].

This paper assumes a knowledge of the underlying logical and set notations, and describes some aspects of the Abstract Machine Notation (AMN) and the Generalised Substitution Language (GSL); a full description of AMN and GSL may be found in [14].

J-R Abrial has given a set theoretic model for the notation to ensure its soundness, and his book is due for publication [15].

The B-Toolkit which supports the B-Method is an integrated suite of computer programs which covers many aspects of software development, including:

1. Full configuration and dependency management of all source files, including abstract specification documents as well as binary code files.
2. Syntax and type-checking of specification documents as well as low level design documents, with comprehensive error reporting .
3. Verification condition generation (which generates the proof-obligations necessary in order to guarantee specification consistency and correctness of refinement).
4. Automatic & interactive provers for discharging the verification conditions.
5. Specification animation, enabling the specification to be 'run' and tested.
6. Code generation from declarative descriptions, facilitated by a re-usable library of code modules.
7. A Library of reusable code modules which integrate automatically with models from the reusable specification library.
8. A translator for translating low level design documents into C.
9. Rapid prototyping, facilitated by an interface generator, built on the re-usable library.
10. Automatic Markup and cross-referencing and indexing of documentation.
11. Remaking facilities when a source file is altered; the scope of the remake is determined by the toolkit, and the system is rebuilt (if possible automatically) to its former state, including code and documentation.
12. A multi-user facility for team software development.
13. All the individual tools are integrated and designed to run automatically.
14. All tools are activated from a simple to use window-based control panel; an X Motif Interface will be complete by the end of 1994.

The B-Toolkit is commercially available from B-Core(UK); a more detailed description of its components is given in section 6.

2 Abstract Machine Notation

In the B-Method, specifications, refinements and implementations are presented as Abstract Machines. Clauses within the Abstract Machine Notation declare a context of global constraints, given constants, abstract sets and their properties; further AMN clauses provide a list of abstract state variables and operations on those variables, the latter having optional preconditions, parameters and results. The operations provide the interface for the outside world to the state variables encapsulated within the Abstract Machine. Abstract Machines may be parameterised (through parameter constraints) so that instances of machines can be reused in the incremental construction of more complex machines.

2.1 The Generalised Substitution Language

In order to describe operations, a "before-after" predicate for each operation could be used, as done in VDM [8] and in Z [9]. However, the B-Method uses instead an extension of the notion of substitution (as used in mathematics) called the Generalised Substitution Language (GSL) for specifying an operation in a dynamic systems. AMN is an extended form of GSL, itself a generalisation of Dijkstra's guarded command notation [12], enhanced with preconditioning and unbounded choice, similar to the generalisations proposed in [13].

The statements in AMN (and GSL) are very concise and easy to read; there is no difference between the specification and programming notation; the generalised substitution notation, in common with all programming notations, does not refer to variables which do not change. Each generalised substitution S is defined as a predicate transformer which transforms a postcondition R into the weakest precondition for S to establish R. We write this condition as $[S]R$ and it is read : S establishes R. S may take one of the following forms: the simple substitution ($v \;:=\; e$); the `skip` substitution; the *pre-conditioned* substitution ($\mid$); the *bounded choice* substitution ($[]$); the *guarded* substitution ($\Longrightarrow$); the *unbounded choice* substitution (@). These generalised substitutions are axiomatised:

GSL Syntax	Definition
$[vl := el]\, R$	substititions within R
$[\mathtt{skip}]\, R$	R
$[P \mid S]\, R$	$P \;\wedge\; [S]\, R$
$[P \Longrightarrow S]\, R$	$P \Rightarrow [S]\, R$
$[S [] T]\, R$	$[S]\, R \;\wedge\; [T]\, R$
$[@l.S]\, R$	$\forall\, l\, .\, [S]\, R$, if l is not free in R

Note that, thanks to the two choice substitutions, it is possible to specify the behaviour of a dynamic systems in a non-deterministic way: this is important, since a specification should not always give the precise and determinate details of a future systems. Such details will be added later in the refinement/implementation phases leading eventually to code.

Below, are examples of syntactic rewriting rules (sugars) allowing us a very rich "pseudo-programming" specification notation (AMN).

AMN Substitution	GSL Substitution
BEGIN S **END**	S
PRE P **THEN** S **END**	$P \mid S$
CHOICE S **OR** $\cdots$ **OR** T **END**	$S \; [] \; \cdots \; [] \; T$
IF P **THEN** S **ELSE** T **END**	$P \Longrightarrow S \; [] \; \neg P \Longrightarrow T$
IF P **THEN** S **END**	$P \Longrightarrow S \; [] \; \neg P \Longrightarrow \mathtt{skip}$
ANY z **WHERE** P **THEN** S **END**	$@z \; . \; (P \Longrightarrow S)$
VAR z **IN** S **END**	$@z \; . \; S$
$x \; := \mathtt{bool}(P)$	$P \Longrightarrow x := \mathtt{TRUE} \; [] \; \neg P \Longrightarrow x := \mathtt{FALSE}$

3 Abstract Machine Specification

Proof of internal consistency of an Abstract Machine specification requires demonstration that its context may exist - the formal parameters, constants and variables - and that within this context of the machine, the initialisation establishes the invariant, and each machine operation maintains that invariant. For example, the following Abstract Machine:

MACHINE
  $Machine\_name(x)$
CONSTRAINTS
  P
CONSTANTS
  y
PROPERTIES
  Q
VARIABLES
  z
INVARIANT
  R
INITIALISATION
  T
OPERATIONS
  $Operation\_name \;\; = \;\;$ **PRE** L **THEN** S **END ;**
  ...
END

gives rise to the following proof obligations:

$$\exists\, x \,.\, P$$

$$P \;\Rightarrow\; \exists\, y \,.\, Q$$

$$(P \;\wedge\; Q) \;\;\Rightarrow\;\; \exists\, z \,.\, R$$

$$(P \;\wedge\; Q) \;\;\Rightarrow\;\; [T]R$$

$$(P \;\wedge\; Q \;\wedge\; R \;\wedge\; L) \;\;\Rightarrow\;\; [S]R$$

In the above, $[S]R$ is read as "S establishes R"; the first three are concerned with consistency of contextual information (i.e if these conditions are not met then we have given a contradictory specification, which we should not attempt to implement); the fourth concerns the initialisation, and checks whether the invariant condition is satisfied initially; the last proof obligation concerns the operations (there will be one such proof obligation for each operation of the machine), and this checks whether an operation maintains the invariant condition.

3.1 Structuring and Constructing Specifications

In order to manage the complexity of the specification process it is essential that specifications can be constructed and verified in an incremental way, and that existing specifications can be used to build new specifications. Whilst in the construction of programs we are concerned with separate compilation, in the construction of large specifications from component parts we are concerned with separate verification and proof.

The key to the problem is a new generalised substitution construct, the multiple substitution ($\|$), allowing us to put together two substitutions working on distinct variables. Thus we extend the syntax of GSL to permit multiple substitution, and we may now derive such laws as:

GSL Syntax	Equivalent Syntax
$v := x \parallel w := y$	$v, w := x, y$
$S \parallel \texttt{skip}$	S
$S \parallel (P \mid T)$	$P \mid (S \parallel T)$
$S \parallel (T \; [] \; U)$	$(S \parallel T) \; [] \; (S \parallel U)$
$S \parallel (P \implies T)$	$P \implies (S \parallel T)$, *if* S *"terminates"*
$S \parallel @z \,.\, T$	$@z \,.\, (S \parallel T)$, *if* z *is not free in* S

From these properties we can show that multiple substitution, $\|$, can always can be eliminated in any formal reasoning about AMN operations.

We extend AMN with five Abstract Machine Clauses. These clauses will make operations from other machines visible, and these operations can be used in the construction of the specification of new operation (e.g. using $\|$). The clauses are described in the two sections that follow.

3.1.1 The INCLUDES, PROMOTES and EXTENDS Clauses

These concerns are provided for by employing what is known as the *semi-hiding principle*: a variable from one Abstract Machine can be included within another Abstract Machine but not modified directly from within that Abstract Machine (and so the invariant of an included machine does not add to the proof obligations of the machine in which it is included). The included variables are thus "read-only", but may appear in the invariant of the including machine. The only way in which the included variables may be changed is by "calling" one of the included operations from within the body of an operation of the including machine.

AMN allows for the inclusion of existing Abstract Machine specifications with distinct variables to be included to build new specifications, following the semi-hiding principle by the use of the INCLUDES clause. Machine parameterisation and renaming permit particular instances of different machines, as well as distinct copies of the same machine, to be included within a new machine. The variables of the new machine include the variables from each of the included machines; the invariant of the new machine includes the conjunction of the invariants of all included machines; the initialisations from the included machines are also inherited. New variables can be introduced and new invariant conditions (a "gluing invariant") can be imposed; each operation of an included machine is accessible to each operation of the new machine. The INCLUDES clause is transitive.

The PROMOTES clause contains the list of operation names of those operations of the included machines that become, without alteration, operations of the new machine. The EXTENDS clause has exactly the same form as the INCLUDES clause: all extended machines are included and, moreover, all operations of extended machines are automatically promoted to the new machine.

3.1.2 The USES and SEES Clauses

The INCLUDES mechanism does not allow for the "sharing" of one specification by several others, however, the USES and SEES constructs allow for one or more machines to have knowledge of another, provided that the used/seen and all using/seeing machines are eventually included in a single new machine. Hence neither USES nor SEES permits reference to formal parameters, and neither clause is transitive.

Machines may call operations of used/seen machines provided those operations do not change the state (i.e. they may call only "enquiry" operations). Both clauses allow for the renaming of machines. The difference between the two clauses concerns variable scope; variables of a used machine may appear in the invariant of a using machine, but variables of a seen machine may not appear in the invariant of the machine that sees it. This restriction means that a seen machine may be refined independently of a machine that sees it.

4 Abstract Machine Refinement

An Abstract Machine specification is not necessarily executable, and certainly the abstract model of the state of the machine as well as the formulation of the operations may not be directly translatable into executable code (for example, non-determinism plays an important role in many specifications). To facilitate translation into code, the Abstract Machine must be refined.

A refinement can be regarded as a step towards code; it is not the final step (see page 28), but permits a change of state in order to get closer to the data structures of the final implementation - *data refinement* - if required, and also allows for *algorithmic refinement*, and for a combination of the two.

Data refinement is achieved by a change of variables and a new set of operations which refine each of those in the original specification. The relationship between the abstract variables and the new variables is given through new invariant conditions. Algorithmic refinement permits the reformulation of operations without changing the state model.

An Abstract Machine specification does not permit sequencing: substitutions are either simple or multiple (parallel). In a refinement, however, sequencing is permitted, the GSL extension being provided by the following law:

GSL Syntax	Equivalent Syntax
$[S\ ;\ T]R$	$[S]([T]R)$

The refinement relation defined within the B-Method is based on that presented in [8],[13] and [16] and possesses some fundamental properties which are of great importance in performing incremental development and proof. The refinement relation is *transitive*, which means that the final implementation may be verified by verification of each refinement step, The refinement relation is also *monotonic* with respect to all constructs of GSL, which means that subcomponents of an operation can be refined independently.

The refinement of an Abstract Machine incurs a number of proof obligations. We consider the most general case of simultaneous data and algorithmic refinement.

MACHINE
  Machine_name
VARIABLES
  x
INVARIANT
  P
INITIALISATION
  S
OPERATIONS
  $z \longleftarrow Op\_Name \;=$
    PRE
      Q
    THEN
      T
    END ;
  ...
END

REFINEMENT
  Refinement_name
REFINES
  Machine_name
VARIABLES
  y
INVARIANT
  R
INITIALISATION
  U
OPERATIONS
  $z \longleftarrow Op\_Name \;=$
    PRE
      L
    THEN
      V
    END ;
  ...
END

gives rise to the following proof obligations:

$$\exists (x, y) \cdot (P \;\wedge\; R)$$

$$[U] \neg ([S] \neg R)$$

$$\forall (x, y) \;.\; (\;(P \;\wedge\; R \;\wedge\; Q) \;\;\Rightarrow \mathrm{L})$$

$$\forall (x, y) \;.\; (\;(P \;\wedge\; R \;\wedge\; Q) \;\;\Rightarrow \\ (\;[V'] \neg ([T] \neg (R \;\wedge\; z = z'))\;)\;)$$

In the above, V' stands for substitution V within which the output variable z has been replaced by z'. The first proof obligation concerns the existence for the new variables of the refinement, and this checks whether the new invariant condition, R, contradicts the invariant condition already given - P; the second deals with the correctness of the initialisation, and it verifies that the new initialisation, U, establishes a situation where the specified initialisation, S, cannot fail to establish the condition - R. (R being the new invariant and the abstraction relationship); the third and fourth ensures the correctness of the operations (there will be a pair of such proof obligation for each operation), and it verifies that an operation, V, establishes a situation where the specified operation, T, cannot fail to maintain R, while preserving output.

5 Abstract Machine Implementation

An Abstract Machine implementation represents the last refinement and one which may be directly translated into code; thus an implementation may contain no non-determinism (unlike a refinement), complete concrete algorithms being provided for each operation. Loop constructs are permitted in an implementation and thus GSL is extended by the following syntactic rewriting rule for a while loop:

AMN Substitution	GSL Substitution
WHILE P **DO** S **END**	$(P \Longrightarrow S)\hat{}\ ; (\neg P \Longrightarrow \texttt{skip})$

For a description of ˆ, see [15].

Usually a loop is used together with an initialisation (substitution), an invariant (predicate) and, a variant (expression giving a natural number), giving the proof rule enabling us to verify loop constructs without using the definition above:

$$
\begin{array}{l}
\quad [T]Q \ \wedge \\
\quad Q \ \Rightarrow \ E \in \mathbb{N} \ \wedge \\
\quad \forall\, l \,.\, (\ (Q \wedge P) \ \Rightarrow \ [S]Q\ ) \ \wedge \\
\quad \forall\, l \,.\, (\ (Q \wedge P) \ \Rightarrow \ [n := E]([S]E) < n\ ) \ \wedge \\
\quad \forall\, l \,.\, (\ (Q \wedge \neg P) \ \Rightarrow \ R\ ) \\
\Rightarrow \\
\quad [T\ ;\ \textbf{WHILE}\ P\ \textbf{DO}\ S\ \textbf{INVARIANT}\ Q\ \textbf{VARIANT}\ E\ \textbf{END}]\,R
\end{array}
$$

The first antecedent expresses the fact that the initialisation, T, establishes the loop invariant - Q; the second that, the loop variant is a natural number expression; the third that, under the guard P, the body, S, maintains the loop invariant; the fourth that, under the guard, the body decreases the loop variant - E; the last that the post-condition, R, is established when the loop terminates, i.e. when the negation of the guard holds.

The proof obligations of an Abstract Machine implementation are exactly those give for an Abstract Machine refinement on page 27.

5.1 Structuring and Implementing Designs

An important difference between an Abstract Machine refinement and an Abstract Machine implementation is that in the latter all of the data is completely encapsulated in other machines; encapsulated variables may not be seen or used from within the algorithms of an implementation. This is known as the *full-hiding principle*, and means that an implementation need have access only to the specifications of the encapsulating machines; such machines may be independently verified, refined and implemented, so long as the specification is met. This object oriented approach has clear and considerable benefits in the construction of large software systems, where teams of designers may work in parallel on different aspects of the system. This also means that Implementations are machine independent descriptions.

Another important consideration in the design of software systems is that of reuse. A library of encapsulating machines is available to the designer, each having an Abstract Machine specification and matching implementation (for example a set machine, sequence machine and so on), to facilitate the process of reaching code from abstract specification quickly, and also to impose standards that are easily-understood and followed.

5.1.1 The IMPORTS and SEES Clauses

The IMPORTS clause implements the full-hiding principle, allowing for the introduction of encapsulating machines into an Abstract Machine implementation. As with the INCLUDES clause for specification construction, it permits machine parameterisation to permit particular instances of machines (including library machines) to be imported.

Since all data is imported, an Abstract Machine implementation may not have a variables clause, but the imported variables may be referenced in the INVARIANT clause (to enable the introduction of "gluing invariants") and also in the invariant part of a loop.

The SEES clause allows for sharing of encapsulating machines which are separately developed. An imported machine can be shared by other machines. These machines SEES the imported machine. While the importing machine has access to all the operations provided by the shared machine, the seeing machines have limited access, and cannot use the operations which modify the abstract state of the shared machine.

6 The B-Toolkit Components

The B-Toolkit provides powerful fully-integrated tool support for every aspect of the B-Method. These tools are implemented on the B-Platform (the successor to the B-Tool [17], [18], thus providing theorem-proving capabilities). The B-Toolkit and the B-Method underwent seven years of research and development at the PRG in Oxford and at BP Research in Sunbury, and development is now continuing at B-Core (UK) Ltd in Oxford, from whom the product is commercially available.

6.1 The B-Toolkit Managers

Integration of the toolkit is achieved by several cooperating managers; the two most important are the configuration manager, which incorporates a file manager and is responsible for the status of each single construct in a development, and the dependency manager, responsible for maintaining the inter-dependence between constructs.

These managers provide for the introduction (and removal) of constructs within a development, and ensure that when edits are to be committed, the extent of those edits is made known to the developer.

One of their most important functions is to ensure that all parts of the development are up-to-date, and reflect all changes that have been committed to the system (for example, it is impossible to produce code or documents that do not reflect the latest changes made in a specification). Equally important is the facility to automatically remake the system to some previous "high-water" mark once changes have been made (this process may be fine-tuned by the setting of appropriate flags).

Configuration Management is achieved through the Reset tool which, in cooperation with the B-Toolkit Managers, permits a development (at any stage) to be collapsed to its source files (which together with the toolkit dependency and configuration managers enable the complete system to be automatically rebuilt - including proof, documents and code). This facility provides for conformance with emerging standards such as BS 5750 and DEF STAN 00-55.

The B-Toolkit Managers also provide a full range of status and query functions.

6.2 Analysis

Processing of Abstract Machine specifications, refinements and implementations begin with the Analyser, providing syntax-checking and the "de-sugaring" of AMN into GSL. The former, of course, provides a rich "pseudo-programming" notation, very convenient for humans, the latter providing the concise substitution language which lends itself very well to machine processing and automation. All subsequent processing takes place on the GSL form of the construct.

The Analyser also ensures compliance with the semi-hiding and full-hiding principles.

The TypeChecker will report on the type correctness of each construct; each expression in a construct is assigned a type constructed on the given sets using the two operators $\mathbb{P}$ and $\times$ denoting respectively the power set constructor and the cartesian product constructor. Moreover, in a substitution the types of the elements situated on each side of the $:=$ operator must be the same.

6.3 Animation

It is of fundamental importance in any formal development of a computer system that the mathematics "captures" what is actually required; verification can ensure that a specification is internally consistent, or that refinement is correct with respect to its abstraction, but cannot demonstrate compliance with the (usually informal) requirements.

The B-Toolkit Animator provides an important validation facility, often overlooked in formal developments, that enables Abstract Machines to be "executed" and tested against expected behaviour. It may thus may be viewed as a very high-level prototyper (although no design need be provided by the developer).

It is envisaged that as well as a software house building computer systems on the B-Toolkit, the software procurer having the B-Toolkit could take delivery of specifications and refinements as they are written (rather than waiting for the final code to be delivered) and use validation tools such as the Animator to check progress.

6.4 Proof

One of the major advantages that a formal development method provides is the facility to provide verification conditions or proof obligations. A criticism of formal methods in the past has been the vast number of proof obligations provided by even the most modest of developments: thus automated machine support in this area is of fundamental importance.

The B-Toolkit POGenerator is an tool that automatically produces all proof obligations associated with Abstract Machine specifications, refinements and implementations. It encodes an efficient calculus implementing the rules presented on pages 21, 23, 24, 26, 27 and 28; a file of proof obligations is produced which may be subsequently presented to the B-Toolkit Provers (or, indeed, external provers if required).

The B-Toolkit provides for automated proof - the Autoprover - and for interactive proof - the InterProver. Both are built on a mathematical library containing a rich collection of mathematical laws for the underlying set-theoretic notation. The InterProver provides an environment in which the developer may supplement the library with further rules (such rules should, of course, be verified independently).

Both the POGenerator and AutoProver participate in the automatic Remake facility (if required); user-supplied theory is reused in an attempt to discharge as many of the new proof obligations as possible.

These tools also co-operate with the DocumentMarkUpTool [17] so that all proof obligations and proofs are automatically marked up, and may be either viewed independently or incorporated into the documentation of the development.

6.5 Specification/Module Library

The B-Toolkit Specification/Module Library provides a collection of off-the-shelf Abstract Machine specifications together with matching code, which may be employed in the construction of specifications (using the SEES, USES or INCLUDES facilities) or in the design of implementations (using the IMPORTS or SEES facility); such machines require no further development; in particular, they do not require analysis or proof (since they are guaranteed to be internally consistent) or refinement (since the code is provided).

The use of the library machines facilitate greatly the process of building an abstract specification and of its subsequent implementation.

Library machines provided include those for:

1. Manipulating types of objects: character string, bit string, scalar and boolean.
2. Encapsulation of a single object: variable, array, sequence, partial function and character string.
3. Encapsulation of a collection of objects: sets, sequences, strings and functions.
4. Simple i/o: terminal i/o for basic types of objects (numbers and strings etc), token i/o for entering and displaying elements of a specified type, persistent data facilities.

Each library machine provides a rich collection of operations for manipulating and accessing the encapsulated data.

A facility is also provided for the establishment of team libraries, so that specification/module pairs already developed are made accessible to other developers, as well as a facility for supplementing this system library.

All library machines may be submitted to the Animator and to the automatic InterfaceGenerator (see below).

6.6 Code Translators

The B-Toolkit Translator translates appropriate Abstract Machine implementations into object code; library modules participate fully in this process. When a development is complete, the Linker will produce executable code.

A C translator is provided; other code translators have been developed for IBM(UK) to produce PL/X and CICS code, and further translators are planned.

The Translator and Linker participate in the automatic Remake facility (if required).

6.7 Interface Generator

The InterfaceGenerator will automatically provide an interface with executable code for any Abstract Machine specification provided that it, and all subordinate constructs, have implementations built on the toolkit library; the i/o library machines will be used to provide appropriate functionality for each operation parameter.

The InterfaceGenerator thus provides a rapid-prototyping facility, and participates in the automatic Remake facility (if required).

6.8 Enumerator and Base Object Generators

The Enumerator generates the access functions associated with a given enumerated set, and an Abstract Machine specification and code modules are generated; thus the output from this generator may be viewed as a library machine.

The Base Object Generator is a powerful tool which interprets a declarative description of a database requirement (a syntactic form of entity relationship modelling) and automatically generates an Abstract Machine specification modelling that requirement, together with a rich set of operations to manipulate and access the data; an Abstract Machine implementation is also automatically generated, and built on the library object machines. The operations generated may be limited to those that are actually required in the development (this process is automatic) thus resulting in the provision of a tailor-made "instruction set".

All such generated Abstract Machines may be incorporated in the construction and design of other Abstract Machine components; they may be submitted to the Animator and to the InterfaceGenerator. The Enumerator and Base generator participate in the automatic Remake facility (if required).

6.9 Documentation

All Abstract Machine components (including interfaces, enumerations, bases, proof obligations, proofs and so on) may be submitted to the DocumentMarkUp tool which will automatically produce and compile LaTeX source code. Informal annotations may be included (which will, of course, be ignored by all other tools).

Components may be gathered together into a single Document, which provides full cross-referencing of variable, set, constant, operation and construct names at the clause and construct level, together with an index for the entire document. These features are automatic, but may be turned off.

As with all other tools, the DocumentMarkUp tool participates in the automatic Remake facility, if required.

7 Conclusion

The B-Method is designed to provide a homogeneous language, methodology and integrated tool support for the formal specification, design and implementation of real-life software systems. Therefore, the features of incremental construction and proof have been guiding principles in its development. It has now reached a mature and stable form, and a several books are in an advanced state of preparation providing a full account of the B-Method and its theoretical foundations. [15].

The B-Toolkit has been under alpha- and beta-test since January 1992, and was released in June 1994; The B-Method has been in use at GEC-Alsthom, which has for some years used an in-house tool environment (under licence from BP) to specify and develop railway control systems [19] [20],[21]. A prototype version of the B-Toolkit was used at BP International in building a fully operational graphics system [22] for visualising the flow of oil inside a refinery. MATRA Transport is using the current version of the B-Toolkit for software specification, and IBM (UK) is using it in the complete development of a new sub-system within CICS/ESA. The Parisian subway authority, RATP, and the French railways authorities, SNCF and INRETS, are funding the introduction of the B-Method into the French transport industries.

References

[1] Abrial et al, J-R., B-Technology Technical Overview, B-Core(UK) Ltd. 1993. (available from B-Core(UK) on request)

[2] Abrial, J-R., A Formal Approach to Large Software Construction, in van de Snepscheut, J.L.A (editor) Mathematics of Program Construction, Springer-Verlag, Berlin, 1989

[3] Abrial, J-R., On Constructing Large Software Systems, in van Leeuwen,J (editor), Algorithms, Software,Architecture. Information Processing 92, Vol. 1, Elsevier Science Publishers B.V., North Holland..

[4] Abrial, J-R., A Refinement Case Study (Using the Abstract Machine Notation), in *Proceedings of the Fourth Refinement Workshop*, Springer-Verlag, 1991.

[5] Lee, M.K.O., Scharbach, P.N., Sørensen, I.H., Engineering Real Software Using Formal Methods, in *Proceedings of the Fourth Refinement Workshop*, Springer-Verlag, 1991.

[6] Bicarregni, J., Richie, B., Invariants, Frames and Postconditions: a Comparison of the VDM and B Notations, in *FME'93: Industrial-Strength Formal Methods*, Springer-Verlag, 1993.

[7] Jones, C.B., *Software Development. A Rigorous Approach*, Prentice Hall, 1980.

[8] Jones, C.B., *Systematic Software Development Using VDM* Prentice Hall, Englewood Cliffs, NJ 1986.

[9] Spivey, J.M., *Understanding Z*, Cambridge University Press, 1988.

[10] Zermelo, E., *Untersuchung über die Grundlagen der Me genlehre*, 1908.

[11] Zermelo, E., *Neuer Beweis für die Möglichkeit ei er Wohlordnung der Mengenlehre*

[12] Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall, 1976.

[13] Morgan, C.C., *Programming From Specifications*, Prentice Hall, 1990.

[14] B-Core (UK), *B-Method Abstract Machine Summary.*

[15] Abrial, J-R., *The B-Method*, book to appear

[16] He, J. et al, *Data Refinement Refined*, PRG Oxford,1985

[17] Abrial, J-R., *B-Tool Reference Manual*, Edinburgh Portable Compiler, 1991

[18] N. Brown, D. Mery, *A proof Environment for Concurrent Programs*, in *FME'93: Industrial-Strength Formal Methods*, Springer-Verlag, 1993.

[19] Chapront,P. Vital Coded Processor and Safety Related Software Design, in Frey,H.H. (editor). Safety of Computer Control Systems 1992 (SAFECOMP 92), Computer System in Safety Critical Applications, Proceedings of the IFAC Symposium, Switzerland. Pergamon Press, 1992.

[20] Carnot, M., DaSilva, C., Dehbonei, B., Mejia, F., Error-free Software Development for Critical Systems Using the B-Methodology, in *Proceedings of the Third IEEE International Conference on Software Reliability Engineering*, October 92, North Carolina, USA.

[21] DaSilva, C., Dehbonei, B., Mejia, F., Formal Specification in the Development of Industrial Applications: The Subway Speed Control Mechanism, in *Proceedings of the Fifth International Conference on Formal Description Techniques, FORTE'92*, October 92, Lannion, France.

[22] Blamey, W., McNeil, I., *Productivity Gains Through the Use of the B-Method*, ITRU/273, BP Internal Report, 1991.

A Framework for Ad-Hoc Type Inference
(Extended Abstract)

Hosein Askari, Ole I. Hougaard*, Michael I. Schwartzbach
{hosask,hougaard,mis}@daimi.aau.dk
BRICS†
Department of Computer Science
University of Aarhus
8000 Aarhus C, Denmark

Abstract

Languages based on variations of the lambda calculus are designed to permit the slick, unification-based technique for type inference, which is by now a well-established discipline.

Other widely used languages have been created less by design and more by coincidence and compromise. It seems therefore that the question of type inference for such languages could be infeasible or should at least permit only ad-hoc solutions.

In this paper we argue the existence of a uniform framework for developing type inference algorithms, even for seemingly ad-hoc languages. This framework provides useful generalizations of well-known concepts related to type inference. It is, however, by no means a universal algorithm. In fact, the essential algorithmic problem must generally be solved from scratch.

We demonstrate the viability of our approach by developing a type inference algorithm for a full version of the Turbo Pascal language.

1 Introduction

The concept of *type inference* has been firmly established as a strong quality in a programming language. It has also traditionally been associated with *functional* programming languages, such as ML, which are based on variations of the lambda calculus and hence allow the slick, unification-based technique for polymorphic type inference.

However, type inference may certainly be just as relevant for ad-hoc languages, which have been created less by design and more by coincidence and

*Direct communications to this author
†Basic Research in Computer Science, Centre of the Danish Research Foundation

compromise. Clearly, if the type constraints are rather obscure, then automatic support will be correspondingly more useful.

It can naturally be argued that these ad-hoc languages should be eliminated and replaced by simpler, theoretically well-founded alternatives. For the purposes of this paper, we will merely observe that this process is sufficiently slow to justify an interest in building tools for these existing languages. Besides, the algorithmic challenge should certainly be met.

Note that if a language has a notion of type checking, then the problem of type inference is perfectly well-defined: Given a program (partially) without type annotations, does there exist a full annotation that allows the program to be type checked? This is simply an algorithmic problem, which one may attempt to solve.

It could be expected that the type inference problem for ad-hoc languages will turn out to be infeasible or at best permit only ad-hoc solutions. Certainly, it does not seem possible to generalize the unification-based approach.

In this paper we argue the existence of a uniform framework for developing type inference algorithms that to a large extent allows the ad-hoc nature of a programming language to be factored out. This framework should be viewed as a set of formalized rules-of-thumb that may guide the development of the algorithm; by no means do we present a universal algorithm.

This paper has two main parts, where we first provide useful generalizations of well-known concepts related to type inference and then show how to adapt them to ad-hoc situations.

In Section 2 we review the well-known unification-based technique for type inference in ML-style languages. For each fundamental concept do we provide suitable generalizations that enable the inclusion of ad-hoc features. The type domain is generalized from simple regular terms to term automata that capture the essence of name equivalence. The type constraints are generalized from equalities to more arbitrary relations respecting equivalence preserving homomorphisms. Based on these generalizations we propose a technique for representing solutions to type constraints, which may suggest the structure of a type inference algorithm. The essential algorithmic problem must however be solved from scratch. Finally, we show how polymorphism may be introduced through code expansion even in the likely absence of principal type schemes.

In Section 3 we demonstrate the viability of our framework by developing a polymorphic type inference algorithm for a full version of the Turbo Pascal language. It is safe to say that this language contains no special structure that will give us an unfair advantage when applying our framework. As predicted, we must solve an algorithmic problem specific to this application. However, we do this in a more general context that should cover several variations of the problem. The Turbo Pascal type inference algorithm has been implemented as a working preprocessor and suggests some interesting possibilities.

2 Generalized Concepts

We now recapitulate the well-known technique for performing type inference in ML-style languages. While presenting the familiar concepts, we will also introduce the generalizations necessary to handle more ad-hoc languages.

2.1 Type Domain

The types of core-ML are the expressions generated by the grammar below:

$$\begin{aligned} T_{\mathrm{ML}} ::= \quad & \alpha \mid \beta \mid \gamma \mid \ldots \\ & \mid T_{\mathrm{ML}} \to T_{\mathrm{ML}} \end{aligned}$$

where $\alpha, \beta, \gamma, \ldots$ are type variables. Such type expressions can be described as finite *terms* as defined by Courcelle [2]. Here we adopt the notation of Kozen [5].

Definition 2.1 *Let ω denote the set of natural numbers and let Σ be a ranked alphabet. A (Σ-)*term *is a partial function $t : \omega^* \to \Sigma$ whose domain is nonempty, prefix-closed, and respects arities in the sense that if $t(\alpha)$ is defined then*

$$\{i \mid t(\alpha i) \text{ is defined}\} = \{1, 2, \ldots, arity(t(\alpha))\}$$

A term is regular *if it has only finitely many subterms up to isomorphism.*

In the case of ML we have $\Sigma_{\mathrm{ML}} = \{\to, \alpha, \beta, \ldots\}$, $arity(\to) = 2$, and $arity(\alpha) = arity(\beta) = \cdots = 0$.

From [2] we adopt the notion of a *subtree* of t rooted at a node α, t/α, defined as $t/\alpha(\beta) = t(\alpha\beta)$. We will not distinguish between the singleton tree $\epsilon \mapsto l$ and the label l.

In our framework we represent types as finite *term automata* taken from Kozen [5].

Definition 2.2 *A* pure (Σ-)term automaton *is a tuple*

$$M = (Q, \Sigma, l, \delta)$$

where:

- *Q is a finite set of* states.
- *Σ is a ranked alphabet.*
- *$l : Q \to \Sigma$ is a labeling.*
- *$\delta : Q \times \omega \to Q$ is a partial function such that for all $q \in Q$,*

$$\{i \mid \delta(q, i) \text{ is defined}\} = \{1, 2, \ldots, arity(l(q))\}$$

Note that the automata used here differs from the ones in [5] in that they are finite. We will make another addition to the definition.

Definition 2.3 *A* (Σ-)*term automaton with* initial state *is a tuple*

$$M = (Q, q_0, \Sigma, l, \delta)$$

where (Q, Σ, l, δ) *is a pure* (Σ-)*term automaton, and* $q_0 \in Q$ *is an initial state. We write* $A(\Sigma)$ *for the set of* Σ*-term automata with initial state.*

In both kinds of term automata the function δ extends into a partial function $\hat{\delta} : Q \times \omega^* \to Q$ according to the inductive definition:

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= q \\ \hat{\delta}(q, \alpha i) &= \delta(\hat{\delta}(q, \alpha), i) \end{aligned}$$

with the understanding that the last expression is undefined if $\hat{\delta}(q, \alpha)$ is undefined.

Let $M = (Q, q_0, \Sigma, \delta, l)$ or $M = (Q, \Sigma, \delta, l)$ and $q \in Q$. The *subautomaton* of M rooted at q is

$$M[q] = (Q, q, \Sigma, \delta, l)$$

For $M = (Q, q_0, \Sigma, \delta, l)$ we also define M/α defined as

$$M/\alpha = M[\hat{\delta}(q_0, \alpha)]$$

For any automaton $M = (Q, q_0, \Sigma, \delta, l)$ the partial function defined by:

$$t^M(\alpha) = l(\hat{\delta}(q_0, \alpha))$$

is a regular Σ-term. For $q \in Q$ we write $t^M_q = t^{M[q]}$. Similarly for pure term automata. It is easy to see that $t^{M/\alpha} = t^M/\alpha$.

Conversely, every regular term corresponds to t^M for some term automaton, M, with initial state [7, lemma 8]. So we have the following.

Proposition 2.1 *The set of regular* Σ*-terms is isomorphic to the set of* Σ*-term automata with initial state modulo term equality.*

We can thus consider the ML types to be Σ_{ML}-term automata with initial state.

Let us now consider the set $\{t^M \mid M \in A(\Sigma)\}$. Clearly, it forms a free Σ-algebra. But there are type domains with restrictions on the types to which an operator can be applied. In order to deal with such situations we apply restrictions to the terms of the automata in the type domain.

Definition 2.4 *Let* Σ *be a ranked alphabet, and let* L *be a regular tree language. The class of legal automata with respect to* Σ *and* L *is*

$$\mathcal{A}(\Sigma, L) = \{M \in A(\Sigma) \mid t^M \in L\}$$

In ML the type domain is then $\mathcal{A}(\Sigma_{\mathrm{ML}}, \{t^M \mid M \in A(\Sigma_{\mathrm{ML}})\}) = A(\Sigma_{ML})$

2.2 Type Equivalence

The notion of type equivalence a central concept in a language definition, reflecting the essential properties of types. Most languages incorporate either *structural equivalence* or *name equivalence*. In the following we will generalize structural equivalence in a way that provides a simple framework for dealing with both equivalences.

It follows from the previous subsection that structural equivalence between term automata is simply equality between their terms. It is clear that for any two terms, t and t', we have that

$$t = t' \Leftrightarrow t(\epsilon) = t'(\epsilon) \wedge \forall i \in \{1, 2, \ldots, arity(t(\epsilon))\} : t/i = t'/i$$

In the case of finite trees, this is actually an unambiguous definition of the relation $=$. In the more general case of regular trees, there is a number of relations having the same property as $=$. In fact, $=$ is the most general of the equivalence relations with the above property. We would expect that the equivalence relation $t^M = t^{M'}$ would have a similar property, and it does as stated in the lemma below.

Lemma 2.2 *The equivalence relation $t^M = t^{M'}$ is the most general of all equivalence relations, $\approx$, with the property that for all M and M':*

$$M \approx M' \Leftrightarrow t^M(\epsilon) = t^{M'}(\epsilon) \wedge \forall i \in \{1, \ldots, arity(t^M(\epsilon))\} : M/i \approx M'/i$$

That is for any $\approx$ with the above property $M \approx M' \Rightarrow t^M = t^{M'}$.

The property in the above lemma implies that every label in Σ is equivalence preserving. Now, let us consider a situation where only *some* of the labels have this property. Choose a subset $S \subseteq \Sigma$. We want the above to apply only if the label is in S. If we write

$$M \approx M' \Leftrightarrow t^M(\epsilon) = t^{M'}(\epsilon) \in S \wedge \forall i \in \{1, \ldots, arity(t^M(\epsilon))\} : M/i \approx M'/i$$

then we wind up with a relation that is not necessarily reflexive. Hence we need to take the reflexive closure. The definition is as follows.

Definition 2.5 *Let $S \subseteq \Sigma$. The equivalence relation $\approx_S$ is the most general among the equivalence relations, $\approx$, with the property that for all M and M':*

$$M \approx M'$$
$$\Updownarrow$$
$$(t^M(\epsilon) = t^{M'}(\epsilon) \in S \wedge \forall i \in \{1, \ldots, arity(t^M(\epsilon))\} : M/i \approx M'/i) \vee (M = M')$$

Definition 2.5 is unpleasantly indirect. The proposition below gives a more direct definition of $\approx_S$.

Proposition 2.3 *$\approx_S$ is the relation:*

$$M \approx_S M' \Leftrightarrow \forall \alpha \in \omega^* : t^M(\alpha) = t^{M'}(\alpha) \in S \vee M/\alpha = M'/\alpha$$

with the understanding that undefined equals undefined.

2.3 Type Variables

A given term in core-ML gives rise to a finite collection of *type variables*, in principle one for each subterm or parse tree node. However, the type variables corresponding to different occurrences of the same identifier are identified. We shall use the notation $[\![E]\!]$ for the type variable corresponding to the subterm E. We denote by $\mathcal{V}$ the set of type variables.

2.4 Type Constraints

In general, a type constraint is a relation on types. We can, however, be more specific than simply to allow arbitrary relations. Even seemingly ad-hoc constraints share some common structure.

Given a term in core-ML, we generate the required constraints inductively in the syntax as follows. For a subterm of the form $\lambda x.E$ we have the constraint

$$[\![\lambda x.E]\!] = [\![x]\!] \to [\![E]\!],$$

and for a subterm of the form $E_1 E_2$ the constraint

$$[\![E_1]\!] = [\![E_2]\!] \to [\![E_1 E_2]\!].$$

The type constraint $[\![E]\!] = [\![E']\!] \to [\![E'']\!]$ can be described as follows. If M, M' and M'' are automata for the values of the type variables in $[\![E]\!] = [\![E']\!] \to [\![E'']\!]$, then we require

$$M = M' \to M'' \Leftrightarrow (t^M(\epsilon) = \to) \wedge (M/1 \approx_\Sigma M') \wedge (M/2 \approx_\Sigma M'').$$

Since types are automata, it is possible to define very general constraints. In particular, it is possible to define constraints which are dependent on the actual initial state (or other states) of the automaton, e.g. by demanding that the initial state should be an integer in the range from 1 to 7. We consider constraints of this sort to be an abuse of this framework. In order to avoid it, we demand that all the constraints should be resistant to replacement of the set of states as long as this replacement does not violate the type equivalence. We need a further concept to formalize this.

Definition 2.6 *A* homomorphism *is a function:*

$$f : (Q, q_0, \Sigma, \delta, l) \to (Q', q_0', \Sigma, \delta', l')$$

where

$$f : Q \to Q'$$
$$f(q_0) = q_0'$$
$$\delta'(f(q), i) = f(\delta(q), i)$$
$$l'(f(q)) = f(l(q))$$

It is clear that any homomorphism is term preserving. That is, if $f : M \to M'$ is a homomorphism, then $M \approx_\Sigma M'$. This means that any relation over terms is resistant to any homomorphism (i.e. $R(M) \Leftrightarrow R(M')$, where M and M' are as above). For general S, though, we need a stronger restriction on the homomorphisms.

Definition 2.7 *An indexed family of homomorphisms:*

$$(f_i : M_i \to M'_i)_{i \in I}$$

is $\approx_S$-preserving *iff:*

$$\forall i, j \in I, q \in Q_i, q' \in Q_j : M_i[q] \approx M_j[q'] \Leftrightarrow M'_i[f_i(q)] \approx M'_j[f_j(q')]$$

where $M_i = (Q_i, q_{i0}, \Sigma, \delta_i, l_i)$ *and* $M'_i = (Q'_i, q'_{i0}, \Sigma, \delta'_i, l'_i)$.

For an indexed family of homomorphisms, $\Phi = (f_i : M_i \to M'_i)_{i \in I}$, we introduce the following notation:

$$\begin{aligned} \Phi(M_i) &= M'_i \\ dom(\Phi) &= \{M_i \mid i \in I\} \\ codom(\Phi) &= \{M'_i \mid i \in I\} \end{aligned}$$

Definition 2.8 *A* constraint alphabet *is a ranked alphabet,* Γ*, of relations, where each* $R \in \Gamma$ *satisfies the following: For all automata* $M_1, \ldots, M_n$*, and for any* $\approx_S$*-preserving family of homomorphisms,* Φ*, such that* $\{M_1, \ldots, M_n\} \subseteq dom(\Phi)$*, we have:*

$$R(M_1, ..., M_n) \Leftrightarrow R(\Phi(M_1), ..., \Phi(M_n))$$

where $n = arity(R)$*. A* constraint *from* Γ *is of the form* $R(v_1, \ldots, v_n)$*, where* $R \in \Gamma$*,* $v_i \in \mathcal{V}$*, and* $n = arity(R)$*.*

In general, the type constraints generated for a given term can be described using simple rules following the syntax. For each production in the context-free grammar, such as $X ::= aYbZc$, a finite number of constraints of the form $R([\![X]\!], [\![Y]\!], [\![Z]\!])$ are generated.

2.5 Solutions

The hardest problem in a type inference arises when trying to find a solution to the set of constraints generated from a specific program. This is often done by at any point to represent the set of solutions to the constraints read so far. We present here a uniform representation of a single solution that may sometimes suggest how to solve the constraints.

A solution to a set of constraints is an assignment of a type to each type variable that occurs in the constraints, such that the constraints are satisfied when the type variables are replaced with the assigned types. We can thus represent a solution by a partial function: $\mathcal{S} : \mathcal{V} \to \mathcal{A}(\Sigma, L)$. We will look at an alternative representation of solutions that are more closely connected to automata, the *base automaton*.

Definition 2.9 *A* base automaton *over a set $\mathcal{V}$ of type variables and an alphabet, Σ, is a pure Σ-term automaton, $M = (Q, \Sigma, \delta, l)$, where Q is a set of non-empty, finite, pairwise disjoint subsets of $\mathcal{V}$. We write $\langle v \rangle_M$ for the element of Q containing v if it exists.*

Definition 2.10 *The solution corresponding to a base automaton $M = (Q, \Sigma, \delta, l)$ is the partial function $\mathcal{S}_M$ defined by $\mathcal{S}_M(v) = (Q, \langle v \rangle_M, \Sigma, \delta, l)$.*

In the following we will show that it is sufficient to look at base automata. Actually, S-minimal base automata can always be found. An automata is S-minimal, if it cannot be smaller and still have the same $\approx_S$ properties as in the definition below.

Definition 2.11 *A base automaton, M, is S-minimal, iff for each automaton M', $\mathcal{S}_{M'}(v) \approx_S \mathcal{S}_{M'}(v') \Leftrightarrow \mathcal{S}_M(v) \approx_S \mathcal{S}_M(v')$ implies that M has fewer states than M'.*

The advantage with S-minimal automata is that the relation $\approx_S$ becomes equality between initial states.

Proposition 2.4 *A base automaton M is minimal, iff $\mathcal{S}_M(v) \approx_S \mathcal{S}_M(v') \Rightarrow \langle v \rangle_M = \langle v' \rangle_M$.*

Proposition 2.5 *Let $\mathcal{S} : \mathcal{V} \to \mathcal{A}(\Sigma, L)$. There exists an S-minimal base automaton, M, and a $\approx_S$-preserving family of homomorphisms, Φ, such that for all $v \in dom(\mathcal{S})$: $\mathcal{S}_M(v) = \Phi(\mathcal{S}(v))$.*

Corollary 2.6 *Let C be a set of constraints. If C has a solution, then there is a minimal base automaton, M, such that $\mathcal{S}_M$ is a solution to C.*

This means that we can restrain ourselves to looking at types of the form $M[q]$, where M is an S-minimal base automaton. For such automata equivalence actually reduces to simple equality.

2.6 The Algorithm

In the end, type inference reduces to an algorithmic problem that is clearly outlined by the above definitions. Since the relations involved in the type constraints can be quite expressive, it is not possible to provide a completely general algorithm. For the simple case of core-ML, it is well-known that the constraints can be solved by means of the unification algorithm in time $O(n)$, where n is the size of the program [10].

It is sometimes thought that for more general languages one must seek to develop a more general notion of unification. It is our experience that this is not so. In spite of the uniform framework we present, it seems that ad-hoc languages give rise to ad-hoc algorithms. We have seen occurrences of algorithms as varied as the following: determinization of non-deterministic finite state automata [9], dynamic transitive closure [6], and in this paper a mixture between a name equivalent version of unification and a special case of general satisfiability.

2.7 Polymorphism

An integral part of the ML language is its approach to polymorphism. It is traditionally linked to the notion of *principal type scheme*, but we shall argue that the proper generalization is that of *code expansion*.

Functions defined by the polymorphic let-construct in ML are assigned a principal type scheme, which is instantiated with fresh type variables at all call sites. Imagine now that we syntactically expand these let-constructs. This will yield a potentially exponentially larger monomorphic term that is typable if and only if the original term is typable. This is not surprising, since type variables are derived from parse tree nodes. Mairson [8] and Kfoury et al [4] have shown that type inference in the polymorphic case is complete for deterministic exponential time, so we have not even compromised efficiency in the asymptotic sense.

For more ad-hoc type systems it is less likely that principal type schemes will exist. Thus, our generalized approach to polymorphism will be that of code expansion, which is clearly always a meaningful transformation. For ML-style languages we obtain in principle the traditional result, even if we are then left to discover the possibilities of compact representations of types and efficient polymorphic code generation.

3 An Application to Turbo Pascal

We now have generalized the concepts underlying the technique for ML-style type inference. We next endeavor to demonstrate that these generalizations are sensible and practical, rather than merely a formal exercise. To that purpose we develop an algorithm for type inference in the Turbo Pascal language—truly a worthy representative of the ad-hoc school of languages.

There is of course not a formalized description of the Turbo Pascal type system. We have therefore had to rely on the informal reference manual, from which we have been able to deduce a formal model that has been verified through experiments with the implementation.

3.1 Type Domain

We define the type domain for Turbo Pascal by giving the alphabet and the regular tree language. This is an example where the regular tree language is different from the free algebra generated by Σ.

Turbo Pascal provides a finite set $\mathcal{C}_{TP}$ of type constants and a ranked alphabet:

$$\begin{aligned}\Sigma_{TP} \quad = \quad & \mathcal{C}_{TP} \\ & \cup \{(a_1, \ldots, a_n) \mid n > 0 \wedge i \neq j \Rightarrow a_i \neq a_j\} \\ & \cup \{\mathrm{record}(a_1, \ldots, a_n) \mid n \geq 0 \wedge i \neq j \Rightarrow a_i \neq a_j\} \\ & \cup \{\mathrm{set}, \mathrm{array}, \wedge, \mathrm{file}\}\end{aligned}$$

where $arity(c) = 0$ for all $c \in \mathcal{C}$, $arity(a_1, \ldots, a_n) = 0$, $arity(\mathrm{set}) = arity(\mathrm{file}) = arity(\wedge) = 1$, $arity(\mathrm{record}(a_1, \ldots, a_n)) = n)$, and $arity(\mathrm{array}) = 2$.

In Turbo Pascal there is a subset $\mathcal{O}_0 \subseteq \mathcal{C}_{TP}$ of *ordinal types*. This set is extended to $\mathcal{O} = \mathcal{O}_0 \cup \{(a_1, \ldots, a_n) \mid n > 0 \wedge i \neq j \Rightarrow a_i \neq a_j\}$. Furthermore we have the set $\overline{F}$ of Σ_{TP}-terms *not* containing the label *file*. The regular tree language L_{TP} is the set of Σ_{TP}-terms such that the subterm of a node with label *set* will be in $\mathcal{O}$, the left subterm of a node with label *array* will be in $\mathcal{O}$, and the subterm of a node with label *file* will be in $\overline{F}$. That is

$$L_{TP} = \{t \mid \forall \alpha : t(\alpha) \in \{\text{set}, \text{array}\} \Rightarrow t/\alpha 1 \in \mathcal{O} \wedge t(\alpha) = \text{file} \Rightarrow t/\alpha 1 \in \overline{F}\}$$

To see that L_{TP} is a regular tree language simply construct the obvious grammar with nonterminals *ord* generating $\mathcal{O}$ and *notfile* generating $\overline{F}$.

All in all, the type domain in Turbo Pascal is $\mathcal{A}(\Sigma_{TP}, L_{TP})$.

3.2 Type Equivalence

The Turbo Pascal type system is distinguished from many others by using the notion of *name equivalence*, whereas ML incorporates *structural equivalence*. That is, two types are considered equal if and only if they have the same tree structure. As we have seen, this corresponds to term equality between automata. In name equivalence, types are equal if and only if they have the same name or they are explicitly defined to be equal in an equational system.

In general, we consider a ranked alphabet, Σ, and a countably infinite set, $\mathcal{V}$, of variables. The set of *right-hand sides* over $\mathcal{V}$ and Σ is $\mathcal{R}(\mathcal{V}, \Sigma) = \mathcal{V} \cup \{l(v_1, \ldots, v_n) \mid l \in \Sigma, \mathit{arity}(l) = n, v_i \in \mathcal{V}\}$. An *equational system*, $\mathcal{E}$, over $\mathcal{V}$ and Σ, is a finite subset $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{R}(\mathcal{V}, \Sigma)$. We will say that $\mathcal{E}$ is *well-defined*, if the left-hand sides are all different, *closed* if any variable on a right-hand side occurs on a left-hand side, and *normalized* if every right-hand side is of the form $l(v_1, \ldots, v_n)$.

If $\mathcal{E}$ is well-defined, we will view it as a partial function $\mathcal{E} : \mathcal{V} \rightarrow \mathcal{R}(\mathcal{V}, \Sigma)$. So for $(v, r) \in \mathcal{E}$, we will write $\mathcal{E}(v) = r$. Similarly, we take $\mathit{dom}(\mathcal{E})$ and $\mathit{codom}(\mathcal{E})$ to be $\{v \mid (v, r) \in \mathcal{E}\}$ and $\{r \mid (v, r) \in \mathcal{E}\}$, respectively.

Definition 3.1 *Let $\mathcal{E}$ be a well-defined, closed type equation system. The name equivalence relation with respect to $\mathcal{E}$ is the reflexive, symmetric, and transitive closure of the relation $\mathcal{E} \cap \mathcal{V} \times \mathcal{V}$. We write this relation as $\sim_{\mathcal{E}}$, and the equivalence classes as $\langle v \rangle_{\mathcal{E}} = \{v' \mid v \sim_{\mathcal{E}} v'\}$.*

Definition 3.2 *Let $\mathcal{E}$ be a well-defined, closed equational system over $\mathcal{V}$ and Σ. The* quotient *of $\mathcal{E}$ is an equational system over $2^{\mathcal{V}}$ and $\Sigma \cup \{\Omega\}$:*

$$\begin{aligned} \mathcal{E}/\sim_{\mathcal{E}} \; = \; & \{(\langle v \rangle_{\mathcal{E}}, l(\langle v_1 \rangle_{\mathcal{E}}, \ldots, \langle v_n \rangle_{\mathcal{E}})) \mid (v, l(v_1, \ldots, v_n)) \in \mathcal{E}\} \\ & \cup \{(\langle v \rangle_{\mathcal{E}}, \Omega) \mid \forall v' \in \langle v \rangle_{\mathcal{E}} : \mathcal{E}(v) \in \mathcal{V}\} \end{aligned}$$

Clearly, $\mathcal{E}/\sim_{\mathcal{E}}$ is well-defined, closed, and normalized.

Definition 3.3 *Let $\mathcal{E}$ and $\mathcal{E}'$ be well-defined, closed equational systems. We say that $\mathcal{E}$ and $\mathcal{E}'$ are* congruent, *iff $\mathcal{E}/\sim_{\mathcal{E}} = \mathcal{E}'/\sim_{\mathcal{E}'}$.*

Definition 3.4 *Let $\mathcal{E}$ be a well-defined, closed, and normalized equational system over $\mathcal{V}$ and Σ. We define* $\textsc{Auto}(\mathcal{E}) = (Q, \Sigma, \delta, l)$, *where* $Q = dom(\mathcal{E})$, *and for each* $(v, \lambda(v_1, \ldots, v_n)) \in \mathcal{E}$, *we have* $\delta(v, i) = v_i$ *and* $l(v) = \lambda$.

For a well-defined, closed equational system, $\mathcal{E}$, we can construct the automaton $\textsc{Auto}(\mathcal{E}/\sim_{\mathcal{E}})$. With this construction we can prove the proposition below.

Proposition 3.1 *The set of base automata over $\mathcal{V}$ and $\Sigma \cup \{\Omega\}$ is isomorphic to the set of well-defined, closed equational systems over $\mathcal{V}$ and Σ modulo congruence.*

In Turbo Pascal the programmer defines an equational system, and then uses the type variables to annotate the program.

Definition 3.5 *A type in Turbo Pascal is a pair $(\mathcal{E}, v)$, where $\mathcal{E}$ is a well-defined, closed equation system over $\mathcal{V}$ and Σ_{TP}, and $v \in dom(\mathcal{E})$.*

Definition 3.6 *Two types, $(\mathcal{E}, v)$ and $(\mathcal{E}', v')$ are* name equivalent, *iff $\mathcal{E}$ and $\mathcal{E}'$ are congruent and $v \sim_{\mathcal{E}} v'$. We write this as $(\mathcal{E}, v) \approx (\mathcal{E}', v')$.*

Lemma 3.2 *Let* $M = \textsc{Auto}(\mathcal{E}/\sim_{\mathcal{E}})$ *and* $M' = \textsc{Auto}(\mathcal{E}'/\sim_{\mathcal{E}'})$. *We have that*

$$(\mathcal{E}, v) \approx (\mathcal{E}', v') \Leftrightarrow \mathcal{S}_M(v) \approx_{\emptyset} \mathcal{S}_{M'}(v')$$

It is impractical to use the relation $\approx_{\emptyset}$. Consider for example the following program:

> **Var** x: Integer;
> **Var** y: Integer;
> x := y

Any programmer would expect this to work properly. So in Turbo Pascal the name equivalence relation is loosened to $\approx_{\mathcal{C}_{TP}}$ (which we shall write $\approx_{\mathcal{C}}$).

3.3 Type Variables

As for ML we generate a finite collection of type variables, but in this case only for parse tree nodes corresponding to expressions or variable references, since these are the only ones that can have types. We identify type variables corresponding to different occurrences of the same variable reference. We use the same notation as for ML.

3.4 Type Constraints

The Turbo Pascal constraints are quite complex. They reflect the ad-hoc nature of the language.

In Turbo Pascal we have a finite set $\mathcal{W}$ of finite sets of terms. The intention with $\mathcal{W}$ is to represent restrictions to overloading of operators. We have the constraint alphabet:

$$\begin{array}{rcl}\Gamma_{TP} & = & \{\in w \mid w \in \mathcal{W}\} \\ & & \cup\,\{=l \mid l \in \Sigma_{TP}\} \\ & & \cup\,\{Rec_\alpha \mid \alpha \text{ is a field name}\} \\ & & \cup\,\{=, Tc, :=, Io, Op\}\end{array}$$

where $\in w$ has arity 1, $=l$ has arity $arity(l)+1$, Rec_α, $=$, Tc, $:=$, and Io has arity 2, and Op has arity 3. Let us look more closely at the constraints from this alphabet.

- $v \in w$ is used to restrict the other constraints, when they are too general. If M is the type corresponding to v, we must have $t^M \in w$.
- $v = l(v_1, \ldots, v_n)$ appears in connection with referencing subvariables. E.g. for the variable reference $\hat{}x$ we have the constraint $[\![x]\!] = \hat{}([\![\hat{}x]\!])$. If $M, M_1, \ldots, M_n$ are the types of $v, v_1, \ldots, v_n$ then we require $t^M(\epsilon) = l \wedge \forall i \in \{1, \ldots, n\} : M/i \approx_{\mathcal{C}} M_i$.
- $Rec_\alpha(v, v')$ is similar to the above. The variable reference $x.\alpha$ gives rise to the constraint $Rec_\alpha([\![x]\!], [\![x.\alpha]\!])$. If v and v' have types M and M' there must be a sequence $\alpha_1, \ldots, \alpha_n$ and an i between 1 and n, such that $t^M(\epsilon) = record(\alpha_1, ..., \alpha_n) \wedge \alpha = \alpha_i \wedge M' \approx_{\mathcal{C}} M/i$. This definition differs from the above by allowing records to be extended.
- $v = v'$ arises in a number of places. For example in procedure calls, where the actual type must equal the formal type of a variable parameter. M and M' is a solution to $v = v'$, iff $M \approx_{\mathcal{C}} M'$.
- $v\ Tc\ v'$, $v := v'$, $v\ Io\ v'$, and $Op(v, v', v'')$. These constraints are concerned with type coercion, overloading and the like. They are the most complex of the Turbo Pascal constraints. As examples of where they arise, we can take $x := e$, where the constraint is $[\![x]\!] := [\![e]\!]$ or $e + e'$, where we generate the constraints $Op([\![e]\!], [\![e']\!], [\![e+e']\!])$, $[\![e]\!] \in w$, $[\![e']\!] \in w$, and $[\![e+e']\!] \in w$, where

  $$w = \{\text{Integer}, \text{Real}, \text{String}, \text{Char}, \text{array}(\text{Integer}, \text{Char})\} \cup \{t \mid t(\epsilon) = \text{set}\}.$$

  The definitions of these relations show how complex they are. For example Tc is defined as the least symmetric relation such that $M\ Tc\ M'$, if one of the following holds:

  - $M \approx_{\mathcal{C}} M'$
  - $t^M, t^{M'} \in \{\text{Integer}, \text{Real}\}$
  - $t^M(\epsilon) = t^{M'}(\epsilon) = \text{set} \wedge M/1\ \ Tc\ \ M'/1$
  - $t^M = t^{M'} = \text{array}(\text{Integer}, \text{Char})$
  - $t^M = \text{String} \wedge t^{M'} \in \{\text{String}, \text{Char}, \text{array}(\text{Integer}, \text{Char})\}$

– $t^M = \text{Pointer} \wedge t^M(\epsilon) = \hat{}$

To see that the relations in Γ_{TP} are resistant to $\approx_{\mathcal{C}}$-preserving families of homomorphisms, it is enough to note that these are defined only by the terms of the automata and the relation $\approx_{\mathcal{C}}$.

3.5 Solutions

In Turbo Pascal it becomes very natural to represent the solution as a base automaton, since this is isomorphic to type equations. As we have hoped for the concept of a base automaton suggests an algorithm that can solve some of the constraints. For the rest of the constraints we have to be more inventive. The construction of the algorithm is outlined in the next subsection.

3.6 The Algorithm

The Turbo Pascal type constraints may seem quite arbitrary and without any redeeming structure. For example, the relation Tc is reflexive and symmetric, but not transitive. Even worse, the relation := is neither reflexive, symmetric, transitive, nor antisymmetric. Nonetheless, these relations have *some* structure that can be exploited to obtain an algorithm.

The basic idea of the algorithm is to represent a set of solutions (base automata) by a base automaton labeled by sets of labels. The idea is to label each state with the set of labels that it might take according to the constraints read so far. For each label we have the possible transitions out of the state. If we have yet to read a constraint for any of the variables in the state, we say that those variables are *fresh*. We mark this by a special label, $*$.

By using base automata over label sets we can represent the set of solutions to the relations $\in w$, $=l$, Rec_α, and $=$. Furthermore, by using name equivalent unification we can maintain a representation of the set of solutions to a set of constraints over these relations. What remains is to deal with the relations Tc, :=, Io, and Op. As one might expect, these relations are too complex to fit into the model of base automata over sets of labels.

In general, we consider relations $R(M_1, \ldots, M_n)$, which can be described as finite disjunctions of clauses, where each clause is a finite conjunction of literals, and each literal is either of the form $l(q) = \lambda$ or $M_i[q] \approx_{\mathcal{C}} M_j[q']$. Finding solutions to such relations is in general NP-hard, so we add extra requirements to the relations R. If these requirements are satisfied we can find a solution to a set of $O(n)$ constraints over $O(n)$ variables in time $O(n^2 \log n)$.

Turbo Pascal constraints violates these requirements in a few places, but in each case we can argue that the algorithm succeeds here as well. It seems that in general the requirements on the strong labels and the partition of the labels can be weakened somewhat without affecting the correctness of the algorithm, but it is not clear at the moment how to describe the full generality of the algorithm.

3.7 Polymorphism

As predicted in the previous section, Turbo Pascal does not allow principal types schemes. This can be demonstrated by the following simple example.

```
Procedure P(var x; y);
begin
    x := 2*y;
end;
```

In the above procedure, both the type of $[\![x]\!]$ and the type of $[\![y]\!]$ can be any of the types Integer and Real but not independently. If the type of $[\![x]\!]$ is Integer then the type of $[\![y]\!]$ must be Integer as well. If the type of $[\![x]\!]$ is Real then the type of $[\![y]\!]$ can be any of the types Integer and Real. Clearly, it is impossible to express this interdependence through type schemas, where the interdependence of two types is expressed through common type variables. It might be possible to express the interface of a procedure in a more compact form than just the set of constraints. At this moment, however, it is not clear to us how this should be done. The only remaining possibility is to resort to code expansion. In any case, we would have to generate expanded code since Turbo Pascal has no native polymorphism.

We proceed by taking a procedure, which is called from two places within the program. We make a copy of the procedure with a new name, and replace one of the two calls by a similar call to the new procedure. We repeat this until no procedure is called from two different places. In the presence of recursive procedures this method will lead to infinite copying. This means, that just as in ML we must disallow recursive polymorphism.

We can view the problem as one concerning graphs in the following way. Any procedure is a node in the graph. In addition there is a main node for the main body of the program. For each procedure call there is an edge from the calling procedure to the called procedure. Assuming that all nodes in the graph are reachable from the main node, we can proceed by a breadth-first search from the main node, coloring the nodes as we meet them and copying a node when we meet it for the second time. In order to deal with cycles in the graph (corresponding to recursive calls) we construct the *component graph* before beginning the search. The component graph has as nodes the maximal, strongly connected components and as edges those going between different components.

By proceeding down the block structure of a program in a depth-first, post-order fashion and using the above method on each scope we have the full code expansion algorithm. In the worst case the size of the expanded graph is exponential, so the asymptotic time of code expansion will be exponential in the size of the program. For a program of size n we get that the total time for type inference with code expansion is $2^{O(d \log n)}$, where d is the longest path through the component graph. This is the same time bound that Kanellakis and Mitchell [3] found for type inference of ML.

4 Conclusion

The framework that we have presented is sufficiently general to cover the spectrum from ML to Turbo Pascal. Even so, it is sufficiently specific to provide substantial assistance in the development of algorithms.

The tool that we have constructed for Turbo Pascal suggests how the benefits of type inference may also come to influence the programming methodology for standard imperative languages. The algorithm we have presented can easily be adapted to, say, Modula-2 or C.

It is safe to assume that ad-hoc languages will continue to have a presence in our culture. We hope by this work to demonstrate that modern tools can provide substantial help in using them.

References

[1] Bruno Courcelle. Infinite trees in normal form and recursive equations having a unique solution. *Mathematical Systems Theory*, 13:131–180, 1979.

[2] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(1):95–169, 1983.

[3] Paris C. Kanellakis and John C. Mitchell. Polymorphic unification and ML typing. In *16th ACM conf. on Principles Of Programming Languages*, pages 105–116, 1989.

[4] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41:368–398, 1994.

[5] Dexter Kozen. Logical aspects of set constraints. TR 94-1421, Department of Computer Science, Cornell University, Ithaca, New York, May 1994.

[6] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. In *Proc. 33rd Symposium on Foundations of Computer Science.*, pages 363–371. IEEE, October 1992.

[7] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *Proc. 20th Symp. Princip. Programming Lang.*, pages 419–428. ACM, January 1992.

[8] Harry G. Mairson. Decidability of ML typing is complete for deterministic exponential time. In *Seventeenth Symposium on Principles of Programming Languages*, pages 382–401. ACM Press, January 1990.

[9] Michael I. Schwartzbach. Type inference with inequalities. In *Proceedings of TAPSOFT'91*. Springer-Verlag (*LNCS* 493), 1991.

[10] M. Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, X:115–122, 1987.

CPO Models for a Class of GSOS Languages

Luca Aceto* Anna Ingólfsdóttir†

BRICS‡
Department of Mathematics and Computer Science
Aalborg University
Fredrik Bajersvej 7E
9220 Aalborg Ø, Denmark

Abstract

In this paper, we present a general way of giving denotational semantics to a class of languages equipped with an operational semantics that fits the GSOS format of Bloom, Istrail and Meyer. The canonical model used for this purpose will be Abramsky's domain of synchronization trees, and the denotational semantics automatically generated by our methods will be guaranteed to be fully abstract with respect to the finitely observable part of the bisimulation preorder. In the process of establishing the full abstraction result, we also obtain several general results on the bisimulation preorder (including a complete axiomatization for it), and give a novel operational interpretation of GSOS languages.

1 Introduction

In this paper, we present a general way of giving denotational semantics to a class of languages equipped with an operational semantics that fits the GSOS format of Bloom, Istrail and Meyer [6]. The canonical model used for this purpose will be Abramsky's domain of synchronization trees $\mathcal{D}$ presented in [1], and the denotational semantics automatically generated by our methods will be guaranteed to be fully abstract with respect to the finitely observable part of the bisimulation preorder studied in, e.g., [9, 11]. Moreover, in the process of establishing the full abstraction result, we also give an algorithm, along the lines of those given in [2], to generate a complete axiomatization of the bisimulation preorder. As a byproduct of our denotational semantics, we shall be able to establish very general results about the bisimulation preorder that would be hard to prove using purely operational definitions. (For an example, cf. Thm 4.10.)

*On leave from School of Cognitive and Computing Sciences, University of Sussex, Brighton BN1 9QH, UK. Partially supported by HCM project EXPRESS. Email: luca@iesd.auc.dk.

†Partially supported by a grant from the Danish Research Council. Email: annai@iesd.auc.dk.

‡Basic Research in Computer Science, Centre of the Danish National Research Foundation.

The class of GSOS systems we shall give denotational semantics to will have the structure of most standard process algebras (see, e.g., [12, 5]). They will consist of a set of operations to construct finite, acyclic process graphs, and a facility for the recursive definition of behaviours. Borrowing a terminology introduced in [10] in the context of denotational semantics, we shall refer to these languages as *compact* GSOS languages. Their operational semantics will be given in terms of a variation on the standard model of labelled transition systems that takes divergence information into account. This will be done in such a way that the bisimulation preorder is a precongruence with respect to all the operators in the language. In order to obtain this substitutivity result, special care must be taken in interpreting negative premises in GSOS rules; in particular, negative premises will only be interpreted over convergent (or fully specified) processes. Intuitively, this is because, in order to find out what a process can*not* do, we need to know precisely what its capabilities are, and the initial behaviour of a divergent process is only partially specified. A consequence of our choice is that, for example, the rule

$$\frac{x \stackrel{a}{\nrightarrow}}{odd(x) \stackrel{a}{\rightarrow} odd(x)} \tag{1}$$

cannot be used to derive that $odd(\Omega)$ has an a-labelled transition to itself, where Ω denote the typical totally divergent process with no transitions. Our first main result is that, with our choice of operational semantics for GSOS languages, the bisimulation preorder is substitutive with respect to all language contexts. Moreover, as a consequence of general results established by Abramsky in [1], we are able to give a characterization of the finitely observable (or finitary) part of the bisimulation preorder for every GSOS language. Intuitively, this is the preorder obtained by restricting the bisimulation preorder to observations of finite depth.

We then show how to automatically give a denotational semantics for a GSOS language in terms of Abramsky's domain of synchronization trees $\mathcal{D}$. To this end, it is sufficient to endow Abramsky's model with an appropriate continuous algebra structure in the sense of [8]. This we do by showing how the GSOS rules defining the operational semantics of an operation symbol f of a compact GSOS language can be used to define a continuous function $\mathbf{f}_{\mathcal{D}}$ of the appropriate arity over the domain of synchronization trees $\mathcal{D}$. In defining the semantic counterparts of the operations in a compact GSOS language, we shall rely on a description of the domain $\mathcal{D}$ presented in [10], where it is shown how to reconstruct $\mathcal{D}$ from a suitable preorder over finite synchronization trees. This view of $\mathcal{D}$ will allow us to define each semantic operation $\mathbf{f}_{\mathcal{D}}$ in stepwise fashion from monotonic operations over finite synchronization trees. We hope that this choice will make the presentation more accessible to readers who are unfamiliar with domain theory [13].

As a result of our general framework, we shall then show that the denotational semantics so obtained is guaranteed to be in complete agreement with the chosen behavioural semantics. More precisely, for every compact GSOS language, the denotational semantics produced by the general approach presented in this paper is always fully abstract with respect to the finitary part of the bisimulation preorder.

Due to strict space limitations, this paper contains no proofs, very few technical details, no comparison with related work and no concluding remarks. These may be found in the full version of this paper [3].

2 Preliminaries

We begin by reviewing the basic notions on transition systems and domain theory that will be needed in this study.

2.1 Labelled Transition Systems with Divergence

The operational semantics of the languages considered in this paper will be given in terms of a variation on the model of labelled transition systems that takes divergence information into account. We refer the interested readers to, e.g., [7, 11, 15] for motivation and more information on (variations on) this semantic model for reactive systems.

Definition 2.1 *A* labelled transition system with divergence (lts) *is a quadruple*

$$(\mathsf{P}, \mathsf{Lab}, \rightarrow, \uparrow)$$

where: P *is a set of* processes*, ranged over by* s, t*;* Lab *is a set of* labels*, ranged over by* ℓ*;* $\rightarrow \subseteq \mathsf{P} \times \mathsf{Lab} \times \mathsf{P}$ *is a* transition relation*;* $\uparrow \subseteq \mathsf{P}$ *is a* divergence predicate*, notation* $s\uparrow$*. As usual, we shall write* $s \xrightarrow{\ell} t$ *in lieu of* $(s, \ell, t) \in \rightarrow$*.*

We write $s\downarrow$, read "s definitely converges", iff it is not the case that $s\uparrow$. A useful source of examples for labelled transition systems with divergence is the set of finite synchronization trees over a set of labels Lab, denoted by $\mathsf{ST}(\mathsf{Lab})$. These are the sets generated by the following inductive definition:

$$\frac{\{\ell_i \in \mathsf{Lab}, t_i \in \mathsf{ST}(\mathsf{Lab})\}_{i \in I}}{\{\langle \ell_i, t_i \rangle \mid i \in I\}[\cup\{\bot\}] \in \mathsf{ST}(\mathsf{Lab})}$$

where I is a finite index set, and the notation $[\cup\{\bot\}]$ means optional inclusion of $\bot$. The set of finite synchronization trees $\mathsf{ST}(\mathsf{Lab})$ can be turned into a labelled transition system with divergence by stipulating that, for $t \in \mathsf{ST}(\mathsf{Lab})$: $t\uparrow$ iff $\bot$ is in t, and $t \xrightarrow{\ell_i} t_i$ iff $\langle \ell_i, t_i \rangle$ is in t. The behavioural relation over processes that we shall study in this paper is that of *prebisimulation* [11, 7, 15] (also known as *partial bisimulation* [1]).

Definition 2.2 *Let* $(\mathsf{P}, \mathsf{Lab}, \rightarrow, \uparrow)$ *be an lts. Let* $\mathsf{Rel}(\mathsf{P})$ *denote the set of binary relations over* P*. Define the functional* $F : \mathsf{Rel}(\mathsf{P}) \rightarrow \mathsf{Rel}(\mathsf{P})$ *by:*

$$\begin{array}{rcl} F(\mathcal{R}) & = & \{(s,t) \mid \forall \ell \in \mathsf{Lab} \\ & & \bullet\ s \xrightarrow{\ell} s' \Rightarrow \exists t' :\ t \xrightarrow{\ell} t' \textit{ and } s' \mathrel{\mathcal{R}} t' \\ & & \bullet\ s\downarrow \Rightarrow t\downarrow \textit{ and } [t \xrightarrow{\ell} t' \Rightarrow \exists s' :\ s \xrightarrow{\ell} s' \textit{ and } s' \mathrel{\mathcal{R}} t']\} \end{array}$$

A relation $\mathcal{R}$ is a prebisimulation iff $\mathcal{R} \subseteq F(\mathcal{R})$. We write $s_1 \lesssim s_2$ iff there exists a prebisimulation $\mathcal{R}$ such that $s_1 \mathrel{\mathcal{R}} s_2$.

The relation $\lesssim$ is a preorder over P based on a variation on bisimulation equivalence [12]. Its kernel will be denoted by $\sim$, i.e., $\sim = \lesssim \cap \lesssim^{-1}$. In this study, we shall be interested in relating the notion of prebisimulation to a denotationally defined preorder on processes. As the preorders induced by a denotational semantics are completely determined by how they act on *finite processes*, we shall be interested in comparing them with the "finitely observable", or *finitary*, part of the bisimulation preorder in the sense of, e.g., [7]. The following definition is from [1].

Definition 2.3 *The* finitary preorder $\lesssim^F$ *is defined on any lts by*

$$s \lesssim^F s' \Leftrightarrow \forall t \in \mathsf{ST}(\mathsf{Lab}).\ t \lesssim s \Rightarrow t \lesssim s' \ .$$

An alternative method for using the functional F to obtain a behavioural preorder is to apply it inductively as in [1, Page 165]. The ω-iterate of this inductive construction will be denoted by $\lesssim_\omega$. The preorders $\lesssim$, $\lesssim_\omega$ and $\lesssim^F$ are related thus: $\lesssim \ \subseteq \ \lesssim_\omega \ \subseteq \ \lesssim^F$. Moreover the inclusions are, in general, strict. The interested reader is referred to [1] for more information on these preorders.

2.2 A Domain Equation for Synchronization Trees

In this section we recall Abramsky's domain equation for synchronization trees, and introduce the background in domain theory that is necessary to understand the paper. The interested reader is referred to, e.g., [13] for more general information on the theory of domains.

The canonical domain we shall use to give a denotational semantics to a class of GSOS languages is the domain of synchronization trees over a countable set of labels Lab considered by Abramsky in his seminal paper [1]. This is defined to be the initial solution $\mathcal{D}(\mathsf{Lab})$ of the domain equation

$$\mathcal{D}(\mathsf{Lab}) \cong (\mathbf{1})_\perp \oplus P[\sum_{\ell \in \mathsf{Lab}} \mathcal{D}(\mathsf{Lab})] \tag{2}$$

where $\mathbf{1}$ is the one point domain, $(\cdot)_\perp$ is lifting, $\oplus$ is coalesced sum, $\sum$ is separated sum, and $P[D]$ denotes the Plotkin powerdomain of D (cf. [13] for details on these domain-theoretic operations). We henceforth omit the parameter Lab as it will always be clear from the context.

To streamline the presentation and make our results more accessible to uninitiated readers, in this study we shall abstract completely from the domain-theoretic description of $\mathcal{D}$ given by (2). Our description of the domain of synchronization trees $\mathcal{D}$ will follow the one given in [10], and we shall rely on results presented in that reference that show how to construct $\mathcal{D}$ starting from a suitable preorder on the set of finite synchronization trees $\mathsf{ST}(\mathsf{Lab})$. Our reconstruction of $\mathcal{D}$ will be given in three steps: First of all, we shall define a preorder $\sqsubseteq$ on the set of synchronization trees $\mathsf{ST}(\mathsf{Lab})$. This preorder will be a reformulation of the Egli-Milner preorder over $\mathsf{ST}(\mathsf{Lab})$ presented in [10]. Secondly, we shall relate the poset of compact elements of $\mathcal{D}$ to the poset of equivalence classes induced by $(\mathsf{ST}(\mathsf{Lab}), \sqsubseteq)$. Finally, we shall use the fact that $\mathcal{D}$ is the ideal completion of its poset of compact elements to relate it to $(\mathsf{ST}(\mathsf{Lab}), \sqsubseteq)$. This approach will allow us to factor the definition of the continuous algebra structure [8] on $\mathcal{D}$ given in Sect. 4 in three similar steps, hopefully making it simpler to understand.

Definition 2.4 *We define $\sqsubseteq$ as the least binary relation over* $\mathsf{ST}(\mathsf{Lab})$ *satisfying:*

$$t \sqsubseteq u \quad \textit{if} \quad \begin{array}{ll} \textit{(1)} & \langle \ell, t' \rangle \in t \Rightarrow \exists \langle \ell, u' \rangle \in u : \ t' \sqsubseteq u' \quad \textit{and} \\ \textit{(2)} & \bot \in u \Rightarrow \bot \in t \quad \textit{and} \\ \textit{(3)} & \langle \ell, u' \rangle \in u \Rightarrow (\bot \in t \ \textit{or} \ \exists \langle \ell, t' \rangle \in t : \ t' \sqsubseteq u') \end{array}$$

The relation $\sqsubseteq$ so defined is easily seen to be a preorder over $\mathsf{ST}(\mathsf{Lab})$, whose kernel will be denoted by $\simeq$. We can now relate the preorder of synchronization trees $(\mathsf{ST}(\mathsf{Lab}), \sqsubseteq)$ with the poset of compact elements of $\mathcal{D}$ in a way that will allow us to define, in a canonical way, continuous operations on $\mathcal{D}$ from monotonic ones on $(\mathsf{ST}(\mathsf{Lab}), \sqsubseteq)$. First of all, we recall here that $\mathcal{D}$ is, up to isomorphism, the algebraic complete partial order (cpo) whose poset of compact elements $(\mathcal{K}(\mathcal{D}), \sqsubseteq_{\mathcal{K}(\mathcal{D})})$ is defined in [1, Def. 3.9]. $\mathcal{K}(\mathcal{D})$ is a subset of the set of finite synchronization trees, and the relations $\sqsubseteq$ and $\sqsubseteq_{\mathcal{K}(\mathcal{D})}$ may be shown to coincide over it. As a consequence of this observation, from now on we shall always use $\sqsubseteq$ as our notion of preorder on $\mathcal{K}(\mathcal{D})$.

For a synchronization tree $t = \{\langle \ell_i, t_i \rangle \mid 1 \leq i \leq n\}[\cup\{\bot\}]$, its recursive convex-closure t^c is inductively defined as follows:

$$t^c \quad \triangleq \quad \mathsf{Con}(\{\langle \ell_i, t_i^c \rangle \mid 1 \leq i \leq n\}[\cup\{\bot\}]) \tag{3}$$

where Con denotes the convex-closure operation given in [1, Def. 3.2]. It is not difficult to see that, for every $t \in \mathsf{ST}(\mathsf{Lab})$, t^c is a compact element of $\mathcal{D}$. Moreover, the function $(\cdot)^c : (\mathsf{ST}(\mathsf{Lab}), \sqsubseteq) \to (\mathcal{K}(\mathcal{D}), \sqsubseteq)$ is monotonic and maps trees that are related by $\simeq$ to the same compact element of $\mathcal{D}$. Assume now that $\mathbf{f}_{\mathsf{ST}} : (\mathsf{ST}(\mathsf{Lab}), \sqsubseteq)^l \to (\mathsf{ST}(\mathsf{Lab}), \sqsubseteq)$ is a monotonic function. Using the fact that each element of $\mathcal{K}(\mathcal{D})$ is a synchronization tree, we may naturally use $\mathbf{f}_{\mathsf{ST}}$ to define a function $\mathbf{f}_{\mathcal{K}(\mathcal{D})} : (\mathcal{K}(\mathcal{D}), \sqsubseteq)^l \to (\mathcal{K}(\mathcal{D}), \sqsubseteq)$ as follows:

$$\mathbf{f}_{\mathcal{K}(\mathcal{D})}(\vec{d}) \quad \triangleq \quad \left(\mathbf{f}_{\mathsf{ST}}(\vec{d})\right)^c \ . \tag{4}$$

It is easy to see that the function so defined is guaranteed to be monotonic. This implies that we can use (4) to lift any Σ-preorder structure [10] on $(\mathsf{ST}(\mathsf{Lab}), \sqsubseteq)$ to a Σ-poset structure on $(\mathcal{K}(\mathcal{D}), \sqsubseteq)$ in the sense of [8]. In Sect. 4.2 we shall take advantage of this fact. Finally, from the theory of powerdomains [13], we know that the domain of synchronization trees $\mathcal{D}$ is, up to isomorphism, the ideal completion of the poset of compact elements $\mathcal{K}(\mathcal{D})$. As a result of this observation, we can extend any monotonic function $\mathbf{f}_{\mathcal{K}(\mathcal{D})} : (\mathcal{K}(\mathcal{D}), \sqsubseteq)^l \to (\mathcal{K}(\mathcal{D}), \sqsubseteq)$ to a continuous function $\mathbf{f}_{\mathcal{D}} : (\mathcal{D}, \sqsubseteq_{\mathcal{D}})^l \to (\mathcal{D}, \sqsubseteq_{\mathcal{D}})$ by:

$$\mathbf{f}_{\mathcal{D}}(\vec{k}) \triangleq \bigsqcup \left\{ \mathbf{f}_{\mathcal{K}(\mathcal{D})}(\vec{d}) \mid \vec{d} \in \mathcal{K}(\mathcal{D})^l \text{ and } \vec{d} \sqsubseteq_{\mathcal{D}} \vec{k} \right\} \ . \tag{5}$$

The interested reader is invited to consult, e.g., [8, Sect. 3.3] for a discussion of the properties afforded by this canonical extension. In what follows, all we shall need to know is that $\mathbf{f}_{\mathcal{D}}$ is guaranteed to be continuous. Thus (5) can be used to conservatively extend any Σ-po algebra structure on $(\mathcal{K}(\mathcal{D}), \sqsubseteq)$ to a continuous algebra structure on $\mathcal{D}$.

3 GSOS Languages

Let Var be a denumerable set of *meta-variables* ranged over by x, y. A *signature* Σ consists of a set of *operation symbols*, disjoint from Var, together with a function *arity* that assigns a natural number to each operation symbol. Throughout this paper, following the standard lines of algebraic semantics (see, e.g., [8]), we shall assume that signatures contain a distinguished function symbol Ω of arity zero to denote the totally unspecified, or divergent, process, i.e., a process about whose behaviour we have no information. The set $\mathbb{T}(\Sigma, \mathsf{Var})$ of *terms* over Σ and Var (abbreviated to $\mathbb{T}(\Sigma)$ when the set of variables is clear from the context or immaterial) is defined as usual. We shall use $P, Q, \ldots$ to range over terms and the symbol $\equiv$ for the relation of syntactic equality on terms. $\mathrm{T}(\Sigma)$ is the set of *closed* terms over Σ, i.e., terms that do not contain variables. Constants, i.e. terms of the form $f()$, will be abbreviated as f. A Σ-*context* $C[\vec{x}]$ is a term in which at most the variables $\vec{x}$ appear. $C[\vec{P}]$ is $C[\vec{x}]$ with x_i replaced by P_i wherever it occurs. Besides terms we have *actions*, elements of some given finite set Act, which is ranged over by a, b, c.

Definition 3.1 (GSOS Rules) *Suppose Σ is a signature. A* GSOS rule *ρ over Σ is an inference rule of the form:*

$$\frac{\bigcup_{i=1}^{l}\left\{x_i \stackrel{a_{ij}}{\rightarrow} y_{ij} | 1 \leq j \leq m_i\right\} \quad \cup \quad \bigcup_{i=1}^{l}\left\{x_i \stackrel{b_{ik}}{\not\rightarrow} | 1 \leq k \leq n_i\right\}}{f(x_1, \ldots, x_l) \stackrel{c}{\rightarrow} C[\vec{x}, \vec{y}]} \tag{6}$$

where all the variables are distinct, $m_i, n_i \geq 0$, f is an operation symbol from Σ with arity l, $C[\vec{x}, \vec{y}]$ is a Σ-context, and the a_{ij}, b_{ik}, and c are actions in Act.

The operation symbol f in rules of the form (6) will be referred to as the *principal operation* of the rule. If, for some i, $m_i > 0$ then we say that ρ *tests its i-th argument positively*. Similarly if $n_i > 0$ then we say that ρ *tests its i-th argument negatively*. An operation f *tests its i-th argument positively (resp. negatively)* if it occurs as principal operation of a rule that tests its i-th argument positively (resp. negatively). We say that an operation f *tests its i-th argument* if it tests it either positively or negatively.

Definition 3.2 (GSOS Systems) *A* GSOS system *is a pair $G = (\Sigma_G, R_G)$, where Σ_G is a finite signature and R_G is a finite set of GSOS rules over Σ_G containing no rules with Ω as principal operation.*

An example of GSOS system, the language $\mathcal{L}$, is presented in Fig. 1. We shall use this concrete language as a running example throughout the paper to illustrate our definitions and results. The language $\mathcal{L}$ is a subset of ACP [5] with action prefixing in lieu of general sequential composition. Its parallel composition operator, denoted by $||$, is parameterized with respect to a partial, commutative and associative communication function $\gamma : \mathsf{Act} \times \mathsf{Act} \rightharpoonup \mathsf{Act}$. An operation in $\mathcal{L}$ that uses the power of negative premises, at least in the presence of a non-trivial priority structure on actions, is the priority operation θ of Baeten, Bergstra and Klop [4]. In order to define this operation, we assume a given partial ordering relation $>$ on Act. Intuitively, $b > a$ is interpreted as "action b has priority over action a". The sub-language of $\mathcal{L}$ consisting only of the operations Ω, δ, $a.\_$ and $+$ will be denoted by $\mathsf{FINTREE}_\Omega$.

Signature	*Arity*	*Rules*
δ	0	no rules
Ω	0	no rules
$a.\_ \quad (a \in \mathsf{Act})$	1	$a.x \xrightarrow{a} x$
$+$	2	$\frac{x \xrightarrow{a} x'}{x+y \xrightarrow{a} x'} \quad \frac{y \xrightarrow{a} y'}{x+y \xrightarrow{a} y'}$
$\|\|$	2	$\frac{x \xrightarrow{a} x'}{x\|\|y \xrightarrow{a} x'\|\|y} \quad \frac{y \xrightarrow{a} y'}{x\|\|y \xrightarrow{a} x\|\|y'}$ $\frac{x \xrightarrow{a} x',\, y \xrightarrow{b} y'}{x\|\|y \xrightarrow{c} x'\|\|y'} \quad \gamma(a,b) \simeq c$
θ	1	$\frac{x \xrightarrow{a} x',\, (\forall b > a)\, x \not\xrightarrow{b}}{\theta(x) \xrightarrow{a} \theta(x')}$

Figure 1: The language $\mathcal{L}$

GSOS systems have been introduced and studied in depth in [6]. Intuitively, a GSOS system gives a language, whose constructs are the operations in the signature Σ_G, together with a Plotkin-style structural operational semantics [14] for it defined by the set of conditional rules R_G. In this study, the operational semantics of a GSOS system will be given in terms of labelled transition systems with divergence. In order to obtain this novel interpretation, we aim at using the rules in a GSOS system G to define a divergence predicate over terms and a transition relation in such a way that our definitions: (1) specialize to those originally given by Bloom, Istrail and Meyer in their seminal study [6] when divergence is not taken into account; (2) give results that are in agreement with those already presented in the literature when applied to known process description languages; and (3) produce operators that are well-behaved with respect to the notion of prebisimulation, i.e., operations for which prebisimulation is a precongruence.

First of all, we shall use the rules in a GSOS systems to define a divergence (or underspecification) predicate on the set of closed terms over Σ_G. In fact, as is common practice in the literature on process algebras, we shall define the notion of convergence, and use it to define the divergence predicate we are after. Intuitively, a term P is convergent if the set of its initial transitions is fully specified. The basic divergent term is Ω, the totally unspecified process. A term of the form $f(\vec{P})$ is convergent iff the set of its initial transitions only depends on those arguments P_is whose initial behaviour is completely known. This informal discussion motivates the following definition.

Definition 3.3 (Convergence) *Let $G = (\Sigma_G, R_G)$ be a GSOS system. The convergence predicate $\downarrow_G$ (abbreviated to $\downarrow$ when the GSOS system G is clear from the context) is the least predicate over $\mathrm{T}(\Sigma_G)$ that satisfies the following clause:*

$$f(P_1, \ldots, P_l) \downarrow_G \text{ if } f \neq \Omega \text{ and, for every argument } i \text{ of } f, \text{ if } f \text{ tests } i \text{ then } P_i \downarrow_G \quad .$$

We write $P \uparrow_G$ iff it is not the case that $P \downarrow_G$.

When applied to the language $\mathcal{L}$, Def. 3.3 gives the following convergence predicate: (1) $\delta \downarrow$; (2) if $P \downarrow$ and $Q \downarrow$, then $P + Q \downarrow$ and $P \| Q \downarrow$; and (3) if $P \downarrow$ then $\theta(P) \downarrow$. The reader familiar with the literature on prebisimulation over CCS-like languages will have noted the similarity of this definition with those given in, e.g., [9, 7, 1].

We shall now present our non-standard operational semantics for GSOS languages. As stated above, we take as our starting point the original theory developed by Bloom, Istrail and Meyer. However, in the presence of divergence information, we shall interpret negative transition formulae over *convergent processes* only. Intuitively, to know that a process cannot initially perform a given action, we need to find out precisely all the actions that it can perform. If a process is divergent, its set of initial actions is not fully specified; thus we cannot be sure whether such a process satisfies a negative transition formula or not. Informally, the intent of a GSOS rule in our setting is as follows. Suppose that we are wondering whether $f(\vec{P})$ is capable of taking a c-step. We look at each rule of the form (6) with principal operation f and action c in turn. We inspect each positive premise $x_i \xrightarrow{a_{ij}} y_{ij}$, checking if P_i is capable of taking an a_{ij}-step for each j and if so calling the a_{ij}-children Q_{ij}. We also check the negative premises; this involves finding out whether P_i is convergent, and, in that case, if P_i is *in*capable of taking a b_{ik}-step for each k. If so, then the rule *fires* and $f(\vec{P}) \xrightarrow{c} C[\vec{P}, \vec{Q}]$. We write $\rightarrow_G$ for the transition relation for G defined in this way. The lts with divergence specified by a GSOS system G is then given by $\mathsf{lts}(G) = (\mathrm{T}(\Sigma_G), \rightarrow_G, \uparrow_G)$. The largest prebisimulation over $\mathsf{lts}(G)$ will be denoted by $\lesssim_G$, and its kernel by $\sim_G$. (The subscript G will be omitted from these relations when this causes no confusion).

Example: We exemplify our approach using our running example, the language $\mathcal{L}$, by considering some identities involving simple terms that use the priority operation θ.

The term $\theta(\Omega)$ is divergent, as Ω is. Moreover it does not have any transition because Ω has none. We thus have that $\theta(\Omega) \sim \Omega$.

Consider a term of the form $P \equiv a.\delta + \Omega$, with a a maximal element in the poset $(\mathsf{Act}, >)$, i.e., with a an action with maximal priority. Then the rule for θ with action a has no negative premises, and it can be used to establish the transition $\theta(P) \xrightarrow{a} \theta(\delta)$. Indeed, this is the only transition that is possible from $\theta(P)$. As $\theta(P)$ is divergent, as P is, it is easy to see that $\theta(P) \sim a.\delta + \Omega$.

On the other hand, if a is *not* maximal in the poset $(\mathsf{Act}, >)$, the rule for θ with action a will have at least one negative premise. As P is divergent, that rule cannot be used to derive a transition from the term $\theta(P)$. It thus follows that $\theta(P) \sim \Omega$, if a is not maximal in the poset $(\mathsf{Act}, >)$. □

We are now ready to establish the first main result of this paper. Namely, we shall prove that the operations of a GSOS system preserve the semantic notion of prebisimulation.

Theorem 3.4 *Let G be a GSOS system. Then $\lesssim_G$ is a precongruence for all operation symbols f of G,* i.e., $(\forall i : P_i \lesssim_G Q_i) \;\Rightarrow\; f(\vec{P}) \lesssim_G f(\vec{Q})$.

3.1 GSOS Languages with Recursion

In this section we consider GSOS languages that may include recursion. Let $G = (\Sigma_G, R_G)$ be a GSOS system, and let PVar be a fresh denumerable set of *process variables* $(X, Y \in \mathsf{PVar})$. The set of recursive terms over Σ_G and PVar, denoted by $\mathsf{REC}(\Sigma_G, \mathsf{PVar})$, is given by the following BNF syntax:

$$P ::= X \mid f(P_1, \ldots, P_l) \mid \mathrm{fix}(X = P)$$

where $X \in \mathsf{PVar}$, f is an operation symbol in Σ_G of arity l and fix is a binding construct. This gives rise to the usual notions of free and bound variables in terms. The set of closed recursive terms (or *programs*) will be denoted by $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$. We shall assume a standard notion of substitution of terms for free process variables, and use $P\{Q/X\}$ to denote term P in which each free occurrence of X has been replaced by Q, after possibly renaming bound variables in P. We shall now define an operational semantics for the set of programs $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$ in terms of an lts with divergence, following the techniques presented in Sect. 3. In this study, the operational semantics of recursion will be given by the standard unfolding rule:

$$\frac{x\{\mathrm{fix}(X = x)/X\} \xrightarrow{a} y}{\mathrm{fix}(X = x) \xrightarrow{a} y} \tag{7}$$

This will ensure that recursive terms are interpreted as fixed-points, and that the following equation holds:

$$\mathrm{fix}(X = P) = P\{\mathrm{fix}(X = P)/X\} \ . \tag{8}$$

In order to define the operational semantics of $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$, we need, first of all, to extend the convergence predicate to $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$.

Definition 3.5 *The convergence predicate $\downarrow_{Grec}$ (abbreviated to $\downarrow$ when the GSOS system G is clear from the context) is the least predicate over $\mathsf{CREC}(\Sigma_G, \mathsf{Var})$ that satisfies the following clauses:*

1. $f(P_1, \ldots, P_l) \downarrow_{Grec}$ *if $f \neq \Omega$, and for every argument i of f, if f tests i then $P_i \downarrow_{Grec}$;*
2. $\mathrm{fix}(X = P) \downarrow_{Grec}$ *if* $P\{\mathrm{fix}(X = P)/X\} \downarrow_{Grec}$.

Again, we write $P \uparrow_{Grec}$ iff it is not the case that $P \downarrow_{Grec}$.

The motivation for the above definition is the following: a term P is divergent if its initial transitions are not fully specified. This occurs either when the initial behaviour of term P depends on underspecified arguments like Ω or in the presence of unguarded recursive definitions. For example, the terms $\mathrm{fix}(X = X)$ and $\mathrm{fix}(X = \mathit{odd}(X))$, where *odd* is the operation given by rule (1), are *not* convergent as the initial behaviour of these processes depends on itself. It is immediate to see that the predicates $\downarrow_G$ and $\downarrow_{Grec}$ coincide over $\mathrm{T}(\Sigma_G)$, the set of recursion-free terms in $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$.

We shall now show how to associate a transition relation with a GSOS language with recursion by putting the extra structure given by the convergence predicate to good use.

We construct the relation $\rightarrow_{Grec}$ over $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$ in two steps. In the first step of the construction, we derive the transitions emanating from *convergent terms* by induction on the convergence predicate following the approach outlined in the previous section, and using (7) to derive the transitions of recursive terms. In the second, we use the information about the transitions that are possible for convergent terms to determine the outgoing transitions for all the terms in $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$. For a divergent term P, a transition $P \stackrel{a}{\rightarrow}_{Grec} Q$ holds iff it is provable from the rules in R_G and those for recursion in the standard sense, with the proviso that every time we need to establish the validity of a negative premise we appeal to the information on the transitions that are possible from convergent terms discovered in the first step of our construction. To exemplify this construction on a pathological case, let us consider the term $\mathrm{fix}(X = \mathit{odd}(X))$, where the operation *odd* is given by rule (1). The operation *odd* is standardly used in the literature to show that negative premises and unguarded recursive definitions can lead to inconsistent specifications. Note, first of all, that $\mathrm{fix}(X = \mathit{odd}(X))$ is a divergent term. It is then easy to see that, because of our requirement that negative premises in rules be interpreted over convergent terms only, rule (1) can never be applied to derive a transition for $\mathrm{fix}(X = \mathit{odd}(X))$. Thus we have that this term has *no* transition and is divergent, i.e., that $\mathrm{fix}(X = \mathit{odd}(X)) \sim \Omega$.

With the above definitions, the operational semantics of a GSOS language with recursion $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$ is given by the lts with divergence

$$\mathsf{lts}(G_{rec}) = (\mathsf{CREC}(\Sigma_G, \mathsf{PVar}), \mathsf{Act}, \rightarrow_{Grec}, \uparrow_{Grec}) \ .$$

As a corollary of general results by Abramsky, we can give the following general characterization of the finitary bisimulation preorder over $\mathsf{lts}(G_{rec})$ for arbitrary G:

Proposition 3.6 *The preorders $\lesssim^F$ and $\lesssim_\omega$ coincide over $\mathsf{lts}(G_{rec})$.*

4 Denotational Semantics

In this section we shall present a general technique to give denotational semantics in terms of the Plotkin powerdomain of synchronization trees (see Sect. 2.2) for a class of GSOS languages with recursion. The denotational semantics will be guaranteed to be *fully abstract*, in the sense of Milner and Plotkin, with respect to the finitary part of the prebisimulation relation $\lesssim_\omega$. The languages that we shall consider have the structure of most standard process calculi (see, e.g., [12, 5]); they will consist of a set of operations to build finite, acyclic labelled transition systems and a facility for recursive definitions of behaviours. Thus we shall consider GSOS languages with recursion in which infinite behaviours can only be defined by means of recursive definitions.

We assume that the reader is familiar with the basic notions of ordered and continuous algebras (see, e.g., [8, 10]); however, in what follows we give a quick overview of the way a denotational semantics can be given to a recursive language like $\mathsf{REC}(\Sigma_G, \mathsf{PVar})$ following the standard lines of algebraic semantics [8]. The interested reader is invited to consult [8] for an explanation of the theory.

Let $\mathcal{A}$ be any Σ_G-cpo, i.e., a Σ_G-algebra whose carrier is an algebraic cpo and whose operations are interpreted as continuous functions. A denotational semantics for the language $\mathsf{REC}(\Sigma_G, \mathsf{PVar})$ is given by the mapping $\mathcal{A}[\![\cdot]\!] : \mathsf{REC}(\Sigma_G, \mathsf{PVar}) \to [\mathrm{ENV}_\mathcal{A} \to \mathcal{A}]$ defined in, e.g., [8, Def. 4.2.3], where $\mathrm{ENV}_\mathcal{A} = [\mathsf{PVar} \to \mathcal{A}]$ is the set of $\mathcal{A}$-environments, ranged over by the metavariable ρ. For each $P \in \mathsf{CREC}(\Sigma_G, \mathsf{PVar})$, $\mathcal{A}[\![P]\!]\rho$ does not depend on the environment ρ. Hence the denotation of a closed term P will be denoted by $\mathcal{A}[\![P]\!]$. A general result about the semantic mapping $\mathcal{A}[\![\cdot]\!]$ states that, for any $P \in \mathsf{REC}(\Sigma_G, \mathsf{PVar})$, there exists an infinite sequence of *finite approximations* $P^n \in \mathbb{T}(\Sigma_G, \mathsf{PVar})$ ($n \in \mathbb{N}$) such that

$$\mathcal{A}[\![P]\!] = \bigsqcup\nolimits_{n\geq 0} \mathcal{A}[\![P^n]\!] \ . \tag{9}$$

For any binary relation $\mathcal{R}$ over $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$, the algebraic part of $\mathcal{R}$, denoted by $\mathcal{R}^A$ is defined as follows [8]:

$$P \; \mathcal{R}^A \; Q \quad \Leftrightarrow \quad \text{for every } P^n\text{, there exists } Q^m \text{ such that } P^n \; \mathcal{R} \; Q^m \ .$$

We say that $\mathcal{R}$ is *algebraic* iff $\mathcal{R}$=$\mathcal{R}^A$. Intuitively, a relation is algebraic if it is completely determined by how it behaves on recursion-free terms. Because of Eqn. 9, the relations over terms induced by a denotational semantics are always algebraic. We shall make use of this fact in the technical developments to follow.

4.1 Compact GSOS Systems

The following notion from [2] will allow us to pin down precisely a class of GSOS operations that map finite processes to finite processes. The semantic counterparts of these operations will have the property of being *compact* in the sense of [10], i.e., of mapping compact elements in the Plotkin powerdomain of synchronization trees to compact elements.

Definition 4.1 *A GSOS rule of the general form (6) is* linear *if each variable occurs at most once in the target and, for each argument i that is tested positively, x_i does not occur in the target and at most one of the y_{ij}'s does. An* *operation from a GSOS system G is* linear *iff all rules for it are linear. Finally, G itself is* linear *iff it only contains linear rules.*

The format of linear rules is a restriction of the general GSOS format in that no copying of arguments is allowed and no argument for which there is a positive premise may appear in the target of a rule. Moreover, there may be possibly many positive premises for an argument x_i in a rule, but at most one of the y_{ij}'s may appear in its target. As far as we know, all the operations occurring in the standard process algebras are linear.

Definition 4.2 ([2]) *A GSOS system G is* syntactically well-founded *iff there exists a function w from operation symbols in Σ_G to natural numbers such that, for each rule $\rho \in R_G$ with principal operation symbol f and target $C[\vec{x}, \vec{y}]$ the following conditions hold:*

- *if ρ has no positive premises then $W(C[\vec{x}, \vec{y}]) < w(f)$, and*
- *$W(C[\vec{x}, \vec{y}]) \leq w(f)$ otherwise,*

where $W : \mathbb{T}(\Sigma_G) \to \mathbb{N}$ is given by

$$\begin{aligned} W(x) &\triangleq 0 \\ W(f(P_1, \ldots, P_l)) &\triangleq w(f) + W(P_1) + \cdots + W(P_l). \end{aligned}$$

For example, the GSOS system in Fig. 1 is linear and syntactically well-founded. In fact, it is sufficient to assign weight 1 to the action prefixing operations and weight 0 to all the other operations. Syntactic well-foundedness is decidable over GSOS systems (cf. [2, Thm. 6.8]), and, for linear GSOS systems, it is sufficient to guarantee that terms are semantically well-founded in the sense of [2].

Definition 4.3 (Compact GSOS Systems) *A GSOS system is said to be* compact *iff it is linear and syntactically well-founded.*

4.2 Full Abstraction for Compact GSOS Systems

Let G be a compact GSOS language. We shall now give a way of defining, for each Σ_G-context $C[\vec{x}]$, a function $\mathbf{C}_{\mathsf{ST}}$ over $\mathsf{ST}(\mathsf{Act})$ of the appropriate arity. The definition of $\mathbf{C}_{\mathsf{ST}}$ will be given using the rules in R_G as a guideline. First of all, note that it is sufficient to define semantic operations $\mathbf{f}_{\mathsf{ST}}$ for each $f \in \Sigma_G$, as derived semantic operations can then be obtained by function composition. The definition of the functions $\mathbf{f}_{\mathsf{ST}}$ is given by the inductive construction in Def. 4.4. Intuitively, the inductive construction of the synchronization tree $\mathbf{f}_{\mathsf{ST}}(\vec{t})$ given below is well-founded because, by the compactness of G, whenever the premises of a rule of the form (6) can be met by $\vec{t}$, then either the weight of $C[\vec{x}, \vec{y}]$ is strictly smaller than that of f, or the weight of $C[\vec{x}, \vec{y}]$ is the same as that of f, and the sum of the sizes of the arguments of $\mathbf{C}_{\mathsf{ST}}$ has decreased.

Definition 4.4 *Let $G = (\Sigma_G, R_G)$ be a compact GSOS system, and let f be an l-ary operation in Σ_G. We define the operation $\mathbf{f}_{\mathsf{ST}} : \mathsf{ST}(\mathsf{Act})^l \to \mathsf{ST}(\mathsf{Act})$ inductively by stipulating that, for every $t_1, \ldots, t_l \in \mathsf{ST}(\mathsf{Act})$:*

- $\perp \in \mathbf{f}_{\mathsf{ST}}(t_1, \ldots, t_l)$ *iff* $f = \Omega$ *or there is an argument* i *for* f *such that* f *tests its* i*-th argument and* $\perp \in t_i$*;*

- $\langle c, t \rangle \in \mathbf{f}_{\mathsf{ST}}(t_1, \ldots, t_l)$ *iff there exist a rule for* f *of the form (6) and a vector of trees* $\vec{u} = u_{11} \ldots u_{1m_1} \ldots u_{l1} \ldots u_{lm_l}$ *such that:*

  1. *for all* $1 \leq i \leq l$*,* $1 \leq j \leq m_i$*,* $\langle a_{ij}, u_{ij} \rangle \in t_i$*,*
  2. *for all* $1 \leq i \leq l$*,* $1 \leq k \leq n_i$*,* $\perp \notin t_i$ *and, for no* $u \in \mathsf{ST}(\mathsf{Act})$*,* $\langle b_{ik}, u \rangle \in t_i$*, and*
  3. $\mathbf{C}_{\mathsf{ST}}[\vec{t}, \vec{u}] = t$*, where* $\mathbf{C}_{\mathsf{ST}}$ *denotes the derived semantic operation associated with the* Σ_G*-context* $C[\vec{x}, \vec{y}]$*. If* $C[\vec{x}]$ *is a variable* x_i*, then* $\mathbf{C}_{\mathsf{ST}}[\vec{t}] = t_i$*.*

When applied to the language $\mathcal{L}$, the construction in Def. 4.4 produces the following functions:

- $\delta_{\mathsf{ST}} = \emptyset$,
- $\Omega_{\mathsf{ST}} = \{\perp\}$,
- for every $t \in \mathsf{ST}(\mathsf{Act})$, $\mathbf{a}_{\mathsf{ST}}(t) = \{\langle a, t \rangle\}$,
- for every $t_1, t_2 \in \mathsf{ST}(\mathsf{Act})$, $t_1 +_{\mathsf{ST}} t_2 = t_1 \cup t_2$,
- for every $t_1, t_2 \in \mathsf{ST}(\mathsf{Act})$, $t_1 \|_{\mathsf{ST}} t_2$ is given by:

  1. $\perp \in t_1 \|_{\mathsf{ST}} t_2$ iff $\perp \in t_1$ or $\perp \in t_2$;
  2. $\langle c, t \rangle \in t_1 \|_{\mathsf{ST}} t_2$ iff one of the following holds:
     (a) there exists $\langle c, t_1' \rangle \in t_1$ such that $t = t_1' \|_{\mathsf{ST}} t_2$, or
     (b) there exists $\langle c, t_2' \rangle \in t_2$ such that $t = t_1 \|_{\mathsf{ST}} t_2'$, or
     (c) there exist $\langle a, t_1' \rangle \in t_1$ and $\langle b, t_2' \rangle \in t_2$ such that $c = \gamma(a, b)$ and $t = t_1' \|_{\mathsf{ST}} t_2'$.

- for every $t \in \mathsf{ST}(\mathsf{Act})$, $\theta_{\mathsf{ST}}(t)$ is given by:

  1. $\perp \in \theta_{\mathsf{ST}}(t)$ iff $\perp \in t$,
  2. $\langle c, t_1 \rangle \in \theta_{\mathsf{ST}}(t)$ iff there exists $\langle c, t' \rangle \in t$ such that:
     (a) either c is maximal in $(\mathsf{Act}, >)$ and $\theta_{\mathsf{ST}}(t') = t_1$,
     (b) or c is not maximal in $(\mathsf{Act}, >)$, for no action $b > c$ and t_2 $\langle b, t_2 \rangle \in d$, $\perp \notin t$, and $\theta_{\mathsf{ST}}(t') = t_1$.

We shall now prove that the definition of the operations given in Def. 4.4 endows the preorder of synchronization trees $\mathsf{ST}(\mathsf{Act})$ with a Σ_G-preorder structure in the sense of [10]. To this end, it is sufficient to prove that each operation $\mathbf{f}_{\mathsf{ST}}$ is monotonic with respect to the preorder $\sqsubseteq$.

Theorem 4.5 (Monotonicity) *Let* G *be a compact GSOS system, and* $f \in \Sigma_G$*. Then the function* $\mathbf{f}_{\mathsf{ST}}$ *given by the construction in Def. 4.4 is monotonic with respect to* $\sqsubseteq$*.*

Because of the above result, the construction of Def. 4.4 allows us to define, for each operation symbol f in the signature of a compact GSOS system, a monotonic function $\mathbf{f}_{\mathsf{ST}}$ over $\mathsf{ST}(\mathsf{Act})$ of the appropriate arity. This is exactly what is needed to endow the preorder $(\mathsf{ST}(\mathsf{Act}), \sqsubseteq)$ with the structure of a Σ_G-preorder in the sense of [10]. Using Eqn. 4, we then automatically obtain a Σ_G-poset structure on the poset of compact elements of $\mathcal{D}$. The unique meaning maps from $\mathrm{T}(\Sigma_G)$ to the resulting algebraic structures, denoted by $\mathsf{ST}[\![\cdot]\!]$ and $\mathcal{K}(\mathcal{D})[\![\cdot]\!]$, are related thus:

Lemma 4.6 *For every $P \in \mathrm{T}(\Sigma_G)$, $(\mathsf{ST}[\![P]\!])^c = \mathcal{K}(\mathcal{D})[\![P]\!]$.*

We now proceed to show that the denotational semantics $\mathcal{K}(\mathcal{D})[\![\cdot]\!]$ for recursion-free terms in $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$ is fully abstract with respect to the bisimulation preorder.

Theorem 4.7 *Let G be a compact GSOS system. Then, for all $P, Q \in \mathrm{T}(\Sigma_G)$, $P \lesssim Q$ iff $\mathcal{K}(\mathcal{D})[\![P]\!] \sqsubseteq \mathcal{K}(\mathcal{D})[\![Q]\!]$.*

Our aim in the remainder of this section will be to extend the above full abstraction result to the whole of $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$. In the process of proving this result, we shall establish very general results for compact GSOS languages that have independent interest. First of all, in order to define an interpretation of programs in $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$ as elements of $\mathcal{D}$, we need to define a continuous Σ_G-algebra structure on $\mathcal{D}$. As $(\mathcal{D}, \sqsubseteq_{\mathcal{D}})$ is, up to isomorphism, the unique algebraic cpo with $(\mathcal{K}(\mathcal{D}), \sqsubseteq)$ as poset of compact elements, this is easily done by using Eqn. (5) to define a continuous function $\mathbf{f}_{\mathcal{D}}$ for each $f \in \Sigma_G$. By the general theory of algebraic semantics (see, e.g., [8]), we then have that, for all $P, Q \in \mathrm{T}(\Sigma_G)$,

$$\mathcal{D}[\![P]\!] \sqsubseteq_{\mathcal{D}} \mathcal{D}[\![Q]\!] \;\Leftrightarrow\; \mathcal{K}(\mathcal{D})[\![P]\!] \sqsubseteq \mathcal{K}(\mathcal{D})[\![Q]\!] \ . \tag{10}$$

In view of Thm. 4.7, our desired full abstraction result will follow if we prove that the behavioural preorder $\lesssim_\omega$ is algebraic. This is because, from general properties of denotational interpretations, the relation $\sqsubseteq_{\mathcal{D}}$ is algebraic, and two algebraic relations that coincide over $\mathrm{T}(\Sigma_G)$ do, in fact, coincide over the whole of $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$. The key to the proof of the algebraicity of $\lesssim_\omega$ is the following general theorem providing a partial completeness result for $\lesssim$ in the sense of Hennessy [7] for arbitrary compact GSOS systems.

Theorem 4.8 (Partial Completeness) *Let G be a compact GSOS system. Then there exist a compact GSOS system G' and a set of $\Sigma_{G'}$-inequations T such that:*

- *G' disjointly extends G and $\mathsf{FINTREE}_\Omega$ in the sense of [2], and*
- *for all $P \in \mathrm{T}(G')$, $Q \in \mathsf{CREC}(\Sigma_{G'}, \mathsf{Var})$, $P \lesssim Q$ iff $T \cup \{\text{Eqn. } 8\} \vdash P \leq Q$.*

The proof of this theorem is rather involved. It relies on an algorithm for generating the compact GSOS system G' and the inequational theory T over it following the lines of [2]. Apart from its intrinsic interest, the main consequence of Thm. 4.8 is the following key result, whose proof uses properties of initial continuous algebras in inequational varieties, that essentially states that finite trees are compact elements with respect to the preorder $\lesssim$.

Theorem 4.9 *Suppose that G is a compact GSOS system. Assume that $t \in \mathsf{ST}(\mathsf{Act})$ and $P \in \mathsf{CREC}(\Sigma_G, \mathsf{Var})$. Then $t \lesssim_{Grec} P$ iff there exists a finite approximation P^n of P such that $t \lesssim_{Grec} P^n$.*

The above result, in conjunction with Propn. 3.6, allows us to prove that $\lesssim_\omega$ is indeed algebraic.

Theorem 4.10 *Let G be a compact GSOS system. Then the relation $\lesssim_\omega$ over $\mathsf{lts}(Grec)$ is algebraic.*

In light of the above results, we can now show that, for any compact GSOS system G, the denotational semantics for $\mathsf{CREC}(\Sigma_G, \mathsf{PVar})$ is fully abstract with respect to $\lesssim_\omega$.

Theorem 4.11 *Let G be a compact GSOS system. Then, for all $P, Q \in \mathsf{CREC}(\Sigma_G, \mathsf{Var})$, $P \lesssim_\omega Q$ iff $\mathcal{D}[\![P]\!] \sqsubseteq_{\mathcal{D}} \mathcal{D}[\![Q]\!]$.*

References

[1] S. Abramsky, *A domain equation for bisimulation*, Information and Computation, 92 (1991), pp. 161–218.

[2] L. Aceto, B. Bloom, and F. Vaandrager, *Turning SOS rules into equations*, Information and Computation, 111 (1994), pp. 1–52.

[3] L. Aceto and A. Ingólfsdóttir, *CPO models for GSOS languages — Part I: Compact GSOS languages*, Research Report RS-94-40, BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation), Department of Mathematics and Computer Science, Aalborg University, December 1994. An extended abstract of the paper will appear in: *Proceedings of CAAP '95*, LNCS, 1995.

[4] J. Baeten, J. Bergstra, and J. Klop, *Syntax and defining equations for an interrupt mechanism in process algebra*, Fundamenta Informaticae, IX (1986), pp. 127–168.

[5] J. Baeten and W. Weijland, *Process Algebra*, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[6] B. Bloom, S. Istrail, and A. Meyer, *Bisimulation can't be traced: preliminary report*, in Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, 1988, pp. 229–239. Full version available as Technical Report 90-1150, Department of Computer Science, Cornell University, Ithaca, New York, August 1990. To appear in the *Journal of the ACM*.

[7] M. HENNESSY, *A term model for synchronous processes*, Information and Control, 51 (1981), pp. 58–75.

[8] ——, *Algebraic Theory of Processes*, MIT Press, Cambridge, Massachusetts, 1988.

[9] M. HENNESSY AND G. PLOTKIN, *A term model for CCS*, in 9^{th} Symposium on Mathematical Foundations of Computer Science, P. Dembiński, ed., vol. 88 of Lecture Notes in Computer Science, Springer-Verlag, 1980, pp. 261–274.

[10] A. INGÓLFSDÓTTIR, *Semantic Models for Communicating Process with Value-Passing*, PhD thesis, School of Cognitive and Computing Sciences, University of Sussex, June 1994. Computer Science Report 8/94. Also available as Report R–94–2044, Department of Mathematics and Computer Science, Aalborg University.

[11] R. MILNER, *A modal characterisation of observable machine behaviour*, in Proceedings CAAP 81, G. Astesiano and C. Bohm, eds., vol. 112 of Lecture Notes in Computer Science, Springer-Verlag, 1981, pp. 25–34.

[12] ——, *Communication and Concurrency*, Prentice-Hall International, Englewood Cliffs, 1989.

[13] G. PLOTKIN, *Lecture notes in domain theory*, 1981. University of Edinburgh.

[14] ——, *A structural approach to operational semantics*, Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[15] D. WALKER, *Bisimulation and divergence*, Information and Computation, 85 (1990), pp. 202–241.

On decidability of simulation and bisimulation for Lossy Channel Systems

Parosh Abdullah   Mats Kindahl

Abstract

By using an infinite-state system consisting of finite-state processes communicating via unbounded FIFO channels it is possible to model link protocols like the Alternating Bit protocol , or the HDLC protocol. It is well known that most interesting verification problems are undecidable for this class of systems. We have previously shown that by altering the behaviour of the channels so that they become *lossy*, several verification problems become decidable. In this paper we develop algorithms for deciding strong bisimulation equivalence and simulation preorder. Furthermore, we show that weak bisimulation equivalence is undecidable.

1 Introduction

During the last decade, the research on methods for algorithmic verification of concurrent and parallel systems has expanded dramatically. Several methods for verifying systems with a *finite* number of states (e.g. [CES86, FM90]) has been developed and investigated. For systems with infinite state-space, algorithmic verification is more difficult. There are many classes of systems that has an infinite state space for which you want to check different properties. Examples of such systems are real-time systems that has clocks, systems with expanding structure, and system that use unbounded FIFO-channels. There are several notations for modelling systems with infinite state space, e.g. Petri nets [Pet77], CCS [Mil89], etc. Unfortunately it is often necessary to properties for these systems manually, as most interesting properties are undecidable.

In [AJ93] a class of finite-state systems that communicate via unbounded but *unreliable* FIFO channels (called *lossy channel systems*), and it was proven that the reachability problem and the eventuality problem were decidable. Lossy channel systems were further investigated in [AJ94] where it was proven that the *model checking* problem, for both *linear* and *branching* time logics, was undecidable; it was also proven that *eventuality properties with fair channels* are undecidable. In this paper we further investigate decidability issues regarding lossy channel systems. Focusing on process algebraic relations we will show that *weak bisimulation equivalence* between lossy channel systems are undecidable. We will further show that *strong bisimulation equivalence* is decidable.

Related work: There are several papers on the analysis of systems that communicate over perfect unbounded FIFO channels. Unfortunately, most verification problems for these kind of systems are undecidable since the channels can be used to simulate the tape of a Turing Machine [BZ83].

Outline: The paper is organized as follows. In section 2 we will present the basic definitions of lossy channel systems. In section 3 we will introduce bisimulation equivalence between lossy channel systems and show some properties of bisimulation equivalence for lossy channel systems, and give an algorithm for deciding bisimulation equivalence. In section 4 we will show that weak bisimulation is undecidable, if you hide the channel operations, by reducing the *Recurrent Path Problem* to the problem of deciding weak bisimulation.

2 Systems with unreliable channels

In this section we present the definition of *lossy channel systems*, systems with unreliable channels. We also define the operational semantics of lossy channel systems.

A lossy channel system consists of two parts, a finite-state *control* and a finite set of *unbounded* FIFO channels. The channels contain sequences of messages taken from a finite alphabet and the control is a finite-state labelled transition system where each label denotes one of three different kinds of operations.

a The operation that perform the visible action a,

$c!m$ the operation that send a message m on a channel c, and

$c?m$ the operation that receive a message m from a channel c.

From [AJ93] we have the following definition of lossy channel systems:

Definition 1 *A* Lossy Channel System *is a tuple* $\mathcal{L} = \langle S, s_0, A, C, M, \delta \rangle$ *where*

S *is a finite set of* control states*,*

$s_0 \in S$ *is an* initial control state*,*

A *is a finite set of* actions*,*

C is a finite set of channels*,*

δ is a finite set of transitions*, each of which is a triple of the form $\langle s, \alpha, s' \rangle$, where s and s' are control states, and α is an* operation *of the form:*

- *$c!m$, where $c \in C$ and $m \in M$,*
- *$c?m$, where $c \in C$ and $m \in M$, or*
- *a, where $a \in A$.*

For a set M, let M^* denote the set of all *finite* strings of elements taken from M. For $x, y \in M^*$, let $x \bullet y$ denote the concatenation of string x and y. The empty string is denoted by ε.

A *string vector* w from C to M is a mapping $C \mapsto M^*$. In addition, we let $w\,[c := x]$ denote a string vector w' such that $w'(c) = x$ and $(\forall c' \neq c)\, w'(c') = w(c')$.

To define the *operational behaviour* we formalize the intuitive behaviour of the system as a transition system with an infinite number of states. Let $\mathcal{L}$ be a lossy channel system $\langle S, s_0, A, C, M, \delta \rangle$. A *global state* of a lossy channel system is a pair $\langle s, w \rangle$ where $s \in S$ and w is a string vector from C to M.

For a lossy channel system $\mathcal{L}$ we define the relation $\longrightarrow$ as a set of triples $\langle \gamma, \alpha, \gamma' \rangle$ where γ and γ' are global states. Let $\gamma \xrightarrow{\alpha} \gamma'$ denote $\langle \gamma, \alpha, \gamma' \rangle \in \longrightarrow$. Define the set of *events Event* to be the union of A, and the set of all $c!m$ and $c?m$, where $c \in C$ and $m \in M$.

Definition 2 *For a lossy channel system $\mathcal{L} = \langle S, s_0, A, C, M, \delta \rangle$, define $\longrightarrow$ to be the smallest trinary relation over global states such that:*

1. *if $\langle s, c!m, s' \rangle \in \delta$, then $\langle s, w \rangle \xrightarrow{c!m} \langle s', w\,[c := w(c) \bullet m] \rangle$, i.e. the control state changes from s to s' and m is appended to the end of channel c,*

2. *if $\langle s, c?m, s' \rangle \in \delta$, then $\langle s, w\,[c := m \bullet w(c)] \rangle \xrightarrow{c?m} \langle s', w \rangle$, i.e. the control state changes from s to s' and m is removed from the head of channel c,*

3. *if $w(c) = m \bullet y$, then $\langle s, w \rangle \xrightarrow{\tau} \langle s, w\,[c := y] \rangle$, i.e. the first message m in channel c is lost without changing the control state, and*

4. *if $\langle s, a, s' \rangle \in \delta$, then $\langle s, w \rangle \xrightarrow{a} \langle s', w \rangle$, i.e. the control state changes from s to s' while the action $a \in A$ is performed. The contents of the channels are unchanged.*

Note that this definition of $\longrightarrow$ differs from the definition in [AJ93] in that messages are only lost at the beginning of a channel, and that channel operations are visible to the environment.

For $x_1, x_2 \in M^*$, let $x_1 \preceq x_2$ denote that x_1 is a (not necessarily contiguous) substring of x_2. If w_1, w_2 are string vectors from C to M, then $w_1 \preceq w_2$ denotes that $w_1(c) \preceq w_2(c)$ for all $c \in C$. Let $\langle s_1, w_1 \rangle \preceq \langle s_2, w_2 \rangle$ denote that $s_1 = s_2$ and $w_1 \preceq w_2$.

3 Decidability results

In this section we present bisimulation equivalence and simulation preorder between lossy channel systems and prove that they are decidable by outlining an algorithm for checking bisimulation equivalence and simulation preorder. We also give argument for the correctness of the algorithms.

To simplify the proofs we limit ourself to using states with only one channel. It is straightforward to extend the proofs to systems with multiple channels. We introduce an alternative equivalence relation $\dot{\sim}$ and prove that it is equal to the standard definition of bisimulation equivalence when we restrict ourself to states where the channel contents are equal.

The definition of bisimulation equivalence between lossy channel systems follows the definition of bisimulation equivalence found in [Mil89].

Definition 3 *The relation R is a* bisimulation *iff; for any global states γ_1 and γ_2 such that $\gamma_1 R \gamma_2$ then for any event $\alpha \in Event \cup \{\tau\}$:*

- *if $\gamma_1 \xrightarrow{\alpha} \gamma_1'$, for some state γ_1', then there exists global state γ_2' such that $\gamma_2 \xrightarrow{\alpha} \gamma_2'$ and $\gamma_1' R \gamma_2'$, and*
- *if $\gamma_2 \xrightarrow{\alpha} \gamma_2'$, for some state γ_2', then there exists global state γ_1' such that $\gamma_1 \xrightarrow{\alpha} \gamma_1'$ and $\gamma_1' R \gamma_2'$.*

Two global state γ_1 and γ_2 are *bisimilar*, written $\gamma_1 \sim \gamma_2$, if there exists a relation R such that $\gamma_1 R \gamma_2$. Two control states $s_1, s_2 \in S$ are bisimilar, written $s_1 \sim s_2$, iff $\langle s_1, \varepsilon \rangle \sim \langle s_2, \varepsilon \rangle$. Further, for global states $\gamma_1 = \langle s_1, w_1 \rangle$ and $\gamma_2 = \langle s_1, w_1 \rangle$, $\gamma_1 \approx_c \gamma_2$ iff $w_1 = w_2$, i.e. the states have the same channel contents, but not necessarily the same control states.

An alternative characterization of bisimilarity, which we will use later, is given using the k-bisimilarity relation, which can be found in [Mil89]. Informally, two states are k-bisimilar if they behave bisimilar in the first k steps. It is also shown that for any image-finite labelled transition system graph we have $\sim = \bigcap_{k \in \mathbf{N}} \sim_k$. It is obvious that the transition graph for any LCS is image-finite.

Definition 4 *The relation R_c is a* bisimulation with equal channels *iff; for any global states γ_1 and γ_2 such that $\gamma_1 R_c \gamma_2$ and $\gamma_1 \approx_c \gamma_2$, then for any event $\alpha \in Event \cup \{\tau\}$:*

- *if $\gamma_1 \xrightarrow{\alpha} \gamma_1'$, for some state γ_1', then there exists global state γ_2' such that $\gamma_2 \xrightarrow{\alpha} \gamma_2'$, $\gamma_1' R_c \gamma_2'$, and $\gamma_1' \approx_c \gamma_2'$, and*
- *if $\gamma_2 \xrightarrow{\alpha} \gamma_2'$, for some state γ_2', then there exists global state γ_1' such that $\gamma_1 \xrightarrow{\alpha} \gamma_1'$ and $\gamma_1' R_c \gamma_2'$, and $\gamma_1' \approx_c \gamma_2'$.*

Two global states γ_1 and γ_2 are *bisimilar with equal channels*, written $\gamma_1 \dot{\sim} \gamma_2$, if there exists a relation R_c such that $\gamma_1 R_c \gamma_2$. Two control states $s_1, s_2 \in S$ are bisimilar, written $s_1 \dot{\sim} s_2$, iff $\langle s_1, \varepsilon\rangle \dot{\sim} \langle s_2, \varepsilon\rangle$.

To prove that we can decide bisimulation by deciding bisimulation with equal channels we prove that they are equivalent when you start with equal channels. To show this we need the following proposition:

Proposition 3.1 *For any global states γ_1 and γ_2 such that $\gamma_1 \approx_c \gamma_2$ and operation $\alpha \in Event \cup \{\tau\}$, if $\gamma_1 \xrightarrow{\alpha} \gamma_1'$ and $\gamma_2 \xrightarrow{\alpha} \gamma_2'$, then $\gamma_1' \approx_c \gamma_2'$.*

Proof. By case analysis on the transitions of $\longrightarrow$. $\square$

Lemma 3.2 *For any two global states γ_1 and γ_2 such that $\gamma_1 \approx_c \gamma_2$, then $\gamma_1 \sim \gamma_2$ iff $\gamma_1 \dot{\sim} \gamma_2$.*

Proof by induction on k. Where $\gamma_1 \sim_k \gamma_2$ iff $\gamma_1 \dot{\sim}_k \gamma_2$, for any $k \in \mathbf{N}$. For the base case ($k = 0$) we see that $\gamma_1 \sim_0 \gamma_2$ iff $\gamma_1 \dot{\sim}_0 \gamma_2$ holds trivially. For the induction case, we assume $\gamma_1 \approx_c \gamma_2 \supset (\gamma_1 \sim_k \gamma_2) = (\gamma_1 \dot{\sim}_k \gamma_2)$, $\gamma_1 \approx_c \gamma_2$, and prove $(\gamma_1 \sim_k \gamma_2) = (\gamma_1 \dot{\sim}_k \gamma_2)$ by proving each direction separately.

$\Longrightarrow$ Assume that $\gamma_1 \sim_{k+1} \gamma_2$ and $\gamma_1 \approx_c \gamma_2$. From the definition of $\sim_{k+1}$ we then get; for any event $\alpha \in Event$, whenever $\gamma_1 \xrightarrow{\alpha} \gamma_1'$ there exists a γ_2' such that $\gamma_2 \xrightarrow{\alpha} \gamma_2'$ and $\gamma_1' \sim_k \gamma_2'$. By proposition 3.1 we conclude that $\gamma_1' \approx_c \gamma_2'$, from which we, together with the induction hypothesis, can conclude that $\gamma_1' \dot{\sim}_k \gamma_2'$. In a similar manner we can prove the symmetric case. From the definition of $\dot{\sim}_{k+1}$ we can see that this implies $\gamma_1 \dot{\sim}_{k+1} \gamma_2$.

$\Longleftarrow$ Assume $\gamma_1 \dot{\sim}_{k+1} \gamma_2$. From the definition of $\dot{\sim}_{k+1}$ we then get; for any event $\alpha \in Event$, whenever $\gamma_1 \xrightarrow{\alpha} \gamma_1'$ there exists a γ_2' such that $\gamma_2 \xrightarrow{\alpha} \gamma_2'$ and $\gamma_1' \dot{\sim}_k \gamma_2'$. Observing that $\gamma_1' \dot{\sim} \gamma_2'$ implies $\gamma_1' \approx_c \gamma_2'$, and using the induction hypothesis, we can conclude that $\gamma_1' \sim_k \gamma_2'$. In a similar manner we can show the symmetric case. From the definition of $\sim_{k+1}$, we can see that this implies $\gamma_1 \sim_{k+1} \gamma_2$.

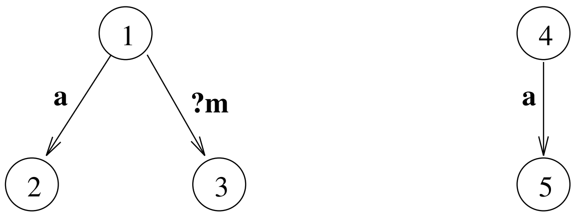


Figure 1: Counterexample for upper closure of bisimilarity

□

Bisimilarity is not upper closed, if two states are bisimilar it does not automatically mean that states that are greater are bisimilar. A trivial counterexample can be found by using the automata in figure 1 and noting that $\langle 1, \varepsilon \rangle \dot{\sim} \langle 4, \varepsilon \rangle$. Instead we can prove that *non-bisimilarity* is upper closed. If two states are non-bisimilar, then all greater states are also non-bisimilar. This can be formalized in the following lemma:

Lemma 3.3 *For global states $\gamma_1 = \langle s_1, w \rangle$ and $\gamma_2 = \langle s_2, w \rangle$, if $\gamma_1 \dot{\nsim} \gamma_2$, then for any state γ_1' and γ_2' such that $\gamma_1 \preceq \gamma_1'$ and $\gamma_2 \preceq \gamma_2'$, follows $\gamma_1' \dot{\nsim} \gamma_2'$.*

Proof by contradiction. Assume that $\gamma_1 \dot{\nsim} \gamma_2$, that there exists γ_1' and γ_2' such that $\gamma_1 \preceq \gamma_1'$, $\gamma_2 \preceq \gamma_2'$, and $\gamma_1' \dot{\sim} \gamma_2'$. From the definition of $\dot{\sim}$ we observe that there exists a global state γ_2'' such that $\gamma_2'' \preceq \gamma_2'$ and $\gamma_2'' \dot{\sim} \gamma_1$. From the definition of $\dot{\sim}$ we then know that γ_1' and γ_2' has the same channel contents w'. From the definition of $\preceq$ we know that γ_1 and γ_1' has the same control state s_1, and that γ_2 and γ_2' has the same control state s_2. From the definition of $\preceq$ we know that γ_2' and γ_2'' has the same control state s_2. From the definition of $\dot{\sim}$ we know that γ_1 and γ_2'' has the same channel contents w, from which we can conclude that $\gamma_2'' = \langle s_2, w \rangle = \gamma_2$, i.e. that $\gamma_1 \dot{\sim} \gamma_2$, which is a contradiction. □

Algorithm

The basic idea of the algorithm is to check "non-bisimilarity" and do it backwards. Starting with states that are not 1-bisimilar and working backwards to get states that are non-bisimilar.

As a convenience we let ϑ denote a pair (γ_1, γ_2) such that $\gamma_1 \approx_c \gamma_2$. Let $S_\alpha(\vartheta)$ denote the set of α-successors to bisimulation state ϑ and let $P_\alpha(\vartheta)$ denote the set of α-predecessors to bisimulation state ϑ. More formally,

$$\begin{aligned} S_\alpha(\vartheta) &= \{\, \vartheta' \mid \vartheta \xrightarrow{\alpha} \vartheta' \,\} \\ P_\alpha(\vartheta) &= \{\, \vartheta' \mid \vartheta' \xrightarrow{\alpha} \vartheta \,\}. \end{aligned}$$

The algorithm accepts a lossy channel system $\mathcal{L} = \langle S, s_0, A, C, M, \delta \rangle$ and two control states c_1 and c_2 and will decide if $c_1 \dot{\sim} c_2$. W denotes the "working set", which is a set of all states that have been concluded non-bisimilar but that has not yet been processed. The set V contains all sets that have been concluded non-bisimilar but have been processed.

The steps of the algorithm to check bisimulation equivalence (simulation preorder) are:

1. Set V to the empty set.
2. Let W be the set of all $\vartheta = (\gamma_1, \gamma_2)$ such that γ_1 and γ_2 have empty channels and are not 1-bisimilar.
3. If W is empty, goto step 7, otherwise proceed with step 4 to 6.
4. Pick one state ϑ in W and remove that state from W.
5. If there are no state ϑ' in V such that $\vartheta' \preceq \vartheta$, then
   (a) remove all states ϑ' in V such that $\vartheta \preceq \vartheta'$,
   (b) add ϑ to V, and
   (c) add to W all state-pairs $\vartheta' = (\gamma_1', \gamma_2')$ in $P_\alpha(\vartheta)$ such that:
      i. there is a state γ_1'' such that $\gamma_1' \xrightarrow{\alpha} \gamma_1''$ and for all γ_2'' such that $\gamma_2' \xrightarrow{\alpha} \gamma_2''$, $(\gamma_1'', \gamma_2'') \in V$, or
      ii. there is a state γ_2'' such that $\gamma_2' \xrightarrow{\alpha} \gamma_2''$ and for all γ_1'' such that $\gamma_1' \xrightarrow{\alpha} \gamma_1''$, $(\gamma_1'', \gamma_2'') \in V$.
6. Go back to step 3.
7. If $(\langle c_1, \varepsilon \rangle, \langle c_2, \varepsilon \rangle) \in V$, return "not bisimilar", else return "bisimilar".

This algorithm will add all non-bisimilar state-pairs to the set V because: for any non-bisimilar state-pair the is always a transition sequence to a state-pair that is non-1-bisimilar, i.e. a state-pair for which it can be decided that they are not bisimilar in one transition, and these state-pairs were in the set W initially.

The reason to why the algorithm terminates is that only a finite number of state-pairs can be added to V. This can be explained as follows; whenever a new state-pair ϑ is added to V it is ensured that there is no state-pair ϑ' in V such that $\vartheta' \not\preceq \vartheta$. Each sequence of state-pairs added to V can be viewed as a sequence $\vartheta_1, \vartheta_2, \ldots$ such that $\vartheta_i \not\preceq \vartheta_j$ for all $i < j$. From *Higman's theorem* it follows that there is no such sequence that is infinite. Higman's theorem can be found in [Lot83], but a version that suits our purpose better is:

Theorem 3.4 (Higman's theorem) *Let M be a finite set. There is no infinite sequence $w_1, w_2, \ldots$ of sequences such that $w_i \not\preceq w_j$ for all $i < j$.*

It is straightforward to generalize Higman's theorem to sequences of pairs of global states.

The above algorithm can be altered to cover simulation preorder by removing step 5(c)ii and replacing step 2 with:

2b. Let W be the set of all $\vartheta = (\gamma_1, \gamma_2)$ such that γ_1 and γ_2 have empty channels and are not 1-similar.

4 Undecidability results

In this section we introduce the problem of checking weak bisimulation between two lossy channel systems and prove that it is undecidable by reducing the *Recurrent Path Problem* (RPP) to weak bisimulation. For the problems in this section we use the original $\longrightarrow$, as defined in [AJ93], where we allow any message in the channel to be lost. Also, channel operations are invisible to the environment (they appear as τ transitions).

For a lossy channel system $\mathcal{L}$, a computation π of $\mathcal{L}$ and a control state s of $\mathcal{L}$, we say that π *visits s infinitely often* if there are infinitely many occurrences of global states in π with control state c.

The definition of the Recurrent Path Problem is

Definition 5 (Recurrent Path Problem) *The definition of RPP is:*

Instance: *A lossy channel system $\mathcal{L}$ and a control state c in $\mathcal{L}$*

Question: *Is there a computation of $\mathcal{L}$ visiting c infinitely often?*

In a similar manner, we can define the problem of deciding weak bisimilarity between lossy channel systems as:

Definition 6 (Weak Bisimulation) *The definition of weak bisimulation is:*

Instance: *Two lossy channel systems $\langle S_1, s_{01}, A_1, C_1, M_1, \delta_1\rangle$ and $\langle S_2, s_{02}, A_2, C_2, M_2, \delta_2\rangle$.*

Question: *Is there a computation of $\mathcal{L}$ visiting c infinitely often?*

Theorem 4.1 *RPP is undecidable.*

The proof of this can be seen in [AJ94].

Theorem 4.2 *RPP is reducible to weak bisimulation.*

Proof. We are given an instance Π of RPP where we have lossy channel system $\mathcal{L} = \langle \mathcal{S}, \int_{\prime}, \mathcal{A}, \mathcal{C}, \mathcal{M}, \delta \rangle$ and a control state c. We will construct two lossy channel systems $\mathcal{L}_1$ and $\mathcal{L}_2$ such that Π has a solution if and only if $\mathcal{L}_1$ and $\mathcal{L}_2$ are weakly bisimilar.

We define $\mathcal{L}_1 = \langle S \cup \{c'\}, s_0, A \cup \{a'\}, C, M, \delta' \rangle$, where

$$\delta' = \delta - \{ \langle c_t, a, c \rangle \mid \langle c_t, a, c \rangle \in \delta \} \cup \{ \langle c_t, a, c' \rangle \mid \langle c_t, a, c \rangle \in \delta \} \cup \{ \langle c', a', c \rangle \},$$

a' is a new action not in A, and c' is a new control state not in S. We define $\mathcal{L}_2 = \langle \{t\}, t, \{a'\}, \emptyset, \emptyset, \{ \langle t, a, t \rangle \mid a \in A \cup \{a'\} \} \rangle$. $\square$

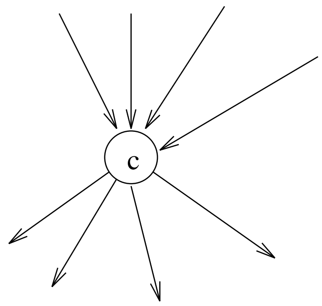


Figure 2: Instance of RPP problem

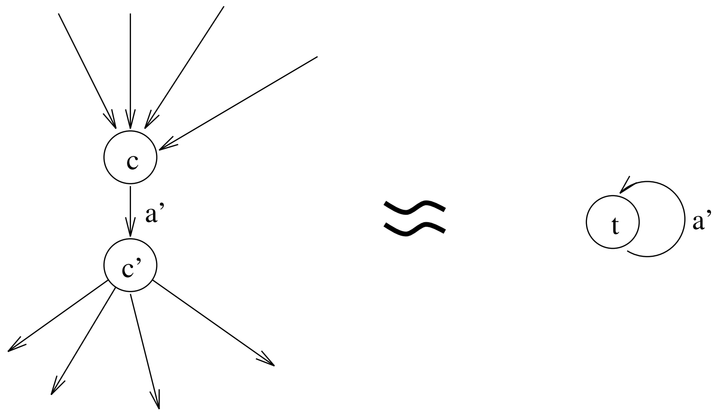


Figure 3: Weak bisimulation instance of RPP problem

On an informal level, given an instance of RPP (as in figure 2) we split the state c into two states, with a transition between them (see figure 3). These two states are visited infinitely often iff the transition between them is taken infinitely often. As the transition is labeled with a unique action a', this is equivalent to checking weak bisimulation between the modified automata and an automata that can do a' infinitely often.

5 Conclusion

In this article we have show that strong bisimulation (and simulation) between systems of finite-state processes that communicate over unbounded but lossy FIFO channels are decidable. We have also shown that weak bisimulation is undecidable, if you hide the channel operations.

Our result generalize to other types of sequences that can lose elements, e.g. "lossy stacks", "lossy tapes" (for "lossy Turing machines"). It follows that the halting problem for Turing machines with "lossy" tapes is decidable.

6 Future work

It is, as shown, possible to extend the algorithm for strong bisimulation to cover also simulation, but a more interesting alternative is to decide simulation by deciding simulation preorder between a lossy channel system and a finite-state "specification" which does *not* use FIFO channels. This would make both weak and strong simulation preorder decidable, it can be argued that many specifications usually are written in a manner which makes them finite-state, but they are used to check an infinite state system. For which simulation preorder algorithms would be useful.

Further, the model we describe here is very deterministic, messages are only lost at the beginning of a channel and every operation gives us a very predictable channel state. This high level of control unfortunately limits our modeling capacity, which is undesirable. We are currently investigating the possibility of allowing a more general model of losses, in the same manner as in [AJ93] while still maintaining bisimilarity.

Another interesting area, which we are also investigating, is if we can allow weak bisimilarity between lossy channel systems by making the channel operation visible, or by restricting one of the automata to be finite-state in the same manner as described above for simulation preorder.

References

[AJ93] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *Proc. 8^{th} IEEE Int. Symp. on Logic in Computer Science*, 1993.

[AJ94] Parosh Aziz Abdulla and Bengt Jonsson. Undecidable verification problems for programs with unreliable channels. Technical Report 94/47, Dept. of Computer Systems, Uppsala University, Sweden, 1994.

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 2(5):323–342, April 1983.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.

[FM90] J.C. Fernandez and L. Mounier. Verifying bisimulations on the fly. In *Proc. FORTE '90*, pages 91–105, Nov. 1990.

[Lot83] M. Lothaire. *Combinatorics on Words*, volume 17 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, 1983.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Pet77] J.L. Peterson. Petri nets. *Computing Surveys*, 9(3):221–252, 1977.

PMC:
A Process Algebra for Real-Time Systems

Henrik Reif Andersen and Michael Mendler

Department of Computer Science, Technical University of Denmark
Building 344, DK-2800 Lyngby, Denmark
E-mail: {hra,mvm}@id.dtu.dk

Abstract

Most timed process algebras view a real-time system as operating under the regime of a global time parameter constraining the occurrence of actions. By the use of quantitative timing constraints they aim at describing completely the global real-time behaviour of timed systems in a fairly detailed fashion. Based on an industrial case study we believe that these approaches are often overly realistic with disadvantages for both the specification and the modelling of real-time systems. We propose a rather different, abstract approach to the specification and modelling of real-time systems that captures the *qualitative* aspects of timing constraints through the use of *multiple clocks*. Clocks enforce global synchronization of actions without compromising the abstractness of time by referring to a concrete time domain. Technically, we present the process algebra PMC as a non-trivial extension of CCS by *multiple clocks* with associated *timeout* and *clock ignore* operators.

We describe the object of investigation in the industrial case study, a highly sophisticated instrument – the Brüel & Kjær 2145 Frequency Analyzer – and show how the central real-time constraints are expressed concisely in PMC. Focus is on the actual specification of the instrument and the technical results that have been obtained for PMC is only touched briefly.

Relevant references

For the algebraic theory, see [AM94] (proofs in [AM93a, AM93b]). For more details on the case study, see [AM95]. These papers are available by WWW (URL http://www.id.dtu.dk/~hra) or anonymous ftp (ftp.id.dtu.dk, directory pub/hra).

References

[AM93a] Henrik R. Andersen and Michael V. Mendler. A process algebra with multiple clocks. Technical Report ID-TR: 1993-122, Department of Computer Science, Technical University of Denmark, August 1993.

[AM93b] Henrik R. Andersen and Michael V. Mendler. Complete axiomatization of observation congruence for PMC. Technical Report ID-TR: 1993-126, Department of Computer Science, Technical University of Denmark, December 1993.

[AM94] Henrik R. Andersen and Michael V. Mendler. An asynchronous process algebra with multiple clocks. In Donald Sannella, editor, *ESOP'94. Proceedings of the 5th European Symposium on Programming, Edinburgh, Scotland, 11-13 April 1994*, volume 788 of *LNCS*, pages 58–73. Springer-Verlag, April 1994.

[AM95] Henrik R. Andersen and Michael V. Mendler. Describing a signal analyzer in the process algebra PMC – a case study. To appear in proceedings of FASE'95, LNCS, 1995. Preprint available as technical report ID-TR: 1994-144, Department of Computer Science, Technical University of Denmark, August 1994.

Type and Behaviour Reconstruction for Higher-Order Concurrent Programs

Torben Amtoft, Flemming Nielson, Hanne Riis Nielson
DAIMI, Aarhus University
Ny Munkegade, DK-8000 Århus C, Denmark
{tamtoft,fn,hrn}@daimi.aau.dk

Abstract: In this paper we develop a sound and complete type and behaviour inference algorithm for a fragment of CML (Standard ML with primitives for concurrency). Behaviours resemble terms of a process algebra and yield a concise presentation of the communications taking place during execution; types are mostly as usual except that function types and "delayed communication types" are labelled by behaviours expressing the communications that will eventually take place. The development of the present paper improves a previously published algorithm in not only achieving a sound algorithm for type inference but also one that is complete, due to an alternative strategy for generalising over types and behaviours. Although we show how to solve special kinds of constraints the problem of a general solution procedure for the constraints generated remains; this is related to an open problem concerning whether all programs typable in Standard ML (with concurrency primitives) are also typable in our system.

1 Introduction

It is well-known that testing can only demonstrate the presence of bugs, never their absence. This has motivated a vast amount of software related activities on guaranteeing statically (that is, at compile-time rather than run-time) that the software behaves in certain ways; a prime example is the formal (and usually manual) verification of software. In this line of activities various notions of type systems have been put forward because these allow static checks of certain kinds of bugs: while at run-time there may still be a need to check for division by zero there will never be a need to check for the addition of booleans and files. As programming languages evolve in terms of features, like module systems and the integration of different programming paradigms, the research on "type systems" is constantly pressed for new problems to be treated.

Our research has been motivated by the integration of the functional and concurrent programming paradigms. Example programming languages are CML [Rep91] that extends Standard ML with primitives for concurrency, Facile [PGM90] that follows a similar approach but more directly contains syntax for expressing CCS-like process composition, and LCS [BS94]. The overall communication structure of such programs may not be immediately clear and hence one would like to find compact ways of recording the communications taking place during execution. One such representation is *behaviours*, a kind of process algebra expressions.

In [NN93] and [NN94a] inference systems are developed that extend the usual notion of types with a formal way of linking behaviours to the syntax of programs. Applications of such information are demonstrated in [NN94a] and [NN94c].

The question remains: how to implement the inference system, i.e. how to reconstruct the types and behaviours? It seems suitable to use a modified version of algorithm W [Mil78]. This algorithm works by unification, but since our behaviours constitute a non-free algebra (due to the laws imposed on them) this approach is not immediately feasible in our framework. Instead we employ the technique of algebraic reconstruction [JG91, TJ92a]; that is the algorithm unifies the free part of the type structure and generates constraints to cater for the non-free parts.

This idea is carried out in [NN94b], where a reconstruction algorithm is presented which is sound but not complete. The algorithm returns two kind of constraints: C-constraints and S-constraints. The C-constraints represent the "monomorphic" aspects of the analysis whereas the S-constraints are needed to cope with polymorphism: they express that instances of polymorphic variables should remain instances even after applying a solution substitution. Using S-constraints is *not* a standard tool when analysing polymorphic languages; they seem to be needed because the C-constraints apparently lack a "principal solution property" (a phenomenon well-known in unification theory). Finding a solution to C-constraints is feasible as shown in [NN94b]. It is harder to solve S-constraints and in fact they can be seen as a special case of semi-unification; the relationship between type inference for polymorphic languages and semiunification is discussed in [Hen93].

The purpose of the present paper is to present a sound and *complete* reconstruction algorithm. This algorithm also returns S-constraints together with the C-constraints, albeit in a different version than [NN94b]. However, the problem of finding an algorithm for solving these constraints remains.

Overview of paper

In Section 2 we introduce the basic concepts; apart from the treatment of type schemes and from the introduction of S-constraints and from the notion of "downwards closure" they are pretty standard. In Section 3 we present our version of the inference system from [NN94a]. In Section 4 the reconstruction algorithm W is presented. In Section 5 the algorithm is shown to be sound. In Section 6 we motivate our non-standard notion of type schemes, leading to a slightly different (but equivalent) inference system. This inference system allows us to prove the completeness of W in Section 7. In Section 8 we discuss our choice of generalisation strategy for W and compare it to the one used in [NN94b]. Then in Section 9 we show how to solve the constraints generated for some special cases; this solution in effect loses all polymorphism. Section 10 concludes.

2 Preliminaries

CML expressions. Expressions e are built from identifiers x, constants c, applications $e_1\ e_2$, monomorphic abstractions $\lambda x.e_0$, polymorphic abstractions $\mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$, conditionals $\mathsf{if}\ e_0\ e_1\ e_2$ and recursive constructs $\mathsf{rec}\ f(x) \Rightarrow e_0$.

This is much like ML, the concurrent aspects being taken care of by the constants c. Some of these will be listed in Section 3 together with their types. We refer to [NN94a] for a more thorough motivation.

Free and bound variables. We shall employ two kinds of variables, and use γ to range over either kind: behaviour variables β and type variables α. We write $\mathsf{fv}(\_)$ for the free variables and $\mathsf{bv}(\_)$ for the bound variables – the definition of these sets is mostly obvious. It is convenient also to define $\mathsf{var}(\_) = \mathsf{fv}(\_) \cup \mathsf{bv}(\_)$ and to use F to range over sets of free variables.

Substitutions. These are total functions that for each β produces a behaviour b and for each α produces a type t. We use g to range over types and behaviours collectively; similarly the letters ϕ, ψ and θ range over substitutions. The result of applying a substitution ϕ to some entity g is denoted $g[\phi]$. The substitution which works by first applying ϕ_1 and then applying ϕ_2 is denoted $\phi_1;\phi_2$. The substitution which maps γ into g and leaves all other variables unchanged is denoted $[\gamma \mapsto g]$. If $\gamma[\phi_1] = \gamma[\phi_2]$ for all $\gamma \in F$ we write $\phi_1 \stackrel{F}{=} \phi_2$. The following standard conventions apply for a substitution ϕ: the domain $\mathsf{dom}(\phi) = \{\gamma \mid \gamma[\phi] \neq \gamma\}$; the range $\mathsf{ran}(\phi) = \{\gamma \mid \exists \gamma' \in \mathsf{dom}(\phi) : \gamma \in \mathsf{fv}(\gamma'[\phi])\}$; and $\mathsf{fv}(\phi) = \mathsf{dom}(\phi) \cup \mathsf{ran}(\phi)$.

Behaviours. Behaviours b are built using the syntax

$$b ::= \beta \mid \epsilon \mid\ !t \mid\ ?t \mid t\ \mathsf{CHAN} \mid \mathsf{FORK}\ b \mid \mathsf{REC}\beta.b \mid b_1;b_2 \mid b_1 + b_2$$

that is they can be one of the following: a behaviour variable β; the empty behaviour ϵ (no communications take place); an output action $!t$ (a value of type t is sent); an input action $?t$ (a value of type t is received); a channel action $t\ \mathsf{CHAN}$ (a channel able to transmit values of type t is created); a fork action $\mathsf{FORK}\ b$ (a process with behaviour b is created); a recursive behaviour $\mathsf{REC}\beta.b$ (where b can "call" itself recursively via β); a sequential composition $b_1;b_2$ (first b_1 is performed and then b_2); a non-deterministic choice $b_1 + b_2$ (either b_1 or b_2 are performed).

For a behaviour b we define $\mathsf{bv}(b) = \emptyset$ whereas $\mathsf{fv}(b)$ is defined such that $\mathsf{fv}(\mathsf{REC}\beta.b') = \mathsf{fv}(b') \setminus \{\beta\}$. In other words, the β in $\mathsf{REC}\beta.b'$ is considered a "nameless dummy", i.e. we have some sort of implicit deBruijn-notation. Accordingly we do not distinguish between behaviours which are equivalent modulo alpha-conversion, so $\mathsf{REC}\beta.b$ is considered identical to $\mathsf{REC}\beta'.b[\beta \mapsto \beta']$ if β' does not occur in b (in [NN94b, Table 3] this was listed as axiom R2). Thus it makes sense to define $(\mathsf{REC}\beta.b)[\psi] = \mathsf{REC}\beta.b[\psi]$ where $\beta \notin \mathsf{var}(\psi)$.

Just as in [NN94a, Table 3] we impose a preorder $\sqsupseteq$ on behaviours. The intuition behind $b_1 \sqsupseteq b_2$ is that any action performed by b_2 can be performed by b_1 as well (but not necessarily vice-versa). This preorder is listed in Fig. 1, where $b_1 \equiv b_2$ denotes that $b_1 \sqsupseteq b_2$ and $b_2 \sqsupseteq b_1$.

P1	$b \sqsupseteq b$		**S1**	$b_1;(b_2;b_3) \equiv (b_1;b_2);b_3$
P2	$b_1 \sqsupseteq b_2 \;\wedge\; b_2 \sqsupseteq b_3 \;\Rightarrow\; b_1 \sqsupseteq b_3$		**S2**	$(b_1+b_2);b_3 \equiv (b_1;b_3)+(b_2;b_3)$
C1	$b_1 \sqsupseteq b_2 \;\wedge\; b_3 \sqsupseteq b_4 \;\Rightarrow\; b_1;b_3 \sqsupseteq b_2;b_4$		**E1**	$b \equiv \epsilon; b$
C2	$b_1 \sqsupseteq b_2 \;\wedge\; b_3 \sqsupseteq b_4 \;\Rightarrow\; b_1+b_3 \sqsupseteq b_2+b_4$		**E2**	$b;\epsilon \equiv b$
C3	$b_1 \sqsupseteq b_2 \;\Rightarrow\; \mathsf{FORK}\; b_1 \sqsupseteq \mathsf{FORK}\; b_2$		**J1**	$b_1+b_2 \sqsupseteq b_1 \;\wedge\; b_1+b_2 \sqsupseteq b_2$
C4	$b_1 \sqsupseteq b_2 \;\Rightarrow\; \mathsf{REC}\beta.b_1 \sqsupseteq \mathsf{REC}\beta.b_2$		**J2**	$b \sqsupseteq b+b$
R1	$\mathsf{REC}\beta.b \equiv b[\beta \mapsto \mathsf{REC}\beta.b]$			

Figure 1: The preorder $\sqsupseteq$ on behaviours with equivalence $\equiv$.

Fact 2.1 If $b_1 \sqsupseteq b_2$ then $\mathsf{fv}(b_1) \supseteq \mathsf{fv}(b_2)$, and for any ψ we also have $b_1[\psi] \sqsupseteq b_2[\psi]$.

Types. Types t can be either a type variable α, a base type like Int or Bool or Unit, a function type $t_1 \rightarrow^b t_2$ (given a value of type t_1 a computation is initiated that behaves as indicated by b and that returns a value of type t_2), a list type t **list**, a communication type t **com** b (if such a communication possibility if activated it behaves as indicated by b and returns a value of type t), or a channel type t **chan** (a channel able to transmit values of type t).

For all types t we have $\mathsf{bv}(t) = \emptyset$; so as is the case for behaviours there is no difference between writing $\mathsf{fv}(t)$ ($\mathsf{fv}(b)$) and writing $\mathsf{var}(t)$ ($\mathsf{var}(b)$).

Type schemes. These are denoted by ts and are of form $\forall \overline{F}.t$ where F are the *free* variables: to be more precise we have $\mathsf{fv}(ts) = F$ and $\mathsf{bv}(ts) = \mathsf{fv}(t) \setminus F$. If $F \subseteq \mathsf{fv}(t)$ we say that ts is a *kernel* type scheme; if this is not necessarily the case we say that ts is an *enriched* type scheme. For every ts we define a kernel type scheme $\kappa(ts)$ by $\forall \overline{F'}.t$ where $F' = F \cap \mathsf{fv}(t)$.

> *Remark:* These definitions are highly non-standard. Although there is a natural bijection between "standard" type schemes (of form $\forall \vec{\gamma}.t$ where $\vec{\gamma}$ is the list of bound variables) and kernel type schemes we shall have to deal with enriched type schemes, as will be explained in Section 6.
>
> Further notice that the names of the bound variables *do* matter; they are not just nameless dummies (as they are recorded in S-constraints): so $\forall \overline{\emptyset}.\alpha \rightarrow^\epsilon \alpha$ is not interchangeable with $\forall \overline{\emptyset}.\alpha' \rightarrow^\epsilon \alpha'$.

Substitutions on type schemes. Given a type scheme $ts = \forall \overline{F}.t$ and a substitution ψ, we define $ts[\psi] = \forall \overline{F'}.(t[\psi])$ where $F' = \mathsf{var}(F[\psi])$. With this definition we have

$$\mathsf{fv}(ts[\psi]) = \mathsf{var}(\mathsf{fv}(ts)[\psi])$$

but it does *not* necessarily hold that $\mathsf{bv}(ts[\psi]) = \mathsf{var}(\mathsf{bv}(ts)[\psi])$; the latter situation arises when $\beta_1[\psi] = \beta_2[\psi]$ where $\beta_1 \in \mathsf{fv}(ts)$ and $\beta_2 \in \mathsf{bv}(ts)$.

Relations on type schemes. We say that t' is an instance of $\forall \overline{F}.t$, written $\forall \overline{F}.t \succ t'$, iff there exists ψ with $\mathsf{dom}(\psi) \cap F = \emptyset$ such that $t' = t[\psi]$.

Let $ts = \forall\overline{F}.t$ and $ts' = \forall\overline{F'}.t'$ be type schemes. We then say that ts is more general than ts', written $ts \succeq ts'$, iff $t = t'$ and $F \subseteq F'$. Notice that this is a more "syntactic" (and more restrictive) definition than is standard in the literature; we have chosen the current definition to simplify some proofs later.

Fact 2.2 Suppose $ts \succeq ts'$ and that $ts' \succ t'$. Then also $ts \succ t'$.

Substitution relations. Stating that $\forall\overline{F}.t \succ t'$ amounts to specifying what the bound variables of t (i.e. those not in F) are instantiated to. This motivates the introduction of *substitution relations*, of form $\forall\overline{F}.\vec{g} \succ \vec{g'}$ where one can think of $\vec{g}$ as the bound variables. Such a relation will hold iff there exists ψ with $\mathsf{dom}(\psi) \cap F = \emptyset$ such that $\vec{g'} = \vec{g}[\psi]$. Notice that with $\mathsf{fv}(\vec{g''}) = F$ the relation will hold iff $\forall\overline{\emptyset}.(\vec{g}\vec{g''}) \succ (\vec{g'}\vec{g''})$; hence substitution relations can be represented as semi-unification problems (at the cost of losing some structure). We shall refrain from doing so, however, since semi-unification is undecidable in general and it is unclear if our problem is a general semi-unification problem.

Environments. An environment E is a list of the form $[[x_1 : ts_1] \ldots [x_n : ts_n]]$ (the x_i's not necessarily distinct). We shall often write a type t_i in place of an ts_i: this is a short hand for $\forall\overline{F_i}.t_i$ where $F_i = \mathsf{fv}(t_i)$.

We say that $E(x) = s$ if $s = s_i$ for an i satisfying that $x = x_i$ and that $x \neq x_j$ for all $j > i$ (in other words we look up in E by searching from the right). The result of updating E with $[x : s]$, to be written $E \oplus [x : s]$, is the environment $[[x_1 : s_1] \ldots [x_n : s_n][x : s]]$.

We shall feel free to use the relations and predicates on type schemes also on environments; as an example we say that E is a kernel environments if s_i is a kernel type scheme for all $i \in \{1 \ldots n\}$.

Constraints. There are two kinds of constraints c:

- *C-constraints* (for containment), of form $b_1 \dot{\sqsupseteq} b_2$. This constraint is satisfied iff $b_1 \sqsupseteq b_2$. The result $c[\psi]$ of applying a substitution ψ to this constraint is $b_1[\psi] \dot{\sqsupseteq} b_2[\psi]$;
- *S-constraints* (for substitutions), of form $\forall\overline{F}.\vec{g_1} \dot{\succ} \vec{g_2}$. This constraint is satisfied iff the substitution relation $\forall\overline{F}.\vec{g_1} \succ \vec{g_2}$ holds. The result $c[\psi]$ of applying a substitution ψ to this constraint is $\forall\overline{F'}.\vec{g_1}[\psi] \dot{\succ} \vec{g_2}[\psi]$ where $F' = \mathsf{var}(F[\psi])$.

If $c[\psi]$ is satisfied we say that ψ satisfies c and write $\psi \models c$. Notice that if $\psi \models c$ then also $\psi; \phi \models c$ if c is a C-constraint (by Fact 2.1), but not necessarily if c is an S-constraint – this fact is a major obstacle for solving constraints "sequentially" as we shall see in Section 9. Further notice that $\psi; \phi \models c$ holds iff $\phi \models c[\psi]$.

The capital letter C ranges over lists of constraints and we say that $\psi \models C$ iff $\psi \models c$ holds for all elements c of C. The concatenation of C_1 and C_2 is written as $C_1 \oplus C_2$. The reason for choosing lists rather than sets is that this extra structure makes it possible to formulate some properties of the constraints generated by the reconstruction algorithm – due to space limits this line will not be pursued further in the present paper.

Principal solutions to C-constraints. With the ordering on behaviours axiomatised in Fig. 1 there seems to be no notion of "principal solutions" to C-constraints. For an example of this, consider the constraint $\beta \sqsupseteq \beta; !\mathsf{Int}; !\mathsf{Int}$. The substitution ψ_1 which maps β into $\mathsf{REC}\beta.(\beta; !\mathsf{Int}; !\mathsf{Int})$ as well as the substitution ψ_2 which maps β into $\mathsf{REC}\beta.(\beta; !\mathsf{Int})$ will satisfy this constraint; but using the axiomatisation in Fig. 1 it seems hard to find a sense in which ψ_1 and ψ_2 are comparable.

A remedy, inspired by [CC91] where a similar observation is made concerning recursive types, might be to consider $\mathsf{REC}\beta.b$ equivalent to its infinite unfolding (cf. rule R1 in Fig. 1 which states that $\mathsf{REC}\beta.b$ is equivalent to its finite unfoldings). It is not yet clear whether such a change in the axiomatisation will be sufficient to gain some notion of principality.

Downwards closure. Given a set of variables F and a list of constraints C we define the downwards closure of F with respect to C, written F^C, as the smallest set F' such that $F \subseteq F'$ and such that whenever $\beta \dot{\sqsupseteq} b \in C$ and $\beta \in F'$ then also $\mathsf{fv}(b) \subseteq F'$. It is easy to see that F^C can be computed constructively.

A brief motivation: think of the variables in F as the "non-polymorphic" ones; i.e. those the reconstruction algorithm is not allowed to "make copies of". Then if $\beta \dot{\sqsupseteq} b \in C$ with $\beta \in F$ also the variables in b should be non-polymorphic. We shall elaborate on this in Section 8; more formally we have the following result which is a consequence of Fact 2.1:

Fact 2.3 If $\psi \models C$ then $\mathsf{fv}(F[\psi]) = \mathsf{fv}(F^C[\psi])$.

3 The type and behaviour inference system

In Fig. 2 we list (part of) the inference system for which we are going to present a reconstruction algorithm. A judgement is of the form $E \vdash e : t \;\&\; b$ and says that in (kernel) environment E one can infer that expression e has type t and behaviour b.

For constants a function CTypeOf() is employed, assigning an *extended type scheme* to each constant. An extended type scheme ts is of form $\forall\overline{\emptyset}.(t, C)$ (so $\mathsf{fv}(ts) = \emptyset$). Consistent with previous definitions we say that t' is an instance of $\forall\overline{\emptyset}.(t, C)$, written $\forall\overline{\emptyset}.(t, C) \succ t'$, iff there exists ψ such that $t' = t[\psi]$ and such that $\psi \models C$. The value of CTypeOf() on some constants is tabulated below (adopted from [NN94b, Table 4]).

$$
\begin{array}{lcl}
\mathrm{CTypeOf}(+) & = & \forall\overline{\emptyset}.(\mathsf{Int} \rightarrow^{\beta_1} \mathsf{Int} \rightarrow^{\beta_2} \mathsf{Int}, [\beta_1 \sqsupseteq \epsilon, \beta_2 \sqsupseteq \epsilon]) \\
\mathrm{CTypeOf}(\mathsf{sync}) & = & \forall\overline{\emptyset}.((\alpha\ \mathbf{com}\ \beta_1) \rightarrow^{\beta_2} \alpha, [\beta_2 \sqsupseteq \beta_1]) \\
\mathrm{CTypeOf}(\mathsf{send}) & = & \forall\overline{\emptyset}.(\alpha\ \mathbf{chan} \times \alpha \rightarrow^{\beta_1} \alpha\ \mathbf{com}\ \beta_2, [\beta_1 \sqsupseteq \epsilon, \beta_2 \sqsupseteq !\alpha]) \\
\mathrm{CTypeOf}(\mathsf{receive}) & = & \forall\overline{\emptyset}.(\alpha\ \mathbf{chan} \rightarrow^{\beta_1} \alpha\ \mathbf{com}\ \beta_2, [\beta_1 \sqsupseteq \epsilon, \beta_2 \sqsupseteq ?\alpha]) \\
\mathrm{CTypeOf}(\mathsf{channel}) & = & \forall\overline{\emptyset}.(\mathsf{Unit} \rightarrow^{\beta} \alpha\ \mathbf{chan}, [\beta \sqsupseteq \alpha\ \mathsf{CHAN}]) \\
\mathrm{CTypeOf}(\mathsf{fork}) & = & \forall\overline{\emptyset}.((\mathsf{Unit} \rightarrow^{\beta_1} \alpha) \rightarrow^{\beta_2} \mathsf{Unit}, \beta_2 \sqsupseteq \mathsf{FORK}\ \beta_1)
\end{array}
$$

The inference system is much as in [NN94a], but there are the following differences: (i) we use the non-standard notation for type schemes that was introduced in Section 2; (ii) we have removed regions from behaviours; (iii) we

$$E \vdash x : t \mathbin{\&} b \qquad \text{if } E(x) \succ t \text{ and } b \sqsupseteq \epsilon$$

$$E \vdash c : t \mathbin{\&} b \qquad \text{if CTypeOf}(c) \succ t \text{ and } b \sqsupseteq \epsilon$$

$$\frac{E \vdash e_1 : t_1 \mathbin{\&} b_1,\ E \vdash e_2 : t_2 \mathbin{\&} b_2}{E \vdash e_1\ e_2 : t \mathbin{\&} b} \qquad \text{if } t_1 = t_2 \rightarrow^{b_0} t \text{ and } b \sqsupseteq b_1; b_2; b_0$$

$$\frac{E \oplus [x : t_1] \vdash e_0 : t_0 \mathbin{\&} b_0}{E \vdash \lambda x.e_0 : t \mathbin{\&} b} \qquad \text{if } t = t_1 \rightarrow^{b_0} t_0 \text{ and } b \sqsupseteq \epsilon$$

$$\frac{E \vdash e_1 : t_1 \mathbin{\&} b_1,\ E \oplus [x : \forall \overline{F}.t_1] \vdash e_0 : t \mathbin{\&} b_0}{E \vdash \textsf{let } x{=}e_1 \textbf{ in } e_0 : t \mathbin{\&} b} \qquad \text{if } b \sqsupseteq b_1; b_0$$

$$\text{where } F = (\textsf{fv}(E) \cup \textsf{fv}(b_1)) \cap \textsf{fv}(t_1)$$

Figure 2: Selected parts of the type and behaviour inference system.

avoid coding the constraints present in the definition of CTypeOf() into the types (as in TypeOf of [NN94a]).

Example 3.1 Consider the following CML-program map2, a version of the well-known map function where a process is forked for each tail while the forking process itself works on the head. A rather informal notation is employed; for instance we write $e_1; e_2$ as an abbreviation for $(\lambda d.e_2)\ e_1$.

```
rec mp2(f) ⇒ λxs.if xs = [] then []
                 else let ch = channel ()
                      in fork (λd.(sync (send ⟨ch,mp2 f (tail xs)⟩)));
                         cons (f (head xs)) (sync (receive ch))
```

We expect to have $\emptyset \vdash \textsf{map2} : ((\alpha_1 \rightarrow^{\beta_1} \alpha_2) \rightarrow^{\epsilon} (\alpha_1 \textbf{ list} \rightarrow^{\beta_2} \alpha_2 \textbf{ list})) \mathbin{\&} \epsilon$ where $\beta_2 = \textsf{REC}\beta.(\epsilon + ((\alpha_2 \textbf{ list})\ \textsf{CHAN}; \textsf{FORK}\ (\beta; !(\alpha_2 \textbf{ list})); \beta_1; ?(\alpha_2 \textbf{ list})))$.

We now list some properties of the inference system (the latter due to Fact 2.2):

Fact 3.2 Suppose $E \vdash e : t \mathbin{\&} b$ and $b' \sqsupseteq b$. Then also $E \vdash e : t \mathbin{\&} b'$.

Fact 3.3 Let E and E' be kernel environments with $E' \succeq E$. Suppose that $E \vdash e : t \mathbin{\&} b$. Then also $E' \vdash e : t \mathbin{\&} b$.

4 A reconstruction algorithm

Our goal is to produce an algorithm which works in the spirit of the well-known algorithm W[Mil78]. Hence we have to perform unification over types; but in order to unify $t_1 \rightarrow^{b} t_2$ with $t_1' \rightarrow^{b'} t_2'$ we have to unify b and b', and if for example $b = b_1; b_2$ and $b' = b_1'; b_2'$ this cannot always be done "compositionally" by first unifying b_1 with b'_1 and then unifying b_2 with b'_2 – for then we will lose the possibility that say $b_1 = b'_2 = \epsilon$ and $b'_1 = b_2 = !\mathsf{Int}$.

The remedy (as done in [TJ92a] and [NN94b] but not in [JG91] where no unification is performed) is to introduce the notion of *simplicity*: a type is simple if all the behaviours it contains are behaviour variables (so $t_1 \rightarrow^b t_2$ is simple iff t_1 and t_2 are both simple and $b = \beta$ for some β); a behaviour is simple if all the types it contains are simple (so e.g. $!t$ is simple iff t is simple) and if it does not contain sub-behaviours of form $\mathsf{REC}\beta.b$; a C-constraint is simple if it is of form $\beta \dot{\sqsupseteq} b$ with b simple; an S-constraint is simple if it is of form $\forall \overline{F}.(\vec{t}\vec{\beta}) \dot{\succ} (\vec{t'}\vec{\beta'})$, with t and t' simple types[1]; a substitution is simple if it maps type variables into simple types and maps behaviour variables into behaviour variables (rather than simple behaviours).

Fact 4.1 Let t be a simple type; let b be a simple behaviour; let C be a list of simple constraints; and let ϕ and ϕ' be simple substitutions. Then $t[\phi]$ is a simple type; $b[\phi]$ is a simple behaviour; $C[\phi]$ is a list of simple constraints; and $\phi;\phi'$ is a simple substitution.

We can now define a procedure UNIFY which takes two simple types t_1 and t_2 and returns the most general unifier if one unifier exists – otherwise UNIFY fails. Some of the more interesting clauses of UNIFY are listed below:

$$\begin{array}{lll}
\mathsf{UNIFY}(\alpha, t) = \theta & \text{iff} & \theta = [\alpha \mapsto t] \text{ and } (t = \alpha \text{ or } \alpha \notin \mathsf{fv}(t)) \\[2ex]
\mathsf{UNIFY}(t_1 \ \mathbf{com}\ \beta_1, t_2\ \mathbf{com}\ \beta_2) = \theta & \text{iff} & \mathsf{UNIFY}(t_1, t_2) = \theta' \\
& \text{and} & \beta'_1 = \beta_1[\theta'] \text{ and } \beta'_2 = \beta_2[\theta'] \\
& \text{and} & \theta = \theta'; [\beta'_1 \mapsto \beta'_2]
\end{array}$$

Lemma 4.2 Let t_1 and t_2 be simple types. First suppose that $\mathsf{UNIFY}(t_1, t_2) = \theta$. Then θ is simple and $t_1[\theta] = t_2[\theta]$.

Conversely assume that $t_1[\phi] = t_2[\phi]$. Then $\mathsf{UNIFY}(t_1, t_2)$ succeeds with result θ; and there exists ϕ' such that $\phi = \theta; \phi'$.

We are now ready to define the reconstruction algorithm W, as is done in Figure 3. The algorithm fails if and only if a call to UNIFY fails.

The most interesting cases are the ones for identifiers and for polymorphic abstractions:

An identifier x is looked up in the environment and fresh copies $\vec{\gamma'}$ of the bound variables $\vec{\gamma}$ of their type schemes are generated. Moreover, it is recorded (via an S-constraint) that a solution should map $\vec{\gamma'}$ to an instance of what $\vec{\gamma}$ is mapped into in such a way that it does not interfere with the free variables.

For a polymorphic abstraction $\mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$ the algorithm first finds the type t_1 and behaviour b_1 of e_1; then e_0 is analysed in an environment where x is bound to a type scheme with type part t_1. The free variables of the type scheme (i.e. the non-polymorphic ones) are built from those which are free in the environment together with those which are free in b_1. This corresponds

[1] In fact it holds that t will be of form $\vec{\alpha}$ for all simple S-constraints produced by W.

$$
\begin{array}{ll}
 & W(x,E) = (t,\epsilon,C,id) \\
\text{iff} & E(x) = \forall\overline{F_x}.t_x \\
\text{and} & \vec{\gamma} = \mathsf{fv}(t_x) \setminus F_x \\
\text{and} & \vec{\gamma}' \text{ are fresh copies of } \vec{\gamma} \\
\text{and} & t = t_x[\vec{\gamma} \mapsto \vec{\gamma}'] \\
\text{and} & C = [\forall\overline{F_x}.\vec{\gamma} \mathrel{\dot{\succ}} \vec{\gamma}']
\end{array}
$$

$$
\begin{array}{ll}
 & W(c,E) = (t,\epsilon,C,id) \\
\text{iff} & \text{CTypeOf}(c) = \forall\overline{\emptyset}.(t_c, C_c) \\
\text{and} & \vec{\gamma} = \mathsf{fv}(t_c) \cup \mathsf{fv}(C_c) \\
\text{and} & \vec{\gamma}' \text{ are fresh copies of } \vec{\gamma} \\
\text{and} & t = t_c[\vec{\gamma} \mapsto \vec{\gamma}'] \\
\text{and} & C = C_c[\vec{\gamma} \mapsto \vec{\gamma}']
\end{array}
$$

$$
\begin{array}{ll}
 & W(\lambda x.e_0, E) = (t,\epsilon,C,\theta) \\
\text{iff} & \alpha_1 \text{ and } \beta_0 \text{ are fresh} \\
\text{and} & W(e_0, E \oplus [x:\alpha_1]) \\
 & = (t_0, b_0, C_0, \theta_0) \\
\text{and} & t = \alpha_1[\theta_0] \to^{\beta_0} t_0 \\
\text{and} & C = C_0 \oplus [\beta_0 \mathrel{\dot{\sqsupseteq}} b_0] \\
\text{and} & \theta = \theta_0
\end{array}
$$

$$
\begin{array}{ll}
 & W(e_1\ e_2, E) = (t,b,C,\theta) \\
\text{iff} & W(e_1,E) = (t_1,b_1,C_1,\theta_1) \\
\text{and} & W(e_2, E[\theta_1]) = (t_2,b_2,C_2,\theta_2) \\
\text{and} & \alpha \text{ and } \beta_0 \text{ are fresh} \\
\text{and} & \mathsf{UNIFY}(t_1[\theta_2], t_2 \to^{\beta_0} \alpha) = \theta_0 \\
\text{and} & t = \alpha[\theta_0] \\
\text{and} & b = b_1[\theta_2;\theta_0]; b_2[\theta_0]; \beta_0[\theta_0] \\
\text{and} & C = C_1[\theta_2;\theta_0] \oplus C_2[\theta_0] \\
\text{and} & \theta = \theta_1;\theta_2;\theta_0
\end{array}
$$

$$
\begin{array}{ll}
 & W(\mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0, E) = (t,b,C,\theta) \\
\text{iff} & W(e_1,E) = (t_1,b_1,C_1,\theta_1) \\
\text{and} & W(e_0, E[\theta_1] \oplus [x : \forall\overline{\mathcal{NQ}}.t_1]) \\
 & = (t_0,b_0,C_0,\theta_0) \\
\text{and} & t = t_0 \\
\text{and} & b = b_1[\theta_0]; b_0 \\
\text{and} & C = C_1[\theta_0] \oplus C_0 \\
\text{and} & \theta = \theta_1;\theta_0 \\
\multicolumn{2}{l}{\text{where } \mathcal{NQ} = (\mathsf{fv}(E[\theta_1]) \cup \mathsf{fv}(b_1))^{C_1}}
\end{array}
$$

Figure 3: Selected clauses of Algorithm W.

to the inference system in Fig. 2, but notice the following differences: (a) in contrast to the inference system the algorithm does *not* intersect with $\mathsf{fv}(t_1)$ and thus an enriched type scheme is generated; (b) the algorithm takes the downwards closure with respect to the constraints C_1. We shall elaborate upon the former point in Section 6 and upon the latter point in Section 8.

Fact 4.3 Suppose that $W(e,E) = (t,b,C,\theta)$ where the types in E are simple. Then t is a simple type, b is a simple behaviour, C is a list of simple constraints, and θ is a simple substitution.

5 Soundness of algorithm W

We are now going to prove that the algorithm is sound; this means that a solution to the constraints gives rise to a valid inference in the inference system of Figure 2.

Theorem 5.1 Suppose that $W(e,\emptyset) = (t,b,C,\theta)$
and suppose that ψ is such that $\psi \models C$ and $t' = t[\psi]$ and $b' \sqsupseteq b[\psi]$.
Then it holds that $\emptyset \ \vdash \ e : t' \ \&\ b'$.

This theorem follows easily from Fact 3.2 and the following proposition that allows a proof by induction:

Proposition 5.2 Suppose that $W(e,E) = (t,b,C,\theta)$. Then for all ψ such that $\psi \models C$ we have $\kappa(E[\theta][\psi]) \vdash e : t[\psi]$ & $b[\psi]$. (Recall that W deals with enriched environments whereas the inference system of Figure 2 deals with kernel environments.)

Proof: We perform induction on e; to conserve space we use the same terminology as in the definition of the relevant clause for W. There are seven cases; here we shall list the two most interesting ones only.

The case $W(x,E)$**:** Let $F' = \mathsf{var}(F_x[\psi])$ and let $F'_x = F' \cap \mathsf{fv}(t_x[\psi])$.

Our aim is to show that $\kappa(E[\psi])(x) \succ t[\psi]$, which amounts to

$$\forall \overline{F'_x}.(t_x[\psi]) \succ t_x[\vec{\gamma} \mapsto \vec{\gamma'}][\psi]. \tag{1}$$

Since $\psi \models C$ we have $\forall \overline{F'}.\vec{\gamma}[\psi] \succ \vec{\gamma'}[\psi]$; so there exists a ϕ' with $\mathsf{dom}(\phi') \cap F' = \emptyset$ such that $\psi;\phi' \stackrel{\vec{\gamma}}{=} [\vec{\gamma} \mapsto \vec{\gamma'}];\psi$. This implies, since $\mathsf{var}(t_x) \subseteq \vec{\gamma} \cup F_x$, that we even have $\psi;\phi' \stackrel{\mathsf{var}(t_x)}{=} [\vec{\gamma} \mapsto \vec{\gamma'}];\psi$. But this shows that (1) holds, with ϕ' as the "instance substitution".

The case $W(\mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0, E)$**:** First a *remark:* the only property of $\mathcal{NQ}$ we shall exploit is the following which is a consequence of Fact 2.3 (the condition may seem superfluous but is added to make (2) and (5) dual):

$$\psi \models C_1 \text{ implies } \mathsf{var}(\mathcal{NQ}[\psi]) \supseteq \mathsf{var}(\mathsf{fv}(E[\theta_1])[\psi]) \cup \mathsf{var}(b_1[\psi]). \tag{2}$$

First observe that $\theta_0;\psi \models C_1$; so we can apply the induction hypothesis on the call $W(e_1,E)$ and the substitution $\theta_0;\psi$ to get

$$\kappa(E[\theta][\psi]) \vdash e_1 : t_1[\theta_0;\psi] \ \&\ b_1[\theta_0;\psi].$$

Since $\psi \models C_0$ we can apply the induction hypothesis on the call $W(e_0, E[\theta_1] \oplus [x : \ldots])$ and the substitution ψ to get

$$\kappa(E[\theta][\psi]) \oplus [x : \forall \overline{F'}.(t_1[\theta_0;\psi])] \vdash e_0 : t_0[\psi] \ \&\ b_0[\psi]$$

where $F' = \mathsf{fv}(\mathcal{NQ}[\theta_0;\psi]) \cap \mathsf{fv}(t_1[\theta_0;\psi])$.

Let $F'' = (\mathsf{fv}(\kappa(E[\theta][\psi])) \cup \mathsf{fv}(b_1[\theta_0;\psi])) \cap \mathsf{fv}(t_1[\theta_0;\psi])$. Applying (2) on the substitution $\theta_0;\psi$ tells us that $F'' \subseteq F'$ so by Fact 3.3 we infer that

$$\kappa(E[\theta][\psi]) \oplus [x : \forall \overline{F''}.t_1[\theta_0;\psi]] \vdash e_0 : t_0[\psi] \ \&\ b_0[\psi].$$

This shows that we have the desired judgment

$$\kappa(E[\theta][\psi]) \vdash \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0 : t_0[\psi] \ \&\ b[\psi].$$

6 The need for enriched type schemes

Enriched type schemes in algorithm W. The reconstruction algorithm from [NN94b] carries constraints around, together with type schemes in kernel form. Our algorithm carries no constraints in the environment, but in order to preserve soundness the type schemes have to be enriched. This is informally illustrated by the following example.

Let $e = \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$ and suppose that $W(e_1, E) = (t_1, b_1, C_1, \theta_1)$ where $t_1 = \mathsf{Unit} \rightarrow^{\beta} \mathsf{Unit}$ and where C_1 contains the constraint $\beta \dot{\sqsupseteq} !\alpha$, where $\alpha \in \mathsf{fv}(E)$. The intuitive interpretation of this is that when analysing e_0, β is a polymorphic variable with a non-polymorphic subpart α. Thus β cannot be instantiated freely; hence the need to carry α around in the environment (via the type scheme $\forall\overline{\{\alpha\}}.t_1$).

Enriched type schemes in the completeness proof. It turns out to be hard to give a direct proof of the completeness of W with respect to the inference system of Fig. 2, due to the fact that for inferences in this system it does not necessarily hold that "bound variables are unique" (space limits prevent us from giving a more precise definition and/or an example of this phenomenon). We shall alleviate this by presenting an alternative (but equivalent) inference system having the aforementioned property:

An enriched inference system. Inferences are of form $E \vdash_2 e : t \ \&\ b$, where E is an enriched environment. The system is as in Fig. 2, except that the rule for polymorphic abstraction is as follows:

$$\frac{\begin{array}{c} E \vdash_2 e_1 : t_1 \ \&\ b_1 \\ E \oplus [x : \forall\overline{F}.t_1] \vdash_2 e_0 : t \ \&\ b_0 \end{array}}{E \vdash_2 \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0 : t \ \&\ b} \text{ if } F = \mathsf{fv}(E) \cup \mathsf{fv}(b_1) \text{ and } b \sqsupseteq b_1; b_0. \quad (3)$$

The two inference systems are equivalent in the following sense:

Theorem 6.1 $\emptyset \vdash e : t \ \&\ b$ holds iff $\emptyset \vdash_2 e : t \ \&\ b$ holds.

This is a consequence of the following two propositions:

Proposition 6.2 Suppose $E \vdash_2 e : t \ \&\ b$. Then also $\kappa(E) \vdash e : t \ \&\ b$.

Proposition 6.3 Suppose $E \vdash e : t \ \&\ b$ and that $\kappa(E') = E$. Then also $E' \vdash_2 e : t \ \&\ b$.

7 Completeness of algorithm W

We are now in a position to prove that algorithm W is complete: if there exists a valid inference then the algorithm will produce a set of constraints that can be satisfied.

Theorem 7.1 Suppose $\emptyset \vdash e : t' \ \&\ b'$. Then $W(e, \emptyset) = (t, b, C, \theta)$ and there exists a ψ such that $\psi \models C$ and $t' = t[\psi]$ and $b' \sqsupseteq b[\psi]$.

Notice the symmetry with Theorem 5.1. The theorem follows easily from Theorem 6.1 and the following proposition that admits a proof by induction:

Proposition 7.2 Suppose $E' \vdash_2 e : t' \,\&\, b'$ and $E[\phi] \succeq E'$. Then $W(e, E) = (t, b, C, \theta)$ and there exists a ψ such that $\theta; \psi \overset{\mathsf{var}(E)}{=} \phi$ and $\psi \models C$ and $t[\psi] = t'$ and $b' \sqsupseteq b[\psi]$.

Proof: Induction on the derivation tree for $E' \vdash_2 e : t' \,\&\, b'$; to conserve space we use the same terminology as in the definition of the relevant clause for W and we write e.g. $\psi \overset{E}{=} \phi$ instead of $\psi \overset{F}{=} \phi$ where $F = \mathsf{var}(E)$. There are seven cases; here we shall list the two most interesting ones only.

The case $e = x$: Suppose $E' \vdash_2 x : t' \,\&\, b'$ holds because $E'(x) = \forall \overline{F'_x}.t'_x$, because $t' = t'_x[\phi']$ with $\mathsf{dom}(\phi') \cap F'_x = \emptyset$, and because $b' \sqsupseteq \epsilon$.

Since $E[\phi] \succeq E'$ it holds that $t_x[\phi] = t'_x$ (thanks to our syntactic definition of $\succeq$) and that $\mathsf{var}(F_x[\phi]) \subseteq F'_x$. From this we infer that

$$\mathsf{dom}(\phi') \cap \mathsf{var}(F_x[\phi]) = \emptyset. \tag{4}$$

Now define ψ as follows: it maps $\vec{\gamma'}$ into $\vec{\gamma}[\phi; \phi']$; and otherwise it behaves like ϕ. This ensures that $\theta; \psi \overset{E}{=} \phi$; and it is trivial that $b' \sqsupseteq b[\psi]$.

For our remaining claims, observe that due to (4) we have

$$[\vec{\gamma} \mapsto \vec{\gamma'}]; \psi \overset{\vec{\gamma} \cup F_x}{=} \psi; \phi' \text{ with } \mathsf{dom}(\phi') \cap \mathsf{fv}(F_x[\psi]) = \emptyset$$

showing that $\forall \overline{F_x[\psi]}.\vec{\gamma}[\psi] \succ \vec{\gamma'}[\psi]$ implying $\psi \models C$; and that (since $\mathsf{var}(t_x) \subseteq F_x \cup \vec{\gamma}$) $t[\psi] = t_x[[\vec{\gamma} \mapsto \vec{\gamma'}]; \psi] = t_x[\psi; \phi'] = t_x[\phi; \phi'] = t'_x[\phi'] = t'$.

The case $e = \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$: First a *remark:* the only property of $\mathcal{NQ}$ we shall exploit is the following:

$$\psi \models C_1 \text{ implies } \mathsf{var}(\mathcal{NQ}[\psi]) \subseteq \mathsf{var}(\mathsf{fv}(E[\theta_1])[\psi]) \cup \mathsf{var}(b_1[\psi]). \tag{5}$$

Notice the duality between (2) and (5); and notice that the conjunction of these two is a consequence of Fact 2.3.

Some terminology: we define $V_1 = \mathsf{var}(t_1, b_1, C_1, \theta_1)$ and $V_0 = \mathsf{var}(t_0, b_0, C_0, \theta_0)$ and $V_e = \mathsf{var}(E[\theta_1]) \cup \mathcal{NQ} \cup \mathsf{fv}(t_1)$; and say that a variable is "internal" if it occurs in V_1 but not in V_e. We can assume that no internal variable occurs in V_0.

Suppose $E' \vdash_2 \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0 : t' \,\&\, b'$ because of $E' \vdash_2 e_1 : t'_1 \,\&\, b'_1$ and of $F' = \mathsf{fv}(E') \cup \mathsf{fv}(b'_1)$ and of $E' \oplus [x : \forall \overline{F'}.t'_1] \vdash_2 e_0 : t' \,\&\, b'_0$ and because of $b' \sqsupseteq b'_1; b'_0$. By induction we see that $W(e_1, E)$ succeeds and that there exists ψ_1 such that $\theta_1; \psi_1 \overset{E}{=} \phi$, such that $\psi_1 \models C_1$, such that $t_1[\psi_1] = t'_1$, and such that $b'_1 \sqsupseteq b_1[\psi_1]$. It holds that

$$\mathsf{var}(\mathcal{NQ}[\psi_1]) \subseteq F'$$

for by (5) we have $\mathsf{var}(\mathcal{NQ}[\psi_1]) \subseteq \mathsf{var}(\mathsf{fv}(E[\theta_1])[\psi_1]) \cup \mathsf{var}(b_1[\psi_1])$, by assumption we have $\mathsf{var}(\mathsf{fv}(E[\theta_1])[\psi_1]) = \mathsf{fv}(E[\theta_1][\psi_1]) = \mathsf{fv}(E[\phi]) \subseteq \mathsf{fv}(E')$, and by Fact 2.1 we have $\mathsf{fv}(b_1[\psi_1]) \subseteq \mathsf{fv}(b'_1)$.[2]

[2] It is not possible to replace this subset inclusion by set equality. This is the motivation for making $E[\phi] \succeq E'$ (and not $E[\phi] = E'$) be part of the induction hypothesis.

Using the above observation it is straightforward to verify that

$$(E[\theta_1] \oplus [x : \forall\overline{\mathcal{NQ}}.t_1])[\psi_1] \succeq E' \oplus [x : \forall\overline{F'}.t_1']$$

so by induction we see that $W(e_0, \_)$ succeeds and that there exists ψ_0 such that $\theta_0; \psi_0 \stackrel{V_e}{=} \psi_1$, such that $\psi_0 \models C_0$, such that $t_0[\psi_0] = t'$ and such that $b_0' \sqsupseteq b_0[\psi_0]$.

Now define ψ to behave as ψ_0 except that is behaves as ψ_1 on internal variables. We have the following relations:

$$\psi \stackrel{V_0}{=} \psi_0 \text{ and } \theta_0; \psi \stackrel{V_1 \cup V_e}{=} \psi_1 \tag{6}$$

where the second part follows from the following reasoning: if γ is internal then $\gamma[\theta_0; \psi] = \gamma[\psi] = \gamma[\psi_1]$; and if γ is not internal (and hence belongs to V_e) then $\gamma[\theta_0]$ does not contain any internal variables so $\gamma[\theta_0; \psi] = \gamma[\theta_0; \psi_0] = \gamma[\psi_1]$.

Using (6) enables us to infer the desired properties of ψ: (i) if $\gamma \in \mathsf{var}(E)$ then $\gamma[\theta; \psi] = \gamma[\theta_1][\theta_0; \psi] = \gamma[\theta_1][\psi_1] = \gamma[\phi]$; (ii) $\psi \models C$ holds because $\psi \models C_1[\theta_0]$ (which follows from $\psi_1 \models C_1$) and because $C_0[\psi] = C_0[\psi_0]$; (iii) $t[\psi] = t_0[\psi_0] = t'$; (iv) we infer that $b' \sqsupseteq b_1'; b_0' \sqsupseteq b_1[\psi_1]; b_0[\psi_0] = b_1[\theta_0][\psi]; b_0[\psi] = b[\psi]$.

8 Choice of generalisation strategy

In this section we shall discuss alternative ways of defining the entity $\mathcal{NQ}$ occurring in Figure 3 (the clause for polymorphic abstractions). Recall that $\mathcal{NQ}$ satisfies the following property:

$$\beta \dot{\sqsupseteq} b \in C_1 \text{ and } \beta \in \mathcal{NQ} \text{ implies } \mathsf{fv}(b) \subseteq \mathcal{NQ}$$

stating that if a variable is not polymorphic then neither of its "subparts" can be polymorphic. We now consider two variations on this theme.

8.1 An algorithm not taking the downwards closure

First define a version of W, to be called W^n for <u>n</u>o closure, where simply $\mathcal{NQ} = \mathsf{fv}(E[\theta_1]) \cup \mathsf{fv}(b_1)$. It is trivial that (2) and (5) still hold and hence W^n will be sound as well as complete. On the other hand, failures are reported at a later stage as witnessed by the expression $e =$ let ch=channel () **in** e_1 where in e_1 an integer as well as a boolean is transmitted over the newly generated channel ch. W^n applied to e will terminate successfully and return constraints including the following

$$[\beta \dot{\sqsupseteq} \alpha\ \mathsf{CHAN},\ \forall\overline{\{\beta\}}.\alpha \dot{\succ} \mathsf{Bool},\ \forall\overline{\{\beta\}}.\alpha \dot{\succ} \mathsf{Int}]$$

which are easily seen to be unsolvable; whereas W applied to e will fail immediately (since α is considered a non-polymorphic variable and hence is not copied, causing UNIFY to fail). So it seems that W should be preferred to W^n on the basis that failures should be reported as early as possible.

Note that e above can be typed if ch is assigned the type scheme $\forall\overline{\emptyset}.\alpha$ **chan** but cannot be typed if ch is assigned the type scheme $\forall\overline{\{\alpha\}}.\alpha$ **chan**. The former case will arise if α **chan** is handled as α **list**; whereas the latter case will arise if α **chan** is handled by the techniques for α **ref** developed in [TJ92b] and [BD93].

8.2 An algorithm taking the upwards closure

Next define a version of W, to be called W^{ud} for upwards and downwards closure, where $\mathcal{NQ}$ is defined in such a way that

$$\beta \dot{\sqsupseteq} b \in C_1 \text{ and } (\mathsf{fv}(b) \cup \{\beta\}) \cap \mathcal{NQ} \neq \emptyset \text{ implies } (\mathsf{fv}(b) \cup \{\beta\}) \subseteq \mathcal{NQ}$$

which in effect states that in order for a variable to be polymorphic, all of its "subparts" should be polymorphic as well. As we shall see this approach to separating polymorphic and non-polymorphic variables is too strong. On the positive side (2) still holds so W^{ud} is sound. On the negative side (5) does not hold; in fact completeness fails because we have found a well-typed expression upon which W^{ud} does not succeed.

The algorithm presented in [NN94b] essentially works similar to W^{ud}.

9 Solving the constraints

In this section we discuss how to solve the constraints generated by Algorithm W. As stated in Fact 4.3, the C-constraints are of form $\beta \dot{\sqsupseteq} b$ and the S-constraints are of form $\forall \overline{F}.\vec{t}\vec{\beta} \dot{\succ} \vec{t'}\vec{\beta'}$ (where actually, among other things, $\vec{t}$ is of form $\vec{\alpha}$ and where $\vec{\alpha}$ as well as $\vec{\beta}$ are disjoint variables). The right hand sides of the C-constraints may be quite lengthy, for instance they will often involve sub-behaviours of form $\epsilon;\epsilon;\ldots$, but we have implemented an algorithm that applies the behaviour equivalences from Fig. 1 so as to decrease the size of behaviours significantly (in most cases).

Solving constraints sequentially. Given a set of constraints C, a natural way to search for a substitution ψ that satisfies C is to proceed sequentially:

- if C is empty, let $\psi = id$;
- otherwise, let C be the disjoint union of C' and C''. Suppose we can find ψ' which solves C'. Next suppose we (recursively) can find ψ'' which solves $C''[\psi']$. Then return $\psi = \psi'; \psi''$.

It is easy to see that $\psi \models C$ provided $C'[\psi']$ is such that for all ϕ we have $\phi \models C'[\psi']$. This will be the case if $C'[\psi]$ only contains C-constraints (due to Fact 2.1) and S-constraints of form $\forall \overline{F}.\vec{g} \dot{\succ} \vec{g}$, the latter kind to be denoted S-equalities. So we arrive at the following (informally specified) condition:

$$\text{S-constraints are not solved unless they become S-equalities.} \tag{7}$$

Moreover, we have the following (also informally specified) condition:

$$\text{applying } \psi' \text{ to } C'' \text{ should preserve any "nice structure" that } C'' \text{ has.} \tag{8}$$

Some constraints we know how to solve. If C is a list of constraints, such that all S-constraints in C are of form $\forall \overline{F}.(\vec{\alpha}, \vec{\beta}) \dot{\succ} (\vec{\alpha'}, \vec{\beta'})$, we can apply the scheme for sequential solution outlined in the preceding section (we shall not deal with other kinds of S-constraints even though some of those might have simple solutions as well).

First we convert the S-constraints into S-equalities, cf. (7). This is done by identifying all type and behaviour variables occurring in "corresponding positions" (thus going from a polymorphic world into a monomorphic world).

Next we have to solve the C-constraints sequentially; and the invariant to be preserved during this process (cf. (8)) is that they should be of form $\beta \dot{\sqsupseteq} b$ (where b is no longer assured to be a simple behaviour, as it may contain recursion). It is easy to see that this invariant will be maintained provided we can solve a constraint set of the form $\{\beta \dot{\sqsupseteq} b_1, \ldots, \beta \dot{\sqsupseteq} b_n\}$ by means of a substitution whose domain is $\{\beta\}$. But this can easily be achieved, due to rule R1 in Figure 1: just map β into $\mathsf{REC}\beta.b_1 + \ldots + b_n$ (if β does not occur in the b_i's, we can omit the recursion). This is also the way that C-constraints are solved in [NN94b].

10 Conclusion

In this paper we have adapted the traditional algorithm W to our type and behaviour system. The algorithm has been implemented and has provided quite illuminating analyses of example CML programs.

One difference from the traditional formulation of W is that we generate so-called C-constraints that then have to be solved. This is a consequence of our behaviours being a non-free algebra and is an phenomenon found also in [JG91].

Another and major difference from the traditional formulation, as well as that of [JG91], is that we generate so-called S-constraints that also have to be solved. This phenomenon is needed because our C-constraints would seem not to have principal solutions (as was discussed in Section 2). This in not the case for the traditional "free" unification of Standard ML, but it is a phenomenon well-known in unification theory [Sie89]. As a consequence we have to ensure that the different solutions to the C-constraints (concerning the polymorphic definition and its instantiations) are comparable and this is the purpose of the S-constraints. At present it is an open problem how hard it is to solve S-constraints in the presence of C-constraints and whether it is as hard as general semi-unification (which is undecidable). This problem is closely related to the question of whether all expressions typeable in (C)ML are also typable in our system. More precisely consider an expression e that is typeable in Standard ML when types α **chan** are handled by the techniques for α **ref** developed in [TJ92b]; the (open) question then is whether this expression is always typeable in our system. If this should ever fail it means that the causality espressed in the behaviours constitute an unwarranted restriction upon the language.

Future work will show how to overcome these problems. One avenue of research is to extend our axiomatization of $\sqsupseteq$ such that principal solutions exist; perhaps by identifying recursive behaviours with their infinite unfolding just as done in [CC91] for recursive types. Another avenue is to accept that there definitely will be applications of "effect systems" in theories that are known not to have principal solutions; in this case we may try to reduce our problems by incorporating a more general rule for subtyping into the inference system.

Acknowledgements This research has been supported by the DART and LOMAPS projects.

References

[BD93] Dominique Bolignano and Mourad Debabi. A coherent type system for a concurrent, functional and imperative programming language. In *AMAST '93*, 1993.

[BS94] Bernard Berthomieu and Thierry Le Sergent. Programming with behaviours in an ML framework: the syntax and semantics of LCS. In *ESOP '94*, volume 788 of *LNCS*, pages 89–104. Springer-Verlag, 1994.

[CC91] Felice Cardone and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92:48–80, 1991.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[JG91] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages*, pages 303–310. ACM Press, 1991.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[NN93] Flemming Nielson and Hanne Riis Nielson. From CML to process algebras. In *CONCUR '93*, volume 715 of *LNCS*. Springer-Verlag, 1993. An expanded version appears as DAIMI technical report no. PB-433.

[NN94a] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *ACM Symposium on Principles of Programming Languages*. ACM Press, January 1994.

[NN94b] Flemming Nielson and Hanne Riis Nielson. Constraints for polymorphic behaviours of concurrent ML. In *Constraints in Computational Logics (CCL '94)*, volume 845 of *LNCS*. Springer-Verlag, September 1994.

[NN94c] Hanne Riis Nielson and Flemming Nielson. Static and dynamic processor allocation for higher-order concurrent languages. Technical Report PB-483, DAIMI, University of Aarhus, Denmark, 1994. Also see this volume.

[PGM90] Sanjiva Prasad, Alessandro Giacalone, and Prateek Mishra. Operational and algebraic semantics for Facile: A symmetric integration of concurrent and functional programming. In *ICALP 90*, 1990.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991.

[Sie89] Jörg H. Siekmann. Unification theory. *J. Symbolic Computation*, 7:207–274, 1989.

[TJ92a] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programing*, 2(3):245–271, 1992.

[TJ92b] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *IEEE Symposium on Logic in Computer Science*, pages 162–173, 1992. Also see Information and Computation **111** *2*, 1994.

Applications of Summation and Product Operators in the Refinement Calculus

Ralph-Johan Back Michael Butler
Dept. of Computer Science, Åbo Akademi, Finland
{backrj,mbutler}@abo.fi

30 November 1994

Abstract

Product and summation operators for predicate transformers were introduced by Naumann [20] and by Martin [15] using category theoretic considerations. In this paper, we formalise these operators in the higher order logic approach to the refinement calculus of [6], look at various algebraic properties of these operators, and examine several of their applications. We look at how the product operator provides a model of simultaneous execution of statements, while the summation operator provides a simple model of late binding. We also generalise the product operator slightly to form an operator that corresponds to conjunction of specifications. We show how the product operator may be used to model extension and modification operators for programs, and how a combination of the product and summation operators may be used to model inheritance in an object-oriented programming language.

1 Introduction

Dijkstra introduced weakest-precondition predicate transformers as a means of verifying total correctness properties of sequential programs [9]. In the *refinement calculus* of Back, Morgan, and Morris [2, 16, 18], specifications and programs are regarded uniformly as predicate transformers, and refinement laws are derived from properties of predicate transformers.

The refinement calculus provides various choice and assignment operators that are generalisations of Dijkstra's operators, and the applications of these operators are well-known. However, the applications of an operator representing simultaneous execution of program statements are less well developed in the refinement calculus. Such an operator was introduced by Naumann [20] and by Martin [15] using category theoretic considerations. This *product* operator combines predicate transformers by forming the cartesian product of their state spaces. In this paper, we examine the product operator using the higher-order logic formalisation of the refinement calculus of Back & von Wright [6]. We examine various distributivity and refinement preserving properties of the operator and show that it can be used to model simultaneous execution and to extend the state spaces of statements so they can be more easily matched with other statements. We also generalise the definition of the product operator slightly to form what we call a *fusion* operator and show that the product operator is a special case of the fusion operator. The fusion operator can also be applied to conjoining or amalgamating specification statements.

The *summation* (or *co-product*[1]) operator, which is the categorical dual of the product operator and combines statements by forming the disjoint union of their state spaces, is also described in [20] and [15]. The summation operator is a form of choice operator and we show that it is a special case of the existing choice operators of the refinement calculus. We show that this operator provides a simple yet powerful model of late binding, and that when combined with the product operator, provides a model of inheritance in an object-oriented programming language.

The paper is organised as follows. First the refinement calculus basics are described. Then Section 3 examines properties of the summation operator, while Section 4 examines properties of the product and fusion operators. In Section 5 we look briefly at the categorical properties of the summation and product operators. Section 6 looks at various applications of the operators including simultaneous execution, extending state spaces, late binding, and inheritance. Note that proofs are omitted here, but may be found in [3].

2 Refinement Calculus Basics

A *predicate* over a set of states Σ is a boolean function $p : \Sigma \to \mathbf{Bool}$ which assigns a truth value to each state. The set of predicates on Σ is denoted $\mathcal{P}\,\Sigma$:

$$\mathcal{P}\,\Sigma \;\;\widehat{=}\;\; \Sigma \to \mathbf{Bool}.$$

We define the *entailment ordering* on predicates by pointwise extension: for $p, q : \mathcal{P}\,\Sigma$,

$$p \leq q \;\;\widehat{=}\;\; (\forall \sigma : \Sigma \cdot p\ \sigma \;\Rightarrow\; q\ \sigma).$$

The identically false predicate is denoted $\bot$, and the identically true predicate is denoted $\top$. Negation, conjunction, and disjunction of (similarly-typed) predicates are defined by pointwise extension, so that, e.g.,

$$(p \wedge q)\ \sigma \;\;\widehat{=}\;\; (p\ \sigma \;\wedge\; q\ \sigma).$$

Predicates form a complete lattice under the entailment ordering. Conjunction and disjunction represent meet and join respectively, while $\top$ and $\bot$ represent top and bottom respectively.

A *relation* from Σ to Γ is a function $P : \Sigma \to \mathcal{P}\,\Gamma$ that maps each state σ to a predicate on Γ. We write

$$\Sigma \leftrightarrow \Gamma \;\;\widehat{=}\;\; \Sigma \to \mathcal{P}\,\Gamma.$$

This view of relations is isomorphic to viewing them as predicates on the cartesian space $\Sigma \times \Gamma$. The domain and range of relation P, are denoted $dom\,P$ and $ran\,P$ respectively. Conjunction and disjunction of relations is defined pointwise, so that, e.g., $(P \wedge Q)\ \sigma\ \gamma \;=\; (P\ \sigma\ \gamma) \wedge (Q\ \sigma\ \gamma)$. We write $P;Q$ for the relational composition of $P : \Sigma \leftrightarrow \Gamma$ and $Q : \Gamma \leftrightarrow \Lambda$, and P^{-1} for

[1]The direction of arrows in [20] and [15] are the reverse of what we use, so our product operator corresponds to their co-product operator, and vice versa.

the inverse relation. For function $f : \Sigma \to \Gamma$, we write f^{-1} for the relation $(\lambda\gamma \cdot (\lambda\sigma \cdot \gamma = f\sigma)) : \Gamma \leftrightarrow \Sigma$. A partial function $g : \Sigma \to_p \Gamma$ is a relation from Σ to Γ such that $g\ \sigma\ \gamma\ \wedge\ g\ \sigma\ \gamma' \ \Rightarrow\ \gamma = \gamma'$.

A *predicate transformer* is a function $S : \mathcal{P}\,\Gamma \to \mathcal{P}\,\Sigma$ from predicates to predicates. We write,

$$\Sigma \longmapsto \Gamma \quad \widehat{=} \quad \mathcal{P}\,\Gamma \ \to\ \mathcal{P}\,\Sigma.$$

Note the reversal of Γ and Σ: program statements in the refinement calculus are identified with weakest-precondition predicate transformers that map a postcondition $q : \mathcal{P}\,\Gamma$ to the weakest precondition $p : \mathcal{P}\,\Sigma$ such that the program is guaranteed to terminate in a final state satisfying q whenever the initial state satisfies p. For program statement $S : \Sigma \longmapsto \Gamma$, we say that S has *source* Σ and *target* Γ. Programs need not have identical initial and final state spaces, though if they do, we write $S : \Delta(\Sigma)$ instead of $S : \Sigma \longmapsto \Sigma$. Note that, in order to aid intuition, we sometimes discuss commands from an operational viewpoint, though our formal reasoning is always in terms of predicate transformers.

The *refinement ordering* on predicates is defined by pointwise extension: for $S, T : \Sigma \longmapsto \Gamma$,

$$S \leq T \quad \widehat{=} \quad (\forall q : \mathcal{P}\,\Gamma \cdot S\ q\ \leq\ T\ q).$$

The refinement ordering on predicates models the notion of total-correctness preserving program refinement. A total-correctness specification is typically given as a precondition-postcondition pair $(pre, post)$, and program (i.e., predicate transformer) S satisfies $(pre, post)$ if $pre \leq (S\ post)$. Now, for programs S and T, $S \leq T$ holds if and only if T satisfies any specification satisfied by S.

Predicate transformers form a complete lattice under the refinement ordering. The bottom element is the predicate transformer *abort* that maps each postcondition to $\bot$, and the top element is the predicate transformer *magic* that maps each postcondition to $\top$. The *abort* statement is never guaranteed to terminate, while the *magic* statement is *miraculous* since it is always guaranteed to establish any postcondition. A miraculous statement cannot be implemented. For statement S, $halt\ S \ \widehat{=}\ S\ \top$ describes those initial states under which S is guaranteed to terminate, while $gd\ S \ \widehat{=}\ \neg\,(S\ \bot)$ (called *guard* of S) describes those initial states under which S behaves non-miraculously.

Conjunction and disjunction of (similarly-typed) predicate transformers is defined pointwise, so that, e.g., $(S \wedge T)\ q = (S\ q) \wedge (T\ q)$. Conjunction of statements models *demonic nondeterministic choice* between executing S and T (i.e., each alternative must establish the postcondition), whereas disjunction models *angelic nondeterministic choice* (i.e., some alternative must establish the postcondition). If $halt\ S = \neg\ halt\ T$, then $S \vee T$ becomes a deterministic choice, while if $gd\ S = \neg\ gd\ T$, then $S \wedge T$ becomes a deterministic choice.

Sequential composition of program statements is modelled by functional composition of predicate transformers, i.e., for $S : \Sigma \longmapsto \Gamma$, $T : \Gamma \longmapsto \Lambda$, $p : \mathcal{P}\,\Lambda$, $\quad S;T\ p \ \widehat{=}\ S\ (T\ p)$. The program statement $skip_\Sigma$ is modelled by the identity predicate transformer on $\mathcal{P}\,\Sigma$.

Given a relation $P : \Sigma \leftrightarrow \Gamma$, the *angelic update statement* $\{P\} : \Sigma \longmapsto \Gamma$ and *demonic update statement* $[P] : \Sigma \longmapsto \Gamma$ are defined by

$$\{P\}\ q\ \sigma \quad \widehat{=} \quad (\exists \gamma : \Gamma \cdot (P\ \sigma\ \gamma) \wedge (q\ \gamma))$$

$$[P]\ q\ \sigma \quad \widehat{=} \quad (\forall \gamma : \Gamma \cdot (P\ \sigma\ \gamma) \Rightarrow (q\ \gamma)).$$

When started in a state σ, $\{P\}$ angelically chooses a new state γ such that $P\sigma\gamma$ holds, while $[P]$ demonically chooses a new state γ such that $P\sigma\gamma$ holds. If no such new state exists then $\{P\}$ aborts, while $[P]$ behaves as *magic*.

For predicate $p : \mathcal{P}\,\Sigma$, let $\overline{p} : \Sigma \leftrightarrow \Sigma$ be the corresponding test relation for p, i.e., $(\lambda\sigma \cdot \lambda\sigma' \cdot p\ \sigma\ \wedge\ \sigma = \sigma')$. Then we write $\{p\}$ for $\{\overline{p}\}$ and $[p]$ for $[\overline{p}]$. $\{p\}$ models the *assert* statement that behaves as *skip* if p holds, otherwise it aborts. $[p]$ models the *guard* statement that behaves as *skip* if p holds, otherwise it behaves as *magic*.

Given a function $f : \Sigma \to \Gamma$, the *deterministic update statement* $\langle f \rangle : \Sigma \longmapsto \Gamma$ is defined by

$$\langle f \rangle\ q\ \sigma \quad \widehat{=} \quad q\ (f\ \sigma).$$

Let $\tilde{f}$ be the the deterministic relation corresponding to function f, i.e., $(\lambda\sigma \cdot \lambda\gamma \cdot f\ \sigma = \gamma)$. Then we have $\{\tilde{f}\} = \langle f \rangle = [\tilde{f}]$.

Ordinary program constructs such conditionals, recursions, and assignments may be modelled using the basic operators presented above. For example, **if** g **then** S **else** T **fi** may be modelled by $[g]; S\ \wedge\ [\neg\ g]; T$ or alternatively by $\{g\}; S \vee \{\neg\ g\}; T$. Program variables may be modelled as cartesian components of the state space, and a multiple assignment $x, y := e, f$ in a state space with three components, representing program variables (x, y, z), may be modelled by the deterministic update:

$$\langle \lambda x, y, z \cdot (e, f, z) \rangle.$$

It is easy to show, for predicates *pre* and *post*, that $\{pre\}; [post]$ is the least statement satisfying the specification pair $(pre, post)$. Thus, implementing the specification $(pre, post)$ involves constructing a statement S such that $\{pre\}; [post] \leq S$. If *post* is a relation rather than a predicate, we can model postconditions on the before and after states in the manner of VDM [14] and Z [22]. Rules for the stepwise refinement of specification statements of the form $\{pre\}; [post]$ into more familiar program constructs may be found in [2, 16, 18].

All predicate transformers S constructed using the operators described above will be *monotonic*, i.e., $p \leq q\ \Rightarrow\ S\ p \leq S\ q$. A predicate transformer S is *bottom homomorphic* if $S\ \bot = \bot$, and *top homomorphic* if $S\ \top = \top$. *Conjunctive* predicate transformers form a subset of the monotonic predicate transformers satisfying $S\ (\forall i \in I \cdot q_i) = (\forall i \in I \cdot S\ q_i)$, for non-empty I. S is *universally conjunctive* if it is conjunctive and top homomorphic. *Disjunctive* and *universally disjunctive* predicate transformers are defined dually. A predicate transformer is *continuous* if it is disjunctive over non-empty chains of predicates. Demonic update statements are universally conjunctive, angelic update statements are universally disjunctive, and deterministic update statements are both. The operators $\wedge$ and ; preserve the conjunctivity of their operands, while $\vee$ and ; preserve the disjunctivity of their operands. Also, each of $\wedge$, $\vee$, and ; preserve refinement of monotonic predicate transformers, e.g., $S \leq S'\ \Rightarrow\ S; T \leq S'; T$.

In Dijkstra's original calculus, all statements were conjunctive and non-miraculous. Also, for conjunctive statements, disjunctivity corresponds to determinism, while continuity corresponds to bounded nondeterminism [9].

3 Summation Operators

In this section, we describe two summation operators for predicate transformers: the first only sums the operand sources, while the second sums both the operand sources and the operand targets.

The sum (or disjoint union) of two state spaces Σ_1 and Σ_2 is written $\Sigma_1 + \Sigma_2$. Associated with the summation are two *injections* ϕ_1 and ϕ_2 which map elements of the base type to elements of the summation ($A \rightarrow_\iota B$ represents the set of injective functions from A to B):

$$\phi_1 : \Sigma_1 \rightarrow_\iota \Sigma_1 + \Sigma_2 \tag{1}$$

$$\phi_2 : \Sigma_2 \rightarrow_\iota \Sigma_1 + \Sigma_2. \tag{2}$$

Since any element of $\Sigma_1 + \Sigma_2$ must either come from Σ_1 or Σ_2, but not both, the injections also satisfy:

$$(ran\, \phi_1, ran\, \phi_2) \text{ partition } \Sigma_1 + \Sigma_2. \tag{3}$$

We shall assume that all injection pairs (ϕ_1, ϕ_2) associated with summations satisfy (1), (2), and (3).

Since ϕ_1 and ϕ_2 are injective, their inverses ϕ_1^{-1} and ϕ_2^{-1} are partial functions:

$$\phi_1^{-1} : \Sigma_1 + \Sigma_2 \rightarrow_p \Sigma_1 \qquad \phi_2^{-1} : \Sigma_1 + \Sigma_2 \rightarrow_p \Sigma_2.$$

For $\sigma \in \Sigma_1 + \Sigma_2$, $(\phi_1^{-1}\sigma)$ is only defined if $\sigma \in (dom\, \phi_1^{-1})$.

Summation of predicates is given by:

Definition 1 *For $p_1 : \mathcal{P}\,\Sigma_1,\ p_2 : \mathcal{P}\,\Sigma_2, \qquad p_1 + p_2$ is of type $\mathcal{P}\,(\Sigma_1 + \Sigma_2)$, where for $\sigma : \Sigma_1 + \Sigma_2$,*

$$(p_1 + p_2)\ \sigma \ \ \widehat{=}\ \ \sigma \in (dom\, \phi_1^{-1}) \wedge p_1(\phi_1^{-1}\sigma) \quad \vee \quad \sigma \in (dom\, \phi_2^{-1}) \wedge p_2(\phi_2^{-1}\sigma).$$

Given an injection $\phi_1 : \Sigma_1 \rightarrow_\iota \Sigma_1 + \Sigma_2$, the deterministic update

$$\langle \phi_1 \rangle : \Sigma_1 \longmapsto \Sigma_1 + \Sigma_2 \quad (\ =\ \mathcal{P}\,(\Sigma_1 + \Sigma_2) \rightarrow \mathcal{P}\,\Sigma_1)$$

maps predicates in the summation type to predicates in the first base type. The updates can be used to select the components of a summation of predicates:

Theorem 1 *For $p_1 : \mathcal{P}\,\Sigma_1,\ p_2 : \mathcal{P}\,\Sigma_2$,*

$$\langle \phi_1 \rangle\ (p_1 + p_2)\ =\ p_1 \quad \textit{and} \quad \langle \phi_2 \rangle\ (p_1 + p_2)\ =\ p_2.$$

The following theorem shows that a predicate in the summation type can always be separated into a summation of predicates:

Theorem 2 *For $p : \mathcal{P}\,(\Sigma_1 + \Sigma_2), \qquad p\ =\ \langle \phi_1 \rangle\ p\ +\ \langle \phi_2 \rangle\ p$.*

3.1 Summation of Predicate Transformers

The summation of predicate transformers S_1 and S_2 is written $S_1 \oplus S_2$, where S_1 and S_2 have the same target, but possibly distinct sources:

Definition 2 *For $S_1 : \Sigma_1 \longmapsto \Gamma,\ \ S_2 : \Sigma_2 \longmapsto \Gamma$, $S_1 \oplus S_2$ is of type $\Sigma_1 + \Sigma_2 \longmapsto \Gamma$, where for $q : \mathcal{P}\,\Gamma$:*

$$(S_1 \oplus S_2)\ q \ \ \widehat{=}\ \ (S_1\ q)\ +\ (S_2\ q).$$

The selection and separation properties of the summation operator follow directly from Theorems 1 and 2:

Theorem 3 *For* $S_1 : \Sigma_1 \longmapsto \Gamma,\ S_2 : \Sigma_2 \longmapsto \Gamma,\ S : \Sigma_1 + \Sigma_2 \longmapsto \Gamma,$

$$\langle\phi_1\rangle;(S_1 \oplus S_2) \;=\; S_1 \quad \textit{and} \quad \langle\phi_2\rangle;(S_1 \oplus S_2) \;=\; S_2$$

$$S \;=\; \langle\phi_1\rangle;S \;\oplus\; \langle\phi_2\rangle;S.$$

Summation of predicate transformers is a form of choice operator since the effect of executing $S_1 \oplus S_2$ in some initial state σ depends on the base type of σ: if it is of type Σ_1, then S_1 is executed, while if it is of type Σ_2, then S_2 is executed. This may be seen more clearly in the following theorem where the summation operator is characterised purely in terms of the existing choice operators. The theorem uses the following angelic and demonic updates formed from the inverse injections:

$$\{\phi_1^{-1}\} : \Sigma_1 + \Sigma_2 \longmapsto \Sigma_1 \qquad\qquad [\phi_1^{-1}] : \Sigma_1 + \Sigma_2 \longmapsto \Sigma_1.$$

Theorem 4 *For* $S_1 : \Sigma_1 \longmapsto \Gamma,\ S_2 : \Sigma_2 \longmapsto \Gamma,$

$$S_1 \oplus S_2 \;=\; \{\phi_1^{-1}\};S_1 \,\vee\, \{\phi_2^{-1}\};S_2 \;=\; [\phi_1^{-1}];S_1 \,\wedge\, [\phi_2^{-1}];S_2.$$

It is also clear from Theorem 4 that the choice between S_1 and S_2 is deterministic since ϕ_1^{-1} and ϕ_2^{-1} have disjoint domains, so that

$$\mathit{halt}\ \{\phi_1^{-1}\} = \neg\, \mathit{halt}\ \{\phi_2^{-1}\} \quad \text{and} \quad \mathit{gd}\ [\phi_1^{-1}] = \neg\, \mathit{gd}\ [\phi_2^{-1}].$$

3.2 Derived Summation Operator

Next we define a summation operator on predicate transformers with distinct sources and distinct targets. Let ψ_1 and ψ_2 be the injections for the summation $\Gamma_1 + \Gamma_2$ (thus, e.g., $\langle\psi_1\rangle$ has type $\Gamma_1 \longmapsto \Gamma_1 + \Gamma_2$).

Definition 3 *For* $S_1 : \Sigma_1 \longmapsto \Gamma_1,\quad S_2 : \Sigma_2 \longmapsto \Gamma_2,\quad S_1 + S_2$ *is of type* $\Sigma_1 + \Sigma_2 \longmapsto \Gamma_1 + \Gamma_2$, *where*

$$S_1 + S_2 \;\;\widehat{=}\;\; S_1;\langle\psi_1\rangle \;\oplus\; S_2;\langle\psi_2\rangle.$$

The components of $S_1 + S_2$ may be selected as follows:

Theorem 5 *For* $S_1 : \Sigma_1 \longmapsto \Gamma_1,\ S_2 : \Sigma_2 \longmapsto \Gamma_2,$

$$\langle\phi_1\rangle;(S_1+S_2);\{\psi_1^{-1}\} \;=\; \langle\phi_1\rangle;(S_1+S_2);[\psi_1^{-1}] \;=\; S_1$$

$$\langle\phi_2\rangle;(S_1+S_2);\{\psi_2^{-1}\} \;=\; \langle\phi_2\rangle;(S_1+S_2);[\psi_2^{-1}] \;=\; S_2.$$

Summation distributes through sequential composition as follows:

Theorem 6

$$\textbf{(i)}\quad (S_1+S_2)\ ;\ (T_1 \oplus T_2) \;=\; (S_1;T_1) \oplus (S_2;T_2).$$

$$\textbf{(ii)}\quad (S_1+S_2)\ ;\ (T_1 + T_2) \;=\; (S_1;T_1) + (S_2;T_2).$$

In order to consider the summation of angelic and demonic updates, we require the following definition of summation of relations:

Definition 4 *For* $P_1 : \Sigma_1 \leftrightarrow \Gamma_1,\ P_2 : \Sigma_2 \leftrightarrow \Gamma_2,\ \sigma : \Sigma_1 + \Sigma_2,\ \gamma : \Gamma_1 + \Gamma_2,$

$$\begin{aligned}(P_1+P_2)\ \sigma\ \gamma \;\;\widehat{=}\;\; & \sigma \in (\mathit{dom}\,\phi_1^{-1}) \;\wedge\; \gamma \in (\mathit{dom}\,\psi_1^{-1}) \;\wedge\; P_1\ (\phi_1^{-1}\sigma)\ (\psi_1^{-1}\gamma) \quad \vee \\ & \sigma \in (\mathit{dom}\,\phi_2^{-1}) \;\wedge\; \gamma \in (\mathit{dom}\,\psi_2^{-1}) \;\wedge\; P_2\ (\phi_2^{-1}\sigma)\ (\psi_2^{-1}\gamma).\end{aligned}$$

Summation distributes through updates, assert and guard statements as follows:

Theorem 7 *For* $P_1 : \Sigma_1 \leftrightarrow \Gamma_1,\ P_2 : \Sigma_2 \leftrightarrow \Gamma_2,\ p_1 : \mathcal{P}\,\Sigma_1,\ p_2 : \mathcal{P}\,\Sigma_2,$

$$\{P_1\} + \{P_2\} = \{P_1 + P_2\} \quad \textit{and} \quad [P_1] + [P_2] = [P_1 + P_2] \tag{4}$$

$$\{p_1\} + \{p_2\} = \{p_1 + p_2\} \quad \textit{and} \quad [p_1] + [p_2] = [p_1 + p_2]. \tag{5}$$

Both the summation operators preserve refinement allowing us to refine summands separately:

Theorem 8

$$S_1 \leq S_1' \;\wedge\; S_2 \leq S_2' \quad \Rightarrow \quad (S_1 \oplus S_2) \leq (S_1' \oplus S_2')$$
$$T_1 \leq T_1' \;\wedge\; T_2 \leq T_2' \quad \Rightarrow \quad (T_1 + T_2) \leq (T_1' + T_2').$$

4 Product Operators

The cartesian product of state spaces Σ_1 and Σ_2 is denoted $\Sigma_1 \times \Sigma_2$. Given two predicates q_1 and q_2 their product is denoted $q_1 \times q_2$ and is defined as follows:

Definition 5 *For $q_1 : \mathcal{P}\,\Sigma_1$, $q_2 : \mathcal{P}\,\Sigma_2$, $\quad q_1 \times q_2$ is of type $\mathcal{P}\,(\Sigma_1 \times \Sigma_2)$ where for $\sigma_1 : \Sigma_1$, $\sigma_2 : \Sigma_2$,*

$$(q_1 \times q_2)\;(\sigma_1, \sigma_2) \quad = \quad (q_1\;\sigma_1) \wedge (q_2\;\sigma_2).$$

Note that, in contrast to summation of predicates where any predicate of type $\mathcal{P}\,(\Sigma_1 + \Sigma_2)$ may be represented by the summation of two predicates of types $\mathcal{P}\,\Sigma_1$ and $\mathcal{P}\,\Sigma_2$ respectively, predicates of the form $q_1 \times q_2$ only form rectangular subsets of $\mathcal{P}\,(\Sigma_1 \times \Sigma_2)$. Also, the components of a product of predicates cannot always be selected, e.g., p_1 cannot be retrieved from $p_1 \times \perp$. As we shall see, these facts mean that the product operators for predicate transformers do not satisfy as many properties as the summation operator.

Consider the following two predicate transformers which have a common source but distinct targets:

$$S_1 : \Sigma \longmapsto \Gamma_1, \qquad S_2 : \Sigma \longmapsto \Gamma_2.$$

The product operator combines these two commands to form a command with target $\Gamma_1 \times \Gamma_2$ as follows:

Definition 6 *For $S_1 : \Sigma \longmapsto \Gamma_1$, $S_2 : \Sigma \longmapsto \Gamma_2$, $S_1 \otimes S_2$ is of type $\Sigma \longmapsto \Gamma_1 \times \Gamma_2$, where for $q : \mathcal{P}\,(\Gamma_1 \times \Gamma_2)$:*

$$(S_1 \otimes S_2)\;q \quad \widehat{=} \quad (\exists q_1 : \mathcal{P}\,\Gamma_1;\; q_2 : \mathcal{P}\,\Gamma_2 \,|\, q_1 \times q_2 \leq q \cdot S_1 q_1 \;\wedge\; S_2 q_2).$$

We shall see later that execution of $S_1 \otimes S_2$ has the same effect as simultaneous execution of S_1 and S_2.

Before investigating properties of the product operator further, we first look at two related operators. The first is derived from the product operator. Let π_1 and π_2 be the projections from $\Sigma_1 \times \Sigma_2$ to Σ_1 and Σ_2 respectively:

$$\pi_1 : \Sigma_1 \times \Sigma_2 \to \Sigma_1, \qquad \pi_2 : \Sigma_1 \times \Sigma_2 \to \Sigma_2,$$
$$\pi_1(\sigma_1, \sigma_2) = \sigma_1, \qquad \pi_2(\sigma_1, \sigma_2) = \sigma_2.$$

The projections give the following deterministic updates:

$$\langle \pi_1 \rangle : \Sigma_1 \times \Sigma_2 \longmapsto \Sigma_1 \quad \text{and} \quad \langle \pi_2 \rangle : \Sigma_1 \times \Sigma_2 \longmapsto \Sigma_2.$$

The statement $\langle \pi_1 \rangle$ starts in a state (σ_1, σ_2) and ends in the state σ_1 simply discarding the second component. The derived product operator combines commands with distinct sources and distinct targets, and is defined as follows:

Definition 7 *For $S_1 : \Sigma_1 \longmapsto \Gamma_1$, $S_2 : \Sigma_2 \longmapsto \Gamma_2$, $S_1 \times S_2$ is of type $\Sigma_1 \times \Sigma_2 \longmapsto \Gamma_1 \times \Gamma_2$, where*

$$S_1 \times S_2 \quad \widehat{=} \quad \langle \pi_1 \rangle; S_1 \;\otimes\; \langle \pi_2 \rangle; S_2.$$

We shall see later that the derived product operator models simultaneous execution of two commands with distinct initial states and distinct final states.

Definition 6 may be generalised to define what we term a *fusion* operator that combines commands with common sources and common targets as follows:

Definition 8 *For $S_1, S_2 : \Sigma \longmapsto \Gamma$, $S_1 \odot S_2$ is of type $\Sigma \longmapsto \Gamma$, where for $q : \mathcal{P}\,\Gamma$:*

$$(S_1 \odot S_2)\ q \ \ \widehat{=}\ \ (\exists q_1, q_2 : \mathcal{P}\,\Gamma \mid q_1 \wedge q_2 \leq q \cdot S_1 q_1 \ \wedge\ S_2 q_2).$$

Neither product operator is commutative, since $\Sigma_1 \times \Sigma_2$ is different to $\Sigma_2 \times \Sigma_1$ when Σ_1 and Σ_2 are different. However, it is clear from Definition 8 that the fusion operator is commutative. We proceed by investigating further properties of the fusion operator and showing that the product operators are special cases of the fusion operator.

4.1 Properties of Fusion Operator

The fusion operator preserves refinement:

Theorem 9 $\quad S_1 \leq S_1' \ \wedge\ S_2 \leq S_2' \ \ \Rightarrow\ \ (S_1 \odot S_2) \leq (S_1' \odot S_2')$.

A program specification takes the form $\{p\};[P]$, where p is a state predicate and P is a state relation. The following theorem provides a simple way of calculating the fusion of two such specifications:

Theorem 10 *For relations $P, Q : \Sigma \leftrightarrow \Gamma$, and predicates $p, q : \mathcal{P}\,\Sigma$,*

$$\{p\};[P] \ \odot\ \{q\};[Q] \ \ =\ \ \{p \wedge q\};[P \wedge Q].$$

The theorem illustrates that the effect of the fusion operator is to reduce the (demonic) nondeterminism of the terminating behaviour of both commands. The theorem also illustrates that the fusion operator can be used as a (refinement preserving) way of combining program specifications.

Given a predicate transformer $S : \Sigma \longmapsto \Gamma$, we derive a relation, called $rel\ S$, of type $\Sigma \leftrightarrow \Gamma$ as follows:

$$(rel\ S)\ \sigma\ \gamma \ \ \widehat{=}\ \ \neg\ S(\lambda\gamma' \cdot \gamma \neq \gamma')\ \sigma.$$

This definition is from [10]. It can be shown that any conjunctive predicate transformer may be characterised as a specification involving $halt\ S$ and $rel\ S$:

Lemma 1 *For conjunctive predicate transformer S,* $\quad S \ =\ \{halt\ S\};[rel\ S]$.

Using this lemma and Theorem 10, we arrive at the following theorem:

Theorem 11 *For conjunctive predicate transformers S and T,*

$$S \odot T \ \ =\ \ \{\ (halt\ S) \wedge (halt\ T)\ \}\ ;\ [\ (rel\ S) \wedge (rel\ T)\ ].$$

Abrial [1] describes a parallel operator for statements whose definition is similar to the right-hand side of Theorem 11. Thus our fusion of statements is similar to Abrial's parallel composition, for conjunctive statements.

We say that two specifications $\{p\};[P]$ and $\{q\};[Q]$ are *contradictory* in some initial state σ if σ has a successful outcome in each specification, but does not have a successful outcome in their fusion, i.e., σ holds in p, q, $dom\ P$, and $dom\ Q$, but not in $dom\ (P \wedge Q)$. For those contradictory initial states, the fusion of both specifications behaves miraculously. Ward [25] has defined a slight variant of Abrial's parallel combinator where the combination behaves as *abort* when both specifications are contradictory. Thus, in Ward's case, a contradictory combination may be refined by any statement whereas, in our case, it is unimplementable. Furthermore, Ward's combinator is not refinement preserving.

4.2 Properties of Product Operator

Now we show that the product operator is a special case of the fusion operator. Let ϖ_1 and ϖ_2 be the projections from $\Gamma_1 \times \Gamma_2$ to Γ_1 and Γ_2 respectively:

$$\varpi_1 : \Gamma_1 \times \Gamma_2 \to \Gamma_1, \qquad \varpi_2 : \Gamma_1 \times \Gamma_2 \to \Gamma_2.$$

The inverses of ϖ_1 and ϖ_2 are relations:

$$\varpi_1^{-1} : \Gamma_1 \leftrightarrow \Gamma_1 \times \Gamma_2, \qquad \varpi_2^{-1} : \Gamma_2 \leftrightarrow \Gamma_1 \times \Gamma_2,$$

so that the demonic update $[\varpi_1^{-1}]$ transforms predicates in $\Gamma_1 \times \Gamma_2$ to predicates in Γ_1:

$$[\varpi_1^{-1}] : \Gamma_1 \longmapsto \Gamma_1 \times \Gamma_2, \qquad (= \mathcal{P}(\Gamma_1 \times \Gamma_2) \to \mathcal{P}\,\Gamma_1),$$

and similarly for $[\varpi_2^{-1}]$. The command $[\varpi_1^{-1}]$ starts in a state $\gamma_1 : \Gamma_1$ and ends in a state $(\gamma_1', \gamma_2') : \Gamma_1 \times \Gamma_2$ such that $\gamma_1 = \gamma_1'$ and γ_2' is chosen nondeterministically. Given $S_1 : \Sigma \longmapsto \Gamma_1$, the predicate transformer $S_1; [\varpi_1^{-1}] : \Sigma \longmapsto \Gamma_1 \times \Gamma_2$ is a 'lifted' version of S_1 that behaves as S_1 on the first component of the final state, and nondeterministically writes any value to the second component. Similarly for $S_2; [\varpi_2^{-1}]$. Now we have:

Theorem 12 *For $S_1 : \Sigma \longmapsto \Gamma_1$, $S_2 : \Sigma \longmapsto \Gamma_2$,*

$$S_1 \otimes S_2 \quad = \quad S_1; [\varpi_1^{-1}] \ \odot \ S_2; [\varpi_2^{-1}].$$

Preservation of refinement follows from Theorems 9 and 12. The product operator does not, in general, satisfy selection and separation properties, though it does satisfying the following:

Theorem 13 *For monotonic $S_1 : \Sigma \longmapsto \Gamma_1$, $S_2 : \Sigma \longmapsto \Gamma_2$, $q : \mathcal{P}\,\Gamma_1$,*

$$(S_1 \otimes S_2); \langle \varpi_1 \rangle\ q \quad = \quad (S_1 q \ \wedge \ S_2 \top) \ \vee \ (S_1 \top \ \wedge \ S_2 \perp).$$

The separation property does not hold for the product operator, though we do have the following inequality for conjunctive S:

Theorem 14 *For conjunctive $S : \Sigma \longmapsto \Gamma_1 \times \Gamma_2$,* $\quad S; \langle \varpi_1 \rangle \ \otimes \ S; \langle \varpi_2 \rangle \ \leq \ S.$

Assume that for functions $f_1 : \Sigma \to \Gamma_1$, $f_2 : \Sigma \to \Gamma_2$ and relations $P_1 : \Sigma \leftrightarrow \Gamma_1$, $P_2 : \Sigma \leftrightarrow \Gamma_2$ we have:

$$f_1 \otimes f_2 \ \ \sigma \quad \widehat{=} \quad (f_1\ \sigma, f_2\ \sigma)$$

$$P_1 \otimes P_2\ \sigma\ (\gamma_1, \gamma_2) \quad \widehat{=} \quad P_1\ \sigma\ \gamma_1 \quad \wedge \quad P_2\ \sigma\ \gamma_2.$$

The product operator then combines with specification statements and deterministic updates in the following manner:

Theorem 15 *For $p_1, p_2 : \mathcal{P}\,\Sigma$, and $P_1 : \Sigma \leftrightarrow \Gamma_1$, $P_2 : \Sigma \leftrightarrow \Gamma_2$,*

$$\{p_1\}; [P_1] \ \otimes \ \{p_2\}; [P_2] \quad = \quad \{p_1 \wedge p_2\}; [P_1 \otimes P_2].$$

Theorem 16 *For $f_1 : \Sigma \to \Gamma_1$, $f_2 : \Sigma \to \Gamma_2$,* $\quad \langle f_1 \rangle \otimes \langle f_2 \rangle \quad = \quad \langle f_1 \otimes f_2 \rangle.$

4.3 Properties of the Derived Product Operator

Theorem 12 shows that the derived product operator is also a special case of the fusion operator:

Theorem 17 *For* $S_1 : \Sigma_1 \longmapsto \Gamma_1$, $S_2 : \Sigma_2 \longmapsto \Gamma_2$,

$$S_1 \times S_2 \quad = \quad \langle \pi_1 \rangle ; S_1 ; [\varpi_1^{-1}] \;\odot\; \langle \pi_2 \rangle ; S_2 ; [\varpi_2^{-1}].$$

Here, the command $\langle \pi_1 \rangle$ starts in a state $(\sigma_1, \sigma_2) : \Sigma_1 \times \Sigma_2$ and ends simply in the state σ_1. Given $S_1 : \Sigma_1 \longmapsto \Gamma_1$, the predicate transformer $\langle \pi_1 \rangle ; S_1 ; [\varpi_1^{-1}] : \Sigma_1 \times \Sigma_2 \longmapsto \Gamma_1 \times \Gamma_2$ is a 'lifted' version of S_1 that behaves as S_1 on the first components of the before and after state, and behaves nondeterministically on the second components. Similarly for $\langle \pi_2 \rangle ; S_1 ; [\varpi_1^{-1}]$.

The derived product operator preserves refinement and it is easy to show that it combines with specification statements and deterministic updates in a similar manner to the product operator.

5 Applications

In Section 4.1 (c.f. Theorem 10), we saw that the fusion operator corresponds to a form of conjunction operator for specifications. Here we look more closely at applications of the other operators.

5.1 Simultaneous Assignment

As mentioned in Section 2, variables may be modelled as cartesian components of the state space. We take the view that the predicate transformer $S_1 : \Sigma_1 \times \Sigma_2 \longmapsto \Sigma_1$ models a statement that reads from program variables x and y, and assigns a value to program variable x, while $S_2 : \Sigma_1 \times \Sigma_2 \longmapsto \Sigma_2$ reads from the same variables and assigns a value to y. The product $S_1 \otimes S_2$ has type $\Delta(\Sigma_1 \times \Sigma_2)$ and models simultaneous assignment to x and y. For example, we have

$$\begin{array}{rcl} x := f(x,y) \;\otimes\; y := g(x,y) & = & \langle \lambda x, y \cdot f(x,y) \rangle \;\otimes\; \langle \lambda x, y \cdot g(x,y) \rangle \\ & = & \qquad \{\text{Theorem 16}\} \\ & & \langle \lambda x, y \cdot f(x,y), g(x,y) \rangle \\ & = & x, y := f(x,y), g(x,y). \end{array}$$

If S_1 and S_2 read from distinct state components, i.e., S_1 has type $\Delta(\Sigma_1)$ and S_2 has type $\Delta(\Sigma_2)$, then their simultaneous execution is modelled by the derived product $S_1 \times S_2$. With both product operators we know that if either assignment is nonterminating, then the combined assignment is nonterminating, e.g., $S_1 \otimes \mathit{abort} \;=\; \mathit{abort}$.

The *action system* formalism of Back & Kurki-Suonio [4] uses predicate transformers to model parallel programs. An action system consists of an initialisation predicate transformer and a set of action predicate transformers. Execution of an action system proceeds by firstly executing the initialisation, then, repeatedly, executing an enabled action (an action A is enabled when $gd\ A$ holds). Two action systems are composed in parallel by composing their initialisations such that they are executed simultaneously and forming the union of their actions. Conventionally, the initialisations are demonic updates $[I_1]$, $[I_2]$, and their composition is simply $[I_1 \wedge I_2]$. The product operator provides a way of composing more general initialisations achieving the same effect.

In [8], the actions of an action system are given labels and a correspondence with Hoare's CSP is established. A version of parallel composition of action systems is described in which commonly labelled actions from the respective action systems are composed such that they are executed simultaneously. This composition is defined by the properties that it should satisfy. One such property is that the composition should only be enabled when both actions are enabled. The product operator almost satisfies all the required properties: it fails if one of the actions can abort. However, by guarding the product of the actions as follows, we get an operator that satisfies the required properties:

$$A_1 \| A_2 \quad \widehat{=} \quad [(gd\ A_1) \times (gd\ A_2)]\ ;\ (A_1 \times A_2).$$

5.2 Superposition

Consider the following two predicate transformers:

$$S_1 : \Delta(\Sigma_1), \qquad S_2 : \Sigma_1 \times \Sigma_2 \longmapsto \Sigma_2.$$

S_1 only operates on Σ_1, while S_2 reads from $\Sigma_1 \times \Sigma_2$ and writes to Σ_2. Superposition of S_2 on to S_1 allows S_2 to add extra program variables which it writes to, as well as allowing S_2 to read the program variables of S_1. Such an operation may be defined as a special case of the product operator:

Definition 9 $\quad S_1 \,\mathbf{sup}\, S_2 \;\widehat{=}\; \langle\pi_1\rangle; S_1 \;\otimes\; S_2.$

Here, $\langle\pi_1\rangle$ reads from $\Sigma_1 \times \Sigma_2$ though it discards the Σ_2 component. For example, we have:

$$x := f(x) \quad \mathbf{sup} \quad y := g(x, y) \quad = \quad x, y := f(x), g(x, y).$$

Superposition refinement of action systems is described in [5], where superposition on individual actions is described in terms of sequential composition. Our superposition operator could be used instead.

5.3 Modification

Consider the following two predicate transformers:

$$S_1 : \Delta(\Sigma_1 \times \Sigma_2), \qquad S_2 : \Sigma_1 \times \Sigma_2 \longmapsto \Sigma_2.$$

S_1 operates on $\Sigma_1 \times \Sigma_2$, while S_2 reads from $\Sigma_1 \times \Sigma_2$ and writes to Σ_2. For example, we could have

$$S_1 \;=\; x, y := f(x, y), g(x, y), \qquad S_2 \;=\; y := h(x, y).$$

We wish to modify S_1 with S_2 so that S_1 writes to Σ_1 as before, but the value written to Σ_2 is determined instead by S_2. Again, such an operation may be defined as a special case of the product operator:

Definition 10 $\quad S_1 \,\mathbf{mod}\, S_2 \;\widehat{=}\; S_1; \langle\pi_1\rangle \;\otimes\; S_2.$

Here, $\langle\pi_1\rangle$ discards the Σ_2 component written to by S_1 preserving only the Σ_1 component. For example,

$$x, y := f(x, y), g(x, y) \quad \mathbf{mod} \quad y := h(x, y) \quad = \quad x, y := f(x, y), h(x, y).$$

5.4 Rearranging and Extending State Components

Sometimes when combining statements, their state components may not match in the required way. For example, the statements $S_1 : \Delta(\Sigma_1 \times \Sigma_2 \times \Sigma_3)$ and

$S_2 : \Delta(\Sigma_2 \times \Sigma_3 \times \Sigma_1)$ may not be combined directly as $S_1; S_2$, (even if Σ_1, Σ_2, and Σ_3 are the same sets, constructing $S_1; S_2$ directly will result in the state components being mismatched). Instead, the state components may be rearranged using an update statement before combining the statements:

$$S_1; \langle \delta \rangle; S_2,$$

where $\delta(\sigma_1, \sigma_2, \sigma_3) = (\sigma_2, \sigma_3, \sigma_1)$.

The same technique can be used to match statements modulo associativity. For example, although

$$(S_1 \times S_2) \times S_3 \quad \neq \quad S_1 \times (S_2 \times S_3),$$

we do have (let $\kappa((\sigma_1, \sigma_2), \sigma_2) \quad = \quad (\sigma_1, (\sigma_2, \sigma_2))$):

$$(S_1 \times S_2) \times S_3 \quad = \quad \langle \kappa \rangle; (S_1 \times (S_2 \times S_3)); \langle \kappa^{-1} \rangle.$$

Sometimes the state components of a statement need to be extended in order to combine it with another statement. For example, suppose we wish to combine $S_1 : \Delta(\Sigma_1 \times \Sigma_2)$ and $S_2 : \Delta(\Sigma_2)$. Here, S_2 is independent of Σ_1, so when combining it with S_1 we would most likely expect it to leave the Σ_1 component unchanged. To achieve this, we first combine S_2 with *skip*:

$$S_1; (skip_{\Sigma_1} \times S_2).$$

Extension of the state space is required, for example, when composing action systems: the state space of all actions must be extended to the state space of the combined action system.

5.5 Shared Variable Assignment

The fusion operator may be viewed as a way of combining statements such as that they both assign a value that they agree on to the same state component. In the case that two statements have only partially overlapping state components, then they need to be extended firstly with the *havoc* statement, before being combined using the fusion operator. For example, $S_1 : \Delta(\Sigma_1 \times \Sigma_2)$ and $S_2 : \Delta(\Sigma_2 \times \Sigma_3)$ may be combined as follows:

$$(S_1 \times havoc_{\Sigma_3}) \odot (havoc_{\Sigma_1} \times S_2).$$

Since *havoc* acts as a unit for the fusion operator, the effect of this statement on the Σ_1 component will be determined purely by S_1, and the effect on the Σ_3 component will be determined by Σ_3. The effect on the Σ_2 component will be the intersection of the nondeterministic possibilities allowed by S_1 and S_2. If that intersection is empty, then, provided S_1 and S_2 terminate, the combination behaves as magic.

Note that the above operands of the fusion operator, do not match exactly: the first is of type $\Delta((\Sigma_1 \times \Sigma_2) \times \Sigma_3)$, while the second is of type $\Delta(\Sigma_1 \times (\Sigma_2 \times \Sigma_3))$. However, these can easily be made to match using the function $(\lambda((\sigma_1, \sigma_2), \sigma_3) \cdot (\sigma_1, (\sigma_2, \sigma_3)))$ as described in Subsection 5.4.

5.6 Late Binding of Procedures

We model a procedure by simply associating a statement with a procedure name. Calling a procedure then involves executing the statement associated with the procedure name. The following example shows the concrete syntax used to describe procedures:

$$\begin{array}{c}\textbf{proc } Increment\\ \quad\textbf{var } x : \textbf{Int}\\ \quad x := x + 1\end{array}$$

The concept of *late binding* of procedures is important in object-oriented programming. Late binding means that the effect of executing a procedure depends on the value of the state on which it is to be executed. For example, Utting & Robinson [23] model late binding in the refinement calculus by regarding a procedure as being a function from values to statements:

$$proc \ : \ \Sigma \rightarrow \Delta(\Sigma).$$

The effect of executing procedure $proc$ in state σ is then determined by the statement $(proc\ \sigma)$. This is sometimes referred to as *instance-centered* late binding [12].[2]

A simpler notion of late binding uses types rather than values to determine which statement is selected when a procedure is called (so-called *class-centered* late binding [12]). Here we show how the summation operator may be used to model this form of late binding.

Firstly, the following operator lifts predicate transformers in either base type of a summation to the summation type:

Definition 11 *Let $S_1 : \Sigma_1 \longmapsto \Gamma_1$ and $S_2 : \Sigma_2 \longmapsto \Gamma_2$ be predicate transformers. Then,*

$$S_1^+ \mathrel{\widehat{=}} S_1 + abort \qquad\qquad S_2^+ \mathrel{\widehat{=}} abort + S_2.$$

Now, given predicate transformers

$$\begin{array}{ll} S_1 : \Sigma_1 \longmapsto \Gamma_1 & \qquad S_2 : \Sigma_2 \longmapsto \Gamma_2 \\ T_1 : \Gamma_1 \longmapsto \Lambda_1 & \qquad T_2 : \Gamma_2 \longmapsto \Lambda_2, \end{array}$$

we have the following result:

Theorem 18

$$S_1^+;(T_1+T_2) \ = \ (S_1;T_1)^+ \qquad\qquad S_2^+;(T_1+T_2) \ = \ (S_2;T_2)^+.$$

That is, the effect of (T_1+T_2) depends on whether S_1 or S_2 is executed beforehand.

Consider the following two procedures:

$$\begin{array}{ll} \textbf{proc } Invert_1 & \qquad \textbf{proc } Invert_2 \\ \quad \textbf{var } x : \textbf{Int} & \qquad \quad \textbf{var } x : \textbf{Bool} \\ \quad x := -x & \qquad \quad x := \neg\, x \end{array}$$

Let $Invert \mathrel{\widehat{=}} Invert_1 + Invert_2$. Then

$$\begin{array}{rcl} (x := 10)^+; Invert & = & (x := 10)^+; (Invert_1 + Invert_2) \\ & = & (x := 10; Invert_1)^+ \end{array}$$

[2] Sometimes the terms "late binding" and "dynamic binding" are used interchangeably in the literature. We prefer to reserve the term "dynamic binding" for the case where the procedure associated with an instance/class may change during execution. Such an effect may be achieved, for example, by using stored procedures as provided in Oberon [19] where procedures are themselves values. A predicate transformer model of stored procedures may be found in [21].

$$= \quad (x := -10)^+,$$

while

$$\begin{aligned}(x := true)^+; Invert &= (x := true)^+; (Invert_1 + Invert_2)\\ &= (x := true; Invert_2)^+\\ &= (x := false)^+.\end{aligned}$$

Thus the effect of *Invert* depends on whether an integer or boolean value has been assigned to x beforehand.

5.7 Inheritance

Consider the following object-oriented definition of a procedure:

proc R
inherits S
extended by E
modified by M

Here, R inherits the behaviour of S, extends the behaviour of S by E and modifies the behaviour of S by M.

Suppose S, E and M have the following types:

$$\begin{aligned}&S : \Delta(\Sigma_1 \times \Sigma_2)\\ &E : \Sigma_1 \times \Sigma_2 \times \Sigma_3 \longmapsto \Sigma_3\\ &M : \Sigma_1 \times \Sigma_2 \times \Sigma_3 \longmapsto \Sigma_2.\end{aligned}$$

That is, E introduces a new component of type Σ_3, while M modifies how S behaves on the Σ_2 component. One way to define R would be as follows:

$$R \quad \widehat{=} \quad (S \,\mathbf{sup}\, E) \,\mathbf{mod}\, M.$$

However, in object-oriented programming, as well as performing the extended and modified behaviour on the extended type of R, we would expect R to behave as S on values of the type of S. But some of the behaviour of S has been discarded in $(S \,\mathbf{sup}\, E) \,\mathbf{mod}\, M$. So instead we define R as follows:

$$R \quad \widehat{=} \quad S + ((S \,\mathbf{sup}\, E) \,\mathbf{mod}\, M).$$

Now, R uses late binding when choosing whether to perform the inherited behaviour or the extended/modified behaviour.

This form of inheritance preserves refinement:

THEOREM 19

$$\begin{aligned}&S \leq S' \ \wedge \ M \leq M' \ \wedge \ E \leq E' \ \Rightarrow\\ &S + ((S \,\mathbf{sup}\, E) \,\mathbf{mod}\, M) \ \leq \ S' + ((S' \,\mathbf{sup}\, E') \,\mathbf{mod}\, M').\end{aligned}$$

It should also be possible to define multiple inheritance in a similar way, with the fusion operator being used in the case where two inherited procedures act on the same state component.

6 Conclusions

We have investigated summation and product operators for predicate transformers and shown that they satisfy a rich set of algebraic laws. These are mostly distributivity laws and refinement preservation laws that experience with other operators in the refinement calculus has shown to be useful.

We examined possible applications of these operators such as simultaneous execution, conjunction of specifications, late binding and inheritance. Simultaneous execution is important, for example, for various forms of action system parallel composition. The fusion operator provides a refinement-preserving way of composing requirements (specifications) which could also be viewed as a form of multiple inheritance. Late binding and inheritance are important features of object-oriented programming languages, and although our approach is limited in that we only model class-centered late binding, we believe it to be sufficiently powerful and find the rich set of associated laws encouraging.

Acknowledgements

The work reported here was carried out within the IRENE-project supported by the Academy of Finland. We are grateful to David Naumann and Xu Qiwen for useful comments.

References

[1] J.R. Abrial. *B-Technology Technical Overview.* B-Core(UK) Ltd., 1993.

[2] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.* Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[3] R.J.R. Back and M.J. Butler. *Exploring Summation and Product Operators in the Refinement Calculus.* Åbo Akademi reports on Computer Science and Mathematics, Series A, No.152, 1994.

[4] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[5] R.J.R. Back and K. Sere. Superposition refinement of parallel algorithms. In K.A. Parker and G.A. Rose, editors, *FORTE'91.* North–Holland, 1992.

[6] R.J.R. Back and J. von Wright. Refinement concepts formalised in higher order logic. *Formal Aspects of Computing*, 5:247–272, 1990.

[7] R.J.R. Back and J. von Wright. *Refinement Calculus.* Book in preparation, Åbo Akademi, 1994.

[8] M.J. Butler. *Refinement and Decomposition of Value-Passing Action Systems.* In E. Best, editor, *CONCUR'93*, volume LNCS 715. Springer–Verlag, 1993.

[9] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[10] E.W. Dijkstra. *From Predicate Transformers to Predicates.* Manuscript EWD821, April 1982.

[11] P.H.B. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[12] J. Gutknecht. *Object-Oriented Programming with Oberon.* Lecture Notes from Eastern Finland Universities International Summer School on Novel Computing, Lapeenranta University of Technology, Finland, 1994.

[13] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In *European Symposium on Programming*, volume LNCS 213. Springer–Verlag, 1986.

[14] C.B. Jones. *Systematic Software Development using VDM – Second Edition*. Prentice–Hall, 1990.

[15] C.E. Martin. *Preordered Categories and Predicate Transformers*. D.Phil. Thesis, Programming Research Group, Oxford University, 1991.

[16] C.C. Morgan. *Programming from Specifications*. Prentice–Hall, 1990.

[17] C.C. Morgan. The cuppest capjunctive capping, and Galois. In A.W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*. Prentice–Hall, 1994.

[18] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comp. Prog.*, 9(3):298–306, 1987.

[19] H.P. Mossenboeck. *Object-Oriented Programming in Oberon-2*. Springer-Verlag, 1994.

[20] D.A. Naumann. *Two-Categories and Program Structure: Data Types, Refinement Calculi, and Predicate Transformers*. Ph.D. Thesis, University of Texas at Austin, 1992.

[21] D.A. Naumann. *On the Essence of Oberon*. Southwestern University, Georgetown, Texas, 1993.

[22] J.M. Spivey. *The Z Notation - A Reference Manual*. Prentice–Hall, 1989.

[23] B.M. Utting and K. Robinson. Modular reasoning in an object-oriented refinement calculus. In *Mathematics of Program Construction, 1992*, volume LNCS 669. Springer–Verlag, 1993.

[24] J. von Wright. The lattice of data refinement. *Acta Informatica*, 31(2):105–135, 1994.

[25] N. Ward. Adding specification constructs to the refinement calculus. In *FME'93*, volume LNCS 670. Springer–Verlag, 1993.

Programming with Data Dependencies: a Comparison of Two Approaches

Vytautas Čyras
Faculty of Mathematics
Vilnius University
Naugarduko 24
LT-2600 Vilnius, Lithuania
Vytautas.Cyras@maf.vu.lt

Magne Haveraaen
Department of Informatics
University of Bergen
Høyteknologisenteret
N-5020 Bergen, Norway
Magne.Haveraaen@ii.uib.no

Abstract

We present two methods for expressing computations based on recurrence relations and discuss their relative merits. One method, the *structural blanks* approach, is built on top of traditional programming languages like Fortran or Pascal. The other method, the *constructive recursive* approach, is based on recursive relations over graphs.

1 Introduction

Recurrence relations play an important role in the formulation of many problems, such as differential equations from mathematical physics or dynamic programming from operations research. A recurrence may be viewed as composed from a structural part, an expression part, and an initialization part. In this paper we discuss two programming methods for solving generalized recurrences: the *structural blanks* approach and the *constructive recursive* approach. Both approaches rely on a separation between the structural part describing the *data dependency pattern* of the recurrence, the definition of the expressions to be computed as the *computational aspect*, and the initialization and definition of outputs.

The structural blanks approach was first presented by Čyras [Čyras83], then by Greshnev, Lyubimskii and Čyras [GLČ85], and by Čyras [Čyras86]. The approach was inspired by problems in mathematical physics, where driver routines for sets of mutually dependent recurrences were needed. One of the aims was to develop a framework where the correctness of the driver routine need only be proved once, while the scheduling it defines may be reused for different problems with the same basic dependency structures. The solution to this was to define driver routines (S-modules) based on the structure of the recurrence, and requiring that the routines (F-modules) for solving each recurrence included a declaration of its dependency structure. The driver routine could then be applied to all recurrences with a compatible structure. Compatibility was shown by exhibiting an injective function from the S-module to the global arrays underlying the F-modules.

The constructive recursive approach was developed by Haveraaen [Hav93] from a programming perspective. Traditionally a recursive program implicitly defines an exponentially growing graph. In the constructive recursive case the graph is explicitly defined by the user, allowing linear solution time for recurrences, even in the higher order cases. The information in the graph also allows the translation of the recursive program to recursion free loop programs.

This paper is structured as follows: first we discuss some basic properties of recurrences and present the two programming methods. Then we show their similarities and difference by comparing them on some examples. Finally we discuss the relative merits of the approaches.

2 Basic concepts

2.1 Generalized recurrences

An order k linearly dependent recurrence r with the natural numbers as *index domain* is defined by a set of equations

$$\begin{aligned} r_n &= \phi(r_{n-1}, r_{n-2}, \ldots, r_{n-k}), \\ r_{k-1} &= \varepsilon_{k-1}, \\ &\cdots \\ r_0 &= \varepsilon_0 \end{aligned} \tag{1}$$

where the indices are natural numbers, ϕ is a k-ary expression not referring to r, except via its arguments, and the ε_i, representing initial values, are expressions not referring to r. The choice of $r_0, \ldots, r_{k-1}$ as initial elements is arbitrary, and one may even envision cases where the recurrence is infinite or where different recurrences only differ in the choice of initial elements. The archetypical second order recurrence relation is the Fibonacci function $F_n = F_{n-1} + F_{n-2}$, where $F_1 = 1$ and $F_0 = 0$, defining the sequence $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$ The dependency pattern of this function is illustrated in figure 1.

A straight forward method to compute the n values of the recurrence (1) is to start with an array R[0:n], initializing R[i] with ε_i for $0 \leq i \leq k-1$, and then computing

$$\mathsf{R[j]} := \phi(\mathsf{R[j-1]}, \mathsf{R[j-2]}, \ldots, \mathsf{R[j-k]}) \tag{2}$$

for $j = k, k+1, \ldots, n$. R[j] will then contain r_j for $0 \leq j \leq n$. Other result sets may also be defined, and have to be mirrored in the declaration and use of the array R. More efficient computation techniques exist for special cases, e.g., if the expression ϕ is linear, the recurrence may be reformulated as a matrix exponentiation problem. Such techniques will not be discussed further in this paper.

Recurrences may be generalized to arbitrary index domains. Given a sufficient set of initial values $\varepsilon_{i_1,\ldots,i_m}$, the m-dimensional order k *general recurrence* has the form

$$r_{n_1,\ldots,n_m} = \phi(r_{\delta_1(n_1,\ldots,n_m)}, \ldots, r_{\delta_k(n_1,\ldots,n_m)}) \tag{3}$$

where the m-ary functions δ_i, each returning an m-tuple of indices, have to be well founded with respect to the set of initial values. This scheme is more powerful than that of conventional recurrences, and algorithms such as the Fast Fourier Transform belong to the class of general recurrences. Since the δ_i have a more complex relationship than the linear dependency in (1), *it is impossible to give a general algorithm for computing* $r_{n_1,\ldots,n_m}$. Moreover, finding such an algorithm for a given set of δ_i, even if they are affine, may be difficult. But the structure of the algorithm to compute the recurrence is dependent only on the δ_i, the *data dependency pattern* of the recurrence, and is independent of the actual ϕ, the *computational aspect* of the recurrence.

Sometimes we will be working with a set of recurrences, all mutually dependent on each other. A *set of mutually dependent recurrences* is a set of ℓ recurrences $r^1, r^2, \ldots, r^\ell$, the recurrence r^i being of dimensionality m_i and order k_i, of the form

$$\begin{array}{rcl} r^1_{n_1,\ldots,n_{m_1}} & = & \phi_1(r^{i_{1,1}}_{\delta_{1,1}(n_1,\ldots,n_{m_1})},\ldots,r^{i_{1,k_1}}_{\delta_{1,k_1}(n_1,\ldots,n_{m_1})}) \\ r^2_{n_1,\ldots,n_{m_2}} & = & \phi_2(r^{i_{2,1}}_{\delta_{2,1}(n_1,\ldots,n_{m_2})},\ldots,r^{i_{2,k_2}}_{\delta_{2,k_2}(n_1,\ldots,n_{m_2})}) \\ & \vdots & \\ r^\ell_{n_1,\ldots,n_{m_\ell}} & = & \phi_\ell(r^{i_{\ell,1}}_{\delta_{\ell,1}(n_1,\ldots,n_{m_\ell})},\ldots,r^{i_{\ell,k_\ell}}_{\delta_{\ell,k_\ell}(n_1,\ldots,n_{m_\ell})}) \end{array} \tag{4}$$

and suitable initial values. Here $i_{j,q} \in \{1, 2, \ldots, \ell\}$, and $\delta_{j,q}$ is an m_j-ary function returning an $k_{i_{j,q}}$-tuple of indices. The mutually dependent recurrences correspond to *course of value recursion* in the terminology of Tucker and Zucker [TZ88]. A mutually dependent recurrence may be reformulated as a general recurrence.

2.2 Structural Blanks

The *structural blanks* (*SB*) approach ([Čyras83, Čyras84, GLČ85, Čyras86]) was developed to express solutions to mutually dependent recurrences as reusable program components defining loops over arrays. The problem of synthesizing the right sequence of array element updates to compute a set of mutually dependent recurrences was formulated by E. Z. Lyubimskii as early as in 1958 (published in [Lyubimskii60]), and later on investigated by I. B. Zadykhailo [Zadykhailo63]. The organization of computations for linear recurrences over multidimensional arrays was studied by R. Karp et al ([KMW67]) independently of the earlier research. The presentation here is a simplification of [HČ95], so the notation and definitions differ from the older papers. We will use a mixed Fortran and Pascal notation when presenting SB examples.

The SB approach, based on the mutually dependent recurrence (4), distinguishes between *structural components* (*S-modules*) and *functional components* (*F-modules*). Each module contains a data dependency part and a procedure part. The S-module describes the data dependencies, the set of initial elements and the set of output elements of a mutually dependent recurrence, and its *S-procedure* defines a driver algorithm for recurrences with this dependency structure.

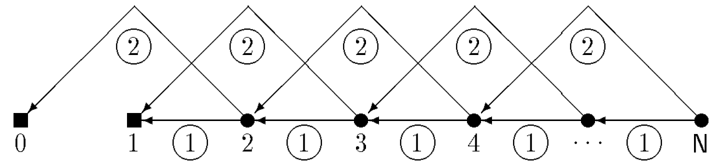


Figure 1: Data dependency graph of a *second order* recurrence, such as the Fibonacci function. The numbers in circles label the two arcs from a node, and the nodes are enumerated by the plain numbers underneath them.

An *F-procedure* defines the algorithm to compute one step of one recurrence expression r^j of (4), and the containing F-module describes the data dependencies of this step. An S-module is *applied* to a collection of F-modules by matching the dependencies of the F-modules with those of the S-module, if necessary using a substitution Ξ on the S-module, producing a new F-module containing an algorithm to compute the full recurrence. The algorithm for the full computation is defined in the new F-procedure, formed by combining the old F-procedures with the S-procedure after the substitutions Ξ have been performed.

In the case of an order k linear recurrence (1) the structural module would capture the computational idea of (2) by

```
S-module LDEP ( Fproc Φ(integer); k, N : integer ) ==
      formal x : array[*]
   internal-template (var q: integer; Φ(q) == (x[t], t=q-k..q-1) ⇝ x[q])
   external-template x[t], t=0..k-1 ⇝ x[t], t=k..N
   procedure                                                        (5)
      var q: integer;
      for q := k to N do
         call Φ(q)
end
```

This is to be interpreted as: given a 1-dimensional (declaration of Φ) order k recurrence over the array x (declared in the internal template), the S-module defines a procedure that will invoke Φ to compute all elements x[k],. . .,x[N] given that x[0],. . . ,x[k-1] are defined (external template). The set of array elements to the left of the "⇝" in the external template is the *set of initial elements*, and the set to the right is the *set of output elements*. The parameters to the formal procedure Φ range over the index domain of the recurrence. Note that the parameters to the S-module must be taken as parameters to the S-procedure when it is looked upon as a separate entity. The formal array x is concidered part of the environment for the argument "**Fproc** Φ", and the S-module needs no type nor size information since it is only used in the templates to declare the dependencies. The data dependency graph of the computation organized by the S-module LDEP when $k = 2$ is shown in figure 1.

The functional module giving the computational aspect of each step of the Fibonacci function is

```
F-module FIBSTEP ( q : integer ) ==
        global X : array[*] of integer
    template X[q-1], X[q-2] ⇝ X[q]
    procedure
        X[q] := X[q-1] + X[q-2]
end
```

This is to be interpreted as: FIBSTEP contains a 1-dimensional (from the declared index domain parameter q) second order recurrence expression over the array X (as declared in the template). The array size is not declared in the F-module, but rather in the program unit that uses the modules. The base type is declared since the operations on the elements require this knowledge.

To be able to use FIBSTEP to compute the Fibonacci function, we need a driver procedure that will schedule the computations of its F-procedure. Driver procedures are part of the S-modules, and are applicable if the internal template of the S-module matches the template of the F-module.

Definition 2.1 An S-module's parameter Φ with internal template given by the pattern (**var** $\vec{q}$: integer; $\Phi(\mathrm{e}(\vec{q}))$ == $\vec{x} \rightsquigarrow \vec{y}$) *matches* an F-module $F(\vec{p}$: integer) with template $\vec{X} \rightsquigarrow \vec{Y}$) if the set $\vec{x}$ of array elements is the same as the set $\vec{X}$ and the set $\vec{y}$ is the same as the set $\vec{Y}$ when the expressions $\mathbf{e}(\vec{q})$ replace the parameters $\vec{p}$ in the template of F. □

In the case of FIBSTEP, we will have to bind the parameter k of the S-module LDEP to 2, and replace the formal array x with the global array X. Then the two templates both describe that the q'th element of 1-dimensional array X is computed from the q-1'th and the q-2'th elements of X. Given the substitution Ξ =[k↦2; x↦X] we can put the modules together to form the F-module FIB = LDEP$|_\Xi$(FIBSTEP). There are three kinds substitution rules, of which the first two are used in the example above.

- The *binding substitution* that replaces an argument of the S-module by a constant, i.e., it removes k from the parameter list, and replaces all occurrences of k in the body of the S-module by 2.
- The *array domain substitution* that embeds the formal array x and its index domain into the global array X and its index domain, in this case a simple renaming was all that was needed.
- The *formal F-module domain substitution* that changes the formal F-module and its parameters in all calls from the S-module, allowing the embedding of the formal index domain in a higher dimensional domain and other manipulations of the index domain.

Unfolding the application above we get

```
F-module FIB( N : integer ) ==
        global X : array[*] of integer
    template X[0], X[1] ⇝ X[2..N]
    procedure
        var q: integer;
        for q := 2 to N do
            X[q] := X[q-1] + X[q-2]
end
```

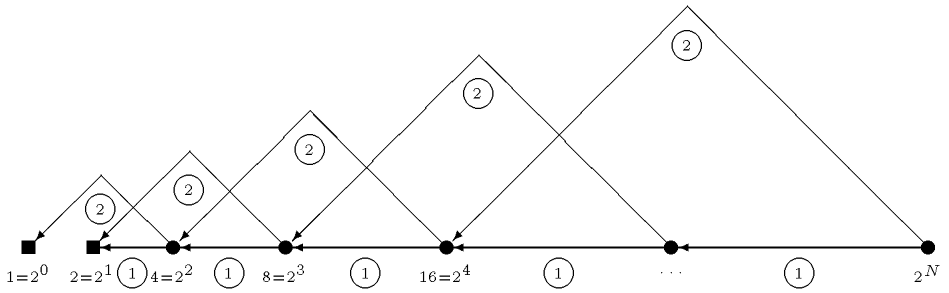


Figure 2: Data dependency graph of the recurrence g defined in (6).

The resulting F-module is to be interpreted as: FIB is a 1-dimensional second order recurrence expression over the array X (declared in the template).

Definition 2.2 An array domain substitution is *safe* if it does not merge any distinct array elements of the S-module's output set. □

Proposition 2.3 *An array domain substitution is safe if it defines an injective function from formal array elements to global array elements.*

This becomes a proof obligation for every application of an S-module to an F-module.

The development methodology of the structural blanks approach can be formulated as three steps. In the **first step** a domain expert, e.g. a physicist, formulates the problem as a set of mutually dependent recurrence equations, which is encoded as a collection of F-modules and global array declarations, comprising the *computational model* for the problem. The sizes of the arrays may be dependent on the size of input data, the number of time-steps the computations are to run for, or may be forced by numerical properties of the discretization technique involved in formulating the recurrence equations.

As an example take the problem that can be formulated as the real valued general recurrence equation

$$\begin{array}{rcl} g(2^{i+2}) & = & \gamma\ (\ g(2^{i+2}/2),\ g(2^{i+2}/4)\ ), \\ g(2^1) & = & \varepsilon_1, \\ g(2^0) & = & \varepsilon_0, \end{array} \tag{6}$$

where we need to know $g(2^i)$ for $i = 0, 1, 2, \ldots, \mathsf{N}$. This may be formulated as the declaration of "Y : **array**[1..2**N] **of** real" together with the F-module

F-module GSTEP (i : integer) ==
    global Y : **array**[*] **of** real
  template Y[2**i], Y[2**(i+1)] $\leadsto$ Y[2**(i+2)]
  procedure Y[2**(i+2)] := γ(Y[2**(i+1)], Y[2**i])
end

The data dependency graph of this recurrence is shown in figure 2.

The **second step** is to devise a driver routine for the computational model, i.e., to find an appropriate S-module. For this purpose there may be a library of S-modules, and one of them may be adapted to the problem at hand by using a substitution.

In the case of the recurrence (6) we may reuse the S-module LDEP with the substitution Ξ =[k$\mapsto$2; x[j]$\mapsto$Y[2**j]; Φ(i)$\mapsto\Phi'$(i-2)] involving all three substitution rules. Here the array domain substitution does the exponential expansion, while the formal F-module domain substitution, shifts the formal F-module parameters two positions in order to adjust the starting point of the loop in the S-procedure to the indices used by the F-module. This yields the application G = LDEP$|_{\Xi}$(GSTEP):

```
F-module G ( N : integer ) ==
      global Y : array[*] of real
   template Y[1], Y[2] ⇝ (Y[2**t], t=2..N)
   procedure
      var q: integer;
      for q := 2 to N do
         call GSTEP(q-2)
end
```

The **third step** is to show that an application is correct by proving that the substitutions are safe. In this case it is obvious since the function on the array index domain, $j \mapsto 2^j$, is injective.

2.3 Constructive recursion

Constructive recursion (*CR*) is an extension of the primitive recursive and μ-recursive schemes over the integers to recursion schemes over any graph structure. It was developed by Haveraaen and early ideas are presented in [Hav90], [Hav93]. Although constructive recursion was developed independently from the work of Tucker and Zucker in [TZ88], CR can be seen as a generalization of the theory of computations on arbitrary algebraic structures developed there. Tucker and Zucker restrict their recursive structures to discrete recursion schemes, while CR may be defined for denser graphs. The development of constructive recursion was inspired by ideas in the programming language Crystal [CCL91], which in turn was inspired by systolic algorithms.

The CR approach defines general recurrences (3) by distinguishing between the definition of the *data dependency graph*, the *recursive relation* defining a value on each node of the data dependency graph, and the specification of initial values at *input nodes* and *output nodes*. A data dependency graph is directed and is usually defined algebraically, which allows for the concise expression of repetitive structures. We will refer to the graph as a *data dependency algebra* (*dda*). The recursive functions define a relation between the value of a node and its neighbors in the direction of the arcs of the dda, so that given a certain set of nodes with initial values, it will be possible to compute the values of some set of dda nodes. The actual outputs will be a subset of the set of nodes whose values may be computed.

Definition 2.4 A *labeled graph* is a set of nodes, node labels and an injective function from nodes to node labels, a set of arcs, a set of labels, and four functions on the arcs: source returning the source node of the arc, target returning the target node of the arc, slabel and tlabel each returning a label for each arc. A *locally uniquely labeled graph* is a labeled graph where slabel is injective on the set of arcs with the same source and tlabel is injective on the set of arcs with the same target. □

Definition 2.5 A *data dependency algebra* is a set P of points and a set D of directions, and on the set $P \times D$ functions canR (*can receive*) and canS (*can send*) with boolean results, partial functions r (*receive from*) and s (*send to*) with results in P, and dirR (*direction received*) and dirS (*direction sent*) with results in D, so that the following equations are satisfied

$$\begin{array}{lcl}
\mathsf{canR(p,d)} & \Rightarrow & \mathsf{canS(r(p,d),dirS(p,d));} \\
\mathsf{canR(p,d)} & \Rightarrow & \mathsf{s(r(p,d),dirS(p,d)) = p;} \\
\mathsf{canR(p,d)} & \Rightarrow & \mathsf{dirR(r(p,d),dirS(p,d)) = d;} \\
\mathsf{canS(p,d)} & \Rightarrow & \mathsf{canR(s(p,d),dirR(p,d));} \\
\mathsf{canS(p,d)} & \Rightarrow & \mathsf{r(s(p,d),dirR(p,d)) = p;} \\
\mathsf{canS(p,d)} & \Rightarrow & \mathsf{dirS(s(p,d),dirR(p,d)) = d;}
\end{array}$$

□

Proposition 2.6 *Any data dependency algebra defines a locally uniquely labeled graph, and a locally uniquely labeled graph defines a data dependency algebra.*

In the case of an order k linear recurrence (1) we can define the recurrence data dependency algebra RDDA with P being the set of natural numbers and D the numbers 1 through k

$$\begin{array}{lcl}
\mathsf{canR(p,d)} & = & \mathsf{(k{<}{=}p)\ and\ (1{<}{=}d{<}{=}k);} \\
\mathsf{r(p,d)} & = & \mathsf{p\text{-}d;} \\
\mathsf{dirS(p,d)} & = & \mathsf{d;} \\
\mathsf{canS(p,d)} & = & \mathsf{(d{=}k)\ or\ ((k\text{-}1{<}{=}p)\ and\ (1{<}{=}d{<}{=}k));} \\
\mathsf{s(p,d)} & = & \mathsf{p{+}d;} \\
\mathsf{dirR(p,d)} & = & \mathsf{d;} \\
\mathsf{lpro(p)} & = & \mathsf{p;} \\
\mathsf{epro(p)} & = & \mathsf{2{*}{*}p;}
\end{array} \tag{7}$$

with the label of an arc being the number in D representing the distance between two nodes. The injective function lpro projects a node to a natural number label, while the injective function epro returns another set of labels for each node. The graph defined by these expressions when $k = 2$ is depicted in figure 1 using the lpro node labels and in figure 2 using the epro node labels.

A recurrence of the form (1) can easily be expressed as a constructive recursive relation. In the Fibonacci case we choose the graph nodes of the RDDA guided by the function lpro, and arrive at the following CR relation

$$\mathsf{canR(p,1)\ and\ canR(p,2)} \Rightarrow \mathsf{Fib(p) = Fib(r(p,1)) + Fib(r(p,2))}$$

This means that whenever there is an arc in direction 1 and an arc in direction 2 from a point p, the value at p is the sum of the values at r(p,1) and r(p,2).

The specification of initial values and the set of output values may take a form inspired by the structural blanks notation

$$\mathsf{Fib(0){=}0,\ Fib(1){=}1} \rightsquigarrow \mathsf{(Fib(t),\ t{=}2..N)}$$

We may now go ahead and compute the Fibonacci numbers by traversing the graph in the s-direction, from inputs 0 and 1, till we have computed all values in the output set.

Theorem 2.7 *A constructive recursive relation* f *defined over a dda may be translated into a recursion free loop which evaluates* f *at most once for each point.*

Proof Assume that each point of the graph only has the arcs the CR relation requires. We may, at each point p, establish a data structure S_p capable of storing the value from all arcs p can receive from, so that the value at p can be computed when S_p has stored the values from all its r-connected points. The algorithm is: create a storage structure M with all initial values v_n for the points n. As long as M is not empty and there still are non-computed output points, take a point p from M, and for all directions d where $\mathsf{canS}(p,d)$ is true, store v_p in S_q for $q = \mathsf{s}(p,d)$ at the arc corresponding to direction $\mathsf{dirR}(p,q)$, and if all data for S_q has arrived compute the value v_q and add it to M. If M is a FIFO type queue, the algorithm will generate the values of all points with exclusively finite paths to the initial values. □

The theorem guarantees that the information from the dda is enough to provide a wave-front computation of any constructively recursive relation. It does not claim anything about the order of evaluation, but this can be specified by providing the dda with a space-time structure. Such a space-time structure may define the communication structure of a parallel computer, allowing compilation of CR relations to parallel computers. See [Hav93] for more information.

We may formulate how to go from a general recurrence of the form (3) to a constructive recursive program as a 3-step approach. **Step 1** is to define the constructive recursive function based on the recurrence equation. Simply take a set D and uniquely label the occurrences of the recurrence variable on the right-hand side of the equation. An occurrence d is replaced by a recursive call with $\mathsf{r}(\mathsf{p},d)$ as argument. In **step 2** we define the dda based on the structure of the recurrence. The r and canR functions follow directly from the recurrence equations and step 1, although the form chosen may vary (see [Lun94] for details). Defining the rest of the dda-functions is more difficult and may require ingenuity if it is to be done efficiently. For affine dependencies and certain other special cases this can be automated. **Step 3** is to verify that the recursive relation, given the dda, defines the recurrence. This is achieved by exhibiting an injective *projection function* pro from the points P of the dda to the index domain of the recurrence (3), s.t. "$\delta_d(\mathsf{pro}(p)) = \mathsf{pro}(\mathsf{r}(p,d))$" whenever $\mathsf{canR}(p,d)$ for a point p and direction d. The initial values, *the inputs*, are then given for the points corresponding to base case of the recurrence, and the output values are the needed solutions.

Let us apply this programming technique on the recurrence defined in (6), using RDDA defined in (7). The relation and input/output sets are

$$\mathsf{canR}(\mathsf{p},1) \text{ and } \mathsf{canR}(\mathsf{p},2) \Rightarrow \mathsf{g}(\mathsf{p}) = \gamma(\mathsf{g}(\mathsf{r}(\mathsf{p},1)), \mathsf{g}(\mathsf{r}(\mathsf{p},2))$$

$$\mathsf{g}(0) = \varepsilon_0,\ \mathsf{g}(1) = \varepsilon_1 \rightsquigarrow \mathsf{g}(\mathsf{t}),\ \mathsf{t} = 2..\mathsf{N}$$

for $\mathsf{g} : P \rightarrow \mathsf{real}$. We use the epro function to label the points consistently with the domain used in (6), and show that "$\mathsf{epro}(\mathsf{p})/2^d = \mathsf{epro}(\mathsf{r}(\mathsf{p},d))$".

3 Comparing the approaches

The structural blanks and constructive recursive approaches are similar in that they break a recurrence equation in a structural part (templates of the S- and F-modules or a graph defined by a dda), a computational part (F-module and recursive relation), and a defined initial value set and output set (external template of S-module or separate input/output description).

Looking at how the input/output description is handled, we see that the CR approach is slightly more flexible. There we define the output set independently from the graph, so we may specify that only the N'th value, or some other subset of the computed values, is to be output. In the SB approach we will have to modify the driver routine to reflect and take advantage of any change in the external template of an S-module. But, as in the CR case, any subset of the computed values may of course be specified as the set of outputs.

In the SB case the data array is declared by the user, separately from the modules, while in the CR approach the points to be computed are given implicitly by the dda graph. In the latter case we may provide the points with a space-time mapping, assigning the computations to explicit time-steps on the processors of a parallel computer. Thus data parallelism is inherent in the CR approach. The data distribution information needed in the SB case is dependent on the driver procedure of the S-module, but has to be declared by the user of the module independently of what S-module that is to be chosen, hence data parallelism is difficult to achieve in the SB approach.

When it comes to programmability the SR approach probably has the edge on the CR approach. SB is based on conventional languages such as Fortran, a notation the practitioner is familiar with. The practitioner is also familiar with the task of adding *pragmas* and additional information to make a program run faster or exploit properties of specific architectures. The CR approach, although clearly related to the mathematical structure of the recurrence, introduces concepts not used in traditional programming, and has a less familiar notation. In the next subsections we show the two approaches on some more complex examples to illustrate the practical similarities and differences.

3.1 Example: mutually dependent equations

Given a set of mutually dependent recurrences

$$\begin{array}{rcl} x_q & = & \varphi_1(x_{q-1}, y_q), \\ y_q & = & \varphi_2(y_{q-1}, z_q), \\ z_q & = & \varphi_3(z_{q-1}, y_{q-1}), \\ \multicolumn{3}{c}{x_0 = \xi_0, \quad y_0 = \upsilon_0, \quad z_0 = \zeta_0.} \end{array} \tag{8}$$

The dependency pattern is shown in figure 3. Expressing this in the structural blanks approach we will define three arrays X[0..N], Y[0..N] and Z[0..N], where N is a constant defining how much we want computed, and for each of the equations define an F-module (only the first of these are shown)

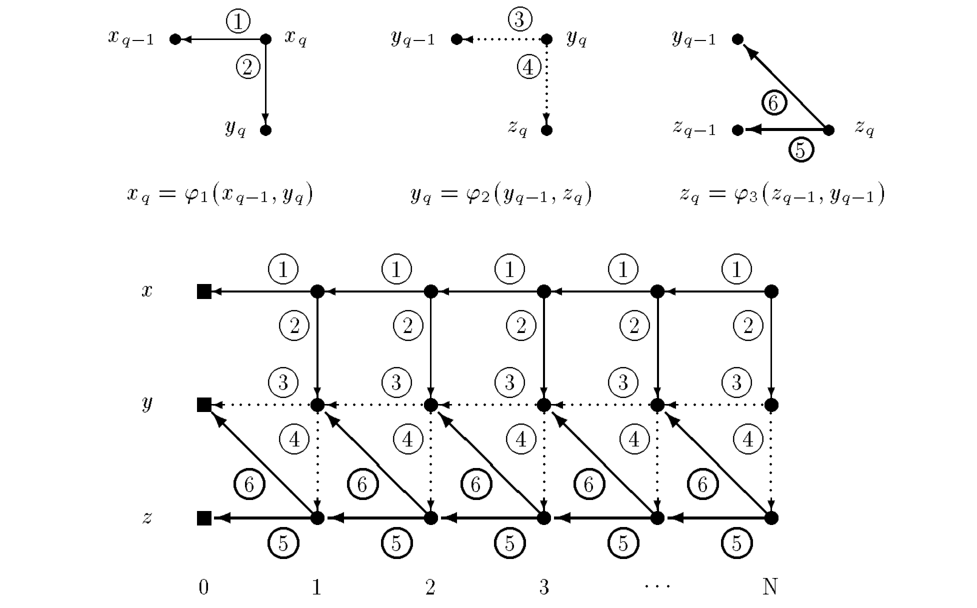


Figure 3: The dependencies of the mutually dependent recurrence in (8) shown separately (on the top) and combined into one large graph (bottom).

```
F-module FX ( q : integer ) ==
      global X, Y, Z : array[*] of <type>
   template X[q-1], Y[q] ⇝ X[q]
   procedure X[q] := φ1(X[q-1], Y[q])
end
```

We suppose that elements X[0], Y[0], Z[0] are initialized. Elements X[1],. . .,X[N], Y[1],. . .,Y[N], Z[1],. . .,Z[N] are to be computed.

The S-module that will structure the computation of the array elements must describe a computation that starting from the initial elements in X[0], Y[0] and Z[0] will compute the next element based on already computed values. Denoting the patterns Φ1, Φ2 and Φ3, we see that this can be achieved by repeating the sequence Φ3(q);Φ2(q);Φ1(q)

```
S-module S3( Φ1, Φ2, Φ3 : Fproc(integer); N : integer) ==
      formal x, y, z : array[*]
   internal-template -- Three internal templates:
      (var q:integer; Φ1(q) == x[q-1], y[q]   ⇝ x[q])
      (var q:integer; Φ2(q) == y[q-1], z[q]   ⇝ y[q])
      (var q:integer; Φ3(q) == z[q-1], y[q-1] ⇝ z[q])                (9)
   external-template x[0], y[0], z[0] ⇝ x[1..N], y[1..N], z[1..N]
   procedure
      var q: integer;
      for q := 1 to N do
            begin call Φ3(q); call Φ2(q); call Φ1(q) end
end
```

Using the injective substitution Ξ=[x[q]↦X[q]; y[q]↦Y[q]; z[q]↦Z[q]], we see that S3 may readily be applied to FX, FY and FZ yielding FXYZ = $S3|_{\Xi}$(FX,FY,FZ).

In the constructive recursion approach the graph depicted in figure 3 may be defined by taking the set of points $P = \{1,2,3\} \times \mathcal{N}$, where $\mathcal{N}$ is the set of natural numbers, and the set of directions $D = \{1,2,3,4,5,6\}$. The functions required to define a dda are

```
canR((t,q),d)=(q>0 and(case t of 1: d in [1,2]; 2: d in [3,4]; 3: d in [5,6]));
canS((t,q),d)=(q=0 and(case t of 1: d in [1]; 2: d in [3,6]; 3: d in [5])) or
              (q>0 and(case t of 1: d in [1]; 2: d in [2,3,6]; 3: d in [4,5]));
r((t,q),d)    =(case d of 1: (1,q-1); 2: (2,q); 3: (2,q-1)
                          4: (3,q); 5: (3,q-1); 6: (2,q-1));                      (10)
s((t,q),d)    =(case d of 1: (1,q+1); 2: (1,q); 3: (2,q+1)
                          4: (2,q); 5: (3,q+1); 6: (3,q+1));
dirS((t,q),d) =d;
dirR((t,q),d) =d;
```

The functions needed to show the correspondence with the graph in figure 3 are

```
xpoint(t,q)=(t=1)
ypoint(t,q)=(t=2)
zpoint(t,q)=(t=3)                                                                  (11)
pro(t,q)    =q
```

Now pro is not injective by itself, but taken together with the information that we are on an xpoint, and so forth, we get an injective embedding that is preserved over the r-function. The function that defines the recurrence in (8) is

$$\text{xpoint(p)} \Rightarrow \text{xyz(p)} = \varphi_1(\text{ f(r(p,1)), f(r(p,2)) })$$
$$\text{ypoint(p)} \Rightarrow \text{xyz(p)} = \varphi_2(\text{ f(r(p,3)), f(r(p,4)) })$$
$$\text{zpoint(p)} \Rightarrow \text{xyz(p)} = \varphi_3(\text{ f(r(p,5)), f(r(p,6)) })$$

The definition of xyz is split into three cases, each corresponding to one equation of (8). Then "xyz(1,0)=ξ_0, xyz(2,0)=υ_0, xyz(3,0)=$\zeta_0 \rightsquigarrow$ xyz(t,p), t=1..3, p=1..N" is the input/output specification.

3.2 Example: embedding in higher-dimensional domain

To illustrate the flexibility of these approaches, we will define a diagonal computation in a 2-dimensional domain W (see figure 4). The computational aspects are defined by the F-modules

F-module F1 (i1, i2 : integer) ==
    global W : **array**[*,*] **of** <*type*>
  template W[i1-1,i2-1], W[i1+1,i2-1] $\rightsquigarrow$ W[i1,i2] (12)
  procedure W[i1,i2] := γ_1(W[i1-1,i2-1], W[i1+1,i2-1])
end
F-module F2 (i1, i2 : integer) ==
    global W : **array**[*,*] **of** <*type*>
  template W[i1-1,i2-1], W[i1-2,i2] $\rightsquigarrow$ W[i1,i2] (13)
  procedure W[i1,i2] := γ_2(W[i1-1,i2-1], W[i1-2,i2])
end

Where F1 is to be used twice in order to define the diagonal marked x. Due to the structural similarity with (8), we may use the S-module S3 (9) as the driver, giving the application FDIAG = S3$|_{\Xi}$((F1,F1,F2)). The substitution needed is

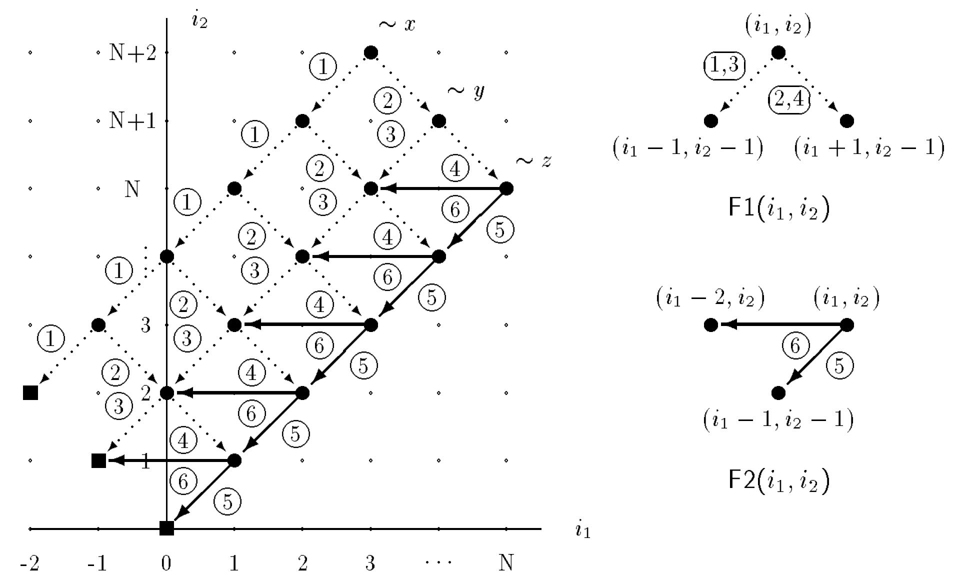


Figure 4: Three mutually dependent 2-dimensional order two recurrences as defined by F-modules F1 (12) (used twice) and F2 (13). The structural similarities with the recurrence (8) as depicted in figure 3 are marked.

Ξ = [x[q]↦W[q-2,q+2]; y[q]↦W[q-1,q+1]; z↦W[q,q];
  $\Phi 1(q) \mapsto \Phi 1'(q-2, q+2)$; $\Phi 2(q) \mapsto \Phi 2'(q-1, q+1)$; $\Phi 3(q) \mapsto \Phi 3'(q, q)$;]

Writing the result of the application out in full we get the F-module

F-module FDIAG (N : integer) ==
  global W : **array**[*] **of** <*type*>
 template W[-2,2], W[-1,1], W[0,0]
  ⇝ (W[t-2,t+2], t=1..N), (W[t-1,t+1], t=1..N), (W[t,t], t=1..N)
 procedure (14)
  var q: integer;
  for q := 1 **to** N **do**
   begin call F2(q,q); **call** F1(q-1,q+1); **call** F1(q-2,q+2) **end**
end

We now need to show that the substitution Ξ is injective, and this follows since each formal array is mapped to a different diagonal of W.

In the constructive recursive solution we need to define an injective projection on 10 to match the labelings of figure 4

wpro(t,q) = (q-3+t,q+3-t)

and the function on the graph corresponding to F1 and F2 (definitions (12) and (13)) is simply

canR(p,1) and canR(p,2) ⇒ f1f2(p) = γ_1(f1f2(r(p,1)), f1f2(r(p,2)))

canR(p,3) and canR(p,4) ⇒ f1f2(p) = γ_1(f1f2(r(p,3)), f1f2(r(p,4)))

canR(p,5) and canR(p,6) ⇒ f1f2(p) = γ_2(f1f2(r(p,5)), f1f2(r(p,6)))

Note how we do the cases on the canR functions since xpoint etc. are not defined in this view of the graph. The template of FDIAG will be the basis for the input/output specification.

4 Summary

We have presented two approaches, structural blanks and constructive recursion, and shown how they may be applied for the transcription of generalized recurrence relations to computer programs.

The structural blanks approach extends a traditional imperative programming language with constructs for defining explicitly the dependency pattern of a recurrence. The program to compute the recurrence is defined as a collection of global arrays and several program components: one for each equation of the recurrence (4), and a scheduler for the entire computation. These components may be reused, and especially the scheduler may be applied on many different recurrence relations. Since the notation used is based on well known programming languages, it should be fairly easy to start using it for a practitioner in a field where recurrences are used.

The constructive recursive approach is a functional programming language where the structure of the directed graph implicitly defined by a recursive expression is made explicit. In most functional languages, s.a. Haskell and Standard ML, the graph is a tree. Using memoization, nodes of this tree may be merged, but the graph can only be traversed in the direction that the functional expression defines it. In the CR approach the arcs can be traversed both in this and in the opposite direction. The latter traversal scheme translates into an efficient loop program to compute the recurrence. The graph may be defined with an assignment of the nodes to the space-time of a parallel computer. The result is then a data parallel program distributed on the processors of the parallel computer. In para-functional programming [Hudak91] it is possible to schedule computations explicitly on a parallel computer, but the underlying graph will be a tree as given by the standard semantics of the programming language, preventing the efficiency obtained here.

Even though this presentation has focused on recurrence relations, the programming techniques presented are not restricted to recurrences in the classical form (1). Most problems with a repetitive or recursive structure can be expressed with the notation presented. Future work includes demonstrating these techniques on a broader set of examples, and building tools to facilitate the practical use of these approaches.

References

[CCL91] M. Chen, Y. Choo and J. Li: "Crystal: Theory and Pragmatics of Generating Efficient Parallel Code". In Boleslaw K. Szymanski (ed.): *Parallel Functional Languages and Compilers*, pp. 255-308.

[Čyras83] V. Čyras: "Loop synthesis on structural data in program packages". In *Computer Programming* [in Russian], Vilnius, Inst. of Math. and Cybern. of the Academy of Sciences of Lithuania, 1983, No. 7, pp. 27-50.

[Čyras84] V. Čyras. *Loop Program Synthesis on Multidimensional Data Structures* [in Russian], M. V. Lomonosov Moscow State University, Faculty of Numerical Mathematics and Cybernetics, Thesis (avtoreferat kandidata fiziko-matematicheskix nauk), 1984, 16 p.

[GLČ85] S. N. Greshnev, E. Z. Lyubimskii and V. A. Chiras: "Synthesis of programs on data structures". In *Programming and Computer Software*, Vol. 11, No. 5, pp. 282-291, 1986. Translated from *Programmirovanie* [in Russian], No. 5, pp. 44-54, 1985.

[Čyras86] V. Čyras: *Loop program synthesis using array traversing modules.* Forschungsberichte Künstliche Intelligenz, Report FKI-93-88, August 1988, Technische Universität München, Institut für Informatik, 25 p. Also in Meldal & Haveraaen: *Proceedings of the 4th Nordic Workshop on Program Correctness*, Reports in Informatics no. 78, Department of Informatics, UiB, 1993, pp. 97-109. Translated from *Lietuvos Matematikos Rinkinys*, Vol. 26, No. 4, 1986, pp. 636-655.

[Hav90] Magne Haveraaen: "Distributing Programs on Different Parallel Architectures". *Proceedings of the 1990 International Conference on Parallel Processing*, ICPP, vol. II Software, 1990, pp. 288-289.

[Hav93] Magne Haveraaen: "How to Create Parallel Programs without Knowing it". In Meldal & Haveraaen: *Proceedings of the 4th Nordic Workshop on Program Correctness*, Reports in Informatics no. 78, Department of Informatics, UiB, 1993, pp. 165-176.

[HČ95] Magne Haveraaen & Vytautas Čyras: *The Structural Blanks approach to solving Generalized Recurrences*. Reports in Informatics no. 100, Department of Informatics, UiB, 1995.

[Hudak91] Paul Hudak: "Para-Functional Programming in Haskell". In Boleslaw K. Szymanski (ed.): *Parallel Functional Languages and Compilers*, pp. 159-196.

[KMW67] R. M. Karp, R. E. Miller, S. Winograd: "The organization of computations for uniform recurrence equations". *Journal of the ACM*, Vol. 14, No. 3, July 1967, pp. 563-590.

[Lun94] Roar Lundevik: *Oversettelse fra Standard ML til Safir.* Thesis, the Department of informatics, University of Bergen, June 1994, 104 p.

[Lyubimskii60] E. Z. Lyubimskii: "Issues of automatic programming". In *Vestnik Akademii Nauk SSSR* [in Russian], No. 8, 1960, pp. 47-55.

[TZ88] J.V. Tucker & J.I. Zucker: *Program Correctness over Abstract Data Types, with Error-State Semantics*, North-Holland, Amsterdam, 1988.

[Zadykhailo63] I. B. Zadykhailo: "The organisation of a cyclical computing process using a parametric representation of special form". In *U.S.S.R. Computational Mathematics and Mathematical Physics*, Vol. 3, No. 2, pp. 442-468, 1963. Translated from *Zhurnal vychislitel'noi matematiki i matematicheskoi fiziki* [in Russian], Vol. 3, No. 2, pp. 337-357, 1963.

A Calculus of Timed Refinement*

EXTENDED ABSTRACT

Kārlis Čerāns
Institute of Mathematics and Computer Science,
University of Latvia
E-mail: `karlis@mii.lu.lv`

1 Introduction

This paper presents CTR - a process algebraic framework for loose specification of time quantity sensitive operational behaviour of reactive systems.

The necessity and role of quantitative time information in reactive (open) system descriptions is nowadays quite appreciated[1], and there are already quite many specification formalisms, which alongside with system structural and/or behaviour event ordering specification permits to specify quantitative time information characterizing the valid behaviours (see e.g. [MF76, CC89, AD90, Wan91, GMMP89] as a few examples of such formalisms).

Quite naturally, many of the existing "timed" specification formalisms do allow in certain natural sense "loose" specifications of time information (it would be not that natural to determine at early system design stage exactly what are the time quantities between specific events going to be[2]), at least [MF76, AD90, Che92, Dan92, CGL93] to be mentioned.

The calculus CTR[3] presented in this paper is also designed to follow this tradition. Specifically, its aim is to provide a framework with mathematically precise and semantically justified notion of timed specification refinement (when one specification can be termed to be "more precise" than the other), concentrating on external behaviour specifications and branching time semantics. The model closest to CTR among those already present in the literature certainly is that of Timed Modal Specifications (TMS) introduced in [CGL93, CGL94]. We discuss in Section 5 in detail, what subset of CTR corresponds to a slight semantic reformulation[4] of TMS.

---

*The full version to be published as a technical report LU-IMCS-TRCS-94-??, Institute of Mathematics and Computer Science, University of Latvia, Riga 1994

[1] Even if the observer of a system is not directly concerned about the actual time quantities between externally observable events, it could be relevant for him/her to put some timing information in the specifications of the system components in order to ensure the correct *qualitative* ordering of the whole system output events.

[2] It may appear impossible even on the level of implementations due to various unpredictable execution environment fluctuations.

[3] A Calculus of Timed Refinement.

[4] Achieving, moreover, more intuitive properties of the specification refinement.

In essence, CTR is obtained by adding to the Milner's CCS [Mil89] the contruct of time interval prefix $[a, b].P$, where a, b are nonnegative real numbers, $0 \leq a \leq b$, possibly $b = \infty$. Such a specification is intended to describe all processes which first delay for some quantity $d \in [a, b]$, and then behave accordingly to some behaviour prescribed by P. (CTR can be seen as a generalization of Timed CCS due to Wang Yi in its version of [Wan90], where only delay prefixes of the form $[d, d].P$ were considered and denoted by $\epsilon(d).P$).

We define an operational semantics for CTR specification terms as a labelled transition system (each transition label denotes either some control action (externally visible or invisible), or it specifies some amount of time passage). Based on this semantics the relations of specification refinement $\trianglelefteq$ and equivalence $\dot{\approx}$ are defined in a bisimulation - like fashion[5].

The CTR specification equivalence and refinement relations are shown to obey the algebraic properties of equivalence/preorder and substitutivity (congruence/pre-congruence). On Timed CCS terms $\trianglelefteq$ induces a so-called divergence refinement relation, which is a slightly stronger property than the weak bisimulation $\approx$ of [Wan90]. $\dot{\approx}$ and $\trianglelefteq$ are also *decidable* for finite control specifications[6] (these include, for instance, networks of regular specifications).

Pragmatically, the CTR refinement relation $\trianglelefteq$ is based on the intuition of resolving underspecification of timing information. So, $[e, f].S \trianglelefteq [d, g].T$ whenever $d \leq e \leq f \leq g$ and $S \trianglelefteq T$ [7] (the tighter the delay interval is, the finer is the specification). On the extreme, the term $[0, \infty].S$ covers any specification of the form $[d, e].S$, together with the "empty" specification nil which never exhibits any observable behaviour but waiting. In fact, the term $[0, \infty].S$ would specify just the *possibility* of the system to exhibit the behaviour prescribed by S after some (possibly none) initial delay[8]. This is the only way of achieving in CTR looseness of "non-timed" transition behaviour of specifications.

On the other hand, the CTR refinement relation $\trianglelefteq$ admits abstraction from internal transitions of specifications and is focusing just on the *externally observable* behaviour. As an example, we have

$$(a.[d, e].\overline{b} \mid b.[f, g].\overline{c})\backslash b \trianglelefteq a.[d + f, e + g].\overline{c}.$$

Here on the right hand side we have a specification of all media which after recieving a message on the channel a wait for some time amount $\delta \in [d+f, e+g]$, after what a message on the channel c is delivered. On the left hand side we have decomposed such a general message transmission specification into more detailed one, where the transmission is done in two stages - first from the channel a to the internal channel b, then from b to c. For each stage the looseness of the delay specification is maintained. Observing the substitutivity property of $\trianglelefteq$, one can during further stages of the design make further refinements of

[5] As largest relations in the set of all specification terms, satisfying certain local "closeness" properties.

[6] To be precise, the decidability results hold for specifications whose initial delay bounds are given by integers (or at least rational numbers). In practice this is not a real restriction.

[7] This includes the case when $S = T$, due to the reflexivity of $\trianglelefteq$.

[8] In the notation of TMS the term $[0, \infty].a.S$ (observe the action label a necessary here) is written as $a_\diamond.S$ and is read semantically as "a transition a leading to S *may* be enabled".

the components responsible for both transmission stages[9] still ensuring that the resulting system[10] will be a refinement of the initial general specification $a.[d+f, e+g].\overline{c}$.

This example illustrates how CTR is conceptually suitable for formal top-down system development by stepwise refinement. One can start with a general and very loose specificaton which captures only a few essential features of the system being designed. The stepwise refinement process allows, in a formal manner, to incorporate into the specification additional design decisions, both of structural and timing nature, each refinement step possibly being done locally in some component of the system[11].

One should note, however, that there are a lot of different notions of specification refinement (or, specification/implementation relations, for that matter), using rather different mathematical models, one could look at, say, [BW94, Lam91, Dil89]. The refinement aspect inherent in CTR is that of time information strenghening in external communication ability specifications, and no claim is being made about any "superiority" of this refinement aspect to others.

In the rest of the paper, after introducing the CTR model in Section 2, in Section 3 we consider the definition and properties of the refinement relation. Sections 4 and 5 are devoted to study of some semantic variations of CTR.

2 CTR: Syntax and Semantics

We start with defining the syntax of CTR specification terms.

Let $\Lambda = \Delta \cup \overline{\Delta}$ with $\overline{\overline{\alpha}} = \alpha$ for all $\alpha \in \Lambda$ be a set of *external actions*, ranged over by α, β, and let τ be a distinct symbol representing the internal actions. $\mathcal{A}ct = \Lambda \cup \{\tau\}$ is a set of all actions, ranged over by a, b.

Let $\mathcal{V}ar$ be a finite set of process variables, ranged over by X. Then the set of *regular* CTR expressions over $\mathcal{V}ar$ is defined by the following grammar:

$$E ::= nil \mid X \mid [c,e].E \mid]c,e].E \mid a.E \mid E+E,$$

where $c \in \mathbf{R}^{+0}, e \in \mathbf{R}^{+0} \cup \{\infty\}, c \leq e$. Here besides the standard process algebraic constructs nil, action prefix $a.E$, choice $E+E$ and variable (see [Mil89]) we have introduced two kinds of delay interval prefix (left-closed and left-open). We let also $\epsilon(d).E$ to be another notation for $[d,d].E$.

We assume furthermore that for every $X \in \mathcal{V}ar$ we are given a regular CTR expression E_X (over $\mathcal{V}ar$) that contains variable occurrences only guarded either by action or some delay interval prefix[12]. We let $\mathcal{E} = \{X \stackrel{def}{=} E_X | X \in \mathcal{V}ar\}$ and call it a system of variable defining equations.

Given such a system $\mathcal{E}$ we call any regular CTR expresion over $\mathcal{V}ar$ a *regular CTR specification* over $\mathcal{V}ar$ and $\mathcal{E}$ (possibly not mentioning $\mathcal{V}ar$ and $\mathcal{E}$, if they are clear from the context).

[9] As a trivial possibility, one could tighten the message delivery delay intervals.

[10] Any system $(S|T)\backslash b$, where $S \trianglelefteq a.[d,e].\overline{b}$ and $T \trianglelefteq b.[f,g].\overline{c}$.

[11] Notice that the decidability of the refinement relation provides with (at least principal) possibility for automatic formal verification of the validity of every performed refinement step.

[12] In other words, all variable occurrences in E_X are allowed only within subterms of the form $a.E$, $[c,e].E$ or $]c,e].E$. Moreover, a delay prefix counts as a valid guard, only if $c > 0$.

Let $\mathcal{G}(\mathcal{V}ar, \mathcal{E})$ be a set of all regular CTR specifications over $\mathcal{V}ar$ and $\mathcal{E}$, ranged over by G. We define the set $\mathcal{S} = \mathcal{S}(\mathcal{V}ar, \mathcal{E})$ of CTR networks (CTR specifications), over $\mathcal{V}ar$ and $\mathcal{E}$, by the grammar:

$$S ::= G \mid S|S \mid S\backslash A \mid S[f],$$

where $A \subseteq \Delta$ and $f : \Delta \to \Delta$ is a bijection, extended to $\Lambda \to \Lambda$ by $f(\overline{\alpha})) = \overline{f(\alpha)}$ and $f(\tau) = \tau$. We let S and T to range over the set of CTR networks.

We define for CTR specifications an operational semantics. Following the design of a standard timed process calculi TCCS [Wan90], we base it on the transition relations $S \xrightarrow{a} T$, meaning that S can execute a control transition $a \in \mathcal{A}ct$ and afterwards exhibit a behaviour prescribed by T, and $S \xrightarrow{\epsilon(d)} T$ meaning that S can become T just by delaying for d time units ($d \in \mathbf{R}^{>0}$).

The new point is in defining the semantics for delay interval prefix. Our choice is to admit the ability of $[c,e].S$ to exhibit the behaviour prescribed by S after any $d \in [c,e]$ time units, the actual value of d to be chosen nondeterministically every time this construct is executed[13]. However, to facilitate the compositionality of the CTR framework, we are to define also what $[c,e].S$ is going to become (can become) after waiting for some smaller time amount (e.g., $d < c$). At least three possible variants initially come into mind, depending on actually *when* during the delay the nondeterminism is being resolved. These variants correspond to (i) choosing in the beginning (early choice), (ii) choosing gradually during the execution (inline choice), and (iii) choosing when to leave (late choice). We shall work out in most detail (iii). (ii) is considered in Section 4 (showing also its relation to the late choice semantics), while (i) can be argued to be not suitable as a basis of timed refinement framework[14].

So, the execution of $[c,e].S$ in time begins with a deterministic and synchronous decreasing of both its higher and lower time bounds: $[c,e].S \xrightarrow{\epsilon(d)} [c-d, e-d].S$ for $d \leq c$. When $[0,e].S$ is reached, the choice comes in. We still let $[0,e].S \xrightarrow{\epsilon(d)} [0, e-d].S$ for $d \leq e$, however, now we want the specification to be able to choose to exhibit the behaviour prescribed by S immedilately. The design decision taken in CTR is to introduce a special new kind of internal transition $\xrightarrow{i}$ whose effect is to "leave the delay prefix": $[0,e].S \xrightarrow{i} S$. This construction materializes the principle of letting the specification itself to make all time nondeterministic choice without any influence from the environment.

Formally, we require $i \notin \mathcal{A}ct$ and let $\mathcal{A}ct^+ = \mathcal{A}ct \cup \{i\}$ to be ranged over by a^+. The whole operational semantics of CTR is given in Table 1 and Table 2 (we assume $f(i) = i$ for any relabelling function f). $x \dot{-} y$ means $max\{x - y, 0\}$ and ⟦ stands for any kind of left end of the interval (] or [). In the delay rules for $\xrightarrow{\epsilon(d)}$ we require $d \in \mathbf{R}^{>0}$. ν stands for either a^+, or $\epsilon(d)$.

For parallel composition, the sorts $Sort(d, S) \subseteq \Lambda$ in the side condition of the delay rule are computed in a standard way, as follows:

---

[13]This would amount to having $[c,e].S \xrightarrow{\epsilon(d)} S$ for all $d \in [c,e]$. As we shall see soon, a proper description of this property in CTR would be $[c,e].S \xrightarrow{\epsilon(d)}_I S$.

[14]In a sense, it requires the choice in the model to be done before the choice is performed in the real modelled system. We discuss this issue more precisely in Section 3.

$$\frac{--}{nil \xrightarrow{\epsilon(d)} nil} \quad \frac{--}{a.S \xrightarrow{a} S} \quad \frac{--}{\alpha.S \xrightarrow{\epsilon(d)} \alpha.S} \quad \frac{S \xrightarrow{\nu} S'}{X \xrightarrow{\nu} S'}[X \stackrel{def}{=} S] \in \mathcal{E}$$

$$\frac{S \xrightarrow{a} S'}{S+T \xrightarrow{a} S'} \quad \frac{T \xrightarrow{a} T'}{S+T \xrightarrow{a} T'} \quad \frac{S \xrightarrow{i} S'}{S+T \xrightarrow{i} S'+T} \quad \frac{T \xrightarrow{i} T'}{S+T \xrightarrow{i} S+T'}$$

$$\frac{S \xrightarrow{\epsilon(d)} S' \;\; T \xrightarrow{\epsilon(d)} T'}{S+T \xrightarrow{\epsilon(d)} S'+T'} \quad \frac{S \xrightarrow{a^+} S'}{S|T \xrightarrow{a^+} S'|T} \quad \frac{T \xrightarrow{a^+} T'}{S|T \xrightarrow{a^+} S|T'} \quad \frac{S \xrightarrow{\alpha} S' \;\; T \xrightarrow{\overline{\alpha}} T'}{S|T \xrightarrow{\tau} S'|T'}$$

$$\frac{S \xrightarrow{\epsilon(d)} S' \;\; T \xrightarrow{\epsilon(d)} T'}{S|T \xrightarrow{\epsilon(d)} S'|T'}[Sort(d,S) \cap \overline{Sort(d,T)} = \emptyset] \quad \frac{S \xrightarrow{a^+} S'}{S \backslash A \xrightarrow{a^+} S' \backslash A}[a, \overline{a} \notin A]$$

$$\frac{S \xrightarrow{\epsilon(d)} S'}{S \backslash A \xrightarrow{\epsilon(d)} S' \backslash A} \quad \frac{S \xrightarrow{a^+} S'}{S[f] \xrightarrow{f(a^+)} S'[f]} \quad \frac{S \xrightarrow{\epsilon(d)} S'}{S[f] \xrightarrow{\epsilon(d)} S'[f]}$$

Table 1: Transition rules for standard process algebraic constructions

$$\frac{-- \quad [d \leq e]}{[c,e].S \xrightarrow{\epsilon(d)} [c \dot{-} d, e-d].S} \quad \frac{--}{[0,e].S \xrightarrow{i} S} \quad \frac{S \xrightarrow{a^+} S'}{[0,0].S \xrightarrow{a^+} S'} \quad \frac{S \xrightarrow{\epsilon(d)} S'}{]c,e].S \xrightarrow{\epsilon(d+e)} S'}$$

$$\frac{-- \quad [d \leq c]}{]c,e].S \xrightarrow{\epsilon(d)}]c-d, e-d].S} \quad \frac{-- \quad [c < d \leq e]}{]c,e].S \xrightarrow{\epsilon(d)} [0, e-d].S}$$

Table 2: Transition rules for delay prefix

- $Sort(d, \alpha.S) = \{\alpha\}, \qquad Sort(d, \tau.S) = Sort(d, nil) = \emptyset,$
- $Sort(d, [c,e].S) = Sort(d \dot{-} e, S),$
- $Sort(d, X) = Sort(d, S)$, where $[X \stackrel{def}{=} S] \in \mathcal{E}$
- $Sort(d, S+T) = Sort(d, S|T) = Sort(d,S) \cup Sort(d,T)$
- $Sort(d, S[f]) = f(Sort(d,S)$ and $Sort(d, S\backslash A) = Sort(d,A) \setminus (A \cup \overline{A})$.

Following the design of Timed CCS, we have incorporated in our framework the principle of *maximal progress*, saying that an enabled internal τ transition has to be executed before the time progresses further[15].

In the semantics one should observe that unlike the $\xrightarrow{\tau}$ transitions, $\xrightarrow{i}$ does not resolve the nondeterministic choice $S+T$ (as do not the delay transitions). It is to be observed also that a specification of the form $[0,0].S$ does not need

[15]There are no delay transitions for the $\tau.S$ terms, also the above side condition with sorts in parallel composition delay rule is just for this reason - a time can pass only until the point where a synchronization between componets is enabled. Note that this does not apply to $\xrightarrow{i}$.

to have any $\xrightarrow{i}$ transitions in order to "leave" the delay prefix. Intuitively this reflects the idea that $\xrightarrow{i}$ corresponds to an active choice, but in case of $[0,0].S$ no (timed) choice has been left.

Using the notation $\epsilon(d).S$ for $[d,d].S$ we can interpret all Timed CCS (network) terms of [Wan90] as CTR terms. It can be seen that this embedding preserves also the TCCS operational semantics (modulo ignoring $\xrightarrow{i}$ transitions which for TCCS terms can be essentially only of the form $S \xrightarrow{i} S$).

3 Refinement and Equivalence

In this section the CTR specification refinement and equivalence relations are defined on the basis of their operational behaviour. We recall also that the refinement relation is going to embody the idea of resolving timed underspecification (what operationally appears as timed non-determinism).

For S being a refinement of T, $S \trianglelefteq T$, we first require that any transition sequence (behaviour) of S is to be matched by a corresponding transition sequence of T[16], the resulting states of these sequences also being in the relation $\trianglelefteq$. In a sense, a more general specification needs to admit more behaviours.

However, our notion of the refinement is not just that of simulation. We require also that certain behaviours of T, namely those not involving $\xrightarrow{i}$ transitions, and not involving also delays, should be matched appropriately on the side of S. Since the $\xrightarrow{i}$ and delay transitions are the only ones possibly resolving the timed nondeterminism ($\xrightarrow{i}$ is doing that in an "active" way, while some delay transitions do that in a "passive" way), this requirement means that all behaviours *not* resolving timed nondeterminism are to be maintained (modulo the abstraction from $\xrightarrow{\tau}$) during the refinement process.

Formally, let $S \xrightarrow{a}_\circ S'$ denote $S(\xrightarrow{i})^* \xrightarrow{a} (\xrightarrow{i})^* S'$, $S \overset{\varepsilon}{\Longrightarrow} S'$ denote $S \xrightarrow{\tau}{}^* S'$ and $S \overset{\alpha}{\Longrightarrow} S'$ stand for $S \overset{\varepsilon}{\Longrightarrow} \xrightarrow{\alpha} \overset{\varepsilon}{\Longrightarrow} S'$. Moreover, let $\overset{\varepsilon}{\Longrightarrow}_\circ = (\xrightarrow{\tau} \cup \xrightarrow{i})^*$ and $S \overset{\alpha}{\Longrightarrow}_\circ S'$ mean $S \overset{\varepsilon}{\Longrightarrow}_\circ \xrightarrow{\alpha} \overset{\varepsilon}{\Longrightarrow}_\circ S'$.

In what follows, we prepare for defining the matching of delay transitions. For $S \in \mathcal{S}$ and $d \in \mathbf{R}^{>0}$ we call any finite sequence $\langle S_0, d_0\rangle, \langle S_1, d_1\rangle, \ldots, \langle S_n, d_n\rangle$ such that $S \overset{\varepsilon}{\Longrightarrow}_\circ S_0$, $d_0 = 0$, $d_n = d$ and for every $i = 1, \ldots, n-1$ we have $S_i \xrightarrow{\epsilon(d_{i+1}-d_i)} S_i' \overset{\varepsilon}{\Longrightarrow}_\circ S_{i+1}$ a *base sequence* for an $\langle S, d\rangle$-trace. The $\langle S, d\rangle$-*trace* itself associated with this base sequence is defined to be the function $f : [0, d] \rightarrow \mathcal{S}$ such that $f(d_i + \delta) = S_i \xrightarrow{\epsilon(\delta)}$, where $i = 1, \ldots, n$ and $\delta \in [0, d_{i+1} - d_i[$ [17]. We denote by $S \overset{\epsilon(d)}{\Longrightarrow}_\circ$ the set of all $\langle S, d\rangle$-traces.

For $f \in S \overset{\epsilon(d)}{\Longrightarrow}_\circ$ and $g \in T \overset{\epsilon(d)}{\Longrightarrow}_\circ$, given a relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$, we say that $\langle f, g\rangle \in \mathcal{R}^t$ if and only if $\langle f(d'), g(d')\rangle \in \mathcal{R}$ for every $d' \in [0, d]$.

For $S \in \mathcal{S}$ we let $S \xrightarrow{\tau^\infty}_\circ$ to denote the fact of existence of infinite chain of $\xrightarrow{\tau}_\circ$ transitions starting at S: $S \xrightarrow{\tau}_\circ S_1 \xrightarrow{\tau}_\circ S_2 \xrightarrow{\tau}_\circ \cdots$.

[16] This matching is abstracting from the externally non-observable transitions $\xrightarrow{i}$ and $\xrightarrow{\tau}$.

[17] By $S \xrightarrow{\epsilon(\delta)}$ we have denoted the unique T such that $S \xrightarrow{\epsilon(\delta)} T$. In Section 4 it will be demonstrated, how a trace can be defined also in the absence of this unicity.

Let μ range over $\Lambda \cup \{\varepsilon\}$ (μ is either α, or ε).

Definition 3.1 *A relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is a refinement relation, if for any $\langle S, T\rangle \in \mathcal{R}$*

1. $S \stackrel{\mu}{\Longrightarrow}_\circ S'$ *implies* $\exists T'. T \stackrel{\mu}{\Longrightarrow}_\circ T'$ *so that* $\langle S', T'\rangle \in \mathcal{R}$;
2. $T \stackrel{\mu}{\Longrightarrow} T'$ *implies* $\exists S'. S \stackrel{\mu}{\Longrightarrow} S'$ *so that* $\langle S', T'\rangle \in \mathcal{R}$;
3. $f \in S \stackrel{\epsilon(d)}{\Longrightarrow}_\circ$ *implies existence of* $g \in T \stackrel{\epsilon(d)}{\Longrightarrow}_\circ$ *such that* $\langle f, g\rangle \in \mathcal{R}^t$;
4. $S \stackrel{\tau^\infty}{\longrightarrow}_\circ$ *implies* $T \stackrel{\tau^\infty}{\longrightarrow}_\circ$.

We say that S is a refinement of T, and write $S \trianglelefteq T$ whenever there exists a refinement relation $\mathcal{R}$ containing $\langle S, T\rangle$ (In other words, $\trianglelefteq$ is the largest refinement relation).

Observational equivalence is a refinement whose inverse is also a refinement[18].

A few further comments on the given definition are in order. The intuitive explanation of the refinement given in the beginning of the section motivates only its first three requirements (observing the specific way of defining a matching for timed transitions). Without including the fourth clause in the definition we would have a so-called *simple* refinement relation, which we denote by $\triangleleft$. It can be shown that $\triangleleft$ obeys the essential mathematical properties of transitivity and substitutivity. However, this relation is in a sense too large since $S \triangleleft T$ holds also in some cases, when it is semantically undesirable, for instance, when $T = a.nil$ and $S \stackrel{def}{=} a.nil + \tau.S$ [19].

The fourth clause in the refinement definition is pragmatically meant to forbid such examples (at the same time not forbidding too much). Though this clause is not of the form of "matching requirement", traditional for process algebraic equivalence and preorder relations, it is hoped that it would pragmatically fit well with the system refinement process.

To facilitate that we suggest a pragmatic requirement that a design in CTR should not admit "instantaneous" loops of internal transitions[20]. As loops of internal transitions are usually essential in the process algebraic system design, this requirement would mean just that any such loop possibly occurring in

[18] In this paper we do not further explore the notion of specification equivalence. However, it is clearly relevant for the theory.

[19] One most likely would not want the "time-blocking" specification S to be a refinement of a perfectly well defined specification willing to synchronize with its environment on a at the moment when the environment is ready to do so (and not necessarily "immedilately"). The simple refinement relation would be satisfactory, however, if one managed to exclude the possibilities of writing "time-blocking" specifications at all. This does not seem feasible while following the process algebraic specification style, and sticking to the maximal progress principle.

[20] In a modelling methodology, where we do take into account the actual time quantities, it might be admitted that some bounded amount of transitions are performed "instantaneously" (if that helps to simplify the model). However, for unbounded chains or loops of internal actions, it is no longer realistic that they could be executed within 0 units of time.

the specification should have its delay being strictly greater than 0 (a $]0,d].S$ construct executed in every round of the loop would be sufficient) [21].

Yet another variation of the refinement notion could be obtained by extending the set of "necessary" behaviours which, when present in a specification, should be matched appropriately also on the side of implementation[22]. A natural such extension is obtained by observing that *some* delay transitions of CTR specifications, namely those of $[c,e].S$ waiting for $d \leq c$ time units, are not resolving any timed non-determinism, and are in one or other form to be present also in all refinements of this specification.

However, a refinement relation which in case of T refined by S would require all "necessary" delay transitions of T to be matched by similar transitions of S (also done in the necessary mode, to ensure the transitivity of the refinement) appears to be too strong. To see that, look at the valid CTR refinement

$$[1,2].\tau.[1,2].\overline{c}.nil \trianglelefteq [2,4].\overline{c}.nil,$$

what is just a transformed and instantiated version of the transmission media refinement example considered in the introduction[23]. One can see that the right hand side is to delay for 2 time units in the "necessary mode". However, while it is clear that the left hand side also will necessarily delay for at least 2 time units before offering $\overline{c}$, there is no single chain of delay transitions of length 2 (possibly interleaved with $\xrightarrow{\tau}$) which could also be performed in the necessary mode, without any active or passive resolving of the time nondeterminism[24]. Thus the requirement of simple matching the "necessary" delay transitions would invalidate the above refinement instance, what could seriosly question the pragmatical applicability of the whole refinement relation[25] [26].

[21] As can be clearly seen from the definition of $\trianglelefteq$, this requirement has not been formally incorporated into the definition. All what has been said is, if the implementation (the stronger specification) admits such a loop for one or other reason, then also the more general specification has to have it admitted (it could be done as, for instance, in $S \stackrel{def}{=} a.nil + \tau.S$). Sometimes indeed some "time-blocking" effects could be useful to say the fact that a certain action should be immediate (ideally, of course, one should have other direct means of expressing that), however, that has nothing in common with the situation when a real chain (loop) of internal transitions is to be executed.

[22] For $\trianglelefteq$ these are just all behaviours not containing $\xrightarrow{i}$, and not containing delays.

[23] $(a.[d,e].\overline{b} \mid b.[f,g].\overline{c})\backslash b \trianglelefteq a.[d+f,e+g].\overline{c}$ with $d=f=1$ and $e=g=2$, here we consider the situation where the initial a has already been performed.

[24] Observe that already after 1 time unit of waiting the further performed delay of $[1,2]$ appears to be only in the possible mode due to the other existing alternative to take $\xrightarrow{i}$ at that moment.

[25] The framework of TMS in [CGL93, CGL94] is based on this idea of requiring the matching of "necessary" delay transitions in the direction from the specification to the implementation. Therefore in that framework only a reduced version of the considered example was possible (one does get a valid refinement instance when either $d=e$, or $f=g$, that is, when one of the components is deterministic).

[26] The same example $[1,2].\tau.[1,2].\overline{c}.nil \trianglelefteq [2,4].\overline{c}.nil$ can be used also to invalidate the "early choice" semantics for CTR, as discussed in Section 2 (provided we keep our schema of defining the refinement).

Indeed, as we want a delay transition of the implementation, say $\xrightarrow{\epsilon(0.1)}$, leading to $\epsilon(1.3).\tau.[1,2].\overline{c}.nil$, to be matched on the side of the specification, we need to determine at

We turn to the mathematical properties of $\trianglelefteq$. Due to the space constraints their proofs are mainly postponed to the full version.

Fact 3.2 *The relation $\trianglelefteq$ is reflexive and transitive.*

Lemma 3.3 *$\trianglelefteq$ is preserved by CTR parallel composition, restriction and relabelling operations.*

Proof: We prove that for any refinement relation $\mathcal{R}$ refinement relations are also $\mathcal{R}^+ = \{\langle S|T, P|T\rangle | \langle S, P\rangle \in \mathcal{R}\}$, $\mathcal{R}^+_{rs} = \{\langle S\backslash A, P\backslash A\rangle | \langle S, P\rangle \in \mathcal{R}\}$ and $\mathcal{R}^+_{rl} = \{\langle S[f], P[f]\rangle | \langle S, P\rangle \in \mathcal{R}\}$. The proof is a rather straightforward check, including the case of timed matching of parallel composition. $\square$

Lemma 3.4 *If $T \xrightarrow{i} S$, then $S \trianglelefteq T$.*

Proof: We show that the relation $\{\langle S, T\rangle | S = T \text{ or } T \xrightarrow{i} S\}$, is a refinement. The proof is a matter of combining definitions, given the following fact.

Fact 3.5 *If $T \xrightarrow{i} S$ and $T \xrightarrow{a} T'$, then either $S \xrightarrow{a} T'$, or there exists an S' such that $S \xrightarrow{a} S'$ and $T' \xrightarrow{i} S'$.*

Proof: By analyzing, whether $T \xrightarrow{i} S$ and $T \xrightarrow{a} T'$ are performed in distinct regular components of T, or in the same. $\square$

In what follows we show what relation the CTR refinement $\trianglelefteq$ induces in the set of time determinstic ("fully specified") process algebraic terms (those of Timed CCS). Let $\mathcal{P}$ denote the set of all TCCS (network) processes ($\mathcal{P} \subseteq \mathcal{S}$).

Following [Wan90], a relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a (weak) bisimulation, if for all $\nu \in \mathcal{A}ct \cup \{\epsilon(d) | d \in \mathbf{R}^{>0}\}$ and all $\langle P, Q\rangle \in \mathcal{R}$

- whenever $P \stackrel{\nu}{\Longrightarrow} P'$ then $\exists Q'.Q \stackrel{\nu}{\Longrightarrow} Q'$ so that $\langle P', Q'\rangle \in \mathcal{R}$ and
- whenever $Q \stackrel{\nu}{\Longrightarrow} Q'$ then $\exists P'.P \stackrel{\nu}{\Longrightarrow} P'$ so that $\langle P', Q'\rangle \in \mathcal{R}$.

Here $X \stackrel{\epsilon(d)}{\Longrightarrow} Y$ is defined to mean $X(\xrightarrow{\tau})^* \xrightarrow{\epsilon(d_1)} (\xrightarrow{\tau})^* \cdots \xrightarrow{\epsilon(d_k)} (\xrightarrow{\tau})^* Y$, where $d_1 + \cdots + d_k = d$. The largest TCCS weak bisimulation is denoted by $\approx$.

Fact 3.6 *For P and Q being TCCS processes, $P \trianglelefteq Q$ implies $P \approx Q$.*

To characterize the relation induced by $\trianglelefteq$ on TCCS terms precisely, let a relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ be called a *divergence refinement* iff

- $\mathcal{R}$ is a weak bisimulation, and
- for any $\langle P, Q\rangle \in \mathcal{R}$ whenever $P \xrightarrow{\tau^\infty}$, then also $Q \xrightarrow{\tau^\infty}$.

Let $\triangleleft_D$ be the largest divergence refinement for TCCS processes.

Theorem 1 *For P and Q being TCCS processes, $P \trianglelefteq Q$ if and only if $P \triangleleft_D Q$.*

---

this very moment, how long the actual delay of $[2, 4]$ is going to be. Doing that, we clearly can not keep the further flexibility inherent in the second implementation's $[1, 2]$ construct.

To prove the theorem we first give an equivalent characterization of $\approx$ and $\lhd_D$ using the form of "continuous matching" for delays[27]. Later on, a determinacy of $\xrightarrow{\epsilon(d)}$ transitions, together with the maximal progress and a few other CTR transition system properties are exploited.

4 Inline Choice Semantics

This section concerns defining alternative semantics for CTR terms following the idea of inline, or ongoing choice, where the "speed" of execution at every moment is contributing to when a specification is going to leave its delay interval prefix. Surprisingly enough, a refinement defined naturally on the basis of this really wildly nondeterministic transition system, coincides with the "standard" CTR refinement $\unlhd$. For the sake of simplicity of technical treatment we consider here only specifications in reduced CTR, where we disallow left-open delay interval prefixes $]c, e]$.

First we define the inline choice operational semantics as the transition relation $\longrightarrow_I$. For that we can take all the rules defining the "standard" semantics $\longrightarrow$ in Table 1 and Table 2 (with changing syntactically $\longrightarrow$ to $\longrightarrow_I$) except for two. First of those to be changed is the main rule for delay of the delay prefix, which is to read now as

$$[c, e].P \xrightarrow{\epsilon(d)} [c', e'].P \text{ whenever } max(0, c-d) \leq c' \leq e' \leq e - d.$$

Intuitively this rule means that during the delay of d time units the lower delay bound is reduced by no more than d, and the higher bound is reduced by not less than d. Any choice in between can be taken during the runtime.

To be able to define adequately the semantics of the delay rule of the parallel composition (we can not work just with the $Sort$ function, as in case of $\longrightarrow$), we have to refine a bit the whole definition of the relation $S \xrightarrow{\epsilon(d)}_I T$ by basing it not solely on the fact of *existence* of such transition, but rather on the explicit exhibition of the whole trace leading in d time units from S to T. Therefore we inductively define the proposition $f \in S \xrightarrow{\epsilon(d)}_I T$, where always $f : [0, d] \rightarrow \mathcal{S}$ satisfying $f(0) = S$ and $f(d) = T$. This definition is in full conformance to the already established clauses of definition of $S \xrightarrow{\epsilon(d)}_I T$. For the sake of brevity, we show only two clauses (the others are more straightforward):

- Whenever $max(0, c-d) \leq c' \leq e' \leq e-d$, we let $f \in [c, e].P \xrightarrow{\epsilon(d)}_I [c', e'].P$ iff $f(x) = [cc(x), ee(x)].P$ for all $x \in [0, d]$ and the functions cc and ee together with $cc(0) = c$, $ee(0) = e$, $cc(d) = c'$ and $ee(d) = e'$ for any $x, y \in [0, d]$ such that $\delta = y - x \geq 0$ satisfy

$$max(0, cc(x) - \delta) \leq cc(y) \leq ee(y) \leq ee(x) - \delta.$$

[27] As can be seen, in the definition of $\approx$ only the endpoints of delay transition are required to be matched.

- Whenever $f \in S \xrightarrow{\epsilon(d)}_I S'$ and $g \in T \xrightarrow{\epsilon(d)}_I T'$, we have $f|g \in S|T \xrightarrow{\epsilon(d)}_I S'|T'$ provided $Sort(f(x)) \cap \overline{Sort(g(x))} = \emptyset$ for all $x \in [0, d[$[28].

We say that $f \in S \overset{\epsilon(d)}{\Longrightarrow}_{Io} S'$ iff $f : [0, d] \to \mathcal{S}$ and $f(d) = S'$, as well as

$$S \overset{\varepsilon}{\Longrightarrow}_\circ S_1 \xrightarrow{\epsilon(d_1)}_I S_1' \overset{\varepsilon}{\Longrightarrow}_\circ \cdots S_k \xrightarrow{\epsilon(d_k)}_I S_k' \overset{\varepsilon}{\Longrightarrow}_\circ S'$$

so that $d_1 + d_2 + \cdots + d_k = d$ and $\forall i \leq k.\ \exists g_i \in S_i \xrightarrow{\epsilon(d_i)}_I S_i'$ such that $\forall \delta \in [0, d_i[.\ f(d_1 + \cdots + d_{i-1} + \delta) = g_i(\delta)$.

$S \overset{\epsilon(d)}{\Longrightarrow}_{Io}$ is defined to be the union of $S \overset{\epsilon(d)}{\Longrightarrow}_{Io} S'$ for all reduced CTR specifications S'.

The refinement $\trianglelefteq_I$ now can be defined by a variant of Definition 3.1 for $\trianglelefteq$, where $\overset{\epsilon(d)}{\Longrightarrow}_{Io}$ is substituted for $\overset{\epsilon(d)}{\Longrightarrow}_\circ$. The principal result of this section (to be proved in the full version) is the following.

Theorem 2 *For any two (reduced) CTR terms S and T $S \trianglelefteq_I T$ if and only if $S \trianglelefteq T$.*

5 (Another) Timed Modal Specifications

In this section we provide yet another semantic variation of (a subset of) CTR, which is based on explicit defining of "necessary" and "possible" modes of transitions, the idea originally appearing in [LT88].

The intuition of the specification refinement in the modal framework is very simple - the more possible and fewer necessary transitions a specification has, the more general it is. This idea has already been adopted to a timed specification formalism in [CGL93, CGL94] resulting in already several times mentioned framework of TMS. Though the TMS model is theoretically very appealing, it turns out that a few of pragmatic properties of the model are a bit unnatural. This applies both to some intuitively desirable refinement examples which are not valid in TMS[29], as well as to the lack of clear operational behaviour intuition due to the having modalities also on the timed transitions.

Here we provide a slight reformulation of the TMS framework (in the light of the developped CTR refinement relation) with the aim of fixing some of its undesirable properties. On the other hand, the theory of A-TMS, sketched in this section, provides yet another semantic variation of the CTR framework, which, as demonstrated by Theorem 3, is consistent with the "standard" development based on $\trianglelefteq$ and explored in Section 3.

Syntactically the A-TMS terms are those of CTR in which every nondeterministic delay prefix is followed by a control action (A-TMS terms have $a_{[c,e]}$,

[28]Here $(f|g)(x)$ is defined to be $f(x)|g(x)$ for all x and $Sort(S) = Sort(0, S)$ for the two-argument $Sort$ function defined in Section 2.

[29]See, for instance, the discussion in Section 3.

$$\frac{--}{nil \xrightarrow{\epsilon(d)} nil} \qquad \frac{--}{a.S \xrightarrow{a}_{\Box} S} \qquad \frac{S \xrightarrow{a}_m S'}{X \xrightarrow{a}_m S'}[X \stackrel{def}{=} S] \in \mathcal{E} \qquad \frac{S \xrightarrow{\epsilon(d)} S'}{X \xrightarrow{\epsilon(d)} S'}[X \stackrel{def}{=} S] \in \mathcal{E}$$

$$\frac{--}{\alpha_{[c,e]}.S \xrightarrow{\epsilon(d)} a_{[c \dot{-} d, e \dot{-} d]}.S} \qquad \frac{-- \qquad [d \leq e]}{\tau_{[c,e]}.S \xrightarrow{\epsilon(d)} \tau_{[c \dot{-} d, e - d]}.S} \qquad \frac{--}{a_{[0,e]}.S \xrightarrow{a}_{\Diamond} S}$$

$$\frac{S \xrightarrow{a}_m S'}{S + T \xrightarrow{a}_m S'} \qquad \frac{T \xrightarrow{a}_m T'}{S + T \xrightarrow{a}_m T'} \qquad \frac{S \xrightarrow{\epsilon(d)} S' \quad T \xrightarrow{\epsilon(d)} T'}{S + T \xrightarrow{\epsilon(d)} S' + T'}$$

$$\frac{S \xrightarrow{a}_m S'}{S|T \xrightarrow{a}_m S'|T} \qquad \frac{T \xrightarrow{a}_m T'}{S|T \xrightarrow{a}_m S|T'} \qquad \frac{S \xrightarrow{\alpha}_m S' \quad T \xrightarrow{\overline{\alpha}}_m T'}{S|T \xrightarrow{\tau}_m S'|T'}$$

$$\frac{S \xrightarrow{\epsilon(d)} S' \quad T \xrightarrow{\epsilon(d)} T'}{S|T \xrightarrow{\epsilon(d)} S'|T'}[Sort(d, S) \cap \overline{Sort(d, T)} = \emptyset]$$

Table 3: Transition rules for A-TMS, $m \in \{\Diamond, \Box\}$

$a_{]c,e]}$ and $\epsilon(d)$ instead of separate $[c, e]$, $]c, e]$ and control action prefixes). We abbreviate $a_{[0,0]}.E$ as $a_{\Box}.E$ (written also as $a.E$) and $a_{[0,\infty]}.E$ as $a_{\Diamond}.E$ [30].

The operational semantics of A-TMS is based on the necessary control transitions $\xrightarrow{a}_{\Box}$, possible control transitions $\xrightarrow{a}_{\Diamond}$ and delays $\xrightarrow{\epsilon(d)}$ [31] [32]. For a fragment of the framework the semantics is summarized in Table 3 (this fragment does not include $\epsilon(d)$ and $a_{]c,e]}$ prefixes, neither restriction and relabelling are considered). The $Sort$ function[33] in the side condition for the parallel composition delay rule is computed, as for CTR terms, following the rules given in Section 2.

In order to define the A-TMS refinement relation let $\mathcal{M}$ be the set of all TMS specifications (over certain sets of specification variables and equations). Let the observational (τ-abstracting) modal transition relations $\stackrel{\mu}{\Longrightarrow}_m$ also be defined in the usual way. The trace set $S \stackrel{\epsilon(d)}{\Longrightarrow}_{\Diamond}$ consists of all traces starting at S and containing interleaving of $\xrightarrow{\epsilon(d_i)}$ and $\xrightarrow{\tau}_{\Diamond}$ transitions, with the total delay amount being d [34]. Also, let μ range over $\Lambda \cup \{\epsilon\}$ (μ is either α, or ϵ).

[30] In the original definition of TMS in [CGL93] the modal prefixes $a_{\Box}.E$ and $a_{\Diamond}.E$ were taken as primitives, and $a_{[c,e]}.E$ was defined as $\epsilon(c).a_{\Diamond}.E + \epsilon(e).a_{\Box}.E$. The $a_{]c,e]}$ prefix here is new, though theoretically it is not principal.

[31] The delays always are in a sense in the possible mode.

[32] A suggested physical interpretation of the necessary and possible transitions is that of transitions of a specification being enabled in a *weak* and *strong* sense. For $a_{[c,e]}.S$ during the first c time units the a transition is not enabled. Afterwards, for the next $c - e$ units this transition is enabled in a weak sense, after what (if it has not been carried out already), it becomes enabled in a strong sense.

[33] This function computes the transition availability in a strong sense.

[34] A formal definition can be given following the style of the trace definition in Section 3. In the same fashion also the relation $\mathcal{R}^t$ for $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$ is defined.

Definition 5.1 *A relation $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$ is a(n) (observational) modal refinement relation, if for any $\langle S, T\rangle \in \mathcal{R}$*

1. $S \stackrel{\mu}{\Longrightarrow}_{\diamond} S'$ *implies* $\exists T'.T \stackrel{\mu}{\Longrightarrow}_{\diamond} T'$ *so that* $\langle S', T'\rangle \in \mathcal{R}$;
2. $T \stackrel{\mu}{\Longrightarrow}_{\Box} T'$ *implies* $\exists S'.S \stackrel{\mu}{\Longrightarrow}_{\Box} S'$ *so that* $\langle S', T'\rangle \in \mathcal{R}$;
3. $f \in S \stackrel{\epsilon(d)}{\Longrightarrow}_{\diamond}$ *implies existence of* $g \in T \stackrel{\epsilon(d)}{\Longrightarrow}_{\diamond}$ *such that* $\langle f, g\rangle \in \mathcal{R}^t$;
4. $S \stackrel{\tau^\infty}{\longrightarrow}_{\diamond}$ *implies* $T \stackrel{\tau^\infty}{\longrightarrow}_{\diamond}$.

We say that S is a modal refinement of T, and write $S \lhd_M T$ whenever there exists a refinement relation $\mathcal{R}$ containing $\langle S, T\rangle$ (In other words, $\lhd_M$ is the largest modal refinement relation).

Let $n : \mathcal{M} \to \mathcal{S}$ be a function giving for any A-TMS term S the same term, viewed as a CTR term[35]. The principal theorem of this section says:

Theorem 3 *For any two modal specifications S and T we have $S \lhd_M T$ if and only if $n(S) \unlhd n(T)$.*

Lemma 5.2

1. *Whenever* $S \stackrel{a}{\longrightarrow}_{\diamond} S'$, *then* $n(S) \stackrel{a}{\longrightarrow}_{\circ} n(S')$.
2. *Whenever* $T \stackrel{a}{\longrightarrow}_{\Box} T'$, *then* $n(T) \stackrel{a}{\longrightarrow} n(T')$.
3. *Whenever* $n(S)(\stackrel{i}{\longrightarrow})^* \stackrel{a}{\longrightarrow} (\stackrel{i}{\longrightarrow})^* Y$, *then* $S \stackrel{a}{\longrightarrow}_{\diamond} S'$ *so that* $Y \unlhd n(S')$.
4. *Whenever* $n(T) \stackrel{a}{\longrightarrow} X$, *then* $T \stackrel{a}{\longrightarrow}_{\Box} T'$ *so that* $n(T') = X$.

To prove $S \lhd_M T \Rightarrow n(S) \unlhd n(T)$ we show that a CTR refinement is the relation $\{\langle X, n(T)\rangle | X \unlhd n(S),\ S \lhd_M T\}$.

The other direction $n(S) \unlhd n(T) \Rightarrow S \lhd_M T$ goes simply by showing that a modal refinement is the relation $\{\langle S, T\rangle | n(S) \unlhd n(T)\}$. In both directions we rely on Lemma 5.2. Details are to be supplied in the full version. $\Box$

6 Decidability

We call a CTR specification *initially integral timed* iff all its initial delay interval prefix bounds are given by integers[36].

Theorem 4 *The refinement relation $\unlhd$ is decidable for initially integral timed CTR specifications.*

The extension of the result of the theorem to allow arbitrary *rational* numbers as the initial delay bounds is straightforward[37]. As a corollary of the theorem we get also the decidability of $\lhd_M$ for TMS.

The main constructions used in the deciding algorithm are those of timed region graph [AD90] and symbolic checking of timed bisimulation [Cer92a, Cer92b], for exposition see also [CGL94, Go94]. The full version will discuss this issue in more detail.

[35] n is an injective mapping of $\mathcal{M}$ into $\mathcal{S}$, introduced for the sake of further convenience.

[36] This requirement is to be applied to the defining equations of all variables which the initial specification relies upon.

[37] Simply by adopting an initial scaling of the time axis.

7 Conclusions

In this paper a mathematically sound and rather general process algebraic framework for refinement of timed system specifications has been proposed, its further fate is to be determined by applicability in CASE studies.

The obtained mathematical results of the coincidence of the three refinement relations $\trianglelefteq$, $\trianglelefteq_I$ and $\triangleleft_M$ show the robustness of the proposed refinement notion - resolving of timed underspecification (nondeterminism). This is a point in favour for trying the proposed calculi on practical examples. It gives an excellent chance for every system designer to understand/imagine the refinement notion in the way he/she likes it most, all choices described here (standard refinement, inline choice, modal specifications) leading to the same formal result.

Still, a somewhat experimental nature of the fourth clauses in all refinement definitions is to be admitted. However, both the theoretical refinement properties and the examples seen so far suggest that this may be a good choice.

A possibility for a further work is to look at incorporating of the introduced timed behaviour underspecification constructs into other process algebraic frameworks (considering disciplines of interprocess communication other than that of CCS may widen the practical applicability of the formalism). It is hoped that the principle of internality of delay nondeterminism, embodied in the semantics of CTR delay interval prefix, should facilitate the possibility of such incorporation.

Though a few comments on the suggested pragmatics of the use of CTR in the real time system design were considered here, there are a few more aspects worth discussing (one of them could be on the pragmatic role of left-open delay interval prefixes).

In a long run one could aim at incorporating both external behaviour and data type refinements into a single framework. For a possible notion of the data type refinement one could look into [BW94] (see also the references therein).

For making CTR a practical specification refinement framework one should work also on practically feasible algorithms for refinement deciding[38]. A possible step into this direction could be a combining of the local validity checking and partition refinement methods for work on deciding "bisimulation - like" properties.

Acknowledgements

The author wants to thank Bengt Jonsson from University of Uppsala and Kim Larsen from University of Aalborg for stimulating discussions and ideas on what a proper notion of operational refinement for timed systems should be.

Thanks are also to BRICS for sponsoring my travel to and attendance at the NWPT'6 conference in Aarhus.

[38]During later stages of the development of the refinement frameworks the possibilities of interactive refinement provers may be considered.

References

[AD90] R. Alur and D. Dill, *Automata for Modelling Real-Time Systems*, in Proceedings of ICALP'90, Lecture Notes in Computer Science No. 443, 1990.

[BW94] R. Back and J. von Wright, *Trace Refinement of Action Systems* in Proceedings of CONCUR'94, Lecture Notes in Computer Science No. 836, 1994.

[CC89] CCITT, *Functional Specification and Description Language (SDL)*, Recommendations Z.100, 1989.

[Cer92a] K. Čerāns, *Decidability of Bisimulation Equivalences for Parallel Timer Processes*, in Proceedings of CAV'92, Lecture Notes in Computer Science No. 663, 1992.

[Cer92b] K. Čerāns, *Algorithmic Problems in Analysis of Real Time System Specifications*, Dr.sc.comp. theses, University of Latvia, 1992.

[CGL93] K. Čerāns, J.Chr. Godskesen and K.G. Larsen, *Timed Modal Specification - Theory and Tools*, in Proceedings of CAV'93, Lecture Notes in Computer Science No. 697, 1993.

[CGL94] K. Čerāns, J.Chr. Godskesen and K.G. Larsen, *Timed Modal Specification - Theory and Tools*, submitted to Formal Aspects of Computing, 1994.

[Che92] L. Chen, *An Interleaving Model for Real Time Systems*, in Proceedings of LFCS'92, Lecture Notes in Computer Science No. 620, 1992.

[Dan92] M. Daniels, *Modelling Real-Time Behaviour with an Interval Time Calculus*, in Proceedings of Conference on Formal Techniques in RT and FT Systems, Nijmegen, 1992

[Dil89] D. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*, ACM Distinguished Dissertations, MIT Press, 1989.

[GMMP89] C. Ghezzi, D. Mandrioli, S. Morasca and M. Pezze *A General Way To Put Time in Petri Nets*, in ACM SIGSOFT Eng. Notes, Vol. 14, No. 3, 1989.

[Go94] J.Chr. Godskesen, *Timed Modal Specifications - A Theory for Verification of Real-Time Concurrent Systems*, Ph.D thesis, Aalborg University, 1994.

[Lam91] L. Lamport, The Temporal Logic of Actions, DEC SRC Technical report No. 79, 1991.

[LT88] K.G. Larsen and B. Thomsen, *A Modal Process Logic*, in Proceedings of the fifth IEEE Symposium on Logic in Computer Science, 1988.

[MF76] P. Merlin and D.J. Farber, *Recoverability of Communication Protocols*, IEEE Trans. on Communication Protocols, Vol. COM-24, No. 9, 1976.

[Mil89] R. Milner, *Communication and Concurrency*, Series in Computer Science, Prentice-Hall International, 1989.

[Wan90] Yi Wang, *Real Time Behaviour of Asynchronous Agents*, in Proceedings of CONCUR'90, Lecture Notes in Computer Science No. 458, 1990.

[Wan91] Yi Wang, *CCS + Time = an Interleaving Model for Real Time Systems*, in Proceedings of ICALP'91, Lecture Notes in Computer Science No. 510, 1991.

Petri Nets, Traces, and Local Model Checking

Allan Cheng*
BRICS†
Computer Science Department, University of Aarhus
Ny Munkegade, DK–8000 Aarhus C, Denmark
e-mail:`acheng@daimi.aau.dk`
Phone: +45 8942 3188 Fax: +45 8942 3255

Abstract

It has been observed that the behavioural view of concurrent systems that all possible sequences of actions are relevant is too generous; Not all sequences should be considered as likely behaviours. By taking progress fairness assumptions into account one obtains a more realistic behavioural view of the systems. In this paper we consider the problem of performing model checking relative to this behavioural view. We present a *CTL*-like logic which is interpreted over the model of concurrent systems labeled 1-safe nets. It turns out that Mazurkiewicz trace theory provides a useful setting in which the progress fairness assumptions can be formalized in a natural way. We provide the first, to our knowledge, set of sound and complete tableau rules for a *CTL*-like logic interpreted under progress fairness assumptions.

keywords: fair progress, labeled 1-safe nets, local model checking, maximal traces, partial orders, inevitability

1 Introduction

Recently attention has focused on behavioural views of concurrent systems in which concurrency or parallelism is represented explicitly [18, 11, 22, 20, 23]. This is is done by imposing more structure on models for concurrent systems, in our case an independence relation on the transitions.

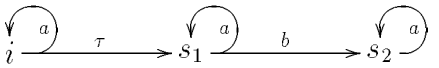
Our main objective is to explore the use of the extra structure of independence in the context of *specification logics*. This paper introduces and studies a *CTL*-like branching time temporal logic, *P-CTL*, interpreted over the reachability graph of labelled 1-safe nets.

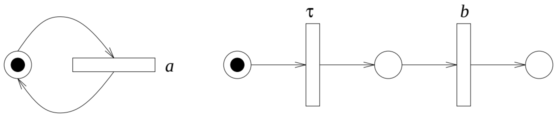
Labeled 1-safe nets are Petri nets whose transitions are labeled by actions from a set *Act* and whose reachable markings have at most one token on any place. Labeled

*This work has been supported by The Danish Research Councils.
†Basic Research in Computer Science, Centre of the Danish National Research Foundation.

1-safe nets are for example obtained by translating agents from various process algebras or constructed as the synchronization of finite automata.

As an example let us consider the process agent *fix* $(X = a.X)|(\tau.b.\mathbf{0})$. Its transition graph is given below to the left. The initial state is i and s_1 and s_2 are the only other reachable states. The agent can also be represented by the labeled 1-safe net to the right, containing three transitions labeled a, τ, and b respectively [14, 23].





The net gives us a more concrete model of the process agent. It shows that the transition labeled a is independent of those labeled τ and b. We can therefore add *more structure* to the above transition system by providing a relation which explicitly states this *independence*. The new transition system is an example of a labeled asynchronous transition system (*l-ATS*) [19, 1, 23, 15]. In general, we can obtain such a labeled asynchronous transition system as the case graph, extended with implicit information about independence, of a labeled 1-safe net. In this paper, we will concentrate on labeled 1-safe nets.

The logic *P-CTL* contains one important feature which is the model-theoretic incorporation of progress. What corresponds to quantified “until” path formulas in *CTL* is in our setting interpreted over firing sequences of labeled 1-safe nets respecting certain *progress assumptions*. This is formalized using maximal traces in the framework of Mazurkiewicz trace-theory, where we make explicit use of the notion of independence between transitions. As an example the formula $Ev(<b>\, tt)$, “eventually b is enabled” (tt means ”True”), is true of the process agent example under the assumption of progress (our interpretation) , but not without (standard *CTL* interpretation). Our interpretation is conservative in the sense that if we interpret *P-CTL* over standard *lts* we get the standard *CTL* interpretation.

Work on expressing progress assumptions and fairness assumptions can be found in for example Manna and Pnueli’s book on temporal logic [10]. Often it involves “coding” these assumptions using linear time temporal logic formulas of the form $\phi_{fair} \Rightarrow \psi$, which require a more detailed knowledge of the particular system. We are able to avoid this obstacle and treat progress assumptions *uniformally*.

In the standard setting of *CTL*-like logics interpreted over labeled transition systems, model checking has been described in [4] using a state based algorithm, and in [8, 21] using tableaux rules. Model checking in the framework of partial order semantics has been described in [17, 16].

In this paper we present the first, to our knowledge, set of sound and complete tableau rules in the style of [8, 21] for a *CTL*-like logic interpreted in the trace theoretic framework. The rules are a generalization of those in [8, 21] in the sense that if we restrict model checking to labeled 1-safe nets without independent transitions, our tableau rules work in the same way. Using the distinction between “local” and “global” model checking as advocated by Stirling and Walker in [21] our method must be classified as “local” model checking. Local model checking has the advantage that it isn’t necessary to have an explicit representation of all the states of the system being investigated. This is however necessary for the global model checking algorithm of [4]. Labeled 1-safe P/T nets are examples of models which can be ”locally” model checked without necessarily generating the entire reachability graph/state space.

In [2] related work is presented. There, based on the approach in [4], we present a global model checking algorithm for *P-CTL* . By choosing this approach, contrary to a tableau based as in [8, 21], we obtain a polynomial time algorithm (in the size of the reachability graph). Also, different extensions of *P-CTL* in the form of modal operators expressing concurrent or conflicting behaviour are considered.

The rest of the paper is organized as follows. In section 2 we provide the necessary definitions. In section 3 we present the logic and its interpretation. Section 4 contains a motivating example followed by the tableau rules and the definition of tableaux. In section 5 we state the main result, soundness and completeness of the proposed tableau rules. Finally section 6 contains the conclusion and suggestions for future work.

2 Basic Definitions

In this section we recall some basic definitions. Furthermore we state some facts and lemmas. First we define *concurrent alphabets*, the fundamental structure in Mazurkiewicz trace theory [11].

Definition 1 Concurrent alphabet and traces

- *A* concurrent alphabet (A, I) *is a set A and a relation $I \subseteq A \times A$, the independence relation, which is symmetric and irreflexive.*

In the following assume a fixed concurrent alphabet (A, I).

- *Given a set A we define $A^\infty = A^* \cup A^\omega$, i.e. A^∞ is the set of all finite and infinite sequences of elements from A.*

- *Define concatenation $\circ$ of elements in A^∞ as:*

$$u \circ v = \begin{cases} u & if\ |u| = \infty \\ uv & else \end{cases}$$

  For notational convenience we will write uv instead of $u \circ v$.

- *Let $\leq_{pref}$ be the usual prefix ordering on sequences and $\pi_{(a,b)}$ the projection on $\{a,b\}^\infty$. We define a preorder $\preceq$ on A^∞ which demands that the relative order of arbitrary elements a and b, which are in conflict, i.e. $(a,b) \notin I$, must be the same when ignoring other elements of the sequences. Formally:*

$$u \preceq v \;\; iff \;\; (\forall (a,b) \notin I.\ \pi_{(a,b)}(u) \leq_{pref} \pi_{(a,b)}(v))$$

- *Define an equivalence relation $\equiv$ on A^∞ as $u \equiv v$ if $u \preceq v$ and $v \preceq u$. Let $[u]$ denote the equivalence class containing u.*

- *Fact: $\equiv$ is a congruence with respect to $\circ$.*

- *The elements of $A^\infty/\equiv$ will be called traces.*

- *For* $[u], [v] \in A^\infty/\equiv$ *we define* $[u] \preceq [v]$ *if* $u \preceq v$. *It can be shown that this relation is a partial order. We will write* $[u] \prec [v]$ *if and only if* $u \preceq v$ *and* $u \not\equiv v$.

- *Fact: for* $u, v \in A^*$:

  - $u \equiv v$ *iff* $u \equiv_M v$
  - $[u] \preceq [v]$ *iff* $(\exists u' \in A^*. [uu'] = [v])$

  where $\equiv_M$ *is the well known equivalence used by Mazurkiewicz when defining finite traces, see* [11].

We have chosen to present traces using projections $\pi_{(a,b)}$. In this way finite as well as infinite traces are handled in a uniform way. Similar definitions can be found in for example [7].

We now define *labeled 1-safe nets*, the labeled version of 1-safe nets[1].

Definition 2 1-safe nets

A 1-safe net, *or just a* net, *is a fourtuple* $N = (P, T, F, M_0)$ *such that*

- P *and* T *are finite disjoint sets; their elements are called* places *and* transitions, *respectively.*

- $F \subseteq (P \times T) \cup (T \times P)$; F *is called the* flow relation.

- $M_0 \subseteq P$; M_0 *is called the* initial marking *of* N; *in general, a set* $M \subseteq P$ *is called a* marking *of* N.

Given $a \in P \cup T$, *the* preset *of* a, *denoted by* ${}^\bullet a$, *is defined as* $\{a' \mid a'Fa\}$; *the* postset *of* a, *denoted by* $a^\bullet$, *is defined as* $\{a' \mid aFa'\}$. *We define the* independence relation I *to be the irreflexive symmetric relation over* T *defined by* $t_1 I t_2$ *iff* ${}^\bullet t_1^\bullet \cap {}^\bullet t_2^\bullet = \emptyset$.

Definition 3 Firing sequences

Let $N = (P, T, F, M_0)$ *be a net.*

- *A transition* $t \in T$ *is* enabled *at a marking* M *of* N *if* ${}^\bullet t \subseteq M$ *and* $(M - {}^\bullet t) \cap t^\bullet = \emptyset$. *We denote the set of enabled transitions at a marking* M *by next(M).*

- *Given a transition* t, *we define a relation* $\xrightarrow{t}$ *between markings as follows:* $M \xrightarrow{t} M'$ *if* t *is enabled at* M *and* $M' = (M - {}^\bullet t) \cup t^\bullet$. *The transition* t *is said to* occur *(or* fire*) at* M. *If* $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} M_n$ *for some markings* $M_1, M_2, \ldots M_n$, *then the sequence* $\sigma = t_1 \ldots t_n$ *is called an* occurrence sequence. M_n *is the marking* reached *by* σ, *and this is denoted* $M_0 \xrightarrow{\sigma} M_n$. *A marking* M *is* reachable *if it is the marking reached by some occurrence sequence.*

- *Given a marking* M *of* N, *the set of reachable markings of the net* (P, T, F, M) *(i.e., the net obtained replacing the initial marking* M_0 *by* M*) is denoted by* $[M\rangle$.

[1] An equivalent definition can be given in terms of Place/Transition nets, see [3].

- *A labeled 1-safe net* $N = (P, T, F, M_0, l)$ *is just a 1-safe net together with a map* $l : T \rightarrow Act$*, mapping each transition to an action in* Act.

The behaviour of a net is captured by the reachability graph.

Definition 4 Reachability graph

- *The reachability graph of a net* N *is the edge-labeled graph,* $(V, E)_N$*, whose vertices,* V*, are the reachable markings of* N*; if* $M \xrightarrow{t} M'$ *for a reachable marking* M*, then there is an edge from* M *to* M' *labeled with* t.

In the following we assume a fixed labeled 1-safe net N and consider its reachability graph $(V, E)_N$. We will use the symbols $p, q, \ldots$ to denote vertices in $(V, E)_N$ and $p \xrightarrow{t} q$ to denote that there is an edge between p and q labeled with t. Notice that (T, I) is a concurrent alphabet. If nothing else is mentioned it is implicitly assumed that (T, I) is used to generate the congruence $\equiv$.

Definition 5 Path

- *Define a* path from $p \in V$ *as a sequence, finite or infinite, of events* $t_1, t_2, \ldots,$ *for which there exists states* $p_1, p_2, \ldots$ *such that* $p \xrightarrow{t_1} p_1 \xrightarrow{t_2} p_2 \cdots$. *Notice that the firing rules of the net ensure the uniqueness of the* p_i*'s if they exist. We therefore also refer to* $p \xrightarrow{t_1} p_1 \xrightarrow{t_2} p_2 \cdots$ *as a path from* p *and use the notation* $p \xrightarrow{\sigma}$ *where* $\sigma = t_1 t_2 \cdots$. *Define* $path(p) \subseteq T^\infty$ *to be all paths from* p *and use the notation* $p \xrightarrow{\sigma}$ *to indicate that* $\sigma \in path(p)$.

- *A path from* p *is* maximal *if it is either infinite or ends in a deadlocked state (or just a deadlock)* p_n*, that is, a state* p_n *such that* $p_n \not\rightarrow$.

- *Due to the firing rules of the nets we have that* $\equiv$ *respect the path property, formally:* $(\forall \sigma \in path(p).\ (\forall \sigma' \in [\sigma].\ p \xrightarrow{\sigma'}))$. *Hence* $path(p)$ *can be partitioned into elements of* $T^\infty/\equiv$. *Moreover if* σ *is finite then* $p \xrightarrow{\sigma} q$ *implies* $(\forall \sigma' \in [\sigma].\ p \xrightarrow{\sigma'} q)$.

- *Given* $\sigma \in path(p), |\sigma| = \infty, \sigma = t_1 t_2 \cdots$. *A transition* t *is said to be* continuously concurrently enabled along $p \xrightarrow{\sigma} = p \xrightarrow{t_1} p_1 \xrightarrow{t_2} p_2 \cdots$ *if and only if* t *is enabled from a certain point and independent of the rest of the transitions taken along* $p \xrightarrow{\sigma}$, *formally:* $(\exists n \in I\!N.\ (\forall j \geq n.\ p_j \xrightarrow{t} \wedge\ t I t_{j+1}))$. *Notice that the irreflexivity of* I *implies that from a certain point* t *is never taken along the path* $p \xrightarrow{\sigma}$. *Whenever* p *is understood we simply say that* t *is continuously concurrently enabled along* σ. *In the process agent example from the introduction,* τ *is continuously concurrently enabled along* $i \xrightarrow{a^\infty}$.

- *Define* $comp(p)$ *as the maximal elements with respect to* $\preceq$ *of* $path(p)/\equiv$. *For* $\sigma \in [\sigma'] \in comp(p)$ *we refer to* $p \xrightarrow{\sigma}$ *as a computation from* p. *In the process agent example,* $\tau b a^\infty$ *is a computation from* i *while* a^∞ *is not, when we use* a*,* b*, and* τ *to refer to the corresponding transitions.*

Lemma 6 *If t is continuously concurrently enabled along $\sigma \in path(p)$ then for any $\sigma' \in [\sigma]$ t is continuously concurrently enabled along σ', that is, $\equiv$ respects* continuously concurrently enabled*. Hence for $\sigma \in path(p)$ we say that $t \in T$ is* continuously concurrently enabled along $[\sigma]$ *if t is* continuously concurrently enabled along *σ.*

Lemma 7 *Given $\sigma \in path(p), |\sigma| = \infty$. Then $(\exists t \in T.\ t$ is continuously concurrently enabled along $\sigma)$ iff $(\exists \sigma' \in path(p).\ [\sigma] \prec [\sigma'])$.*

Above, we have identified the maximal traces as maximal elements in a partial order. Lemma 5 explains why we concentrate on these traces. They represent executions (of a concurrent system) which are fair with respect to progress of independent processes. In [12] the term "concurrency fairness" is used for such behaviours. Compared to other notions of "fairness" in the context of concurrent systems, "progress fairness" is a very weak assumption, see [10] for a comparison.

3 The Logic *P-CTL* and its Interpretation

In this section, we assume a fixed labeled 1-safe net $N = (P, T, F, M_0, l)$. Our logic has the following syntax, where $\alpha \in Act$.

$$A \ ::= \ tt \mid \neg A \mid A_1 \wedge A_2 \mid \Diamond_\alpha A \mid A_1\ U_\exists\ A_2 \mid A_1\ U_\forall\ A_2$$

In Hennessy-Milner logic [13], $<a>\ A$ expresses the fact that one can perform an action a from a state and, in doing so, reach another state at which A holds. Similarly, the $\Diamond_\alpha A$ expresses that a transition labeled α can be performed reaching a state where A holds. tt is an abbreviation for *TRUE*. The "until" operators $U_\exists$ and $U_\forall$ are introduced as generalizations of their counterparts in [4], here interpreted over maximal traces, following Mazurkiewicz [11].

The logic is interpreted over the reachability graph $(V, E)_N$ of N as follows, where $p \in V$, $\alpha \in Act$, and we have written $\models$ instead of $\models_N$ since N was fixed. Only the non trivial cases are presented.

- $p \models \Diamond_\alpha A$ iff $(\exists t \in T, q \in V.\ l(t) = \alpha\ \wedge\ p \xrightarrow{t} q\ \wedge\ q \models A)$
- $p \models A_1\ U_\exists\ A_2$ iff $(\exists\, [\sigma] \in comp(p),\ p \xrightarrow{\sigma}= p_0 \xrightarrow{t_1} p_1 \xrightarrow{t_2} p_2 \cdots.$
  $(\exists\, 0 \leq n \leq |\sigma|.\, (p_n \models A_2)\ \wedge\ (\forall 0 \leq i < n.\ p_i \models A_1)))$
- $p \models A_1\ U_\forall\ A_2$ iff $(\forall [\sigma'] \in comp(p).\ (\forall\, \sigma \in [\sigma'],\ p \xrightarrow{\sigma} = p \xrightarrow{t_1} p_1 \xrightarrow{t_2} p_2 \cdots.$
  $(\exists\, 0 \leq n \leq |\sigma|.\, p_n \models A_2\ \wedge\ \forall 0 \leq i < n.\ p_i \models A_1)))$

Furthermore, we define $f\!f \equiv \neg tt$, $<\alpha>\ A \equiv \Diamond_\alpha A$, $F\ A \equiv tt\ U_\exists A$, $G\ A \equiv \neg F \neg A$, $Ev\ A \equiv tt\ U_\forall A$, and $Al\ A \equiv \neg Ev\, \neg A$. The meaning of EvA is that eventually/inevitably A will hold along any path that satisfies the progress assumptions (fair progress) while AlA means that along some progress fair path A always holds. In the process agent example from the introduction we have $i \models Ev\ \ <b>tt$.

Definition 8 *Given a labeled 1-safe net $N = (P, T, F, M_0, l)$ and a formula A. The* model checking problem of N and A *is the problem of deciding whether or not $M_0 \models A$.*

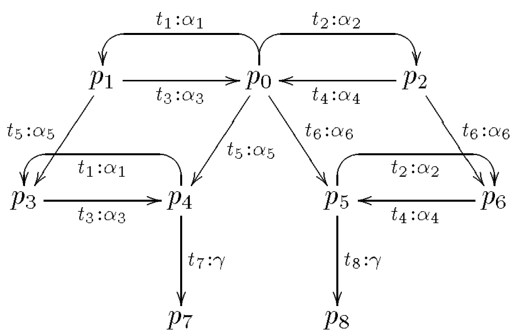
4 A Tableau Method for Model Checking

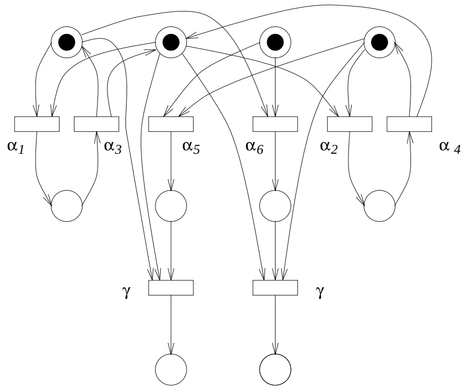
Local model checking as tableau systems has been presented in [21]. As opposed to a global model checker [4] (see also [2], where a global model checker for *P-CTL* is presented), which checks if all states of the system satisfies a formula, a local model checker only checks if a specific state satisfies a given formula. For local model checkers based on tableau systems this is done by only visiting states if the tableau rules require it. Hence the local model checker may well be able to show that a state satisfies a formula without visiting all states of the system. For systems with a compact representation, such as 1-safe nets (where a state of the system/net is considered to be a marking), a local model checker only has to generate new parts of the reachability graph when the tableau rules require it. Since the size of the reachability graph can be exponentially bigger than the size of the net, a local model checker sometimes has an advantage over a global model checker, since it can perform model checking using less memory.

In this section we present a local model checker based on a tableau system for model checking formulas from our logic. We begin by considering an example to give some intuition about the problems we are faced with when looking for a tableau system. Since our interpretation of the logical operators in *P-CTL* coincides with the usual interpretation when there is no concurrency in the nets, we would also like the tableau system to be a conservative extension of those presented in [8, 21]. The main difficulty is how to generalize the unfolding formulas in *P-CTL* which correspond to minimal fixed-point assertions.

4.1 Unfolding Minimal Fixed-Point Assertions

Below we consider a very simple reachability graph, g_1, which is generated by the 1-safe net to the right.





The t_i's are the transitions, the Greek letters the labels, and p_0 the initial marking. The independence relation is the smallest such containing $(t_1, t_5), (t_3, t_5), (t_2, t_6), (t_4, t_6)$. Now $p_0 \models_{g_1} \neg Ev <\gamma> tt$ because of e.g. $[(t_1 t_3 t_2 t_4)^\infty] \in comp(p_0)$ and no state along the computation $(t_1 t_3 t_2 t_4)^\infty$ satisfies $<\gamma> tt$. However if we restrict yourselves to the states p_0, p_1, p_3, p_4, p_7 (referred to as g_2) we do indeed have $p_0 \models_{g_2} Ev <\gamma> tt$, since every computation from p_0 must eventually reach p_4 — t_5 cannot be continuously ignored.

Let us consider how a tableau (proof tree) for $p_0 \models_{g_2} Ev\ {<}\gamma{>}\, tt$ might look like:

$$\dfrac{p_0 \vdash Ev\ {<}\gamma{>}\, tt}{\dfrac{p_1 \vdash Ev\ {<}\gamma{>}\, tt}{p_0 \vdash Ev\ {<}\gamma{>}\, tt \quad \dfrac{p_3 \vdash Ev\ {<}\gamma{>}\, tt}{\dfrac{p_4 \vdash Ev\ {<}\gamma{>}\, tt}{\dfrac{p_4 \vdash {<}\gamma{>}\, tt}{p_7 \vdash tt}}}} \qquad \dfrac{p_4 \vdash Ev\ {<}\gamma{>}\, tt}{\dfrac{p_4 \vdash {<}\gamma{>}\, tt}{p_7 \vdash tt}}}$$

The above tree is constructed according to some intuitive tableau rules. Although informal, the example provides the first important observation. The leftmost branch begins and ends with the sequent $p_0 \vdash Ev\ {<}\gamma{>}\, tt$. In the μ-calculus $Ev\ {<}\gamma{>}\, tt$ is expressed by the formula $\mu X.\ {<}\gamma{>}\, tt \vee [Act]X$. Hence based on the tableau methods from [8, 21], one might expect that the above tree should be discarded as a tableau since in the unfolding of the minimal fixed-point assertion reaches itself. However in the current framework, we interpret the logic over maximal traces, and the detected loop, $(p_0 \xrightarrow{t_1} p_1 \xrightarrow{t_3} p_0)^\infty$, is not a computation from p_0 since the transition t_5 is continuously concurrently enabled. This example suggests that we might allow the unfolding of a minimal fixed-point assertion to reach itself. The cases in which this will be allowed should include the existence of an transition that is continuously concurrently enabled along the loop represented by such a branch. Our solution to this problem is to annotate the logic used in the tableau rules. The idea is to keep track of the transitions which are continuously concurrently enabled and update this information as one unfolds the reachability graph via the tableau rules. So in our case t_5 would be "remembered" along the $p_0 \to p_1 \to p_0$ branch.

Let us consider a second example. This time we use g_1. Again we construct in an intuitive and informal manner a tree rooted in the sequent $p_0 \vdash Ev\ {<}\gamma{>}\, tt$:

$$\dfrac{p_0 \vdash Ev\ {<}\gamma{>}\, tt}{\dfrac{p_1 \vdash Ev\ {<}\gamma{>}\, tt}{p_0 \vdash Ev\ {<}\gamma{>}\, tt \quad \dfrac{p_3 \vdash Ev\ {<}\gamma{>}\, tt}{\dfrac{p_4 \vdash Ev\ {<}\gamma{>}\, tt}{\dfrac{p_4 \vdash {<}\gamma{>}\, tt}{p_7 \vdash tt}}}} \quad \dfrac{p_4 \vdash Ev\ {<}\gamma{>}\, tt}{\dfrac{p_4 \vdash {<}\gamma{>}\, tt}{p_7 \vdash tt}} \quad \dfrac{p_5 \vdash Ev\ {<}\gamma{>}\, tt}{\dfrac{p_5 \vdash {<}\gamma{>}\, tt}{p_8 \vdash tt}} \quad \dfrac{p_2 \vdash Ev\ {<}\gamma{>}\, tt}{\dfrac{p_6 \vdash Ev\ {<}\gamma{>}\, tt}{\dfrac{p_5 \vdash Ev\ {<}\gamma{>}\, tt}{\dfrac{p_5 \vdash {<}\gamma{>}\, tt}{p_8 \vdash tt}}} \quad p_0 \vdash Ev\ {<}\gamma{>}\, tt}}$$

Again the interesting parts are the branches that unfold a minimal fixed-point assertion into itself. There are two such branches, the leftmost and the rightmost. However along both of these there are transitions which are continuously concurrently enabled, t_5 for the left branch and t_6 for the right branch. So according to the previous remarks these branches shouldn't discard the tree from being a tableau. But we do wish to discard the tree as a tableau since $p_0 \models_{g_1} \neg Ev\ {<}\gamma{>}\ tt$. The problem is that by composing the two loops $(p_0 \xrightarrow{t_1} p_1 \xrightarrow{t_3} p_0)$ and $(p_0 \xrightarrow{t_2} p_2 \xrightarrow{t_4} p_0)$ we can obtain an infinite path $(p_0 \xrightarrow{t_1} p_1 \xrightarrow{t_3} p_0 \xrightarrow{t_2} p_2 \xrightarrow{t_4} p_0)^\infty$. Along this path there is no transition which is continuously concurrently enabled, that is, it is a computation from p_0. Moreover no state along the loop satisfies ${<}\gamma{>}\, tt$. This fact should discard the tree from being a tableau.

One solution to the problem of detecting such "combined" loops is to continue to unfold the minimal fixed-point assertions $p_0 \vdash Ev\ {<}\gamma{>}\ tt$. If we unfold the fixed point assertion once more in the above example, still updating and propagating the information kept in the annotation, we will obtain a leaf with the information that we have found a looping path along which no transition is continuously concurrently enabled. This could discard the tree from being a tableau.

It turns out that the remaining problem is to find some general bound on the number of times we allow the unfolding of a minimal fixed-point assertion. In the next subsection we provide the necessary definitions. The bound we use is at most $|T|$, the number of transitions in the labeled 1-safe net.

4.2 Tableau Rules

In this section we consider a fixed labeled 1-safe net N and its reachability graph $(V, E)_N$.

4.2.1 Annotated Logic

Before giving the tableau rules, we define the syntax of an annotated logic which is used in the tableau rules:

$$B \ ::= \ tt \mid \neg B \mid B_1 \wedge B_2 \mid \Diamond_\alpha B \mid B_1 \ U_\exists^{\mathcal{C}} \ B_2 \mid B_1 \ U_\forall^{\mathcal{C}} \ B_2$$

where $1 \leq n < \infty$, $\mathcal{C} \subseteq V$. The model checking will be performed by unfolding parts of the reachability graph into a tree structure. The unfolding will take place under certain constraints which restrict the size of the tree structure. The intuition is that $\mathcal{C}$ keeps track of which states have been visited and prevents unnecessary unfolding.

For the $U_\forall$ operator we need a more elaborate annotation. In the tableau rules we use the following annotations: $B_1 \ U_\forall^{(p,n,T')} \ B_2$, $B_1 \ U_\forall^{(p,n,T',V',\rightarrow)} \ B_2$, and $B_1 \ U_\forall^{(p,n,T',V',\leftarrow)} \ B_2$, where B_1, B_2 are formulas from the annotated logic and $p \in V$, $T' \subseteq T$, $V' \subseteq V$, and $n \in I\!N$. V' plays the same role as $\mathcal{C}$ and T' keeps track of which transitions have been concurrently enabled but ignored along the path corresponding to the current branch in the tree. p is a state from which we try to detect certain "critical loops". In the example from section 4.1 p would correspond to p_0 and the "critical loops" would correspond to $p_0 \xrightarrow{t_1}\xrightarrow{t_3} p_0$ and $p_0 \xrightarrow{t_2}\xrightarrow{t_4} p_0$.

4.2.2 Rules

In this section we present the tableau rules. There is a trade off between the rules and the definition of tableaux. One can obtain simple rules at the cost of a complicated definition of tableaux[2]. At the cost of presenting less simple tableau rules we keep the definition of tableaux simple.

The rules will consist of tableau rules for sequents of the form $p \vdash B$. The rules can be read from top to bottom as: "the top sequent holds (B holds at p) if the bottom sequents and side conditions hold".

[2]The set of simple rules we have identified requires a global side condition in the definition of tableaux.

Tableau Rules

$\wedge, 1)$ $$\frac{p \vdash B_1 \ \wedge \ B_2}{p \vdash B_1 \quad p \vdash B_2}$$

$\Diamond_\alpha, 3)$ $$\frac{p \vdash \Diamond_\alpha B}{q \vdash}$$

– $t \in T, q \in V, p \xrightarrow{t} q$

– $l(t) = \alpha$

$U_\exists^{\mathcal{C}}, 3)$ $$\frac{p \vdash B_1 \ U_\exists^{\mathcal{C}} \ B_2}{p \vdash B_2}$$

– $p \notin \mathcal{C}.$

4) $$\frac{p \vdash B_1 \ U_\exists^{\mathcal{C}} \ B_2}{p \vdash B_1 \quad q \vdash B_1 \ U_\exists^{\mathcal{C}\cup\{p\}} \ B_2}$$

– $p \notin \mathcal{C}, t \in T, q \in V, p \xrightarrow{t} q.$

$U_\forall^{\mathcal{C}}, 5)$ $$\frac{p \vdash B_1 \ U_\forall^{\mathcal{C}} \ B_2}{p \vdash B_2}$$

– $p \notin \mathcal{C}.$

6) $$\frac{p \vdash B_1 \ U_\forall^{\mathcal{C}} \ B_2}{p \vdash B_1 \quad q_1 \vdash B_1 \ U_\forall^{\mathcal{C}\cup\{p\}} \ B_2 \cdots q_m \vdash B_1 \ U_\forall^{\mathcal{C}\cup\{p\}} \ B_2}$$

– $p \notin \mathcal{C}, next(p) = \{t_1, \ldots, t_m\},$
$0 < m \in I\!N,$

– $(\forall 1 \leq i \leq m. \ p \xrightarrow{t_i} q_i)$

7) $$\frac{p \vdash B_1 \ U_\forall^{\mathcal{C}} \ B_2}{p \vdash B_1 \ U_\forall^{(p,|next(p)|,next(p))} \ B_2}$$

– $p \in \mathcal{C}$

8) $$\frac{p \vdash B_1 \ U_\forall^{(p,n,T')} \ B_2}{p \vdash B_1 \ U_\forall^{(p,n-1,T',\emptyset,\rightarrow)} \ B_2}$$

– $0 < n \in I\!N, T' \neq \emptyset$

9) $$\frac{q \vdash B_1 \ U_\forall^{(p,n,T',V',\rightarrow)} \ B_2}{q \vdash B_1 \quad q_i \vdash B_1 \ U_\forall^{(p,n,e_iIT',V'\cup\{q\},\rightarrow)} \ B_2}$$

– $q \notin V', next(q) = \{t_1, \ldots, t_m\},$
$0 < m \in I\!N,$

– $(\forall 1 \leq i \leq m. \ q \xrightarrow{t_i} q_i)$

and for $D \subseteq T$, $t \in T$, we define
$tID = DIt = \{t' \in D \,|\, t'It\}.$

10) $$\frac{q \vdash B_1 \ U_\forall^{(p,n,T',V',\rightarrow)} \ B_2}{q \vdash B_2}$$

– $q \notin V'$

11) $$\frac{q \vdash B_1 \ U_\forall^{(p,n,T',V'\cup\{q\},\rightarrow)} \ B_2}{q \vdash B_1 \ U_\forall^{(p,n,T',\emptyset,\leftarrow)} \ B_2}$$

12) $$\frac{q \vdash B_1 \ U_\forall^{(p,n,T',V',\leftarrow)} \ B_2}{q \vdash B_1 \quad q_i \vdash B_1 \ U_\forall^{(p,n,e_iIT',V'\cup\{q\},\leftarrow)} \ B_2}$$

– $q \notin V', next(q) = \{t_1, \ldots, t_m\},$
$0 < m \in I\!N, q \neq p$

– $(\forall 1 \leq i \leq m. \ q \xrightarrow{t_i} q_i)$

13) $$\frac{q \vdash B_1 \ U_\forall^{(p,n,T',V',\leftarrow)} \ B_2}{q \vdash B_2}$$

– $q \notin V'$

14) $$\frac{p \vdash B_1 \ U_\forall^{(p,n,T',V',\leftarrow)} \ B_2}{p \vdash B_1 \ U_\forall^{(p,n,T')} \ B_2}$$

Some explanation of the rules seems appropriate. Rules 1 to 4 should be reasonably clear. The annotation of the $U_\exists$ operator prevents unnecessary unfolding.

The remaining rules are all concerned with the $U_\forall$ operator. If $p \not\models A_1 \; U_\forall \; A_2$ then by definition of $\models$ there are two fundamental cases which can occur. Either there exists a finite path along which $A_1 \wedge \neg A_2$ holds until either $\neg A_1 \wedge \neg A_2$ holds or a deadlock is reached, or else there exists an infinite computation along which $\neg A_1 \wedge A_2$ holds. The latter situation reduces to the existence of an infinite computation from p which consists of at most $|T|$ loops $\sigma_{p'q_i}\sigma_{q_i-loop}\sigma_{q_ip'}$ from a state p' reachable from p. This is illustrated as follows:

$$\begin{tikzcd} p \arrow[r, "\sigma_{pp'}"] & p' \arrow[r, "\sigma_{p'q_i}"'] & q_i \arrow[l, bend right, "\sigma_{q_ip'}"'] \arrow[loop right, "\sigma_{q_i-loop}"] \end{tikzcd}$$

Rules 6 and 7 take care of the part denoted $\sigma_{pp'}$. Rules 8, 9, and 11 take care of $\sigma_{p'q_i}\sigma_{q_i-loop}$, and rules 12 and 14 take care of $\sigma_{q_ip'}$.

The next step is to define derivation trees which are build up according to the tableau rules.

4.2.3 Derivation Trees and Tableaux

In this section we define the tableaux. This is done by first defining a larger class of trees, derivation trees, which are generated according to the tableau rules. The next step is to restrict the class of derivation trees, using the annotation of the formulae, to a subclass of derivation trees which will be defined to be the tableaux.

Derivation trees are defined inductively in the usual manner, except perhaps for negation. That is, if $T_1, \ldots, T_n$ are derivation trees with roots matching the sequents under the bar of a rule and the side conditions are fulfilled then one obtains a new derivation tree by "pasting the derivation trees together" according to the rule. The root of the new derivation tree is labeled by the sequent above the bar.

First we define the basis, all trees having a single node, the root, labeled with the shown sequent.

- $p \vdash tt$ is a derivation tree.
- $p \vdash \neg B$ is a derivation tree.
- $p \vdash B_1 \; U_\exists^{\mathcal{C}} \; B_2$, where $p \in \mathcal{C}$ is a derivation tree.
- $p \vdash B_1 \; U_\forall^{(p,n,T')} \; B_2$, where $n = 0$ or $T' = \emptyset$.
- $q \vdash B_1 \; U_\forall^{(p,n,T',V',\leftarrow)} B_2$, where $q \in V'$.

By applying the rules we can obtain new derivation trees, for example:

- If T_1 is a derivation tree with root $p \vdash B_1$ and T_2 is a derivation tree with root $q \vdash B_1 \; U_\exists^{\mathcal{C}\cup\{p\}} \; B_2$, where $p \notin \mathcal{C}$ and $\exists t \in T.p \xrightarrow{t} q$ then $\dfrac{p \vdash B_1 \; U_\exists^{\mathcal{C}} \; B_2}{T_1 \quad T_2}$ is a derivation tree with root $p \vdash B_1 \; U_\exists^{\mathcal{C}} \; B_2$.

- If T is a derivation tree with root $p \vdash B_2$ and $p \notin \mathcal{C}$ then $\dfrac{p \vdash B_1 \, U_\forall^{\mathcal{C}} \, B_2}{T}$ is a derivation tree with root $p \vdash B_1 \, U_\forall^{\mathcal{C}} \, B_2$.

Nothing else is a derivation tree. Next, using the annotation of the formulae, we obtain two useful definitions:

- Let Ann be the obvious function which takes a formula A from the first grammar and annotates all its $U_\exists$ and $U_\forall$ operators with $\mathcal{C} = \emptyset$, giving a formula $Ann(A)$ from the second grammar. A formula B from the second grammar is said to be *clean* if there exists a formula A from the first grammar such that B equals $Ann(A)$.
- Sequents of the form $q \vdash B_1 \, U_\forall^{(q,n,\emptyset)} \, B_2$, where $n \in \mathbb{N}$ and $q \in V$, and of the form $q \vdash B_1 \, U_\exists^{\mathcal{C}} \, B_2$, where $q \in \mathcal{C}$, are called terminal sequents.

We can now define tableaux and will hereby restrict our attention to meaningful derivation trees. We get rid of meaningless derivation trees as for example $p \vdash \neg tt$.
A tableau is a derivation tree T with root $p \vdash Ann(A)$ such that either

- $A = tt$ or
- $A = \neg A'$ and there exists no tableau with root $p \vdash Ann(A')$ or
- A is not of the above form and
  - every proper subtree T' of T whose root is labeled with a clean formula is itself a tableau and
  - T has no leaves labeled with terminal sequents.

5 Soundness and Completeness

Having given the necessary definitions we are now ready to state the main result, which holds for the reachability graph of a labeled 1-safe net N, where p is a reachable marking of N:

Theorem 9

Soundness:

If T is a tableau with root $p \vdash Ann(A)$ then $p \models A$.

Completeness*:*

If $p \models A$ then there exists a tableau with root $p \vdash Ann(A)$.

Proof. The proof proceeds by structural induction, showing soundness and completeness simultaneously. The main difficulty is the $U_\forall$ operator. The proof is constructed using the technique from [2]. $\square$

As an example, we show that the process agent from the introduction will eventually be able to fire a transition labeled by a b action (assume the transitions are t_1, t_2, and t_3, and are labeled a, τ, and b). By the previous theorem, to show $i \models Ev({<}b{>}\,tt)$ it is sufficient to construct a tableau with root $i \vdash tt\, U_\forall^{\emptyset}\,({<}b{>}\,tt)$.

$$\dfrac{i \vdash tt\, U_\forall^{\emptyset}({<}b{>}\,tt)}{i \vdash tt \qquad \dfrac{\dfrac{i \vdash tt\, U_\forall^{\{i\}}({<}b{>}\,tt)}{i \vdash tt\, U_\forall^{(i,2,\{t_1,t_2\})}({<}b{>}\,tt)}}{T_1} \qquad \dfrac{\dfrac{s_1 \vdash tt\, U_\forall^{\{i\}}({<}b{>}\,tt)}{s_1 \vdash {<}b{>}\,tt}}{s_2 \vdash tt}}$$

where T_1 is

$$\dfrac{i \vdash tt\, U_\forall^{(i,1,\{t_1,t_2\},\emptyset,\rightarrow)}({<}b{>}\,tt)}{\dfrac{\dfrac{\dfrac{i \vdash tt\, U_\forall^{(i,1,\{t_2\},\{i\},\rightarrow)}({<}b{>}\,tt)}{i \vdash tt\, U_\forall^{(i,1,\{t_2\},\emptyset,\leftarrow)}({<}b{>}\,tt)}}{i \vdash tt\, U_\forall^{(i,1,\{t_2\})}({<}b{>}\,tt)}}{T_2} \qquad \dfrac{\dfrac{s_1 \vdash tt\, U_\forall^{(i,1,\{t_1\},\{i\},\rightarrow)}({<}b{>}\,tt)}{s_1 \vdash {<}b{>}\,tt}}{s_2 \vdash tt} \qquad i \vdash tt}$$

where T_2 is

$$\dfrac{i \vdash tt\, U_\forall^{(i,0,\{t_2\},\emptyset,\rightarrow)}({<}b{>}\,tt)}{i \vdash tt \qquad \dfrac{\dfrac{i \vdash tt\, U_\forall^{(i,0,\{t_2\},\{i\},\rightarrow)}({<}b{>}\,tt)}{i \vdash tt\, U_\forall^{(i,0,\{t_2\},\emptyset,\leftarrow)}({<}b{>}\,tt)}}{i \vdash tt\, U_\forall^{(i,0,\{t_2\})}({<}b{>}\,tt)} \qquad \dfrac{\dfrac{s_1 \vdash tt\, U_\forall^{(i,0,\emptyset,\{i\},\rightarrow)}({<}b{>}\,tt)}{s_1 \vdash {<}b{>}\,tt}}{s_2 \vdash tt}}$$

Notice that if we restrict ourselves to labeled 1-safe nets where the independence relation is empty and translate $A_1\ U_\exists\ A_2$ as $\mu X.\, A_2 \vee (A_1 \wedge {<}Act{>}X)$ and $A_1\ U_\forall\ A_2$ as $\mu X.\, A_2 \vee (A_1 \wedge\ {<}Act{>}\,tt \wedge [Act]X)$ (actually applying this translation recursively on the subformulas A_1 and A_2) our proof rules will work in essentially the same manner as those presented in [8, 21].

6 Conclusion and Future Work

Partial order semantics for concurrent systems have gained interest because interleaving models of concurrency have failed to provide an acceptable interpretation of what it means for events of a concurrent system to be independent. Much work has been devoted to translate obtained results and notions from the interleaving models to the "true concurrency" models [6, 5, 23, 9]. Trying to contribute to the "translation of results" we have provided proof rules for a *CTL*-like logic interpreted over maximal traces. The work which we have tried to "translate" can be found in [8, 21]. Our work supports automatic verification of distributed systems whose liveness properties are only provable under the assumption of progress.

A modal operator specifying concurrent behaviour have been presented in [9]. In [2] we present other modal operators specifying concurrent or conflicting behaviour. By extending *P-CTL* with such modal operators one can also obtain undecidability results such as those in [9]. Also, the operators presented in [2] are all incomparable. The choice of the modal operators has partly been made because we want to draw attention to the fact that there is no obvious choice. This leaves open the problem of identifying a (set of) more general modal operator(s).

Another research area might be to handle a more expressive logic (perhaps one containing a recursion operator) in a similar way, that is, define the interpretation of the formulas over maximal traces and proving soundness and completeness of some tableau proof rules.

Finally the general satisfiability problem for some of the extended logics is still an open problem.

Acknowledgements: I thank Mogens Nielsen and Nils Klarlund for inspiring discussions and comments.

References

[1] M. A. Bednarczyk. *Categories of asynchronous systems.* PhD thesis, University of Sussex, 1988. PhD in computer science, report no.1/88.

[2] Allan Cheng. Local model checking and traces. Technical report, Daimi, Computer Science Department, Aarhus University, May 1994. BRICS Report Series RS-94-17.

[3] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity results for 1-safe nets. In *Proc. FST&TCS 13, Thirteenth Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages 326–337. Springer-Verlag (*LNCS* 761), Bombay, India, December 1993. To appear in TCS, volume 148.

[4] Edmund M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[5] Lalita Jategaonkar and Albert Meyer. Deciding true concurrency equivalences on finite safe nets. In *Proc. ICALP'93*, pages 519–531, 1993.

[6] André Joyal, Mogens Nielsen, and Glynn Winskel. Bisimulation and open maps. In *Proc. LICS'93, Eighth Annual Symposium on Logic in Computer Science*, pages 418–427, 1993.

[7] Marta Z. Kwiatkowska. Event fairness and non-interleaving concurrency. *Formal Aspects of Computing*, 1:213–228, 1989.

[8] Kim G. Larsen. Proof systems for Hennessy-Milner logic with recursion. In *Proceedings of CAAP, Nancy France*, pages 215–230. Springer-Verlag (*LNCS* 299), March 1988.

[9] Kamal Lodaya, Rohit Parikh, R. Ramanujam, and P. S. Thiagarajan. A logical study of distributed transition systems. Technical report, School of Mathematics, SPIC Science Foundation, Madras, 1993. To appear in Information and Computation, a preliminary version appears as Report TCS–93–8.

[10] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1992.

[11] Antoni Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 279–324. Springer-Verlag (*LNCS* 255), 1986.

[12] Antoni Mazurkiewicz, Edward Ochmański, and Wojciech Penczek. Concurrent systems and inevitability. *Theoretical Computer Science*, 64():281–304, 1989.

[13] Robin Milner. *Communication and Concurrency.* Prentice Hall International Series In Computer Science, C. A. R. Hoare series editor, 1989.

[14] Madhavan Mukund and Mogens Nielsen. CCS, Locations and Asynchronous Transition Systems. *Proc. Foundations of Software Technology and Theoretical Computer Science 12*, pages 328–341, 1992.

[15] Ernst R. Olderog. *Nets, Terms and Formulas.* Cambridge University Press, 1991. *Number 23 Tracts in Theoretical Computer Science.*

[16] Doron Peled and Amir Pnueli. Proving partial order liveness properties. In *Proc. ICALP'90*, pages 553–571. Springer-Verlag (*LNCS* 443), 1990.

[17] Wojciech Penzcek. Temporal logics for trace systems: On automated verification. *International Journal of Foundations of Computer Science*, 4 (1):31–67, 1993.

[18] Wolfgang Reisig. *Petri Nets – An Introduction.* EATCS Monographs in Computer Science Vol.4, 1985.

[19] M. W. Shields. Concurrent machines. *Computer Journal*, 28:449–465, 1985.

[20] Eugene W. Stark. Concurrent transition systems. *Theoretical Computer Science*, 64():221–269, 1989.

[21] Colin P. Stirling and David Walker. Local model checking in the modal mu-calculus. Technical Report ECS–LFCS–89–78, Laboratory for Foundations of Computer Science, Department of Computer Science – University of Edinburgh, May 1989.

[22] Glynn Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 325–390. Springer-Verlag (*LNCS* 255), 1986.

[23] Glynn Winskel and Mogens Nielsen. Models for concurrency. Technical Report DAIMI PB–429, Computer Science Department, Aarhus University, November 1992. To appear as a chapter in the Handbook of Logic and the Foundations of Computer Science, Oxford University Press.

Introduction to ALF — a Background

Catarina Coquand

Chalmers Technical University
Inst. for Computer Science
Göteborg, Sweden
E-mail: catarina@cs.chalmers.se

We will give an introduction to type theory, focussing on its use as a logical framework for proofs and programs. We will also discuss the use of inductive definition as specifications and proofs by pattern matching. Many examples will be presented.

The basic idea behind using type theory for developing proofs and programs is the Curry-Howard isomorphism between propositions and types. Take for example the proposition (A implies B implies A). Looking at this proposition as a type we see that this is the type of the K combinator. In type theory we then say that the K combinator proofs the proposition (A implies B implies A).

The language that is used is a functional language with dependent types. Hence programs and proofs can be described in the same language. We use inductive definitions for our specifications. Introduction rules in logic corresponds to defining a type by its constructors. Proofs are functions defined by pattern matching over these types. We shall also mention how this can be extended to proofs about infinite objects.

PROGRAM SEPARATION AS A BASIS FOR DEFINITIONAL HIGHER ORDER PROGRAMMING

GÖRAN FALKMAN

ABSTRACT. We describe a program separation scheme based on a notion of form and content of an algorithm. We show that this type of program separation is one way of making algorithms an explicit part of declarative programming. We also show how this type of program separation can be used as a basis for definitional higher order programming.

1. INTRODUCTION

The basis for all modern declarative programming languages like ML, Haskell, and Prolog is that the programmer should be more concerned with *what* the program should do and less with *how* the program should accomplish some task.

The actual algorithms are often rather *implicit* in such languages since they are hidden under layers of very high-level notions like pattern-matching, unification, and backtracking. So if you want to study algorithms in this context there must be a way of making algorithms a more *explicit* part of declarative programming without losing to much of the declarative reading of the programs.

Just as higher order functions and meta-programming have been subjects to intensive studies in the functional programming and logic programming community respectively, we are in the area of definitional programming interested in the notions of *higher order definitions* and *definitional higher order programming.*

In this paper we describe a program separation scheme based on a notion of *form* and *content* of an algorithm. We show that this type of program

---

Key words and phrases. Programming methodology, definitional programming, program separation.

This work was supported by the Swedish Research Council for Engineering Sciences (TFR) and the Swedish National Board for Industrial and Technical Development (NUTEK).

It is part of the work done within the ESPRIT working group GENTZEN

separation is one way of making algorithms an explicit part of declarative programming. We also show how this type of program separation can be used as a basis for definitional higher order programming.

The work presented here is based on some earlier work on program separation [5].

As an introductory example, consider the recursive definition of addition:

$$\mathcal{P} \quad \begin{cases} plus(0, m) = m, \\ plus(s(n), m) = s(plus(n, m)). \end{cases}$$

The intended algorithm associated with this definition is given implicitly in proving that $plus(n, m) = n + m$ with respect to $\mathcal{P}$. By induction: If $n = 0$, then the definiens of $plus(n, m)$, $\mathcal{P}(plus(n, m))$, is m and thus in this case $plus(n, m) = n + m$. So assume $plus(n, m) = n + m$ with respect to $\mathcal{P}$. Now $\mathcal{P}(plus(s(n), m)) = s(plus(n, m))$, so by the induction hypothesis $plus(s(n), m) = s(n) + m$. The algorithm may be formulated as follows:

(1) if $n = 0$, then take the definiens of $plus(n, m)$ in $\mathcal{P}$, $\mathcal{P}(plus(n, m))$, as the result of the computation of $plus(n, m)$,

(2) if $n = s(k)$, then the result of the computation of $plus(n, m)$ is given by computing $plus(k, m)$ and then applying s to the result.

In this algorithm we really only use primitive operations on definitions; looking up the definiens of an expression and checking whether a given expression is defined or not. So we may reformulate the algorithm in the following manner:

$$Plus(x) = \begin{cases} \mathcal{P}_1(x) & \text{if } \mathcal{P}_1(x) \cap \mathrm{Dom}(\mathcal{P}_1) = \emptyset, \\ \mathcal{P}_2(Plus(\mathcal{P}_1(x))) & \text{if } \mathcal{P}_1(x) \cap \mathrm{Dom}(\mathcal{P}_1) \neq \emptyset. \end{cases}$$

where $\mathcal{P}_1$ and $\mathcal{P}_2$ are the following definitions:

$$\mathcal{P}_1 \quad \begin{cases} plus(0, m) = m, \\ plus(s(n), m) = plus(n, m). \end{cases}$$

$$\mathcal{P}_2 \quad \{\, n = s(n)$$

$\mathcal{P}_1$ and $\mathcal{P}_2$ is the canonical separation of $\mathcal{P}$ in order to find definitions such that addition is locally definable, i.e., to isolate the local operations needed to compute additions on the basis of $\mathcal{P}$. $Plus$ then describes the global structure of the algorithm.

It is natural to think of this type of program separation as a separation of *form* and *content*. $Plus$ gives the recursive form of the algorithm and $\mathcal{P}_1$ and $\mathcal{P}_2$ give the particular content needed to compute addition.

The rest of this paper is organized as follows. In Section 2 we give a short introduction to the target language. In Section 3 we introduce a notion of form and content of an algorithm by studying a couple of examples.

2. Programming with definitions

The target language of this paper is based on an existing definitional programming language called GCLA[1] [1, 2, 3, 4, 7].

Compared to GCLA we use an idealized syntax and, what is more important, we have extended the language by allowing multiple object-level definitions and more general guards and provisos. We use the same operational semantics as in GCLA.

2.1. Basic building blocks.

The basic building blocks of the language are *variables*, *constants*, *terms*, *atoms*, and *conditions*.

2.1.1. *Variables.* Variables are denoted by strings with a capital initial letter, e.g., "X", "Xs", "$A\_long\_variable$" etc.

2.1.2. *Constants.* Constants are denoted by strings beginning with a lowercase letter, e.g., "x", "$aConstant$" etc.

2.1.3. *Terms.* Terms are built up using variables and constants:

(i) All variables are terms,
(ii) All constants are terms,
(iii) If f is a functor and $T_1, \ldots, T_n$ are terms, then $f(T_1, \ldots, T_n)$ is a term.

2.1.4. *Atoms.* Atoms are terms which are not variables:

(i) All constants are atoms,
(ii) If f is a functor and $T_1, \ldots, T_n$ are terms, then $f(T_1, \ldots, T_n)$ is an atom.

2.1.5. *Conditions.* Conditions are built up using, $\bot$, $\top$, and terms:

(i) $\bot$ is condition. (Falsity),
(ii) $\top$ is a condition. (Truth),
(iii) All terms are conditions,
(iv) If C_1 and C_2 are conditions, then $C_1 \to C_2$ is a condition,
(v) If C_1 and C_2 are conditions, then $C_1 ; C_2$ is a condition,
(vi) If C_1 and C_2 are conditions, then C_1 , C_2 is a condition.

2.2. Definitions and programs.

Definitions are built up by *definitional clauses* and using definitions we form *programs*.

2.2.1. *Definitional clauses.* If a is an atom and C is a condition, then $a = C$ is a definitional *clause*.

A is referred to as the *head* of the clause and C is referred to as the *body* of the clause.

We may also have *guarded* definitional clauses. If a is an atom, C is a condition, and $G_1, \ldots, G_n$ are *guards*, i.e., simple syntactical equality and inequality constraints on variables like "$X = Y$", "$f(Y) \neq f(g(Z))$", "$W \geqslant 1$" etc., then $a\#\{G_1 \wedge \cdots \wedge G_n\} = C$ is a guarded definitional clause.

[1] Is to be pronounced "Gisela".

2.2.2. *Definitions.* A definition $\mathcal{D}$ is an ordered set of (guarded) definitional clauses:

$$\mathcal{D} \quad \begin{cases} a = A. \\ b = B. \\ c = C. \\ d = D. \\ \quad \vdots \end{cases}$$

Definitions will be denoted by $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, ...

2.2.3. *Programs.* A *program* $\mathbf{P}$, is a set of definitions, $\mathbf{P} = \{\mathcal{D}_1, \dots, \mathcal{D}_n\}$ for $n > 1$.

2.3. The meaning of a program. A program $\mathbf{P}$ is a set of definitions. In the definitions we define atoms in terms of certain conditions, but these conditions are not defined in the definitions themselves. Instead we select one of the definitions in $\mathbf{P}$ to interpret all the conditions in the other definitions. This definition will be called the *rule definition* (or the meta-level definition) and it will be denoted by $\mathcal{R}$. The definitions in $\mathbf{P} \setminus \{\mathcal{R}\}$ are referred to as the *object-level definitions* (or simply the definitions). We continue to denote object-level definitions by $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, ...

The difference between the object-level definitions and the rule definition is that in the latter we have access to some primitive operations on definitions which are used in the communication between the rule definition and the object-level definitions.

The purpose of $\mathcal{R}$ is to give meaning to the conditions that are used in the object-level definitions. To give this meaning we define *inference rules* that interpret all the different object-level conditions. We also define *search strategies* that guide the search among the inference rules.

When defining inference rules and search strategies we use the conditions defined in Section 2.1.5. Rather than having a "meta-meta-level definition" interpreting these conditions the meaning of the conditions used in $\mathcal{R}$ is given by a sequent calculus, $\mathcal{D}OLD$, which is the same calculus that gives the operational semantics of GCLA [7].

Thus a program $\mathbf{P}$ consists of $\mathcal{R}$ and at least one object-level definition, and an interpretation of $\mathbf{P}$ is obtained only by putting the object-level definitions in the context of $\mathcal{R}$.

2.3.1. *Inference rules.* An inference rule (or simply a rule) interprets one of the object-level conditions. The general form of an inference rule is:

$$\begin{aligned} Rule(PT_1, \dots, PT_n) = \; & P_1, \dots, P_k, \\ & (PT_1 \rightarrow Seq_1), \dots, (PT_n \rightarrow Seq_n) \\ & \rightarrow Seq. \end{aligned}$$

where

- $PT_1, \ldots, PT_n$ are proof terms, i.e., partially instantiated functional expressions representing the proofs of the premises, Seq_i,
- $P_1, \ldots, P_k$ are provisos, i.e., side conditions on the applicability of the rule. k may be equal to 0,
- Seq and $Seq_1, \ldots, Seq_n$ are sequents on the form $\Gamma \vdash C$ where Γ is a finite set of conditions and C is a condition. Intuitively $\Gamma \vdash C$ could be read as: "C follows from Γ."

One way to read this is: "If $P_1, \ldots, P_k$ hold and PT_1 proves Seq_1, ..., and PT_n proves Seq_n, then $Rule(PT_1, \ldots, PT_n)$ proves Seq."

2.3.2. *Search strategies.* A search strategy (or simply a strategy) guides the search among the inference rules. A strategy has the following general form:

$$\begin{aligned} Strategy &= (P_1 \rightarrow Seq_1), \ldots, (P_n \rightarrow Seq_n). \\ Strategy &= PT_1, \ldots, PT_m. \end{aligned}$$

where

- $PT_1, \ldots, PT_m$ are proof terms,
- $P_1, \ldots, P_n$, for $n \geqslant 0$, are provisos,
- $Seq_1, \ldots, Seq_n$, for $n \geqslant 0$, are sequents.

You can read a strategy as: "If P_1 holds and some PT_i, $1 \leqslant i \leqslant m$, proves Seq_1, then $Strategy$ proves Seq_1, ..., or if P_n holds and some PT_i, $1 \leqslant i \leqslant m$, proves Seq_n, then $Strategy$ proves Seq_n."

2.3.3. *Provisos.* A proviso is a condition on the applicability of a rule or a strategy. Intuitively you can think of a proviso as a black box performing some operation that has to succeed in order for the rule or strategy to succeed. Some provisos may have side-effects like adding constraints to variables or generating substitutions.

Some of the provisos are used in the communication between $\mathcal{R}$ and the object-level definitions. These provisos are built up using a set of primitive operations on definitions which are entirely justified by the interpretation of the object-level definitions as definitions.

The most important of these primitive operations are:

- $\mathcal{D}(a)$ which gives the *definiens* of the term a in the definition $\mathcal{D}$. In short, $\mathcal{D}(a)$ is the maximal set of all bodies of clauses in $\mathcal{D}$ whose heads are unifiable with each other and with a, i.e., $\mathcal{D}(a) = \{A \mid a = A \in \mathcal{D}\}$. There can be more than one definiens for a term a.
  The reader is referred to [7] for a formal definition of $\mathcal{D}(a)$ and to [2] for details on the computational aspects of $\mathcal{D}(a)$,
- $\mathrm{Dom}(\mathcal{D})$ which gives the *domain* of a definition $\mathcal{D}$, i.e., $\mathrm{Dom}(\mathcal{D}) = \{a \mid \exists A(a = A \in \mathcal{D})\}$,
- $\mathrm{unify}(A, B)$ which is used to unify the object-level terms A and B.

Using the operations above we can for example construct a proviso for testing whether something is in the domain of a certain definition, i.e., "$a \in \mathrm{Dom}(\mathcal{D})$".

2.4. Queries. If Seq is a sequent on the form $\Gamma \vdash C$ and S is a (meta-level) condition then $S \Vdash Seq$. is a *query*.

We may interpret a query as asking for an object-level substitution σ such that $S\phi$ proves $\Gamma\sigma \vdash C\sigma$ for some meta-level substitution ϕ.

3. Program separation

3.1. A notion of form and content. In this section we are going to introduce a notion of form and content of an algorithm. We will do this by studying an example. This will lead us to a definition of an abstract recursive form of a class of very simple recursive algorithms.

3.1.1. *A motivating example.* As an example we study the canonical definition for finding the last element of a list.

Example 3.1.1. Consider the following recursive definition for finding the last element of a list:

$$\mathcal{L} \quad \begin{cases} last((x :: nil)) = x, \\ last((x :: xs)) = last(xs) \quad \text{if } xs \neq nil. \end{cases}$$

where nil denotes the empty list and $::$ is the cons operator.

The intended algorithm associated with this definition can be described as follows:

(1) if $l = (x :: nil)$, then take the definiens of $last(l)$ in $\mathcal{L}$, $\mathcal{L}(last(l))$, as the result of the computation of $last(l)$,
(2) if $l = (x :: xs)$, where $xs \neq nil$, then the result of the computation of $last(l)$ is given by computing $last(xs)$.

Now, what we want to do is to separate, so to speak, the (recursive) *form*, or the global structure, of the algorithm from the particular *content* needed to compute *last*.

To find the particular content needed to compute *last* we try to separate $\mathcal{L}$ into definitions such that the operation of finding the last element of list is *locally definable*, i.e., we try to isolate the *local operations* needed to compute *last* on the basis of $\mathcal{L}$.

Intuitively *locally defined* means that all operations needed to compute the result from an argument are present in the definitions but *global* information, like which local operation to be performed and when, are not [6].

In the case of *last*, it turns out that we do not need to separate $\mathcal{L}$ any further; whenever we reach something on the form $last((x :: nil))$ in the computation of *last*, the next thing to do is to return x, and whenever we find something on the form $last((x :: xs))$ the next step is to compute $last(xs)$. But this is precisely what $\mathcal{L}$ says, so *last* is already locally defined in $\mathcal{L}$.

To extract the recursive form of the algorithm we study the computation of $last((1::(2::(3::nil))))$. We see that all we have to do to compute $last((1::(2::(3::nil))))$ is to look up the definiens of $last((1::(2::(3::nil))))$ in $\mathcal{L}$ and recursively compute $\mathcal{L}(last((1::(2::(3::nil)))))$, i.e., $last((2::(3::nil)))$. The recursion stops when we find an expression that is not defined in $\mathcal{L}$, i.e., an element.

So in the computation of $last((1::(2::(3::nil))))$ we really only use primitive operations on definitions; looking up the definiens of an expression and checking whether a given expression is defined or not.

We may therefore reformulate the algorithm in the following manner:

$$Last(x) = \begin{cases} \mathcal{L}(x) & \text{if } \mathcal{L}(x) \not\subseteq \mathrm{Dom}(\mathcal{L}), \\ Last(\mathcal{L}(x)) & \text{if } \mathcal{L}(x) \subseteq \mathrm{Dom}(\mathcal{L}). \end{cases} \tag{1}$$

We may read (1) as: "If $\mathcal{L}(x)$, the definiens of x in $\mathcal{L}$, is not defined in $\mathcal{L}$ (the base case), then take $\mathcal{L}(x)$ as the result of the computation of $Last(x)$. Else if $\mathcal{L}(x)$ is defined in $\mathcal{L}$ (the recursive case), then the result of the computation of $Last(x)$ is given by recursively computing $Last(\mathcal{L}(x))$."

It is natural to think of this type of separation as a separation of form and content. We say that (1) gives the (recursive) *form* of the algorithm whereas $\mathcal{L}$ gives the particular *content* needed to compute *last*.

The natural way to implement a separated algorithm, as the one above for computing *last*, is to define the content as object-level definitions and to define the form by strategies and rules in the rule definition.

In the case of *last*, the implementation of $\mathcal{L}$ is straightforward:

$$\mathcal{D}_1 \quad \begin{cases} last([X]) = X. \\ last([X|Xs])\#\{Xs \neq []\} = last(Xs). \end{cases}$$

$\boldsymbol{\mathcal{R}}$ consists of a rule with two clauses corresponding to the two cases of (1):

$$\boldsymbol{\mathcal{R}} \quad \begin{cases} f(1) = (\mathcal{D}_1(X) \not\subseteq \mathrm{Dom}(\mathcal{D}_1)), \\ \qquad\quad \mathrm{unify}(\mathcal{D}_1(X), F) \rightarrow (X \vdash F). \\ f(2) = (\mathcal{D}_1(X) \subseteq \mathrm{Dom}(\mathcal{D}_1)), \\ \qquad\quad (f(\_) \rightarrow (\mathcal{D}_1(X) \vdash F)) \rightarrow (X \vdash F). \end{cases} \tag{2}$$

A typical query would then be:

$$f(\_) \Vdash last([1,2,3,4,5]) \vdash X.$$

which has the solution $X = 5$.

3.1.2. *An abstract recursive form.* In the definition of (1) in Example 3.1.1, we did not use any information about the particular definition at hand. Thus it should be possible to describe the recursive form of the algorithm for finding the last element of a list without referring to any specific definition.

We get the following generalization of (1):

$$F(x) = \begin{cases} \mathcal{D}(x) & \text{if } \mathcal{D}(x) \not\subseteq \mathrm{Dom}(\mathcal{D}), \\ F(\mathcal{D}(x)) & \text{if } \mathcal{D}(x) \subseteq \mathrm{Dom}(\mathcal{D}). \end{cases} \tag{3}$$

where $\mathcal{D}$ is some definition.

In contrast to (1), (3) does not describe the recursive form of a specific algorithm but the *abstract* recursive form of a *class* of algorithms.

Further examples of algorithms belonging to this class are the canonical definitions of enumerating the members of a list, enumerating the members of a binary tree, deciding whether an element is a member of a list or not, generating suffixes of a list, returning the element with index i from a list, and others.

Exactly what constitutes this class remains an open question.

We also generalize (2) in the same way:

$$\mathcal{R} \quad \begin{cases} f(1,\mathcal{D}) = & (\mathcal{D}(X) \not\subseteq \mathrm{Dom}(\mathcal{D})), \\ & \mathrm{unify}(\mathcal{D}(X), F) \to (X \vdash F). \\ f(2,\mathcal{D}) = & (\mathcal{D}(X) \subseteq \mathrm{Dom}(\mathcal{D})), \\ & (f(\_,\mathcal{D}) \to (\mathcal{D}(X) \vdash F)) \\ & \to (X \vdash F). \end{cases} \tag{4}$$

The rule f is now parameterized with a definition and since f itself is a definition, this is a kind of *definitional* higher order programming.

To find the last element in the list $[1, 2, 3, 4, 5]$ we would now pose the query:

$$f(\_,\mathcal{D}_1) \Vdash \mathit{last}([1,2,3,4,5]) \vdash X.$$

where $\mathcal{D}_1$ is the same definition as in Example 3.1.1.

3.2. Splitting and collecting. In many algorithms, a *splitting* and *collecting* part can be discerned. In the splitting part the computation is split up into several sub-computations and in the collecting part the results from these sub-computations are collected together to form the final result.

In some of these algorithms, the result of the collecting part is the result of the whole computation while in others some additional computing is needed in order to get the final result.

In this section we are going to study the abstract recursive forms of very simple recursive algorithms which can be described in terms of a splitting and collecting part.

As we will see these forms are natural extensions of the abstract recursive form given by (3) in Section 3.1.2.

3.2.1. *Algorithms that build structures.* Algorithms that build some kind of a data structure are very common. As an example of such an algorithm we study the canonical algorithm for appending two lists.

Example 3.2.1. Consider the following recursive definition for appending two lists:

$$\mathcal{A} \quad \begin{cases} append(nil, ys) = ys, \\ append((x :: xs), ys) = (x :: append(xs, ys)). \end{cases}$$

The algorithm normally associated with this definition can be described as follows:

(1) if $xs = nil$, then take the definiens of $append(xs, ys)$ in $\mathcal{A}$ as the result of the computation of $append(xs, ys)$,
(2) if $xs = (z :: zs)$, then the result of the computation of $append(xs, ys)$ is given by computing $append(zs, ys)$ and then applying $::$ to z and the result.

The algorithm described above is really *implicit* in $\mathcal{A}$ since there is nothing in $\mathcal{A}$ that indicates that we must first look inside $(x :: append(xs, ys))$ to compute $append(xs, ys)$ and then apply $::$ to x and the result. What we like to do is to make the algorithm a more *explicit* part of $\mathcal{A}$.

We do this by applying the program separation scheme outlined in Section 3.1, i.e., we try to separate the recursive form of the algorithm from the particular content needed to compute *append*.

One way to find the particular content needed to compute *append* is to separate $\mathcal{A}$ into the following definitions:

$$\mathcal{A}_1 \quad \begin{cases} append(nil, ys) = ys, \\ append((x :: xs), ys) = x, \\ append((x :: xs), ys) = append(xs, ys). \end{cases}$$

$$\mathcal{A}_2 \quad \begin{cases} x = (x :: xs) & \text{if } x \neq nil \wedge x \neq (y :: ys), \\ xs = (x :: xs) & \text{if } xs = nil \vee xs = (y :: ys). \end{cases}$$

The intuition behind $\mathcal{A}_1$ is that whenever we reach something on the form $append(nil, ys)$ in the computation of *append* then we return ys and whenever we find something on the form $append((x :: xs), ys)$ the computation is split up into two sub-computations: x and $append(xs, ys)$.

Similarly, the intuition behind $\mathcal{A}_2$ is that whenever we reach something that is not a list, i.e., an element, then we construct a list from that element and some list xs and whenever we find something that is a list then we construct a new list from that list and some element x. $\mathcal{A}_2$ can be seen as the reverse of the inductive definition of lists of a given type.

To find the recursive form of the algorithm we notice that to compute for example $append((1 :: (2 :: nil)), (3 :: nil))$ all we have to do is to look up the definiens of $append((1::(2::nil)),(3::nil))$ in $\mathcal{A}_1$ and continue with the subcomputations 1 and $append((2::nil),(3::nil))$. Now we should put 1 in front of a list, namely the list we get by appending $(3::nil)$ to $(2::nil)$, and we can do that by simply looking up the definiens of 1 in $\mathcal{A}_2$. When the recursive computation of $append((2::nil),(3::nil))$ eventually returns, we should put an element, i.e., 1, in front of the resulting list and we can do that by simply looking up the definiens of the resulting list in $\mathcal{A}_2$. The recursion stops when we find something that is not defined in $\mathcal{A}_1$, i.e., an element.

So in the computation of $append((1::(2::nil)),(3::nil))$ we really only use primitive operations on definitions and we may therefore formulate the algorithm as follows:

$$Append(x) = \begin{cases} \mathcal{A}_1(x) & \text{if } \mathcal{A}_1(x) \cap \mathrm{Dom}(\mathcal{A}_1) = \emptyset, \\ \mathcal{A}_2(Append(\mathcal{A}_1(x))) & \text{if } \mathcal{A}_1(x) \cap \mathrm{Dom}(\mathcal{A}_1) \neq \emptyset. \end{cases} \tag{5}$$

We say that (5) gives the *recursive form* of the algorithm whereas $\mathcal{A}_1$ and $\mathcal{A}_2$ together give the particular *content* needed to compute *append*.

$\mathcal{A}_1$ and $\mathcal{A}_2$ are implemented by two object-level definitions:

$$\mathcal{D}_3 \quad \begin{cases} append([],Ys) = Ys. \\ append([X|Xs],Ys) = X. \\ append([X|Xs],Ys) = append(Xs,Ys). \end{cases}$$

$$\mathcal{D}_4 \quad \begin{cases} X\#\{X \neq [] \wedge X \neq [Y|Ys]\} = [X|Xs]. \\ Xs\#\{Xs = [] \vee Xs = [Y|Ys]\} = [X|Xs]. \end{cases}$$

The form (5) is implemented by the following rule definition:

$$\mathcal{R} \quad \begin{cases} f(1) = (\mathcal{D}_3(X) \cap \mathrm{Dom}(\mathcal{D}_3) = \emptyset), \\ \qquad\quad \mathrm{unify}(\mathcal{D}_3(X), F) \rightarrow (X \vdash F). \\ f(2) = (\mathcal{D}_3(X) \cap \mathrm{Dom}(\mathcal{D}_3) \neq \emptyset), \\ \qquad\quad (\{A_1, \ldots, A_n\} = \mathcal{D}_3(X)), \\ \qquad\quad (g(\_) \rightarrow (A_1 \vdash F)), \ldots, (g(\_) \rightarrow (A_n \vdash F)) \rightarrow (X \vdash F). \\ \\ g(1) = (X \notin \mathrm{Dom}(\mathcal{D}_3)), \\ \qquad\quad \mathrm{unify}(\mathcal{D}_4(X), F) \rightarrow (X \vdash F). \\ g(2) = (X \in \mathrm{Dom}(\mathcal{D}_3)), \\ \qquad\quad (f(\_) \rightarrow (X \vdash FX)), \\ \qquad\quad \mathrm{unify}(\mathcal{D}_4(FX), F) \rightarrow (X \vdash F). \end{cases} \tag{6}$$

The first clause of the rule f corresponds to the base case. If the definiens of an expression consists of more than one member we evaluate all the members to the same value. This is to ensure that the two lists created from the element and the recursive call respectively are equal. This is taken care of in the second clause of f. Thus, it is in this rule that the collecting is performed.

The two clauses of the rule g correspond to the second case in (5).

To concatenate the lists $[1,2]$ and $[3,4]$ we pose the query:

$$f(\_) \Vdash append([1,2],[3,4]) \vdash X.$$

which has the solution $X = [1,2,3,4]$.

What happens in the algorithm given by (5) above is that we use $\mathcal{D}_3$ to present values and recursive calls and then use $\mathcal{D}_4$ to build a structure based on the number of recursive calls.

3.2.2. *Computing in the collecting part.* So far we have only considered examples where the collecting part consists of building a structure on basis of a given datatype. In this section we consider a simple example where we actually compute in the collecting part.

Example 3.2.2. Consider the following (naive) recursive definition for reversing a list:

$$\mathcal{R} \quad \begin{cases} reverse(nil) \;=\; nil, \\ reverse((x::xs)) \;=\; append(reverse(xs),(x::nil)). \end{cases}$$

where *append* is the operation of appending two lists.

The intended algorithm associated with this definition can be described as follows:

(1) if $xs = nil$, then take the definiens of $reverse(xs)$ in $\mathcal{R}$, $\mathcal{R}(reverse(xs))$, as the result of the computation of $reverse(xs)$,
(2) if $xs = (y::ys)$, then the result of the computation of $reverse(xs)$ is given by computing $reverse(ys)$ and then appending $(x::nil)$ to the result.

In order to make the above algorithm a more explicit part of $\mathcal{R}$ we try to separate $\mathcal{R}$ into definitions such that *reverse* is locally definable. One way to do this is to separate $\mathcal{R}$ into the following definitions:

$$\mathcal{R}_1 \quad \begin{cases} reverse(nil) \;=\; nil, \\ reverse((x::xs)) \;=\; x, \\ reverse((x::xs)) \;=\; reverse(xs). \end{cases}$$

$$\mathcal{R}_2 \quad \begin{cases} x \;=\; append(xs,(x::nil)) & \text{if } x \neq nil \wedge x \neq (y::ys), \\ xs \;=\; append(xs,ys) & \text{if } xs = nil \vee xs = (y::ys). \end{cases}$$

We can understand $\mathcal{R}_1$ as: "If we reach something on the form $reverse(nil)$ in the computation of *reverse* then we return *nil* and if we find something on the form $reverse((x::xs))$ the computation is split up into two computations: x and $reverse(xs)$."

Similarly, we can understand $\mathcal{R}_2$ as: "If we reach something that is not a list, i.e., an element, then we construct an append structure from some list xs and the list consisting of that element, and if we find something that is a list then we construct an append structure from that list and another list ys."

It is easy to see that the recursive form of this algorithm must be very similar to (5); in both cases we use the two results from the recursive step to build a data structure. The important difference is that in the case of reversing a list the data structure is intermediate—it is not the final result of the computation—we need to do some additional computing on the structure, i.e., evaluating the append structure to a list.

We get the following recursive form of the algorithm:

$$Rev(x) = \begin{cases} \mathcal{R}_1(x) & \text{if } \mathcal{R}_1(x) \cap \mathrm{Dom}(\mathcal{R}_1) = \emptyset, \\ Append(\mathcal{R}_2(Rev(\mathcal{R}_1(x)))) & \text{if } \mathcal{R}_1(x) \cap \mathrm{Dom}(\mathcal{R}_1) \neq \emptyset. \end{cases} \tag{7}$$

where *Append* is the algorithm for appending two lists.

Perhaps to gain some additional intuition of what this program separation is all about, one may think of the content of the algorithm as the part of the algorithm that defines on *which* objects the next step of the algorithm should act upon, and the form of the algorithm as the part that defines *what* exactly is the next step of the algorithm.

If we translate this algorithm into a program, $\mathcal{R}_1$ and $\mathcal{R}_2$ are implemented by the following two object-level definitions:

$$\mathcal{D}_5 \quad \begin{cases} reverse([]) = []. \\ reverse([X|Xs]) = X. \\ reverse([X|Xs]) = reverse(Xs). \end{cases}$$

$$\mathcal{D}_6 \quad \begin{cases} X\#\{X \neq [] \wedge X \neq [Y|Ys]\} = append(Xs, [X]). \\ Xs\#\{Xs = [] \vee Xs = [Y|Ys]\} = append(Xs, Ys). \end{cases}$$

The form (7) is implemented by the following rule definition:

$$\mathcal{R} \quad \begin{cases} f(1) = (\mathcal{D}_5(X) \cap \mathrm{Dom}(\mathcal{D}_5) = \emptyset), \\ \qquad \mathrm{unify}(\mathcal{D}_5(X), F) \to (X \vdash F). \\ f(2) = (\mathcal{D}_5(X) \cap \mathrm{Dom}(\mathcal{D}_5) \neq \emptyset), \\ \qquad (\{A_1, \ldots, A_n\} = \mathcal{D}_5(X)), \\ \qquad (g(\_) \to (A_1 \vdash FX)), \ldots, (g(\_) \to (A_n \vdash FX)), \\ \qquad (append \to (FX \vdash F)) \to (X \vdash F). \\ \\ g(1) = (X \notin \mathrm{Dom}(\mathcal{D}_5)), \\ \qquad \mathrm{unify}(\mathcal{D}_6(X), F) \to (X \vdash F). \\ g(2) = (X \in \mathrm{Dom}(\mathcal{D}_5)), \\ \qquad (f(\_) \to (X \vdash FX)), \\ \qquad \mathrm{unify}(\mathcal{D}_6(FX), F) \to (X \vdash F). \end{cases} \tag{8}$$

where *append* is some strategy or rule defined elsewhere for appending two lists, e.g., the rule f in (6).

This rule definition is essentially the same as (6) but with the important difference that we use an external strategy (or rule) *append* in the second clause of f for evaluating the intermediate append structure built by g to a list.

If we would like to reverse the list $[1,2,3,4,5]$ we can pose a query like:

$$f(\_) \Vdash reverse([1,2,3,4,5]) \vdash X.$$

which has the solution $X = [5,4,3,2,1]$.

3.2.3. *Abstract recursive forms.* In the definitions of (5) and (7) in Example 3.2.1 and Example 3.2.2 respectively, we did not use any information about the particular definitions at hand. So it should be possible to describe the recursive form of the algorithms for computing *append* and *reverse* without referring to any specific definition.

Consider the following generalization of (5):

$$F(x) = \begin{cases} \mathcal{D}_1(x) & \text{if } \mathcal{D}_1(x) \cap \mathrm{Dom}(\mathcal{D}_1) = \emptyset, \\ \mathcal{D}_2(F(\mathcal{D}_1(x))) & \text{if } \mathcal{D}_1(x) \cap \mathrm{Dom}(\mathcal{D}_1) \neq \emptyset. \end{cases} \tag{9}$$

where $\mathcal{D}_1$ and $\mathcal{D}_2$ are some definitions.

Just like (3) in Section 3.1.2, (9) describes the abstract recursive form of a *class* of algorithms. Further examples of algorithms belonging to this class are the canonical definitions of finding the maximum and minimum of two natural numbers, computing the length of a list, substituting one element for another in a list, duplicating all the members in a list etc.

We also generalize the rule definition in Example 3.2.1 in the same way:

$$\mathcal{R} \quad \begin{cases} f(1,\mathcal{D}_1,\mathcal{D}_2) = & (\mathcal{D}_1(X) \cap \mathrm{Dom}(\mathcal{D}_1) = \emptyset), \\ & \mathrm{unify}(\mathcal{D}_1(X), F) \rightarrow (X \vdash F). \\ f(2,\mathcal{D}_1,\mathcal{D}_2) = & (\mathcal{D}_1(X) \cap \mathrm{Dom}(\mathcal{D}_1) \neq \emptyset), \\ & (\{A_1,\ldots,A_n\} = \mathcal{D}_1(X)), \\ & (g(\_,\mathcal{D}_1,\mathcal{D}_2) \rightarrow (A_1 \vdash F)), \\ & \ldots, \\ & (g(\_,\mathcal{D}_1,\mathcal{D}_2) \rightarrow (A_n \vdash F)) \\ & \rightarrow (X \vdash F). \\ & \\ g(1,\mathcal{D}_1,\mathcal{D}_2) = & (X \notin \mathrm{Dom}(\mathcal{D}_1)), \\ & \mathrm{unify}(\mathcal{D}_2(X), F) \rightarrow (X \vdash F). \\ g(2,\mathcal{D}_1,\mathcal{D}_2) = & (X \in \mathrm{Dom}(\mathcal{D}_1)), \\ & (f(\_,\mathcal{D}_1,\mathcal{D}_2) \rightarrow (X \vdash FX)), \\ & \mathrm{unify}(\mathcal{D}_2(FX), F) \rightarrow (X \vdash F). \end{cases} \tag{10}$$

The rules f and g are now a kind of higher order definitions since they are parameterized with two definitions $\mathcal{D}_1$ and $\mathcal{D}_2$.

To concatenate the two lists $[1,2]$ and $[3,4]$ we would now pose the query:

$$f(\_,\mathcal{D}_3,\mathcal{D}_4) \Vdash append([1,2],[3,4]) \vdash X.$$

where $\mathcal{D}_3$ and $\mathcal{D}_4$ are the same definitions as in Example 3.2.1.

Similarly we get the following generalization of (7):

$$F(x) = \begin{cases} \mathcal{D}_1(x) & \text{if } \mathcal{D}_1(x) \cap \mathrm{Dom}(\mathcal{D}_1) = \emptyset, \\ G(\mathcal{D}_2(F(\mathcal{D}_1(x)))) & \text{if } \mathcal{D}_1(x) \cap \mathrm{Dom}(\mathcal{D}_1) \neq \emptyset. \end{cases} \tag{11}$$

where $\mathcal{D}_1$ and $\mathcal{D}_2$ are some definitions and G is some function defined elsewhere.

The form (11) describes the abstract recursive form of a class of algorithms. Further examples of algorithms belonging to this class are the canonical definitions of flattening a list, computing the factorial of a natural number, computing the fibonacci-numbers, computing the sum and product of all the members in a list of natural numbers, listing a binary tree in pre-, in-, and post-order checking if a binary tree is full etc.

We also generalize the rule definition in Example 3.2.2 in the same way:

$$\mathcal{R} \quad \begin{cases} \begin{aligned} f(1, \mathcal{D}_1, \mathcal{D}_2, PT) = {} & (\mathcal{D}_1(X) \cap \mathrm{Dom}(\mathcal{D}_1) = \emptyset), \\ & \mathrm{unify}(\mathcal{D}_1(X), F) \\ & \rightarrow (X \vdash F). \\ f(2, \mathcal{D}_1, \mathcal{D}_2, PT) = {} & (\mathcal{D}_1(X) \cap \mathrm{Dom}(\mathcal{D}_1) \neq \emptyset), \\ & (\{A_1, \ldots, A_n\} = \mathcal{D}_1(X)), \\ & (g(\_, \mathcal{D}_1, \mathcal{D}_2, PT) \rightarrow (A_1 \vdash FX)), \\ & \ldots, \\ & (g(\_, \mathcal{D}_1, \mathcal{D}_2, PT) \rightarrow (A_n \vdash FX)), \\ & (PT \rightarrow (FX \vdash F)) \\ & \rightarrow (X \vdash F). \\ \\ g(1, \mathcal{D}_1, \mathcal{D}_2, PT) = {} & (X \notin \mathrm{Dom}(\mathcal{D}_1)), \\ & \mathrm{unify}(\mathcal{D}_2(X), F) \\ & \rightarrow (X \vdash F). \\ g(2, \mathcal{D}_1, \mathcal{D}_2, PT) = {} & (X \in \mathrm{Dom}(\mathcal{D}_1)), \\ & (f(\_, \mathcal{D}_1, \mathcal{D}_2, PT) \rightarrow (X \vdash FX)), \\ & \mathrm{unify}(\mathcal{D}_2(FX), F) \\ & \rightarrow (X \vdash F). \end{aligned} \end{cases} \tag{12}$$

Since the definitions f and g are parameterized with two definitions, $\mathcal{D}_1$ and $\mathcal{D}_2$, this is yet another example of how this type of program separation can be used as a basis for definitional higher order programming.

To reverse the list $[1, 2, 3, 4, 5]$ we pose the query:

$$f(\_, \mathcal{D}_5, \mathcal{D}_6, append) \Vdash reverse([1, 2, 3, 4, 5]) \vdash X.$$

where $\mathcal{D}_5$ and $\mathcal{D}_6$ are the same definitions as in Example 3.2.2 and *append* is some strategy or rule defined elsewhere for appending two lists, e.g., the rule f in (6).

References

1. M. Aronsson, *GCLA user's manual*, Research Report T91:21A, SICS, 1991.
2. ———, *GCLA, the design, use, and implementation of a program development system*, Ph.D. thesis, Department of Computer and Systems Sciences, University of Stockholm, 1993.
3. M. Aronsson, L-H. Eriksson, A. Gäredal, L. Hallnäs, and P. Olin, *The programming language GCLA—a definitional approach to logic programming*, New Generation Computing **7** (1990), no. 4, 381–404.
4. M. Aronsson, P. Kreuger, L. Hallnäs, and L-H. Eriksson, *A survey of GCLA—a definitional approach to logic programming*, Extensions of Logic Programming, Proceedings of the 1st International Workshop held at the SNS, Universität Tübingen, 1989 (P. Shroeder-Heister, ed.), Springer Lecture Notes in Artificial Intelligence, vol. 475, Springer-Verlag, 1990.
5. G. Falkman, L. Hallnäs, and O. Torgersson, *Program separation in GCLA*, Proceedings of the Post-Conference Workshop on Proof-Theoretical Extensions of Logic Programming, Santa Margherita Ligure, Italy, 18 June 1994 (A. Momigliano and M. Ornaghi, eds.), June 1994, pp. 31–37.
6. L. Hallnäs, *WM94: program separation in GCLA*, Proceedings of La Wintermöte '94, Lundsbrunn, January 18–20, January 1994, pp. 93–94.
7. P. Kreuger, *GCLA II, a definitional approach to control*, Extensions of Logic Programming, Proceedings of the 2nd International Workshop held at SICS, Sweden, 1991 (L-H. Eriksson, L. Hallnäs, and P. Shroeder-Heister, eds.), Springer Lecture Notes in Artificial Intelligence, vol. 596, Springer-Verlag, 1992, pp. 239–297.

Department of Computing Science, Chalmers University of Technology, S-412 96 Göteborg, Sweden

E-mail address: falkman@cs.chalmers.se

General Algebras

Øyvind B. Fredriksen

University of Bergen
Department of Informatics*

Abstract

We study categories of algebras (for a given signature) constructed from arbitrary categories C with specific finite products. We show how a left adjoint $Set \longrightarrow C$ may be utilized to construct an initial algebra satisfying a given set of equations. Finally, we give a condition under which satisfaction of equations is independent of the set of variables considered.

1 Introduction

This paper reports work in progress. It is concerned with specifications of abstract data types and in particular with the *semantics* of such specifications. Here we will only consider *one-sorted equational* specifications or *presentations*, having two parts:

- the names (called *operation symbols*) of the *operations* involved, together with their arities
- *equations* describing the *properties* of the operations. The two sides of an equation are *terms* constructed from the operation symbols and usually some *variables*.

As for semantics, a presentation is usually interpreted in so-called *total algebras*. Such an algebra comprises a *carrier* set AU_Σ, and for each n-ary operation symbol a function ${AU_\Sigma}^n \longrightarrow AU_\Sigma$. Terms and equations may then be interpreted, and only those algebras *satisfying* the equations are considered. These algebras are the objects of a *category*[1]. Of particular interest are those algebras which are *initial* in this category. The initial algebras identify only those terms whose equality follows from the equations.

However, also other semantic models have been suggested: partial algebras ([13]), ordered/monotonic algebras ([10]), continuous algebras ([6]), and topological algebras ([9]).

As a vehicle for bridging the gap between specification of abstract data types and of systems of parallel processes, one might also consider total algebras

*Postal address: ..., Høyteknologisenteret, N–5020 BERGEN, NORWAY. E-mail: bolme@ii.uib.no

[1]This paper assumes a fair knowledge of category theory. See e.g. [7].

whose carrier elements are models of parallel computation, e.g. partial orders representing the causal order of the events performed by the system.

The *theories* of the various kinds of models are fairly similar: In each case, similar theorems are proved in similar ways. To simplify and clarify matters and to facilitate the consideration of *novel* models, it might be useful to construct a *general* theory subsuming (as many as possible of) the others as special cases.

In [1] we presented our first steps towards this goal. We made a construction based on the following parameters:

- a Cartesian category C
- finite product preserving functors $U : C \longrightarrow Set$ and $F : Set \longrightarrow C$ such that $F \dashv U$ and[2] $F \circ U = I_{Set}$.

However, we felt that this last condition was rather *ad hoc* and too restrictive. In this paper we refine our approach in such a way that the abovementioned restriction becomes unnecessary.

In [1] we demonstrated that our approach subsumes the theories of total and ordered algebras (in the sense that it permits the construction of an initial algebra satisfying a given set of equations). For some speculations on the success or failure of our theory in some of the other cases mentioned above, we refer the reader to the same paper. The same goes for a more detailed introduction.

The rest of the paper is organized as follows: In section 2 we review the theory of total algebras. We present our general theory in section 3. In section 4 we make a few concluding remarks.

In order to structure the next two sections, we consider the successive introduction of the two parts of a presentation: operation symbols and equations.

Omitted proofs are entirely straightforward.

1.1 Notational and Other Conventions

If C is a category, its object class is denoted by C_0 and the (hom-)class of arrows $A \longrightarrow B$ by $C(A, B)$. A finite (product) mediating arrow is denoted by $\langle f_0, \ldots, f_{n-1} \rangle$.

A natural number is the set of the smaller ones, i.e. $n = \{0, \ldots, n-1\}$. In particular, the number 0 is the empty set.

A function space $Set(A, B)$ may also be denoted by ${}^{A}B$, and may alternatively be regarded as an 'A-fold product of B-s'. If A is a natural number n, we will usually use the notation B^n. If $b \in B^n$ and $i \in n$, we usually (but not always) write b_i for ib.

2 Total Algebras

In this section we briefly review the relevant parts of the theory of total algebras. A fuller account is given in e.g. [5]. We also give some lemmas needed later.

[2] Please note that we write composition in an arbitrary category in *diagrammatic* order, i.e. the composite arrow $\xrightarrow{f}\xrightarrow{g}$ is denoted by $f \circ g$. In accordance with this, application of a 'function' (e.g. a functor) f to an 'argument' x is denoted by xf. This application is left associative.

2.1 Signatures

A (*one-sorted*) *signature* Σ is an assignment of a set Σ_n (of *n-ary operation symbols*) to each natural number n. A 0-ary operation symbol is called a *constant symbol.*

If Σ and Σ' are signatures, Σ' is said to *include* Σ, denoted by $\Sigma \subseteq \Sigma'$, if $\Sigma_n \subseteq \Sigma'_n$ for all n.

A *signature chain* is an $\mathbf{N}$-indexed sequence $\Sigma_0, \Sigma_1, \Sigma_2, \ldots$ of signatures such that each $\Sigma_n \subseteq \Sigma_{n+1}$. (A *set chain* is an $\mathbf{N}$-indexed sequence $V_0, V_1, V_2, \ldots$ of sets such that each $V_n \subseteq V_{n+1}$.)

2.1.1 Union of Signatures

Let

- I be a set
- for each $i \in I$ Σ_i be a signature.

Then the *union* (*signature*) $\bigcup_{i \in I} \Sigma_i$ of the Σ_i is defined by

$$\left(\bigcup_{i \in I} \Sigma_i\right)_n = \bigcup_{i \in I} (\Sigma_i)_n .$$

Lemma 1 *Each* $\Sigma_i \subseteq \bigcup_{i \in I} \Sigma_i$.

2.1.2 Algebras and Homomorphisms

Let Σ be a signature. The category Set^{Σ} of *total* Σ*-algebras* and *total* Σ*-homomorphisms* is defined by

- A total Σ-algebra A comprises
  - a *carrier* set AU_Σ
  - for each $n \in \mathbf{N}$ and $f \in \Sigma_n$ a function $f_A : {AU_\Sigma}^n \longrightarrow AU_\Sigma$.
- A total Σ-homomorphism $h : A \longrightarrow B$ is a function $AU_\Sigma \longrightarrow BU_\Sigma$ such that

  (1) $$\forall n \in \mathbf{N}. \forall f \in \Sigma_n . f_A \circ h = \underbrace{h \times \cdots \times h}_{n} \circ f_B ,$$

  which is equivalent to that for all $n \in \mathbf{N}$, $f \in \Sigma_n$, and $a \in {AU_\Sigma}^n$ $a f_A h = (a_0 h, \ldots, a_{n-1} h)\, f_B$.
- Composition is composition in *Set*.
- The identities are defined by $id_A = id_{AU_\Sigma}$.

Lemma 2 *Assume that* Σ *contains a constant symbol* $c \in \Sigma_0$ *and let* A *be a total* Σ*-algebra. Then* AU_Σ *is non-empty.*

Proof: The function $c_A : {AU_\Sigma}^0 \longrightarrow AU_\Sigma$, where ${AU_\Sigma}^0$ is a singleton, entailing that $* c_A \in AU_\Sigma$. ∎

2.1.3 The Forgetful Functor $Set^{\Sigma} \longrightarrow Set$

Extend the assignment $A \mapsto AU_{\Sigma}$ to a functor $U_{\Sigma} : Set^{\Sigma} \longrightarrow Set$ by $hU_{\Sigma} = h$ on arrows.

Even though $hU_{\Sigma} = h$, we will usually write hU_{Σ} when we think of it as a function.

2.1.4 Ground Terms

Let T_{Σ} be the least language (with the inclusion order) such that

$$T_{\Sigma} = \left\{ f\,(t_0, \ldots, t_{n-1}) \,\middle|\, n \in \mathbf{N}\ \mathbf{cand}\ f \in \Sigma_n \wedge \bigwedge_{i \in n} t_i \in T_{\Sigma} \right\}. \tag{2}$$

(When f is a constant symbol, $f\,()$ is written simply f, entailing that $\Sigma_0 \subseteq T_{\Sigma}$.) Each string in the language T_{Σ} is called a *ground term.*

This definition permits us to carry out proofs by *induction* on the structure of ground terms.

Lemma 3 *T_{Σ} is empty iff Σ_0 is.*

Proof: As 0 is the least language, the least language T_{Σ} such that (2) is 0 iff

$$\begin{aligned} 0 &= \left\{ f\,(t_0, \ldots, t_{n-1}) \,\middle|\, n \in \mathbf{N}\ \mathbf{cand}\ f \in \Sigma_n \wedge \bigwedge_{i \in n} t_i \in 0 \right\} \\ &= \Sigma_0. \end{aligned}$$

∎

Lemma 4 *The operation $\Sigma \mapsto T_{\Sigma}$ is monotone w.r.t. (signature and set) inclusion.*

2.1.4.1 Union of Ground Term Languages

Lemma 5 *Let*

- *I be a set*
- *for each $i \in I$ Σ_i be a signature.*

Then $\bigcup_{i \in I} T_{\Sigma_i} \subseteq T_{\bigcup_{i \in I} \Sigma_i}$.

Proof: Straightforward, using lemmata 4 and 1. ∎

Lemma 6 *Let $\Sigma_0 \subseteq \Sigma_1 \subseteq \cdots$ be a signature chain. Then*

$$T_{\bigcup_{n \in \mathbf{N}} \Sigma_n} = \bigcup_{n \in \mathbf{N}} T_{\Sigma_n}.$$

Proof: We have to prove that $T_{\bigcup_{n\in\mathbf{N}}\Sigma_n} \subseteq \bigcup_{n\in\mathbf{N}} T_{\Sigma_n}$. We will prove that $t \in T_{\bigcup_{n\in\mathbf{N}}\Sigma_n} \Rightarrow t \in \bigcup_{n\in\mathbf{N}} T_{\Sigma_n}$ by induction on t. We have to prove that $f(t_0,\ldots,t_{m-1}) \in T_{\bigcup_{n\in\mathbf{N}}\Sigma_n} \Rightarrow f(t_0,\ldots,t_{m-1}) \in \bigcup_{n\in\mathbf{N}} T_{\Sigma_n}$. Therefore, assume the antecedent, which is equivalent to that

- $$\begin{aligned} f &\in \left(\bigcup_{n\in\mathbf{N}} \Sigma_n\right)_m \\ &= \bigcup_{n\in\mathbf{N}} (\Sigma_n)_m, \end{aligned}$$

  which is equivalent to that $\exists n \in \mathbf{N}. f \in (\Sigma_n)_m$. Therefore, let $n \in \mathbf{N}$ be such that $f \in (\Sigma_n)_m$.

- $\bigwedge_{i\in m} t_i \in T_{\bigcup_{n\in\mathbf{N}}\Sigma_n}$, where $t_i \in T_{\bigcup_{n\in\mathbf{N}}\Sigma_n}$ entails that $t_i \in \bigcup_{n\in\mathbf{N}} T_{\Sigma_n}$, which is equivalent to that $\exists n \in \mathbf{N}. t_i \in T_{\Sigma_n}$. Therefore, let for each $i \in m$ $n_i \in \mathbf{N}$ be such that $t_i \in T_{\Sigma_{n_i}}$.

We have to prove the consequent, which is equivalent to that there is an $n \in \mathbf{N}$ such that $f(t_0,\ldots,t_{m-1}) \in T_{\Sigma_n}$.

We claim that $f(t_0,\ldots,t_{m-1}) \in T_{\Sigma_{\max\{n,n_0,\ldots,n_{m-1}\}}}$, which is straightforward to prove, using lemma 4. ∎

2.1.5 Extensions

Let Σ be a signature such that Σ_0 is a 'special' set, disjoint from those in Set_0.

If V is a set, define a signature $\Sigma(V)$ (the *V-extension of Σ*) by

- $\Sigma(V)_0 = \Sigma_0 \cup V$
- $\Sigma(V)_n = \Sigma_n$ otherwise.

Lemma 7 *The extension operation $V \mapsto \Sigma(V)$*

1. *is monotone w.r.t. (set and signature) inclusion*
2. *preserves chains, i.e. takes set chains to signature chains*
3. *preserves/distributes over (arbitrary) unions: $\Sigma(\bigcup_{i\in I} V_i) = \bigcup_{i\in I} \Sigma(V_i)$.*

2.1.6 The Free Algebra Functor

The functor U_Σ has a left adjoint $F_\Sigma : Set \longrightarrow Set^\Sigma$ whose object part is defined by: Let V be a *Set*-object. The Set^Σ-object VF_Σ is defined by

- The set $VF_\Sigma U_\Sigma = T_{\Sigma(V)}$.
- For each $n \in \mathbf{N}$ and $f \in \Sigma_n$ the function $f_{VF_\Sigma} : VF_\Sigma U_\Sigma{}^n \longrightarrow VF_\Sigma U_\Sigma$, i.e. $T_{\Sigma(V)}{}^n \longrightarrow T_{\Sigma(V)}$, is defined by $t f_{VF_\Sigma} = f(t_0,\ldots,t_{n-1})$ (the *string*).

Each string in the language $VF_\Sigma U_\Sigma = T_{\Sigma(V)}$ is called a V-*term*.

In particular,

$$\begin{aligned} V &\subseteq \Sigma_0 \cup V \\ &= \Sigma(V)_0 \\ &\subseteq T_{\Sigma(V)}, \end{aligned}$$

so V-elements ('-variables') are V-terms. Let η_V denote the inclusion function $V \longrightarrow VF_\Sigma U_\Sigma$. Even though each $v\eta_V = v$, we usually write $v\eta_V$ when we think of it as a term.

Each V-term is on exactly one of two forms:

- $v\eta_V$
- $f(t_0, \ldots, t_{n-1})$, where n is a natural number, f is an n-ary operation symbol, and $t_0, \ldots, t_{n-1}$ are V-terms.

This allows for inductive definitions and proofs.

If A is a total Σ-algebra, a function $h : VF_\Sigma U_\Sigma \longrightarrow AU_\Sigma$ is a total Σ-homomorphism $VF_\Sigma \longrightarrow A$ iff for all $n \in \mathbf{N}$, $f \in \Sigma_n$, and $t \in VF_\Sigma U_\Sigma{}^n$ $[f(t_0, \ldots, t_{n-1})]\, h = (t_0 h, \ldots, t_{n-1} h)\, f_A$.

The functor F_Σ being a left adjoint of U_Σ means that each VF_Σ is free over V w.r.t. U_Σ. The universal arrow is the inclusion η_V.

As a special case, $0F_\Sigma$ is free over the initial *Set*-object 0, i.e. is itself initial (in Set^Σ).

VF_Σ being free over V w.r.t. U_Σ with universal arrow η_V means that for each Set^Σ-object A and *Set*-arrow $f : V \longrightarrow AU_\Sigma$ there exists a unique Set^Σ-arrow (denoted by) $f^\# : VF_\Sigma \longrightarrow A$ such that $\eta_V \circ f^\# U_\Sigma = f$.

2.1.6.1 Term Languages

The term language $VF_\Sigma U_\Sigma = T_{\Sigma(V)}$ is empty iff $\Sigma(V)_0 = \Sigma_0 \cup V$ is, i.e. iff both Σ_0 and V are.

Corollary 1 *Let V and V' be sets such that $V \subseteq V'$. Then $VF_\Sigma U_\Sigma \subseteq V'F_\Sigma U_\Sigma$.*

Proof: Straightforward, using lemmata 4 and 7, part 1.. ∎

2.1.6.1.1 Union of Term Languages

Corollary 2 *Let*

- *I be a set*
- *for each $i \in I$ V_i be a set.*

Then $\bigcup_{i \in I} V_i F_\Sigma U_\Sigma \subseteq \left(\bigcup_{i \in I} V_i\right) F_\Sigma U_\Sigma$.

Proof: Straightforward, using corollary 1. ∎

Lemma 8 *Let $V_0 \subseteq V_1 \subseteq \cdots$ be a set chain. Then*

$$\left(\bigcup_{n \in \mathbf{N}} V_n\right) F_\Sigma U_\Sigma = \bigcup_{n \in \mathbf{N}} V_n F_\Sigma U_\Sigma.$$

Proof: Straightforward, using lemmata 6 and 7, parts 2. and 3.. ∎

2.2 Presentations

2.2.1 Equations

If V is a set, a *V-equation* is a pair of V-terms, i.e. an element of $VF_\Sigma U_\Sigma{}^2$. An *equation* is a V-equation for *some* set V.

2.2.1.1 Union of Equation Sets

Corollary 3 *Let*

- *I be a set*
- *for each $i \in I$ V_i be a set.*

Then $\bigcup_{i\in I} V_i F_\Sigma U_\Sigma{}^2 \subseteq \left(\bigcup_{i\in I} V_i\right) F_\Sigma U_\Sigma{}^2$.

Proof: Straightforward, using corollary 1. ∎

Lemma 9 *Let* $V_0 \subseteq V_1 \subseteq \cdots$ *be a set chain. Then* $\left(\bigcup_{n\in\mathbf{N}} V_n\right) F_\Sigma U_\Sigma{}^2 = \bigcup_{n\in\mathbf{N}} V_n F_\Sigma U_\Sigma{}^2$.

Proof: By corollary 3 and lemma 8 we have to prove that $\left(\bigcup_{n\in\mathbf{N}} V_n F_\Sigma U_\Sigma\right)^2 \subseteq \bigcup_{n\in\mathbf{N}} V_n F_\Sigma U_\Sigma{}^2$. Therefore, let $(t,u) \in \left(\bigcup_{n\in\mathbf{N}} V_n F_\Sigma U_\Sigma\right)^2$, which is equivalent to that $t, u \in \bigcup_{n\in\mathbf{N}} V_n F_\Sigma U_\Sigma$, where $t \in \bigcup_{n\in\mathbf{N}} V_n F_\Sigma U_\Sigma$ iff $\exists n \in \mathbf{N}. t \in V_n F_\Sigma U_\Sigma$. Therefore, let $n \in \mathbf{N}$ be such that $t \in V_n F_\Sigma U_\Sigma$. Similarly, let $m \in \mathbf{N}$ be such that $u \in V_m F_\Sigma U_\Sigma$.

We have to prove that $(t,u) \in \bigcup_{n\in\mathbf{N}} V_n F_\Sigma U_\Sigma{}^2$, which is equivalent to that $\exists n \in \mathbf{N}. (t,u) \in V_n F_\Sigma U_\Sigma{}^2$. We claim that $(t,u) \in V_{\max\{m,n\}} F_\Sigma U_\Sigma{}^2$, which is straightforward to prove, using corollary 1. ∎

2.2.2 Satisfaction

Let A be a total Σ-algebra.

2.2.2.1 Satisfaction of a Single Equation What does it mean for A to *satisfy* an equation? The standard answer is as follows: No matter which values (in AU_Σ) are assigned to the variables of the equation, its two sides *evaluate* to the same value. Generalizing a bit, we might also consider variables *not* occurring in the equation. (These should not matter, and indeed they do not. We will return to this issue in paragraph 3.2.1.4.)

Therefore, let V be a set (of variables) and (t,u) be a V-equation. An assignment of values in AU_Σ to the variables in V is simply a function $f : V \longrightarrow AU_\Sigma$, and the evaluation of the V-terms t and u is provided by its Σ-homomorphic extension $f^\# U_\Sigma$. Thus, A *V-satisfies* (t,u), denoted by $A \models_V t = u$, if $\forall f \in \mathit{Set}(V, AU_\Sigma). t\left(f^\# U_\Sigma\right) = u\left(f^\# U_\Sigma\right)$, or, equivalently, if $\forall h \in \mathit{Set}^\Sigma(VF_\Sigma, A). t(hU_\Sigma) = u(hU_\Sigma)$.

2.2.2.2 Satisfaction of a Set of Equations Let E be a set of equations. The total Σ-algebra A is said to satisfy E if it satisfies every equation in E.

The pair (Σ, E) is called a *presentation*. A *total* (Σ, E)*-algebra* is a total Σ-algebra satisfying E.

The total (Σ, E)-algebras determine a full subcategory $Set^{(\Sigma,E)}$ of Set^{Σ}.

2.2.2.2.1 An Initial Algebra Let $\equiv_E$ denote the congruential closure of the ground instances of the equations in E. Then the quotient total Σ-algebra $0F_{\Sigma}/\equiv_E$ is an initial total (Σ, E)-algebra.

3 General Algebras

In the total case, the rôle played by the category *Set* is to provide sets, serving as carriers of algebras, and functions, serving as components of algebras and as homomorphisms. The key idea of this paper is that this rôle may be taken over by some *other* category. (The idea itself is not new ([12, page 221]), but we believe our treatment of initiality and of the independence of satisfaction of the set of variables considered, to be so.)

However, not *any* category will do. We have to be able to form the *finite powers* $AU_{\Sigma}{}^{n}$. To allow headroom for the generalization to *several sorts*, we at once generalize this to finite *products* $B_0 \times \cdots \times B_{n-1}$. A category with all finite products is said to be *Cartesian*.

However, the mere *existence* of products is not quite enough. For each tuple of objects we need a *specific* product (object and projections). The assignment of a product to each object tuple is usually called a *finite product structure* ([3, page 196]), but for the sake of brevity, we will introduce the term *Cartesian structure*. (Of course, any Cartesian category may be equipped with a Cartesian structure. The point is that we have to *decide* on one in particular.)

A functor F from one category with Cartesian structure to another is usually said to be *strict finite product preserving* (also [3, page 196]) if it preserves the structure, i.e. if it takes the product (object and projections) of each tuple $B_0, \ldots, B_{n-1}$ to the product of $B_0F, \ldots, B_{n-1}F$. For the sake of brevity, we will call such a functor *Cartesian*. Cartesian functors also preserve (finite) mediating and product arrows, natural transformations τ from one Cartesian functor to another satisfy $\tau_{B_0 \times \cdots \times B_{n-1}} = \tau_{B_0} \times \cdots \times \tau_{B_{n-1}}$, and the $^{\#}$ operation of an adjunction between Cartesian functors preserves (finite) product arrows.

The categories with Cartesian structure and the Cartesian functors constitute a category *CartStruct*.

3.1 Signatures

3.1.1 Algebras and Homomorphisms

Let C be a category with a Cartesian structure. Define the category C^{Σ} of Σ-*algebras* and Σ-*homomorphisms* by

- A Σ-algebra A comprises

- a *carrier* C-object AU_Σ
- for each $n \in \mathbf{N}$ and $f \in \Sigma_n$ a C-arrow $f_A : {AU_\Sigma}^n \longrightarrow AU_\Sigma$.

- A Σ-homomorphism $h : A \longrightarrow B$ is a C-arrow $AU_\Sigma \longrightarrow BU_\Sigma$ such that (1).
- Composition is composition in C.
- The identities are defined by $id_A = id_{AU_\Sigma}$.

3.1.2 Initiality

In the total case the initial algebra was obtained as the image of an initial object under a left adjoint $\longrightarrow Set^\Sigma$, viz. the left adjoint $Set \longrightarrow Set^\Sigma$ of the forgetful functor $Set^\Sigma \longrightarrow Set$. This suggests that we should look for a left adjoint $\longrightarrow C^\Sigma$, e.g. a left adjoint $C \longrightarrow C^\Sigma$ of the forgetful functor $C^\Sigma \longrightarrow C$. Unfortunately, we have been unable to define such a left adjoint in general. (Moreover, C does not necessarily possess an initial object.)

However, as Set^Σ has an initial object, a left adjoint $Set^\Sigma \longrightarrow C^\Sigma$ would be useful. But, are there any functors between C^Σ and Set^Σ, or more generally, D^Σ, where D is some category with a Cartesian structure? The operation of *lifting* will enable us to construct such functors from Cartesian functors between C and D.

3.1.3 Lifting

Extend the assignment $C \mapsto C^\Sigma$ to a functor $-^\Sigma : \mathit{CartStruct} \longrightarrow \mathit{Cat}$ by: If C and D are *CartStruct*-objects and F is a *CartStruct*-arrow $C \longrightarrow D$, the *Cat*-arrow, i.e. the *lifted* functor, $F^\Sigma : C^\Sigma \longrightarrow D^\Sigma$ is defined by

- If A is a C^Σ-object, the D^Σ-object AF^Σ is defined by
  - The D-object $AF^\Sigma U_\Sigma = AU_\Sigma F$.
  - For each $n \in \mathbf{N}$ and $f \in \Sigma_n$ the D-arrow $f_{AF^\Sigma} = f_A F$.
- $(h : A \longrightarrow B) \longmapsto hF$.

Theorem 1 *Let*

- *C and D be categories with Cartesian structure*
- *$F : C \longrightarrow D$ and $U : D \longrightarrow C$ be Cartesian functors*

such that $F \dashv U$. Then $F^\Sigma \dashv U^\Sigma$. ('Lifting preserves adjunctions.')

Proof: Let $A \in {C^\Sigma}_0$. We have to prove that AF^Σ is free over A w.r.t. U^Σ. A universal arrow from A to U^Σ is an element of $C^\Sigma\left(A, AF^\Sigma U^\Sigma\right) \subseteq C\left(AU_\Sigma, AF^\Sigma U^\Sigma U_\Sigma\right)$, where

$$\begin{aligned} AF^\Sigma U^\Sigma U_\Sigma &= AF^\Sigma U_\Sigma U \\ &= AU_\Sigma FU. \end{aligned}$$

Let $\eta : I_C \stackrel{\cdot}{\longrightarrow} F \circ U$ be the unit of the adjunction. Then η_{AU_Σ} is a C-arrow $AU_\Sigma \longrightarrow AU_\Sigma FU$. By naturality of η it is in fact a C^Σ-arrow $A \longrightarrow AF^\Sigma U^\Sigma$.

We claim that AF^Σ is free over A w.r.t. U^Σ with universal arrow η_{AU_Σ}, which is fairly straightforward to prove. ∎

3.1.4 An Initial Algebra

Let $F : Set \longrightarrow C$ and $U : C \longrightarrow Set$ be Cartesian functors such that $F \dashv U$. Then also $F^\Sigma \dashv U^\Sigma$, where $F^\Sigma : Set^\Sigma \longrightarrow C^\Sigma$.

Corollary 4 *$0F_\Sigma F^\Sigma$ is initial in C^Σ.*

3.2 Presentations

3.2.1 Satisfaction

3.2.1.1 Satisfaction of a Single Equation

3.2.1.1.1 Satisfaction w.r.t. a Set of Variables It seems difficult to generalize the definition of satisfaction of an equation by a total algebra given in subparagraph 2.2.2.1. In [1] we defined a C^Σ-object A to satisfy a set E of equations if AU^Σ did, leading to the restriction $F \circ U = I_{Set}$ mentioned in the introduction.

However, the concept of satisfaction of an equation by a total algebra A may be formulated in another way that is easily generalizable. [11] attributes the idea to Lawvere. In our opinion, the idea may best be explained by appealing to denotational semantics of simple applicative languages, where the denotation of a term is a function from environments to values, where an environment is a function from variables to values. In our case, the set of variables is V and the set of values is AU_Σ, making the set of environments ${}^V AU_\Sigma$ and the set of possible term denotations $Set\left({}^V AU_\Sigma, AU_\Sigma\right)$. The 'semantic function' $h_{AV} : VF_\Sigma U_\Sigma \longrightarrow Set\left({}^V AU_\Sigma, AU_\Sigma\right)$ is then defined in the following way: The function $th_{AV} : {}^V AU_\Sigma \longrightarrow AU_\Sigma$ is defined by $g\,(th_{AV}) = t\left(g^{\#} U_\Sigma\right)$.

Corollary 5 *A total Σ-algebra A V-satisfies a V-equation (t, u) iff $th_{AV} = uh_{AV}$.*

However, we need a *compositional* definition of h_{AV}. Such a definition is provided by the following two corollaries:

Corollary 6 *Let $v \in V$ and $g : V \longrightarrow AU_\Sigma$. Then $g\,(v\eta_V h_{AV}) = vg$.*

Corollary 7 $[f\,(t_0, \ldots, t_{n-1})]\, h_{AV} = \langle t_0 h_{AV}, \ldots, t_{n-1} h_{AV}\rangle \circ f_A$

The intended generalization may seem to require the existence of arbitrary (non-finite) products ${}^V AU_\Sigma$. However, this is not the case, as an equation contains a finite number of variables only. However, this finite number may be arbitrarily large, so we need an infinite set of variables altogether. On the other hand, a countable number is sufficient, as the variables of distinct equations are completely unrelated, although they may have the same names.

To simplify matters, we identify variables with the natural numbers. Furthermore, the only sets of variables (natural numbers) we will consider, are the natural numbers themselves and the set $\mathbf{N}$ of (all) natural numbers.

Corollary 8 $\mathbf{N}F_\Sigma {U_\Sigma}^2 = \bigcup_{n \in \mathbf{N}} nF_\Sigma {U_\Sigma}^2$

Proof: Straightforward, using lemma 9. ∎

Thus, an $\mathbf{N}$-equation is an n-equation for some natural number n. Moreover, there is a *least* such number.

Please notice that $m \leq n$ iff $m \subseteq n$, in which case $mF_\Sigma U_\Sigma \subseteq nF_\Sigma U_\Sigma$ (by corollary 1), entailing that $mF_\Sigma {U_\Sigma}^2 \subseteq nF_\Sigma {U_\Sigma}^2$.

Let n be a natural number. The function $h_{An} : nF_\Sigma U_\Sigma \longrightarrow \mathit{Set}\,(A{U_\Sigma}^n, AU_\Sigma)$ then satisfies $v\eta_n h_{An} =$ the vth projection $A{U_\Sigma}^n \longrightarrow AU_\Sigma$.

Let A be a C^Σ-object. The function[3] $h_{An} : nF_\Sigma U_\Sigma \longrightarrow C\,(A{U_\Sigma}^n, AU_\Sigma)$ is defined by

- $v\eta_n h_{An} =$ the vth projection $A{U_\Sigma}^n \longrightarrow AU_\Sigma$
- $[f\,(t_0, \ldots, t_{m-1})]\, h_{An} = \langle t_0 h_{An}, \ldots, t_{m-1} h_{An} \rangle \circ f_A$.

The Σ-algebra A is now said to n-satisfy an n-equation (t, u), denoted by $A \models_n t = u$, if $th_{An} = uh_{An}$.

Define a function $g_{An} : n \longrightarrow C\,(A{U_\Sigma}^n, AU_\Sigma)$ by $vg_{An} = v\eta_n h_{An}$. (In the total case, $a\,(vg_{An}) = va$.)

3.2.1.1.2 Absolute Satisfaction

Whether the algebra A n-satisfies an n-equation, should ideally not depend on n. In paragraph 3.2.1.4 we will show that under a certain condition this is indeed the case, but for the moment we will content ourselves with the following: Let (t, u) be an $\mathbf{N}$-equation, and let n be the *least* natural number such that (t, u) is an n-equation. The Σ-algebra A is said to *satisfy* the equation (t, u), denoted by $A \models t = u$, if it n-satisfies it.

3.2.1.2 Satisfaction of a Set of Equations

Let E be a set of $\mathbf{N}$-equations, making (Σ, E) a presentation. The Σ-algebra A is said to satisfy E, or to be a (Σ, E)-algebra, if it satisfies every equation in E. The (Σ, E)-algebras determine a full subcategory $C^{(\Sigma,E)}$ of C^Σ.

3.2.1.3 Satisfaction and Lifting

Lemma 10 *Let*

- *C and D be categories with Cartesian structure*
- *$F : C \longrightarrow D$ be a Cartesian functor*
- *A be a C^Σ-object.*

[3] Its domain is a *set*.

Then

1. *Let t be an n-term. Then $th_{AF^{\Sigma},n} = th_{An}F$.*
2. *Let $v \in n$. Then $vg_{AF^{\Sigma},n} = vg_{An}F$.*
3. *An n-equation n-satisfied by A is also n-satisfied by AF^{Σ}.*
4. *An* $\mathbf{N}$*-equation satisfied by A is also satisfied by AF^{Σ}.*
5. *A set of* $\mathbf{N}$*-equations satisfied by A is also satisfied by AF^{Σ}.*

Proof:

1. Straightforward induction on t.
2. Straightforward, using 1..
3. Straightforward, using 1..
4. Straightforward, using 3..
5. Straightforward, using 4..

∎

3.2.1.4 More on Absolute Satisfaction We will now show that under a certain condition n-satisfaction does not depend on n.

Let m and n be natural numbers such that $m \leq n$. Then for each $i \in m$ ig_{An} is a C-arrow $AU_{\Sigma}{}^{n} \longrightarrow AU_{\Sigma}$, entailing that $\langle 0g_{An}, \ldots, (m-1)\,g_{An}\rangle$ is a C-arrow $AU_{\Sigma}{}^{n} \longrightarrow AU_{\Sigma}{}^{m}$.

Lemma 11 *Let t be an m-term. Then $th_{An} = \langle 0g_{An}, \ldots, (m-1)\,g_{An}\rangle \circ th_{Am}$.*

Lemma 12 *Assume that $C = Set$ and that $mF_{\Sigma}U_{\Sigma}$ is non-empty. Then $\langle 0g_{An}, \ldots, (m-1)\,g_{An}\rangle$ is epi.*

Proof: We have to prove that $\langle 0g_{An}, \ldots, (m-1)\,g_{An}\rangle : AU_{\Sigma}{}^{n} \longrightarrow AU_{\Sigma}{}^{m}$ is *surjective*. Therefore, let $a \in AU_{\Sigma}{}^{m}$. We have to prove that there is a $b \in AU_{\Sigma}{}^{n}$ such that

$$\begin{aligned} a &= b\,\langle 0g_{An}, \ldots, (m-1)\,g_{An}\rangle \\ &= (b\,[0g_{An}]\,, \ldots, b\,[(m-1)\,g_{An}]) \\ &= [0b, \ldots, (m-1)\,b]\,, \end{aligned}$$

which is equivalent to that $\bigwedge_{i \in m} ia = ib$, determining b on $m \subseteq n$. If $i \in n \setminus m$, ib may be taken to be any element of AU_{Σ}. Thus, we have to prove that AU_{Σ} is non-empty.

As $mF_{\Sigma}U_{\Sigma}$ is non-empty, m is non-empty or Σ contains a constant symbol. Thus, there are two cases:

- m is non-empty. Let $i \in m$, entailing that $ia \in AU_{\Sigma}$.

- Σ contains a constant symbol, entailing that AU_Σ is non-empty (by lemma 2).

∎

Assume that there is a *faithful* Cartesian functor $V : C \longrightarrow Set$ (this being the abovementioned condition).

Lemma 13 *Assume that* $mF_\Sigma U_\Sigma$ *is non-empty. Then* $\langle 0g_{An}, \ldots, (m-1)\, g_{An} \rangle$ *is* epi*.*

Proof: As the faithful V reflects epics, we may prove this by proving that $\langle 0g_{An}, \ldots, (m-1)\, g_{An} \rangle\, V$ is epi, where

$$\begin{aligned} \langle 0g_{An}, \ldots, (m-1)\, g_{An} \rangle\, V &= \langle 0g_{An}V, \ldots, (m-1)\, g_{An}V \rangle \\ &= \left\langle 0g_{AV^\Sigma,n}, \ldots, (m-1)\, g_{AV^\Sigma,n} \right\rangle, \end{aligned}$$

which is epi (by lemma 12). ∎

Corollary 9 *Let* (t, u) *be an m-equation. Then* $A \models_m t = u$ *iff* $A \models_n t = u$*.*

Proof: We have to prove that

$$\begin{aligned} & th_{Am} = uh_{Am} \\ & \quad \Leftrightarrow \quad th_{An} = uh_{An} \\ & \quad \Leftrightarrow \quad \langle 0g_{An}, \ldots, (m-1)\, g_{An} \rangle \circ th_{Am} = \langle 0g_{An}, \ldots, (m-1)\, g_{An} \rangle \circ uh_{Am}, \end{aligned}$$

where $\Rightarrow$ is trivial, whereas $\Leftarrow$ follows from the fact that $\langle 0g_{An}, \ldots, (m-1)\, g_{An} \rangle$ is epi (in the presence of the m-terms t and u). ∎

This shows that n-satisfaction is indeed independent of n (under the condition that there is a faithful Cartesian functor $C \longrightarrow Set$).

In the total case lemma 12 may be used directly in the proof of corollary 9. Moreover, V-satisfaction is independent of V (for *arbitrary* variable sets V), which may be proved in a similar fashion.

3.2.2 Initiality

Recall the initial total (Σ, E)-algebra $0F_\Sigma / \equiv_E$. Then $(0F_\Sigma / \equiv_E)\, F^\Sigma \in C^{(\Sigma,E)}{}_0$.

Theorem 2 $(0F_\Sigma / \equiv_E)\, F^\Sigma$ *is* initial *in* $C^{(\Sigma,E)}$*.*

Proof: Let $A \in C^{(\Sigma,E)}{}_0$, entailing that $AU^\Sigma \in Set^{(\Sigma,E)}{}_0$ (by lemma 10, part 5). We have to prove that

$$\begin{aligned} \exists_1 h &\in C^{(\Sigma,E)} \left[(0F_\Sigma / \equiv_E)\, F^\Sigma, A \right] \\ &= C^\Sigma \left[(0F_\Sigma / \equiv_E)\, F^\Sigma, A \right], \end{aligned}$$

which is equivalent to that

$$\begin{aligned} \exists_1 h &\in Set^\Sigma \left(0F_\Sigma / \equiv_E, AU^\Sigma \right) \\ &= Set^{(\Sigma,E)} \left(0F_\Sigma / \equiv_E, AU^\Sigma \right), \end{aligned}$$

which holds. ∎

4 Concluding Remarks

We have made a general construction based on some *parameters*: an adjunction $F \dashv U : C \longrightarrow Set$ in *CartStruct*. From C we have constructed the category of Σ-algebras and its full subcategory of (Σ, E)-algebras. Using the functors, we have shown that the two categories have initial objects closely related to the ordinary term and quotient algebras, which we believe to be new results. The same goes for the condition under which we have found satisfaction to be independent of the set of variables considered.

We will conclude the paper by pointing out some (possible) directions for (our) future research:

- To base the approach on algebraic (Lawvere) *theories* ([8]) (instead of on presentations)
- To (try to) apply our theory to some of the cases mentioned in the introduction: topological and continuous algebras, and in particular total algebras whose carrier elements are e.g. partial orders
- Generalization to several sorts
- By considering a *category* of signatures one might possibly obtain an *institution* ([2]) depending on C. This way of constructing institutions should then be compared to others, e.g. *chartering* ([4]).

Acknowledgements Thanks are due to S. Hegner, N. Husberg, S. Meldal, P. D. Mosses, A. Tarlecki, E. G. Wagner, and M. Walicki.

References

[1] Ø. B. Fredriksen. From sets with structure to algebras with structure. In R. J. R. Back and K. Sere, editors, *Proceedings of the 5th Nordic Workshop on Program Correctness*, number 18 in Åbo Akademi Reports on Computer Science and Mathematics, Ser. B, pages 62–71, Åbo Akademi University, Dept. of Computer Science, DataCity, SF–20520 Åbo, Finland, May 1994.

[2] J. A. Goguen and R. M. Burstall. Introducing institutions. In E. Clarke and D. Kozen, editors, *Logics of Programs. Proceedings 1983*, number 164 in Lecture Notes in Computer Science, pages 221–255. Springer-Verlag, 1984.

[3] J. A. Goguen and R. M. Burstall. Some fundamental algebraic tools for the semantics of computation. Part 1: Comma categories, colimits, signatures and theories. *Theoretical Comput. Sci.*, 31:175–209, 1984.

[4] J. A. Goguen and R. M. Burstall. A study in the foundations of programming methodology: Specifications, institutions, charters and parchments. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Category Theory and Computer Programming*, number 240 in Lecture Notes in Computer Science, pages 313–333. Springer-Verlag, 1986.

[5] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. T.-Y. Yeh, editor, *Current Trends in Programming Methodology. Volume IV. Data Structuring*, chapter 5, pages 80–149. Prentice-Hall, Inc., 1978.

[6] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24(1):68–95, Jan. 1977.

[7] H. Herrlich and G. E. Strecker. *Category Theory. An Introduction.* Allyn and Bacon, Inc., 1973.

[8] F. W. Lawvere. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences*, 50:869–872, 1963.

[9] K. Meinke. Topological methods for algebraic specification. Report CSR 21–93, University of Wales, University College of Swansea, Computer Science Department, Singleton Park, Swansea SA2 8PP, Great Britain, May 1993.

[10] B. Möller. On the algebraic specification of infinite objects — ordered and continuous models of algebraic types. *Acta Inf.*, 22:537–578, 1985.

[11] A. Poigné. Algebra categorically. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Category Theory and Computer Programming*, number 240 in Lecture Notes in Computer Science, pages 76–102. Springer-Verlag, 1986.

[12] H. Schubert. *Categories.* Springer-Verlag, 1972.

[13] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, and M. Broy. On hierarchies of abstract data types. *Acta Inf.*, 20:1–33, 1983.

Bisimulations for asynchronous mobile processes

Martin Hansen Hans Hüttel* Josva Kleist

BRICS[†] Department of Math. & Comp. Science
Aalborg University
Fredrik Bajersvej 7E
9220 Aalborg Ø
Denmark

An important question in the theory of process calculi is when two processes can be said to exhibit the same behaviour and a number of *behavioural equivalences* have been proposed. Traditionally, the emphasis has been on studying the equivalences for 'pure' processes where communication does not involve the passing of data values. However, within the past few years there has been renewed interest in the study of value-passing as a consequence of the emergence of the π-calculus. Here Milner et. al. [6] have determined two variants of the notion of *bisimulation*, *late* and *early* bisimilarity, differing in the assumption of when a data value is instantiated. Most recently Sangiorgi [7] has proposed a new notion of bisimulation equivalence which he calls *open* bisimilarity. Sangiorgi formulates a sound and complete *equational theory* for open bisimilarity on finite (i.e. non-recursive) π-calculus terms.

An important theorem is that the three notions of bisimilarity are distinct within the π-calculus, with open bisimulation being strictly finer than late bisimulation which in turn is strictly finer than early bisimulation. The late/early distinction has been studied for the much coarser *testing* equivalence of Hennessy and De Nicola and it has been shown by Ingólfsdótir [5] and Boreale and De Nicola [1] that the late and early testing equivalences coincide, both for a process calculus with simple data values and conditional expression and for the π-calculus.

In this paper we consider Plain LAL, a mobile process calculus which differs from the π-calculus in the sense that the communication of data values happens *asynchronously*. Previously, Honda and Tokoro [4] and Boudol [2] have described so-called asynchronous π-calculi. However, they have not studied the notion of equivalence within this setting. The surprising result is that in the presence of asynchrony, the open, late and early bisimulation equivalences *coincide*. As the equivalences are one and the same, this leads us to formulate an *equational theory* which is sound for finite terms of Plain LAL.

*E-mail: hans@iesd.auc.dk

†Basic Research in Computer Science, Centre of the Danish National Research Foundation

References

[1] Michele Boreale and Rocco De Nicola. Testing equivalences for mobile processes. In W.R. Cleaveland, editor, *CONCUR '92*, number 630 in Springer LNCS, pages 2–16. Springer-Verlag, 1992.

[2] Gérad Boudol. Asynchrony and the π-calculus. Technical report, INRIA Sophia-Antipolis, 1992.

[3] Martin Hansen and Josva Kleist. Process calculi with asynchronous communication. Master's thesis, Aalborg University, 1994.

[4] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP 91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.

[5] Anna Ingólfsdóttir. Late and early semantics coincide for testing. *Theoretical Computer Science*, 1994. To appear.

[6] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes — part I and II. Technical Report ECS-LFCS-89-85 and -86, Department of Computer Science, University of Edinburgh, 1989. Also published in *Information and Computation*, 100:1-77,1992.

[7] Davide Sangiorgi. A theory of bisimulation for the π-calculus. In E. Best, editor, *Proceedings of CONCUR 93*, volume 715 of *Lecture Notes in Computer Science*, pages 127–142. Springer-Verlag, 1993.

G^2-Algebras

Claus Hintermeier*, **Hélène Kirchner**

CRIN-CNRS & INRIA-Lorraine
BP239, F-54506 Vandœuvre-lès-Nancy Cedex, France
E-mail: hinterme@loria.fr, hkirchne@loria.fr

G^2-algebras are polymorphic, order-sorted algebras where sorts are terms in an equational theory. G^2-algebras are still first order and have the classical quotient term algebra as initial model. They extend G-algebras [1] conservatively and are close to a two level, hierarchical fragment of unified algebra [4].

In G^2-algebras, we distinguish between *sorts* (or *types*), used for the description of possible arguments and range of a function, and *combinators*, which are algebraic functions. Domains of functions can be rather complex to describe. E.g. the domain of natural number substraction is the set of all pairs of natural numbers $\langle x, y\rangle$, s.t. $x \geq y$. This is not a regular tree language and therefore, it is not possible to describe this domain with the help of flat, linear function declarations only.

One way to describe this domain is to use term declarations and semantic sorts. However, we want to calculate with the resulting definition of natural number substraction in form of a decorated term rewriting system [1]. We had to realize [2], that term declarations are hard to handle and underly strong restrictions w.r.t. their form, if we want to compute effectively with them. Another solution consists in the use of sort functions. E.g., as shown in the following very simple example, we can define sorts corresponding to natural numbers and intervals on them.

Example 1. Consider the following specification in pseudo-OBJ syntax, where we distinguish between capital and small letters:

```
obj NATS
    kind     Nats
    sorts    zero :       -> Nats
             succ : Nats  -> Nats
    sort-var X :: Nats
    ops      0 :          -> zero
             s :   X      -> succ(X).
```

In the initial algebra of this specification, which is unique up to isomorphism, $succ^n(zero)$ is interpreted as singleton $\{s^n(0)\}$ and $Nats$ is the set of all such singletons. Now intervals on natural numbers can be specified as follows:

```
obj INATS
    kind      INats
    import    NATS
    sorts     leq :     Nats      -> INats
              geq :     Nats      -> INats
              between : Nats Nats -> INats
    sort-var  X,Y :: Nats
    subsorts  X < leq(X) < leq(succ(X))
              X < geq(X) > geq(succ(X))
              X < between(X,X)
              between(succ(X), Y) < between(X, Y)
              between(X, Y) < between(X, succ(Y)).
```

* Visiting Prof. P.D. Mosses at BRICS 09-12/1994, partially supported by ESPRIT Basic Research Working Group COMPASS

Now, substraction maybe defined as follows:

```
obj NAT-SUBSTRACTION
    import    INATS
    sort-vars X::Nats
    vars      x::X, y::leq(X)
    ops       _∸_ :X leq(X) -> geq(zero)
    axioms    x ∸ 0          = x
              s(x) ∸ s(y)    = x ∸ y.
```

The great difference between the last axiom and a conditional many-sorted specification like:

$$y \leq x \Rightarrow s(x) \dot{-} s(y) = x \dot{-} y,$$

is the model notion. G^2-models do not force $\dot{-}$ to be defined over all naturals, as this is the case for models of many-sorted conditional specifications. G^2-algebras allow to specify precisely the domain of a function.

Suppose now that we have an analogous specification module $INTS$ for integers *ints*, using an additional unary minus function $-$. Consequently, there may be two combinator terms representing zero: 0 and -0, a situation very common in real life computing. Addition can then easily be defined as follows:

```
obj INT-ADDITION
    import INTS
    ops    _+_ : ints ints -> ints
    vars   x, y::ints
    axioms x + 0          = x
           x + -0         = x
           x + s(y)       = s(x + y)
           ...
```

Now, addition `plus` may be defined in the standard way on sorts, too, i.e. with only one zero, namely `zero`. A kind of verification of the implementation of addition on the integers results then in proving that `x + y` is of sort `X plus Y` by structural induction.

The presented framework has meanwhile been superseded by an n-level Horn clause approach called G^n-logics [3], having a sound and complete deduction system and initial models. Furthermore, sort terms are still interpreted as sets and can be used to specify function graphs. Another feature of G^n-logics is the possibility to mix combinator and sort terms, such as `create(nat)` for the creation of lists of natural numbers. Finally, let us mention that membership formulas, i.e. typing, can be conditional. Hence, G^n-logics are by far more expressive than G^2-algebras.

References

1. C. Hintermeier, C. Kirchner, and H. Kirchner. Dynamically-typed computations for order-sorted equational presentations —extended abstract. In S. Abiteboul and E. Shamir, editors, *Proc. 21st International Colloquium on Automata, Languages, and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 450–461. Springer-Verlag, 1994.
2. C. Hintermeier, C. Kirchner, and H. Kirchner. Sort inheritance for order-sorted equational presentations. In *Recent Trends in Data Type Specifications*, Lecture Notes in Computer Science. Springer-Verlag, 1994.
3. C. Hintermeier, H. Kirchner, and P. D. Mosses. R^n- and G^n-logics. Technical report, Centre de Recherche en Informatique de Nancy, 1994.
4. P. D. Mosses. Unified algebras and institutions. In *Proceedings 4th IEEE Symposium on Logic in Computer Science, Pacific Grove*, pages 304–312, 1989.

This article was processed using the LaTeX macro package with LLNCS style

Boolean Automata: a Compact Representation of Synchronous Reactive Systems

Leszek Holenderski, Axel Poigné

GMD-SET, Schloss Birlinghoven, D-53757 Sankt Augustin, Germany
August 1994

Boolean automata are compact representations of synchronous reactive systems. The compactness manifests in avoiding the well-known "state explosion" problem which arises from representing parallel composition by a standard product construction.

In the computational model for synchronous reactive systems, time is assumed to be *discrete*, i.e. divided into enumerable number of non-overlapping segments, called *instances of time*. In every instance of time, a synchronous reactive system performs a so-called *reaction step*, which consists in fetching inputs, computing, and emitting outputs. The fetch-compute-emit sequence is considered to be an atomic action which takes exactly one instance of time. A system may consist of several reactive components, running in parallel and communicating with each other. The components perform their reaction steps simultaneously, i.e. in the same instance of time, and the outputs emitted in one component are immediately available to all other components. Thus, communication is based on instantaneous broadcasting, used in the synchronous programming languages Esterel, Lustre and Argos, as opposed to instantaneous hand-shaking, used in synchronous CCS.

In fact, we only consider *pure* synchronous reactive systems, i.e. those in which all inputs and outputs are pure signals. A pure signal does not carry a value. It is either present or absent, in every instance of time.

It is convenient to represent a *pure* synchronous reactive system (with signals S), by a Mealy automaton whose transitions are labelled by pairs e/e', where $e, e' \subset S$. A transition $\sigma \xrightarrow{e/e'} \sigma'$, where σ and σ' are states, describes a possible reaction step of the system. It has the following intuitive reading: provided the system starts an instance in state σ and e is the set of all signals present during the instance, the system emits signals e' and finishes the instance in state σ'. In such a setting, parallel composition of reactive systems is represented by a product of Mealy automata, which leads to the well-known "state explosion" problem.

It turns out that the transition relation of a pure synchronous reactive system can be coded by two sets of Boolean functions. The first one represents a set of (possibly recursive) Boolean equations whose solutions represent pairs e/e'. The second one represents a set of simultaneous Boolean assignments which represent changes of states.

In the paper, we show how to compose such Boolean automata using typical operations, like sequential and parallel composition, choice, iteration and preemption. The main result is that a Boolean automaton grows linearly with the growth of a reactive synchronous system it represents.

Structural Synthesis of Programs Using Regular Data Structures

Mait Harf

Dept. of Computer Software, Institute of Cybernetics,
Akadeemia tee 21, EE 0026 Tallinn, Estonia
mait@cs.ioc.ee

Abstract

This paper discusses a specific feature of structural synthesis of programs. Using regular data structures enables to optimize the method of structural synthesis of programs for several particular cases. The method for program construction described here is implemented in the NUT programming system.

1 Introduction

Automatic program construction from logical specifications is a way to obtain reliable software. Structural synthesis of programs is a method to transform formal specifications into effective programs via automatic proving of solvability of the problem [1, 2]. This technique can be also treated as satisfying functional constraint network using value propagation [3]. The method is powerful under certain conditions (see [3]) but for several particular cases it can be optimized . In this paper cases involving regular data structures like arrays, sequences, vectors etc. are discussed.

The approach is motivated by a number of applications where such data structures appear. In this paper we will discuss more effective handling of these data structures in the framework of program synthesis.

The method for program construction described here has been implemented as a specific feature of the NUT programming system [4]. Several examples of using regular data in specifications in NUT language as well as programs which can be synthesized for solving problems, are considered.

The paper is organized as follows. We start with the description of structural synthesis of programs in general. In section 3 the extension of the formal method for using regular data in synthesized programs is examined. This is followed by the description of specification of the regular data in NUT language in section 4 and examples in section 5.

2 Structural Synthesis of Programs

The method of structural synthesis of programs is described in [1]. Here we give a brief overview of the method.

Let us use objects and functional relations for representing nonprocedural specifications of problem areas as follows:

1. denote objects (problem variables) by $x, y, z, u, v, \ldots$
2. represent dependencies between objects as functional relations of the first or higher order $y = f(x)$, $z = g(u, v)$, $h = r(f)$, $\ldots$

For automatic proving of solvability of computational problems in areas described in such way a logical language of special kind is used, which contains

1. propositional variables $X, Y, Z, U, V, \ldots$, which mean that values of objects $x, y, z, u, v, \ldots$ correspondingly can be computed;
2. computability statements (implications) of two kinds
   - unconditional computability statements of the form $X \underset{f}{\longrightarrow} Y$ which correspond to functional relations of the first order. Here $X \underset{f}{\longrightarrow} Y$ means that from possibility to get value of x follows, that y can be computed using function f;
   - conditional computability statements of the form
   $$(U \underset{\varphi}{\longrightarrow} V) \longrightarrow (X \underset{F(\varphi)}{\longrightarrow} Y),$$
   which correspond to functional relations of higher order. Here
   $$(U \underset{\varphi}{\longrightarrow} V) \longrightarrow (X \underset{F(\varphi)}{\longrightarrow} Y)$$
   means that y can be computed from x using function $F(\varphi)$ if v can be computed from u using function φ.

It is shown in [2] that if we can build a constructive proof for finding W from Z then it is possible to construct a program from this proof, which computes w from z.

For example from the following implications

$U \underset{f_1}{\longrightarrow} W$

$W \underset{f_2}{\longrightarrow} V$

$X \underset{f_3}{\longrightarrow} Y$

$(U \underset{\varphi}{\longrightarrow} V) \longrightarrow (Y \underset{f_4(\varphi)}{\longrightarrow} Z)$

we can get both the proof

$$\frac{X \underset{f_3}{\longrightarrow} Y, \quad \dfrac{U \underset{f_1}{\longrightarrow} W,\ W \underset{f_2}{\longrightarrow} V}{(U \underset{f_1;f_2}{\longrightarrow} V)} \quad \longrightarrow \quad (Y \underset{f_4(f_1;f_2)}{\longrightarrow} Z)}{X \underset{f_3;f_4(f_1;f_2)}{\longrightarrow} Z}$$

and the program $f_3; f_4(f_1; f_2)$ for computing object z from object x.

For building constructive proof of computability structural synthesis rules (SSR) [2] are used. Also in [2] completeness of SSR is shown.

3 Structural Synthesis of Loops

In order to provide the method of structural synthesis of programs with loop synthesis facility, the following additional tools are proposed.

Let us introduce

1. indexed problem variables

$$a_1, a_i, a_{i+1}, a_7, b_i, \ldots,$$

where indice can have natural numbers as their values;

2. functional dependencies, which use indexed problem variables

$$a_{i+1} = f(a_i), \quad b_5 = g(b_3), \ \ldots$$

for representing nonprocedural specifications.

For our logical language let us invoke

1. index propositional variables $I, J, \ldots$, which can have natural numbers as their values. Let us consider indexed propositions

$$A[1],\ B[1], \ldots,\ A[I],\ B[I],\ A[I+1],\ B[I+1],\ \ldots,$$

which use constants, index variables or expressions $I+1,\ J+1$ as indices;

2. computability statements of the following form

$$A[1] \underset{h}{\longrightarrow} A[7];$$

$$A[I] \underset{f}{\longrightarrow} A[I+1];$$

$$A[I+1] \underset{g}{\longrightarrow} A[I].$$

Here any computability statement, containing indexed propositions theoretically represents infinite set of computability statements which are obtained by substituting variable index with natural numbers.

For example

$$A[I] \underset{f}{\longrightarrow} A[I+1]$$

represents computability statements

$$A[1] \underset{f}{\longrightarrow} A[2],\ \ A[2] \underset{f}{\longrightarrow} A[3], \ldots$$

We can see that if formula $A[I] \underset{f}{\longrightarrow} A[I+1]$ can be derived then formula $A[m] \underset{f^{n-m}}{\longrightarrow} A[n]$ can also be derived, where $n > m$; n, m are natural numbers and f^{n-m} denotes application of f $n-m$ times.

Now we can invoke two new inference rules which can be applied to computability statements, containing indexed propositional variables

1.

$$\frac{B \underset{f}{\longrightarrow} A[m], \quad A[I] \underset{g}{\longrightarrow} A[I+1]}{B \underset{f;g^{n-m}}{\longrightarrow} A[n]}$$

program $f; g^{n-m}$ we can write as

f;
$i := m$;
while $i < n$ **do** $g(i)$; $i := i+1$ **od**

2.

$$\frac{B \xrightarrow[f]{} A[n], \quad A[I+1] \xrightarrow[g]{} A[I]}{B \xrightarrow[f;g^{n-m}]{} A[m]}$$

program $f; g^{n-m}$ can be written as

$f;$
$i := n;$
while $i > m$ **do** $g(i);\ i := i - 1$ **od**

4 Extension of the NUT system

NUT is an object-oriented programming system for knowledge-based programming with facilities for automatic program synthesis. The system allows to describe nonprocedural specifications using object-oriented NUT language. Full description of the NUT language can be found in [4, 5]. As in any object-oriented language classes and objects are used as basic notions of the NUT language. Here we consider this part of the language which can be used for computational model specification and especially new constructions which enable us to specify and operate regular data structures.

The automatic program construction involved is based on the method above. The logical language discussed in previous section is used by the system as internal specification language.

Regular data can be described as components of special kind in class specification. For example, having specifications of classes of points and line segments of two-dimensional plane

point
var x, y : *num*;

bar
var P, Q : *point*;
length, angle : num;
rel $(P.x - Q.x)^2 + (P.y - Q.y)^2 = length^2$;
$P.y - Q.y = (P.x - P.y) * \ tan(angle)$;

one can introduce the following model of mechanism

mechanism
var 1.. : *bar length* $= 10$;
rel $\#next.angle = \#curr.angle + 0.1$;
$\#next.P = \#curr.Q$;
init $\#1.P\ := [0, 0]$;
$\#1.angle\ := 0$;

Here the syntactical construction 1.. : *bar length* $= 10$; specifies that a mechanism consists of sequence of bars of length 10. Here elements of the sequence are named by their numbers as $\#1.P$, $\#1.angle, \ldots$. To specify regular relationship between the elements of sequence, two special elements are available, current element ($\#curr$) and next element ($\#next$), which correspond respectively to I-th and $I+1$-th indexed propositions in the logical language described in section 3.

The NUT system includes a program synthesizer, which is in fact an implementation of the method of structural synthesis of programs. Structural synthesis of loops on the basis of inference rules described in previous section has also been implemented as a part of the program synthesizer and a couple of experiments of synthesizing loops are provided.

The program synthesizer is able to construct a loop over a sequence for computing the l-th element, if

1. it is possible to compute the value of one element in the sequence, let us say, the value of the k-th element;
2. it is possible to synthesize a program for computing the value of the next element from the given value of current element, if $k < l$, or vice versa, it is possible to compute the value of current element from the given value of next element, if $k > l$.

Speaking in terms of our logical language the program synthesizer composes a conditional computability statement of the form

$$(A[I] \xrightarrow[g]{} A[I+1]) \longrightarrow (A[k] \xrightarrow[F(g)]{} A[l]),$$

where

$$F(g) \;:\; i := k;\;\; \textbf{while}\; i < l \;\textbf{do}\; g(i);\;\; i := i+1 \;\textbf{od}$$

if $k < l$ and

$$(A[I+1] \xrightarrow[g]{} A[I]) \longrightarrow (A[k] \xrightarrow[F'(g)]{} A[l]),$$

where

$$F'(g) \;:\; i := k;\;\; \textbf{while}\; i > l \;\textbf{do}\; g(i);\;\; i := i-1 \;\textbf{od}$$

if $k > l$

into the knowledge base (formal model) and tries to use this conditional computability statement in the program synthesis.

5 Examples

Example 1

Let us describe Fibonacci numbers as:

element
- **var** $p, c \;:\; num;$

Fibonacci
- **var** $1.. \;:\; element;$
- **rel** $nextelem \;:\; \#next.c = \#curr.p + \#curr.c;$
- $swap \;:\; \#next.p = \#curr.c;$
- $init1 \;:\; \#1.p = 1;$
- $init2 \;:\; \#1.c = 1;$

When computing Fibonacci numbers a relation between three sequential elements is used. As we have no possibility to express this relation explicitly, we have used here a structure named "element" with two components.

c - current element of the sequence of Fibonacci numbers, and

p - predecessor of the element c.

The problem of computing any particular Fibonacci number is solvable on this specification. For example, the program for computing the 7-th number is synthesized as follows (here a simple representation language for synthesized algorithms is used):

$ALGORITHM\ 1$
$(Fibonacci) \longrightarrow$
  $(Fibonacci).init1\ : \longrightarrow \#1.p$
  $(Fibonacci).init2\ : \longrightarrow \#1.c$
  $(Fibonacci).LOOP\ :\ from\ 1\ to\ 7\ do:$
    $ALGORITHM\ 2$
    $(Fibonacci) \longrightarrow \#curr$
      $(Fibonacci).swap\ : \longrightarrow \#next.p$
      $(Fibonacci).nextelem\ : \longrightarrow \#next.c$
    $END\ OF\ ALGORITHM\ 2\ (2\ operators)$
$END\ OF\ ALGORITHM\ 1\ (3\ operators)$

Example 2

Let us specify a sequence of structures consisting of three elements and describe some relations between the elements

structure
  var $x, y, z\ :\ num;$

sequence
  var $1..7\ :\ structure;$
  rel $r1\ :\ \#next.x = \#curr.z + 1;$
  $r2\ :\ \#next.y = \#curr.z + 2;$
  $r3\ :\ \#next.x = \#curr.y + 3;$
  $initx\ :\ \#1.x = 10;$
  $inity\ :\ \#1.y = 20;$
  $initz\ :\ \#1.z = 30;$

This example illustrates the dependence of solvability of different problems on how the problem specification has been presented. In this example the problem of computing the component y of any certain sequence element from the given component y of any other certain sequence element is solvable, as the problems of computing $\#next.y$ from $\#curr.y$ and vice versa as well, are solvable.

Similar problems for components x and z are not solvable, as problems of computing $\#next.x$ from $\#curr.x$ and $\#next.z$ from $\#curr.z$ and vice versa are not solvable.

The synthesized algorithm for computing $\#6.y$ looks as follows

$ALGORITHM\ 1$
$(sequence) \longrightarrow$
  $(sequence).initx\ :\longrightarrow \#1.x$
  $(sequence).inity\ :\longrightarrow \#1.y$
  $(sequence).initz\ :\longrightarrow \#1.z$
  $(sequence).LOOP\ :\ from\ 1\ to\ 6\ do:$
    $ALGORITHM\ 2$
    $(sequence) \longrightarrow \#curr.y$
      $(sequence).r3\ :\longrightarrow \#next.x$
      $(sequence).r1\ :\longrightarrow \#curr.z$
      $(sequence).r2\ :\longrightarrow \#next.y$
    $END\ OF\ ALGORITHM\ 2\ \ (3\ operators)$
$END\ OF\ ALGORITHM\ 1\ \ (4\ operators)$

Example 3

Here we shall observe specification of a two-dimensional table of numbers containing a couple of relations between neighbouring elements

string
  var 1..8 : *num*;
  rel $\#next = \#curr + 1;$

table
  var 1..7 : *num*;
  var $s4,\ s6$: *num*;
  rel $\#next.\#1 = \#curr.\#1 + 1;$
  $\#next.\#3 = \#curr.\#3 + 3;$
  $\#next.\#7 = \#curr.\#7 + 7;$
  $s4 = \#4.\#3;$
  $s6 = \#6.\#7;$
  init $\#1.\#1\ := 0;$

Here follow some remarks about synthesis strategy, used for computation of different elements of the table.

1. Simple loops can be synthesized for computing elements
$$\#1.\#2, \ldots, \#1.\#8$$
as well as
$$\#2.\#1, \ldots, \#7.\#1$$
from the given element #1.#1.
2. More complicated algorithm can be synthesized for computing, for instance, the element #4.#7. This element can be found in the case, when it is possible to synthesize a sequence of simple loops, leading to #4.#7, each of which computes an element of the table, which explicitly occurs in the specification. In our particular case 1-st, 3-rd and 7-th element of 4-th and 6-th row can be computed.
As we can see in Fig. 1. there exist 10 possible routes to the element #4.#7 (only routes without recomputing any element can be synthesized). The NUT system will select one possible route (not neccessarily the shortest).

3. To compute the element #6.#1 for example, the NUT system synthesizes first a loop to compute #4.#1 and thereafter it synthesizes a loop to compute #6.#1 from #4.#1. Here the strategy is to try to synthesize the shortest loop first.

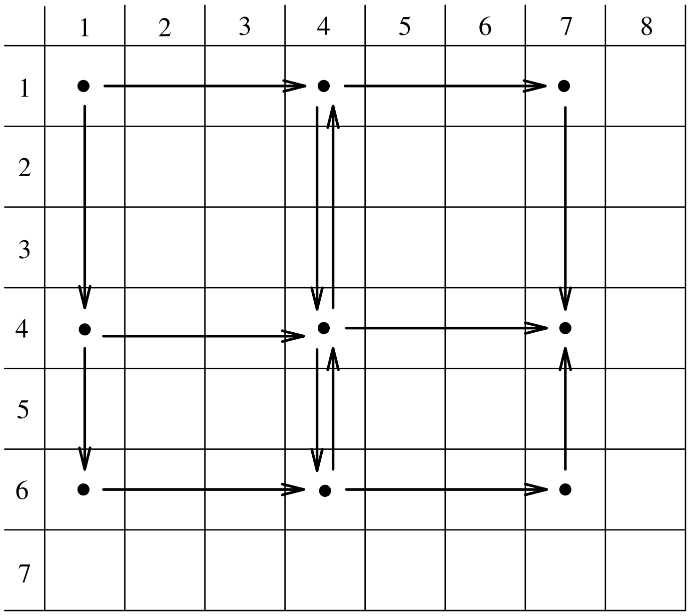


Figure 1:

6 Conclusions

Regular data structures like arrays, sequences, vectors etc. appear in many applications. In such structures there are often functional relations between neighbouring elements.

An example of such a field is logical design of electronic equipment where a large number of components of the same kind are organized into a regular array. The techniques above can be successfully used in modelling and simulating the behavior of such devices.

The method of structural program synthesis creates a large internal model of the problem involving regular data structures. The techniques described in this paper, allow to reduce the complexity to the level where many practical problems can be solved.

References

[1] Tyugu E. Knowledge-Based Programming. Addison-Wesley, N.Y., 1988.

[2] Mints G., Tyugu E. Justification of the Structural Synthesis of Programs. Science of Computer Programming, 2, No.3, 1983, pp. 215 - 240.

[3] Tyugu E., Uustalu T. Higher-Order Functional Constraint Networks. In: Constraint Programming. NATO ASI Series F, Vol. 131. Springer-Verlag, Berlin, 1994, pp. 116 - 139.

[4] Tyugu E., Matskin M., Penjam J., Eomois P. Nut - An Object-Oriented language. Computers and Artificial Intelligence, v.5, No. 6, 1986, pp. 521 - 542.

[5] Uustalu T. et al. The NUT Language Report. Technical Report TRITA-IT R 94:14, Dept. of Teleinformatics, The Royal Institute of Technology, Stockholm, 1994.

A Semantic Theory for Value–Passing Processes Late Approach

Anna Ingólfsdóttir

BRICS[*]
Department of Mathematics and Computer Science
Aalborg University, Denmark

1 Introduction

In the original work of Milner, [Mil80], on *CCS* and Hoare, [Hoa78], on *CSP*, processes are allowed to exchange data in communications. In these original calculi the value-passing calculus is interpreted in terms of the pure calculus in which communication is pure synchronization. A prefixing with an input action, $\overline{a}(x).p$, is interpreted as a non-deterministic choice between pure terms of the form $\overline{a}_v.p[v/x]$, where v ranges over the set of possible values, which in many cases is infinite. In this approach, two processes that synchronize are both supposed to know each other's *channel* and *value*, i.e. the data variable is instantiated by the potential input values already when the process reports the willingness or ability to communicate on the channel c.

In a more recent work on the π-calculus, [MPW89], this semantic approach is referred to as *early semantics* due to the early instantiation of the data variables as described above. Its counterpart, the *late semantics*, is also introduced in the same reference. Here the idea is that the processes *only* synchronize on the channel name and that the inputting process has to accept whatever value the outputting process has to offer. This may be interpreted as if the result of the reception of the value is delayed until the process has received the value. The inputting process reports the willingness to communicate on a channel, c, by performing an action of the form $\overline{c}$, and thereby evolves to a function which waits for the value the outputting counterpart in the communication provides. Symmetrically the result of reporting the willingness to output an uninterpreted value on the channel c is modelled by the action c. By performing this action the process evolves to a term which basically consists of a data expression and a process expressions.

In a more recent version of the π-calculus, the Polyadic π-calculus in [Mil91], the outcome of input and output actions are modelled by extending the syntax

[*]Basic Research in Computer Science, Centre of the Danish National Research Foundation.

with the new constructions *abstractions* and *concretions*. For further motivation for this approach we refer to [MPW89], [Mil91] and [Ing93].

The aim of this paper is to develop a denotational semantics for languages with simple values based on the late approach described above although we use a slightly different notation as we will explain in the next section.

The paper is divided into two main parts: a general theory to handle value-passing in §2, followed by an application to a concrete definition in §3. In the first part we introduce both a general syntax and a general class of mathematical models to model process algebras with values which support the late semantic approach. This is a direct generalization of the standard Σ–terms and Σ-domains that appear in [He88] to model the pure calculus. For this purpose we introduce the general notion of applicative signature (Σ, C) where Σ is a signature and C a set (of channel names) and that of (Σ, C)-terms. This is basically the syntax of the Polyadic π-calculus presented in [Mil91] although we only allow simple values and allow Σ to be any set of operators. We also introduce the general class of applicative (Σ, C)-domains to model the semantics of the (Σ, C)-terms.

Next we apply the general result for algebraic *cpo*s ([He88]): functions which are monotonic on the compact elements of an algebraic *cpo* can be extended to continuous functions on the whole *cpo* in a unique way. This property enables us to turn an algebraic *cpo* into a (Σ, C)-model by defining the operators on the compact elements and making sure that they are monotonic. In this way we may take their unique continuous extension to be their definition on the whole domain.

In §3 a concrete language and its semantics is given. The language is a modification of the original CCS according to the late approach. This is obtained as the (Σ, C)-terms where Σ is instantiated with the standard operators of CCS. Then a concrete denotational model, *Applicative Communication Trees*, is defined, an instantiation of the general class of (Σ, C)-domains. The carrier, an ω-algebraic *cpo*, is defined as a solution to a recursive domain equation, a direct generalization of Abramsky's model for $SCCS$ presented in [Abr92]. Then we apply the described theory to define the operators.

Finally, in 4, we define a proof system to reason about our language and proof it's soundness and completeness with respect to the denotational semantics derived from the model. The ω-algebraicity together with the construction of the operators enables us to reduce the proof of the completeness and soundness to a proof of the same property for a sublanguage which denotes exactly the compact elements of the model.

2 A General Framework for Late Semantics

In this section we will develop a general theoretical framework for describing denotational semantics for value-passing calculi based on the late approach.

2.1 (Σ, C)-Terms

In this subsection we will extend the standard notion of a signature, Σ, and that of Σ-terms used for the pure calculus in order to model processes with value-passing based on the late approach. We do this by introducing the notion of *applicative signature* as a pair, (Σ, C), where Σ is a signature and C is a set (of channel names) and that of (Σ, C)-terms.

We assume some predefined category of expression, Exp, ranged over by e including a countable set of values, Val, ranged over by v, and a set of value variables, Var, ranged over by x. We also assume a predefined category, $BExp$, of boolean expressions, ranged over by be. $BExp$ should at least include equality between the elements of Exp. From that a membership of a finite set can easily be derived. Value expressions are supposed to be equipped with a substitution, denoted by $e[e'/x]$ and an evaluation function $[\![\_]\!] : Exp \times VEnv \longrightarrow Val$ where $VEnv$ is the set of value environments $\sigma : Var \longrightarrow Val$. For closed expression we write $[\![e]\!]$ instead of $[\![e]\!]\sigma$. Further we preassumes an infinite set of process variables, PN, ranged over by P, Q, etc.

The set of (Σ, C)-*terms* is now given as the triplet

$$T_{(\Sigma,C)} = (Proc_{(\Sigma,C)}, Fun_{(\Sigma,C)}, Pair_{(\Sigma,C)})$$

of the sets generated by Σ and C according to the following syntax.

$$\begin{array}{ll} Proc_{(\Sigma,C)}: & p ::= op(\underline{p}) \mid c?.f \mid c!.\pi \mid \tau.p \mid be \rightarrow p, p' \\ Fun_{(\Sigma,C)}: & f ::= [x]p \\ Pairs_{(\Sigma,C)}: & \pi ::= (e, t) \end{array}$$

If the process names in PN are added as primitives to the syntax, we write $T_{(\Sigma,C)}(PN)$, and $T^{rec}_{(\Sigma,C)}(PN)$ if the recursive binding $rec\_.\_$ is also allowed.

We have three kinds of actions, input actions of the form $c?$, $c \in C$, output actions of the form $c!$, $c \in C$ and the silent action τ. We write $C?$ for $\{c? | c \in C\}$ and $C!$ for $\{c! | c \in C\}$. The set $Act = C! \cup C?$ is ranged over by a whereas $Act_\tau = C! \cup C? \cup \{\tau\}$ is ranged over by μ. The action of inputting on channel c is given by $c?$ whereas the action of outputting on that channel is given by $c!$. The abstractions are given by terms of the form $[x]p$, which we call *function terms*, where x is a data variable and p a process term. The input prefixing becomes $c?.[x]p$. The concretions are of the form (e, p), called *pair terms*, where e is a data expression and p a process term and the output prefixing becomes $c!.(e, p)$. We also assume that we have a set of operators, Σ, which at least is supposed to contain the symbol Ω to model the divergent or completely undecided process. Typically Σ contains the standard CCS operators such as NIL, $+$, $\_|\_$, etc. Now the processes are obtained by the input and output prefixing just described, prefixing with the silent action τ and by applying the operators in Σ.

Prefixing by $[x]$ binds the data variable x and the recursion construct is a binding construct for process variables. A value variable, x, is free if it is not in the scope of a prefix, $[x]$, and a process variable P is free if it is not in the scope of a recursion construct, $rec\ P.\_$. We shall mainly be concerned with expressions which contain no free occurrences of value variables. We denote the set of all process terms, functions terms, etc. with no free occurrences of value variables by $CProc_{(\Sigma,C)}$, $CFun_{(\Sigma,C)}$, etc. These will be referred to as processes, functions, etc. ranged over by cp, cf, etc. For $f = [x]p$ we use the convention $f(v) = ([x]p)(v) = p[v/x]$. We assume a notion of substitution for both data and process variables for terms.

2.2 (Σ, C)-Orders and (Σ, C)-Domains

Now we define applicative orders and applicative ordered (Σ, C)-algebras where we use the abbreviations *pro* for pre-order, *po* for partial order and *cpo* for complete partial order:

Definition 2.1 [Applicative Orders] A pair $\langle \overline{A}, \sqsubseteq_A \rangle$ is an applicative *pro/po/cpo* if $\overline{A} = (A_{proc}, A_{fun}, A_{pair})$ is such that

1. $\langle A_{proc}, \sqsubseteq_{A_{proc}} \rangle$ is a *pro/po/cpo*
2. $A_{fun} \subseteq Val \longrightarrow A_{proc}$ and $A_{pair} \subseteq Val \times A_{proc}$ are *pro/po/cpo*s with the standard induced ordering (i.e. $\sqsubseteq_{A_{fun}}$ is the pointwise ordering and $\sqsubseteq_{A_{pair}}$ is defined by: $(v_1, p_1) \sqsubseteq_{A_{pair}} (v_2, p_2)$ if $v_1 = v_2$ and $p_1 \sqsubseteq_{A_{proc}} p_2$).

$\overline{A}$ is said to be *fully applicative* if $A_{proc} = A$, $A_{fun} = Val \longrightarrow A$ and $A_{pair} = Val \times A$ for some A. In that case we refer to $\overline{A}$ as A. An applicative *cpo* is said to be algebraic/ω-algebraic if A_{proc}, A_{fun} and A_{pair} are algebraic/ω-algebraic *cpo*s.

$\square$

Definition 2.2 [(Σ, C)-Orders] A four tuple $\langle \overline{A}, \sqsubseteq_{\overline{A}}, \Sigma_{\overline{A}}, C_{\overline{A}} \rangle$ is an applicative $(\Sigma, C) -$ *pro/po/cpo* if $\overline{A} = (A_{proc}, A_{fun}, A_{pair})$ is such that

1. $\langle \overline{A}, \sqsubseteq_{\overline{A}} \rangle$ is an applicative *pro/po/cpo*.
2. $\langle A_{proc}, \sqsubseteq_{A_{proc}}, \Sigma_A \rangle$ is a $\Sigma -$ *pro/po/cpo*
3. $C_A = C!_A \cup C?_A$ where $C!_{\overline{A}}$ is a set of monotonic/monotonic/continuous functions $c!._{\overline{A}} : A_{pair} \rightarrow A_{proc}$ and $C?_{\overline{A}}$ is a set of monotonic/monotonic/continuous functions $c?_{\overline{A}} : A_{fun} \longrightarrow A_{proc}$.

We refer to the pair (Σ_A, C_A) as a (Σ, C)-pro/po/cpo structure. An ω-algebraic applicative $(\Sigma, C) -$ *cpo* is called a (Σ, C)-*domain*. $\square$

To reduce the notational complexity we allow us to write pro/po/cpo instead of applicative pro/po/cpo. As we are only dealing with applicative orders this should not cause any confusion.

Definition 2.3 A *pro/po/cpo* homomorphism $\overline{h} : \langle \overline{A}, \sqsubseteq_{\overline{A}} \rangle \longrightarrow \langle \overline{B}, \sqsubseteq_{\overline{B}} \rangle$ is a triplet of mappings, $(h_{proc}, h_{fun}, h_{pair})$, where $h_{proc} : A_{proc} \longrightarrow B_{proc}$, $h_{fun} : A_{fun} \longrightarrow B_{fun}$ and $h_{pair} : A_{pair} \longrightarrow B_{pair}$ are monotonic/monotonic/continuous.

A $(\Sigma, C) - $ *pro/po/cpo homomorphism* $\overline{h} : \langle \overline{A}, \sqsubseteq_{\overline{A}}, \Sigma_{\overline{A}}, C_{\overline{A}} \rangle \longrightarrow \langle \overline{B}, \sqsubseteq_{\overline{B}}, \Sigma_{\overline{B}}, C_{\overline{B}} \rangle$, is a triplet, $(h_{proc}, h_{fun}, h_{pair})$, where $h_{proc} : A_{proc} \longrightarrow B_{proc}$ is a $\Sigma -$ *pro/po/cpo* homomorphism, $h_{fun} : A_{fun} \longrightarrow B_{fun}$ and $h_{pair} : A_{pair} \longrightarrow B_{pair}$ are *pro/po/cpo* homomorphisms and satisfy

$$\begin{array}{l} h_{proc}(c?_{\overline{A}}.F) = c?_{\overline{B}}.h_{fun}(F) \text{ and } h_{proc}(c!_{\overline{A}}.\Pi) = c!_{\overline{B}}.h_{pair}(\Pi) \\ h_{fun}(F) = h_{proc} \circ F \\ h_{pair}(v, P) = (v, h_{proc}(P)) \end{array}$$

A pro/po/cpo isomorphism and a (Σ, C)-pro/po/cpo isomorphism are defined in the obvious way. □

Notation 2.4 *For $\overline{A} = (A_1, A_2, A_3)$ and $\overline{B} = (B_1, B_2, B_3)$ we write $\overline{A} \subseteq \overline{B}$ if $A_i \subseteq B_i$ for $i = 1, 2, 3$. If $\overline{f} = (f_1, f_2, f_3)$ then we write $\overline{f} : \overline{A} \longrightarrow \overline{B}$ for $f_i : A_i \longrightarrow B_i$, $i = 1, 2, 3$. We will use a similar pointwise extension of other definitions as well without further explanations.*

We say that

$$\overline{S} = (S_{proc}, S_{fun}, S_{pair}) \subseteq (Proc_{(\Sigma,C)}(PN), Fun_{(\Sigma,C)}(PN), Pairs_{(\Sigma,C)}(PN))$$

is a *recursively closed* subset of $T_{(\Sigma,C)}(PN)$ if whenever $t \in \overline{S}$ then all of its subcomponents are in $\overline{S}$ as well. In this case we write $\overline{S} \subseteq_{rec} T_{(\Sigma,C)}(PN)$. Note that we may apply structural induction when proving properties of recursively closed subsets.

Definition 2.5 Let $\overline{S} \subseteq_{rec} T_{(\Sigma,C)}(PN)$, $\langle \overline{X}, \prec_{\overline{X}}, \Sigma_{\overline{X}}, C_{\overline{X}} \rangle$ an applicative $(\Sigma, C)-$ *pro* and $PEnv_{\overline{X}}$ the set of process environments $\rho : PN \longrightarrow X_{proc}$. A function

$$\overline{X}[\![\_]\!] = (X_{proc}[\![\_]\!], X_{fun}[\![\_]\!], X_{pair}[\![\_]\!]) : \overline{S} \longrightarrow (PEnv_{\overline{X}} \longrightarrow \overline{X})$$

is an evaluation function if whenever $t \in \overline{S}$ then $\overline{X}[\![t]\!]\rho$ satisfies

$$\begin{array}{rcl} X_{proc}[\![op(\underline{p})]\!]\rho &=& op_X(X_{proc}[\![\underline{p}]\!]\rho),\ op \in \Sigma \\ X_{proc}[\![c?.f]\!]\rho &=& c?._X X_{fun}[\![f]\!]\rho \\ X_{proc}[\![c!.\pi]\!]\rho &=& c!._X X_{pair}[\![\pi]\!]\rho \\ X_{proc}[\![\ be \rightarrow p_1, p_2]\!]\rho &=& [\![be]\!] \rightarrow (X_{proc}[\![p_1]\!]\rho), (X_{proc}[\![p_2]\!]\rho) \\ X_{fun}[\![[x]p]\!]\rho &=& \lambda v.X_{proc}[\![p[v/x]]\!]\rho \\ X_{pair}[\![(e,p)]\!]\rho &=& ([\![e]\!], X_{proc}[\![p]\!]\rho) \\ X_{proc}[\![X]\!]\rho &=& \rho(X) \end{array}$$

If A is a *cpo* then, following the standard practice, we may define

$$A_{proc}[\![rec\, X.p]\!]\rho \quad = \quad Y\ \lambda d.A_{proc}[\![p]\!]\rho[d/X]$$

where Y is the least fixed point operator. □

For closed terms the environments do not have any influence on the definition. For terms in $T_{(\Sigma,C)}$ the mapping $\overline{A}[\![ct]\!] = \overline{A}[\![ct]\!]\rho$ may be derived from the definition omitting the last clause of the definition and the occurrence of ρ in the others. We have the following result.

Theorem 2.6 *Let $\overline{S} = (S_{proc}, S_{fun}, S_{pair}) \subseteq_{rec} T_{(\Sigma,C)}(PN)$ and $\langle \overline{X}, \prec_{\overline{X}}, \Sigma_{\overline{X}}, C_{\overline{X}} \rangle$ be an applicative $(\Sigma, C) - pro$. Then there is at most one evaluation mapping*

$$\overline{X}[\![\_]\!] = (X_{proc}[\![\_]\!], X_{fun}[\![\_]\!], X_{pair}[\![\_]\!]) : \overline{S} \longrightarrow (PEnv_{\overline{X}} \longrightarrow \overline{X})$$

If $\langle \overline{X}, \prec_{\overline{X}}, \Sigma_{\overline{X}}, C_{\overline{X}} \rangle$ is fully applicative then such an evaluation mapping exists.

2.3 Properties Derived From the Compact Elements

In this section we will describe how we can turn an algebraic *cpo* into a (Σ, C)-domain if we have a complete description of its compact elements. So let us assume that $\langle \overline{B}, \sqsubseteq_{\overline{B}} \rangle$ is an algebraic *cpo*. Further assume that $\langle \overline{X}, \prec_{\overline{X}} \rangle$ is a preorder where the induced partial order $\langle \overline{X}/\sim, \prec_{\overline{X}/\sim} \rangle$, $\sim = \prec \cap \prec^{-1}$ is isomorphic to $\langle Co(\overline{B}), \sqsubseteq_{Co(\overline{B})} \rangle$ under $\overline{\phi} : \langle \overline{X}/\sim, \prec_{\overline{X}} \rangle \longrightarrow \langle Co(\overline{B}), \sqsubseteq_{Co(\overline{B})} \rangle$. By defining a monotonic (Σ, C) structure on $\langle \overline{X}, \prec_{\overline{X}} \rangle$ we get automatically an induced monotonic structure on $\langle \overline{X}/\sim, \prec_{\overline{X}/\sim} \rangle$ and thereby on the isomorphic $\langle Co(\overline{B}), \sqsubseteq_{Co(\overline{B})} \rangle$. Now, because of the algebraicity this structure may be extended in a unique way to a continuous structure on $\langle \overline{B}, \sqsubseteq_{\overline{B}} \rangle$. This procedure may be described formally by the following Theorem. We note that the function $\overline{\psi} = \overline{inc} \circ \overline{\phi} \circ [\_]$ where $\overline{inc} : \langle Co(\overline{B}), \sqsubseteq_{Co(\overline{B})} \rangle \longrightarrow \langle \overline{B}, \sqsubseteq_{\overline{B}} \rangle$ is the inclusion mapping and $[\_] : \langle \overline{X}, \prec_{\overline{X}} \rangle \longrightarrow \langle \overline{X}/\sim, \preceq_{\overline{X}/\sim} \rangle$ is the quotient mapping, is a *po*-homomorphism.

Theorem 2.7 *Assume that $\langle \overline{X}, \prec_{\overline{X}}, \Sigma_{\overline{X}}, C_{\overline{X}} \rangle$ is a $(\Sigma, C) - pro$ with $\sim = \prec_{\overline{X}} \cap \prec_{\overline{X}}^{-1}$ and the induced partial order $\langle \overline{X}/\sim, \preceq_{\overline{X}/\sim} \rangle$. Further assume that $\langle \overline{B}, \sqsubseteq_{\overline{B}} \rangle$ is an applicative algebraic cpo where the po of compact elements is isomorphic to $\langle \overline{X}/\sim, \preceq_{\overline{X}/\sim} \rangle$ under the isomorphism*

$$\phi : \langle \overline{X}/\sim, \preceq_{\overline{X}/\sim} \rangle \longrightarrow \langle Co(\overline{B}), \sqsubseteq_{Co(\overline{B})} \rangle$$

Then the following holds:

1. *There exists a unique (Σ, C)-cpo structure, $(\Sigma_{\overline{B}}, C_{\overline{B}})$ which extends $\langle \overline{B}, \sqsubseteq_{\overline{B}} \rangle$ to a (Σ, C)-domain and extends $\overline{\psi} = \overline{inc} \circ \overline{\phi} \circ [\_]$ to a (Σ, C)-homomorphism. The structure $(\Sigma_{\overline{B}}, C_{\overline{B}})$ preserves compactness.*

2. *Let $\overline{B}[\![\_]\!] : T^{rec}_{(C,\Sigma)}(X) \longrightarrow \overline{B}$ be an evaluation mapping. If $S \subseteq_{rec} T_{(C,\Sigma)}$ and $X[\![\_]\!] : S \longrightarrow X$ is an evaluation mappings then for all $t \in \overline{S}$:*

$$\overline{B}[\![t]\!] = \overline{\psi}(X[\![t]\!]).$$

3 The Language Late-*CCS* and its Semantics

In this section we will give a concrete language and define a denotational model for it. We also give an equationally based proof system and show its soundness and completeness with respect to the denotational model.

3.1 A Domain Equation for Applicative Communication Trees

The language *Late-CCS*, is a modification of the original CCS in the spirit of the late approach. As described in the Introduction it is basically a sublanguage of the more general π-calculus in which only simple values are transmitted in communications whereas the latter allows port names to be transmitted as well as simple values. Described in our general framework Late-CCS is obtained by instantiating the signature Σ by the standard operator of CCS. So we let Σ consist of the nullary operators NIL and Ω, the families of unary operators $\_ \setminus c, c \in C$ and $\_[R]$ where R is a finite permutation of the process names, and the binary operators $+$ and $|$. For the motivation of these operators we refer to the standard theory of CCS in [Mil89].

Next we construct a (Σ, C)-domain to give a denotational semantics for our language. The definition of this model is motivated by the following existing work.

In 1979 Milne and Milner [MM79], gave a domain theoretical definition of the concept of *communicating processes*. This definition reflects the late semantic approach described above. Each process has a collection of typed ports through which it may communicate with other processes. There are two types of communications; input and output. If we abstract from the types then input capability of a process p along a channel c is modelled as an element of the domain $V \longrightarrow P$ labelled by the channel name c, where the domain of processes is denoted by the *cpo* P and the domain of values by V. An output capability of p on c, on the other hand, is modelled as an element of $V \times P$ labelled by c. A process is modelled as a set of communication capabilities or more precisely as an element of the Smyth power-domain [Sm78], over the domain of communication capabilities. The empty set is embedded into the domain in such a way that it becomes the top element of the domain. This leads to a recursive definition of the process domain which is defined as the initial solution of corresponding domain equation.

In [Abr92] Abramsky pointed out a disadvantage of this model: the use of the Smyth power-domain to model communicating processes rules out the possibility of any correspondence with bisimulation. Also the embedding of the empty set (which corresponds to the inactive and convergent process) as the top element of the model is intuitively incorrect. In the same reference the author defines a model to describe the semantics of pure processes. This model is similar to the model of [MM79] and is also obtained as the initial solution to a recursive domain equation. The main difference is that Abramsky defines his model in terms of the Plotkin power-domain instead of the Smyth power-domain. He adds the empty set to the model as an isolated element only comparable with itself and the bottom element of the model in the obvious way. He then interprets the calculus $SCCS$ in the model and shows the full abstractness of this interpretation with respect to strong ω-prebisimulation.

The model we define is basically the model of [MM79] where the modifications of Abramsky's are adapted. Thus we define a model which models value-passing based on the late approach using the Plotkin power-domain with the empty set adjoined as an isolated element.

Here the main difference is that we use a different representation for Plotkin power-domain to the one used in [Abr92]. The representation we use is the one due to Smyth, [Sm78] and may be described as follows:

Let $\langle A, \sqsubseteq \rangle$ be an ω-algebraic *cpo* and $M[A]$ the family of finite, non-empty sets of compact elements of A. The Egli-Milner preorder on $M[A]$, $\sqsubseteq_{EM(A)}$, is defined by

$$X \sqsubseteq_{EM(A)} Y \text{ if } \forall x \in X \exists y \in Y. x \sqsubseteq_A y \text{ and } \forall y \in Y \exists x \in Y. y \sqsubseteq_A x$$

The Plotkin power-domain of $\langle A, \sqsubseteq \rangle$, $\langle P[A], \sqsubseteq_{P[A]} \rangle$ is the ideal completion of the preorder $\langle M[A], \sqsubseteq_{EM(A)} \rangle$. From the standard theory, [Plo81b], we know that $\langle P[A], \sqsubseteq_{P[A]} \rangle$ is an ω-algebraic *cpo* and the *po* of compact elements is isomorphic to the *po* derived from the preorder $\langle M[A], \sqsubseteq_{M[A]} \rangle$. Given an ω-algebraic *cpo* we write $P^0[D]$ for the Plotkin power-domain over D with the empty set adjoined as an isolated element in the preorder. More precisely the elements of $P^0[D]$ are given by $P[D] \cup \{\emptyset\}$ with the order:

$$X \sqsubseteq_{P^0[D]} Y \quad \text{if} \quad \begin{array}{rl} & X, Y \in P[D] \text{ and } X \sqsubseteq_{P[D]} Y \\ \text{or} & Y = \{\emptyset\} \text{ and } (X = \{\emptyset\} \text{ or } X = \bot) \end{array} \tag{1}$$

Now we can give the following definition:

Definition 3.1 [Applicative Communication Trees] Let $Chan$ (the set of channels) and Val (the set of values) be countable sets and $Act = \{c? | c \in Chan\} \cup \{c! | c \in Chan\} \cup \{\tau\}$ (the set of actions). We define the applicative *cpo* of applicative communication trees, $\langle \overline{ACT}, \sqsubseteq \rangle$, as follows: $\langle ACT, \sqsubseteq \rangle$ is the initial solution in the category of cpos with embeddings of the recursive domain equation:

$$D = P^0[\sum_{e \in Act} D_e]$$

where $D_{c?} = Val \to D$, $D_{c!} = Val \times D$ and $D_\tau = D$. Then we define $ACT_{proc} = ACT$, $ACT_{fun} = Val \longrightarrow ACT$ and $ACT_{pair} = Val \times ACT$ with the usual induced order. $\langle \overline{ACT}, \sqsubseteq_{\overline{ACT}} \rangle$ is a fully applicative *cpo* which we refer to as ACT. □

By unfolding the recursive definition of ACT, we get a description of the compact elements from the standard theory. Thus we define

$$Co(ACT) = \textstyle\bigcup_{n=0}^{\infty} Comp^n$$

where

$$Comp^0 = \{\bot\}$$

and

$$\begin{aligned} Comp^{n+1} &= Co(P^0[\textstyle\sum_{e \in Act} (Comp^n)_e]) \\ &= M[\textstyle\sum_{e \in Act} (Comp^n)_e] \cup \{\emptyset\} \end{aligned}$$

where $(Comp^n)_{c?} = Val \longrightarrow Comp^n$, $(Comp^n)_{c!} = Val \times Comp^n$ and $(Comp^n)_\tau = Comp^n$. We recall that for an algebraic *cpo* A, $M[A]$ is defined as the family of non-empty sets of compact elements of A. The empty set is added to the family $Comp^n$ as we are using the power-domain operator P^0 rather than P. Defining $Co(ACT)$ this way and ordering it by $\sqsubseteq^0_{EM}$, the Egli-Milner preorder over $Co(ACT)$ extended as described above, gives a representation of the compact elements of the model ACT. This suggests an inductive definition to describe the compact elements and the preorder on these. Thus we define the set K and the preorder $\prec$ on it inductively and prove that $\langle K, \prec\rangle$ is a representation of $\langle Co(ACT), \sqsubseteq^0_{EM}\rangle$

Definition 3.2 We define K as the least set which satisfies

1. $\emptyset \in K$
2. $\{\perp\} \in K$
3. $c \in Chan, V \subseteq_{fin} Val$ and $\forall v \in V.\, k_v \in K$ implies $\{\langle c?, \lambda v.x \in V \longrightarrow k_v, \Omega\rangle\} \in K$
4. $c \in Chan, v \in Val, k \in K$ implies $\{\langle c!, (v, k)\rangle\} \in K$
5. $k \in K$ implies $\{\langle \tau, k\rangle\} \in K$
6. $k_1, k_2 \in K$ implies $k_1 \cup k_2 \in K$

The preorder $\prec$ is defined as the least preorder on K which satisfies

1. $\{\perp\} \prec \emptyset$
2. $k_1 \prec k_2$ if $\forall a \in k_1 \exists b \in k_2.\, a \leq b$ and $\forall b \in k_1 \exists a \in k_2.\, a \leq b$ where $\leq$ is defined on the elements of the sets in K by
   (a) $\forall a.\, \perp \leq a$
   (b) $\langle \tau, k\rangle \leq \langle \tau, k'\rangle$ if $k \prec k'$
   (c) $\langle c?, f\rangle \leq \langle c?, g\rangle$ if $\forall v \in Val. f(v) \prec g(v)$
   (d) $\langle c!, (v, k)\rangle \leq \langle c!, (v, k')\rangle$ if $k \prec k'$

We let $\approx = \prec \cap \prec^{-1}$. □

Proposition 3.3 $\langle K/\approx, \prec_{K/\approx}\rangle = \langle Co(ACT), \sqsubseteq^0_{EM}\rangle$ *(where we also use* $\sqsubseteq^0_{EM}$ *for the partial order induced by* $\sqsubseteq^0_{EM}$ *on* $Co(ACT)$*).*

Definition 3.4 We define $\langle \overline{K}, \prec_{\overline{K}}\rangle$ by $K_{proc} = K$, $K_{fun} = Fin(Val \longrightarrow K)$ and $K_{pair} = Val \times K$ and $\prec_{\overline{K}}$ is the induced preorder. □

3.2 Definition of the Operators in the Model

In this subsection we will define the operators in $\Sigma_{\overline{ACT}}$ and prove their continuity. In the definitions we take advantage of Theorem 2.7. Thus we only have to define the operators on the applicative *pro* $\langle \overline{K}, \prec_{\overline{K}}\rangle$ and make sure that they are monotonic.

Definition 3.5 We define $\Sigma_{\overline{K}}$ as follows:

$$
\begin{array}{lrcl}
\textbf{Constants:} & NIL_{\overline{K}} & = & \emptyset \\
& \Omega_{\overline{K}} & = & \{\bot\} \\
\textbf{Prefixing:} & c?_{\overline{K}}.\_ & = & \lambda f.\{\langle c?, f\rangle\} \\
& c!_{\overline{K}}.\_ & = & \lambda(v,k).\{\langle c!, (v,k)\rangle\} \\
& \tau_{\overline{K}}.\_ & = & \lambda k.\{\langle \tau, k\rangle\}
\end{array}
$$

Nondeterminism: $+_{\overline{K}} \;=\; \cup$

Restriction: $\_\backslash c_{\overline{K}} = F_c$ where $F_c : K_{proc} \to K_{proc}$ is defined by

$$
\begin{array}{rcl}
F_c\{\bot\} & = & \{\bot\} \\
F_c(\emptyset) & = & \emptyset \\
F_c\{\langle b?, f\rangle\} & = & \begin{cases} \{\langle b?, F_c \circ f\rangle\} & \text{if } b \neq c \\ \emptyset & \text{otherwise} \end{cases} \\
F_c\{\langle b!, (v,k)\rangle\} & = & \begin{cases} \{\langle b!, (v, F_c k)\rangle\} & \text{if } b \neq c \\ \emptyset & \text{otherwise} \end{cases} \\
F_c\{\langle \tau, k\rangle\} & = & \{\langle \tau, F_c k\rangle\} \\
F_c(k_1 \cup k_2) & = & (F_c k_1) \cup (F_c k_2)
\end{array}
$$

Renaming: $\_[R]_{\overline{K}} = G_R$ where $G_R : K_{proc} \to K_{proc}$ is defined by

$$
\begin{array}{rcl}
G_R\{\bot\} & = & \{\bot\} \\
G_R\emptyset & = & \emptyset \\
G_R\{\langle c?, f\rangle\} & = & \{\langle R(c)?, G_R \circ f\rangle\} \\
G_R\{\langle c!, (v,k)\rangle\} & = & \{\langle R(c)!, (v, G_R k)\rangle\} \\
G_R\{\langle \tau, k\rangle\} & = & \{\langle \tau, G_R k\rangle\} \\
G_R(k_1 \cup k_2) & = & (G_R(k_1)) \cup (G_R(k_2))
\end{array}
$$

Parallel Composition: $\_\,|_{\overline{K}}\,\_ = F$ where

$$
\begin{array}{rcl}
F & = & int \cup comm \cup div \\
int & = & int_{in} \cup int_{out} \cup int_{\tau} \\
int_{in}(x,y) & = & \{\langle c_x?, \lambda v.F(f_x(v)), y\rangle | \langle c_x?, f_x\rangle \in x\} \\
& \cup & \{\langle c_y?, \lambda v.F(x, f_y(v))\rangle | \langle c_y?, f_y\rangle \in y\} \\
int_{out}(x,y) & = & \{\langle c_x!, (v, F(x', y)\rangle | \langle c_x!, (v, x')\rangle \in x\} \\
& \cup & \{\langle c_y!, (v, F(x, y')\rangle | \langle c_y!, (v, y')\rangle \in y\} \\
int_{\tau} & = & \{\langle \tau, F(x', y)\rangle | \langle \tau, x'\rangle \in x\} \\
& \cup & \{\langle \tau, F(x, y')\rangle | \langle \tau, y'\rangle \in y\} \\
comm(x,y) & = & \{\langle \tau, F(f(v), y')\rangle \mid \exists c, v. \langle c?, f\rangle \in x \text{ and } \langle c!, (v, y')\rangle \in y\} \\
& = & \{\langle \tau, F(x', g(v))\rangle \mid \exists c, v. \langle c?, g\rangle \in y \text{ and } \langle c!, (v, x')\rangle \in x\}
\end{array}
$$

$$div(x,y) \quad = \quad \begin{cases} \{\bot\} & \text{if } \bot \in x \cup y \\ \emptyset & \text{otherwise} \end{cases}$$

□

Lemma 3.6 *$\langle \overline{K}, \prec_{\overline{K}}, \Sigma_{\overline{K}}, C_{\overline{K}} \rangle$ is a (Σ, C)-pro.*

We finish this section by summarizing our results. This is done by the following theorem.

Theorem 3.7 *$\langle \overline{ACT}, \sqsubseteq_{\overline{ACT}}, \Sigma_{\overline{ACT}}, C_{\overline{ACT}} \rangle$, where $(\Sigma_{\overline{ACT}}, C_{\overline{ACT}})$ is the (unique) (Σ, C)-po structure induced by $\langle \overline{K}, \prec_{\overline{K}}, \Sigma_{\overline{K}}, C_{\overline{K}} \rangle$, is a fully applicative (Σ, C)-domain.*

3.3 Syntactically Compact Elements

In this last subsection we will show that the compact elements of the model $\overline{ACT}$ may be denoted in our syntax. For this purpose we introduce the *syntactically compact Terms*, $CoTerms = (CoProc, CoFun, CoPair)$.

As usual syntactically finite terms are those without occurrences of recursion. We define compact terms as the syntactically finite ones which only use a finite number of values in a nontrivial way. Note that in this case the number of channels used by the term is automatically finite. We start by introducing some notation.

Notation 3.8 *Let $\underline{v_n} = (v_1, \ldots, v_n)$ and $\underline{p}_n = (p_1, \ldots, p_n)$. We write $x : \underline{v_n} \longrightarrow \underline{x_n}$ for $x = v_1 \longrightarrow p_1, (x = v_2 \longrightarrow p_2, (\ldots x = v_n \longrightarrow p_n, \Omega) \ldots)$. Further we let $\{\underline{v_n}\} = \{v_i | \underline{v_n} = (v_1, \ldots, v_n)\}$ and similarly for $\{p_n\}$.*

Definition 3.9 [Compact Terms] The set of compact terms, $CoTerms$, is the triplet $(CoProc, CoFun, CoPairs)$ where the sets $CoProc$, $CoFun$ and $CoPairs$ are the least sets satisfying:

1. $NIL, \Omega \in CoProc$
2. $\overline{p} \in CoProc$ implies $op(\overline{p}) \in CoProc$, $op = |, +, \_\backslash, \_[R], \tau.\_$
3. $\pi \in CoPair$, $c \in Chan$ implies $c!.\pi \in CoProc$
4. $f \in CoFun$ and $c \in Chan$ implies $c?.f \in CoProc$
5. $p \in CoProc$ and $e \in Exp$ implies $(e, p) \in CoPairs$
6. $\{\underline{p_n}\} \subseteq CoProc$, $\{\underline{v_n}\} \subseteq Val$ and $x \in Var$ implies $[x]. x : \underline{v_n} \longrightarrow \underline{p_n} \in CoFun$.

We say that a term is *compact* if it belongs to $CoTerms$. We let Cot, Cop, etc. range over $CoTerms$, $CoProc$, etc.

□

Note that $CoTerms = (CoProc, CoFun, CoPair) \subseteq_{rec} (Proc, Fun, Pairs)$. We have the following:

Theorem 3.10

1. *$\overline{ACT}[\![\_]\!]$ and $K[\![\_]\!]$ are well-defined and for all $t \in CoTerms$: $\overline{ACT}[\![t]\!]\rho = [\overline{K}[\![t]\!]]_{\approx}$ where $[\_]_{\approx}$ is the quotient mapping with respect to $\prec$.*

2. *For any $Cot \in CoTerms$ $\overline{ACT}[\![Cot]\!] \in Co(\overline{ACT})$*

3. *For all $k \in Co(\overline{ACT})$ there is a $Cot \in CoTerms$ such that $\overline{ACT}[\![Cot]\!] = k$.*

4 Algebraic Laws and Proof Systems

In this section we will introduce a proof system supported by the model $\overline{ACT}$. We proceed by introducing first a system, E, for finite processes which we then extend to a system, E_{rec}, which takes care of recursive processes. We prove the soundness and completeness of E_{rec} with respect to the model.

The proof system E is equationally based where the equations reflect naturally the properties of the operators in the model.

The inference rules describe the structure and the preorder in the model and their interaction with the operators. Because of the two level structure of our syntax, we have the equations

$$\begin{array}{ll} (res\,fun) & ([x]X) \setminus c = [x](X \setminus c) \\ (res\,pair) & (e, X) \setminus c = (e, X \setminus c) \\ (ren\,fun) & ([x]X)[R] = [x](X[R]) \\ (ren\,pair) & (e, X)[R] = (e, X[R]) \end{array}$$

and the rules

$$\begin{array}{ll} (fun) & \dfrac{p[v/x] \sqsubseteq q[v/x] \text{ for every } v \in V}{[x]p \sqsubseteq [x]q} \\ (pair) & \dfrac{[\![e_1]\!] = [\![e_2]\!], p \sqsubseteq q}{(e_1, p) \sqsubseteq (e_2, q)} \end{array}$$

The extended system E_{rec} is then obtained by adding to E three new rules, $(rec1)$, $(rec2)$ and $(\omega - rule)$, to take care of recursion. These rules are all fairly standard. The compact approximations, $p^{(n)}$, used in the rule $(\omega - rule)$ are defined in Definition 4.1.

In the rule $(res\ pre)$ we use the notation $chan(pre)$, where $chan(c?) = chan(c!) = c$. In the interleaving law the summation notation is justified by equations (+1)-(+4) and an empty sum is understood as NIL. $\{+\Omega\}$ indicates that Ω is an optional summand of a term and Ω is a summand of the right hand side if it is either a summand of X or Y. To simplify the notation we assume that i, j etc. in the sums $\sum_i$, $\sum_j$, etc. range over finite index sets I, J, etc.

We now define the compact approximations used in the ω-rule of the proof system.

$$
\begin{array}{lrcl}
(+1) & X+(Y+Z) & = & (X+Y)+Z \\
(+2) & X+Y & = & Y+X \\
(+3) & X+X & = & X \\
(+4) & X+NIL & = & X \\
(res+) & (X+Y)\setminus c & = & X\setminus c+Y\setminus c \\
(res\,pre) & (a.X)\setminus c & = & \begin{cases} a.(X\setminus c) & \text{if } c\neq chan(pre) \\ NIL & \text{otherwise} \end{cases} \\
(res\,fun) & ([x]X)\setminus c & = & [x](X\setminus c) \\
(res\,pair) & (e,X)\setminus c & = & (e,X\setminus c) \\
(res\,NIL) & NIL\setminus c & = & NIL \\
(res\,div) & \Omega\setminus c & = & \Omega \\
(ren+) & (X+Y)[R] & = & X[R]+Y[R] \\
(ren\,pre) & (a.X)[R] & = & R(a).X[R] \\
(ren\,fun) & ([x]X)[R] & = & [x](X[R]) \\
(ren\,pair & (e,X)[R] & = & (e,X[R]) \\
(ren\,NIL) & NIL[R] & = & NIL \\
(ren\,div) & \Omega[R] & = & \Omega \\
(NIL\,par) & NIL\mid X & = & X\mid NIL\ =\ X \\
(div) & \Omega & \sqsubseteq & X
\end{array}
$$

Let $X = \sum_i \tau.X_i + \sum_j a_j?.[x]X'_j + \sum_k a'_k!.(v_k, X''_k)\{+\Omega\}$ and $Y = \sum_l \tau.Y_l + \sum_m b_m?.[y]Y'_m + \sum_n b'_n!.(v_n, Y''_n)\{+\Omega\}$. Then

$$X \mid Y = INTL(X,Y) + COMM(X,Y)\{+\Omega\}$$

where

$$INTL(X,Y) = INTL_\tau(X,Y) + INTL_{in}(X,Y) + INTL_{out}(X,Y)$$

where

$$
\begin{array}{l}
INTL_\tau(X,Y) = \sum_i \tau.(X_i|Y) + \sum_l \tau.(X|Y_l) \\
INTL_{in}(X,Y) = \sum_j a_j?.[x](X'_j|Y) + \sum_m b_m?.[y](X|Y'_m) \\
INTL_{out}(X,Y) = \sum_k a'_k!.(v_k, X''_k|Y) + \sum_n b'_n!.(v'_n, X|Y''_n)
\end{array}
$$

and

$$COMM(X,Y) = \sum_{j,n:a_j=b'_n} \tau.X'_j[v_n/x]|Y''_n + \sum_{k,m:a'_k=b_m} \tau.X''_k|Y'_m[v_k/y]$$

Figure 1: Equations

$$(ref) \quad p \sqsubseteq p \qquad\qquad (trans) \quad \frac{p \sqsubseteq q,\ q \sqsubseteq r}{p \sqsubseteq r}$$

$$(sub) \quad \frac{p_i \sqsubseteq q_i}{op(\underline{p}) \sqsubseteq op(\underline{q})} \qquad op \in \sum$$

$$(pre) \quad \frac{p \sqsubseteq q}{\mu.p \sqsubseteq \mu.q}$$

$$(rec1) \quad \frac{p \sqsubseteq q}{recP.p \sqsubseteq recP.q} \qquad\qquad (rec2) \quad \frac{}{recP.p = p[recP.p/P]}$$

$$(\omega - rule) \quad \frac{p^{(n)} \sqsubseteq q \text{ for all } n}{p \sqsubseteq q}$$

$$(inst) \quad \frac{}{p\sigma \sqsubseteq q\sigma} \text{ for every inequation } p \sqsubseteq q \qquad \text{and instantiation } \sigma$$

$$(cond1) \quad \frac{[\![be]\!] = T}{be \longrightarrow p, q = \ p} \qquad\qquad (cond2) \quad \frac{[\![be]\!] = F}{be \longrightarrow p, q = \ q}$$

$$(pair) \quad \frac{[\![e]\!] = [\![e']\!], p \sqsubseteq q}{(e, p) \sqsubseteq (e', q)}$$

$$(fun) \quad \frac{p[v/x] \sqsubseteq q[v/x] \text{ for every } v \in V}{[x]p \sqsubseteq [x]q}$$

$$(\alpha - red) \quad \frac{}{[x]p = [y]p[y/x]} \qquad \text{if } y \text{ not free in } p$$

Figure 2: The Proof System E_{rec}

Definition 4.1 [Compact Approximations] The n-th compact approximation of a term is defined inductively by following:

I. $i) p^0 = \Omega$
   $ii)$ $1. P^{(n+1)} = P$
   $2. (op(\underline{p}))^{(n+1)} = op(\underline{p}^{(n+1)})$
   $3. (\mu.u)^{(n+1)} = \mu.u^{(n+1)}$
   $5. (recP.p)^{(n+1)} = p^{(n+1)}[(recP.p)^{(n)}/P]$
   $6. (be \longrightarrow p, u)^{(n+1)} = be \longrightarrow p^{(n+1)}, u^{(n+1)}$

II. $([x]p)^{(n+1)} = [x](x \in V_{n+1} \longrightarrow p^{(n+1)}, \Omega)$

III. $((v, p))^{(n+1)} = (v, p^{(n+1)})$

□

The compact approximations have the following property:

Theorem 4.2 *For all n and all $ct \in CTerms$*

1. $ct^{(n)} \in CoTerms$
2. $[\![ct]\!]\rho = \bigsqcup_n [\![ct^{(n)}]\!]\rho$

We will end this section by stating the soundness and completeness of the proof system E_{rec} with respect to the denotational semantics. For the proof we refer to the full version of the paper.

Theorem 4.3 [Soundness and Completeness] *For all closed terms ct, cu in $CTerms$, $ct \sqsubseteq_{E_{rec}} cu$ if and only if $\overline{\mathrm{ACT}}[\![ct]\!] \sqsubseteq \overline{\mathrm{ACT}}[\![cu]\!]$, i.e. the proof system E_{rec} is sound and complete with respect to the denotational semantics.*

5 Conclusion

In the first part of this paper we have set up a general framework for describing the syntax and the denotational semantics for value-passing calculi which support the late approach. For this purpose the standard notion of signature and Σ-algebras and -orders have been extended to the so-called applicative signatures and (Σ, C)-algebras and -orders. Furthermore we show how we may take an advantage of the ω-algebraicity of a model to define the operators in the model.

In the second part of the paper we define the language Late-CCS which is a modification of the standard CCS with values due to the late semantic approach. This language is basically the π-calculus where the values allowed are restricted to be of the simple type only.

A denotational model for Late-CCS is defined, an instantiation of the general class of models we defined. The carrier set of this model is an ω-algebraic cpo and is obtained as a solution to an recursive domain equation and is a direct extension of the model defined by Abramsky in [Abr92].

The operators are constructed by isolating the compact elements, define the operators in the derived partial order, make sure that they are monotonic which then allows us to extend them to continuous functions defined the whole *cpo*.

In the last subsection we define a equationally based proof system and show its soundness and completeness with respect to the model.

As already explained the construction of the domain equation and thereby the definition of the model is strongly inspired by the idea of bisimulation preorder. In the accompanying paper [Ing94a] it is shown that the original late bisimulation is too strong to meet the preorder of the model ACT. The algebraicity of the model implies that the preorder in the model is completely decided by the compact elements. Behaviourally this can be interpreted as that the preorder may be obtained by finite observations. This is not the case for bisimulation as well known even for the pure calculus. In [Ing94a] a finitary version of the preorder is defined by mimicking the ω-algebraicity of the model on the syntactical level. This preorder is shown to coincide with the preorder in the model in the sense that the model is fully abstract with respect to this preorder.

References

[Abr92] S. Abramsky: A Domain Equation For Bisimulation, *Information and Computation*, Vol. 92, pp 161-218, 1992.

[EN86] Engberg, U., Nielsen, M., "A Calculus of Communicating Systems with Label-passing", Report DAIMI PB-208, Computer Science Department, University of Aarhus, 1986.

[Gue81] Guessarian, I., "Algebraic Semantics", *Springer-Verlag Lecture Notes in Computer Science*, vol.99, 1981.

[HI93a] Hennessy, M., Ingólfsdóttir, A. A Theory of Communicating Processes With Value - Passing, *Information and Computation*, Vol. 107, no. , pp 202-236, 1993.

[HM85] M. Hennessy , R. Milner: *Algebraic Laws for Nondeterminism and Concurrency*, Journal of the Association for Computing Machinery, vol. 32, No. 1, pp. 137-161, 1985.

[HP80] Hennessy, M. and Plotkin, G., "A Term Model for CCS", *Springer-Verlag Lecture Notes in Computer Science*, vol.88, 1980.

[He88] Hennessy, M. *Algebraic Theory of Processes*. MIT Press, Cambridge, 1988.

[Hoa78] Hoare, C.A.R. "Communicating Sequential Processes", *Comm. ACM*, 21(8), 666-677 1978.

[Ing93] Ingólfsdóttir, A. Late and Early Semantics Coincide for Testing, Aalborg University Technical Report, IR 93-2008, 1993. To appear in *Theoretical Computer Science*.

[Ing94] Ingólfsdóttir, A., Semantic Models for Communicating Processes with Value-passing, Ph.D Thesis, Technical Report 8/94, Sussex University, 1994.

[Ing94a] Ingólfsdóttir, A., Semantic Theory for Value-Passing Processes, Late Approach, Part II: Behavioural Semantics and Full Abstractness. In preparation.

[IT92] A. Ingólfsdóttir, B. Thomsen. Semantic Models for *CCS* with Values. Technical Report 63, Programming Methodology Group, Chalmers University of Technology, 1992. In Proceedings of the Workshop on Concurrency.

[MM79] G. Milne , R. Milner: *Concurrent Processes and Their Syntax*, Journal of the Association for Computing Machinery, Vol. 26, No. 2, pp. 302-321, 1979.

[Mil80] Milner, R. *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92. Springer-Verlag, Berlin, 1980.

[Mil83] Milner, R. *Calculi for Synchrony and Asynchrony*, Theoretical Computer Science, 25, pp 267–310, 1983

[Mil89] Milner, R., *Communication and Concurrency*, Prentice-Hall, London 1989.

[Mil91] R. Milner, Polyadic π-Calculus. A tutorial. To appear in *Proceedings of the International Summer School on Logic and Algebra of Specifications*, Marktoberdorf, 1991.

[MPW89] R. Milner, J. Parrow & D. Walker, "A Calculus of Mobile Processes" Part I+II, *Information and Computation*, Vol. 100 no. 1, pp 1-77, 1992.

[MPW91] R. Milner, J. Parrow & D. Walker, *Modal Logics for Mobile Processes* Proceedings of Concur '91 , Lecture Notes in Computer Science 327, pp. 45-60, Springer-Verlag, 1991.

[Plo76] G. Plotkin: *A Powerdomain Construction*, SIAM Journal on Computing, 5 (1976), pp. 452-487, 1976.

[Plo81b] Plotkin, G., "Lecture Notes in Domain Theory", University of Edinburgh, 1981.

[Sm78] Smyth, M., Powerdomains, *JCSS* 16(1), 1978

[SP 82] Smyth, M. and Plotkin, G., "The Category-Theoretic Solution of Recursive Domain Equations", *SIAM Journal on Computing*, vol.11, No.4, 1982.

[Win85] Winskel, G., On Powerdomains and Modality, *Theoretical Computer Science*, 36, pp. 127- 137, 1985.

Interpreting Broadcast Communication in CCS with Priority Choice

Claus Torp Jensen *ctjensen@cs.chalmers.se*
Computer Science Department
Chalmers University of Technology

November 29, 1994

1 Introduction

CBS (Calculus of Broadcasting Systems) is a process calculus in which *broadcast* is the fundamental communication paradigm. Communication between processes is performed through unbuffered broadcasts. One of the processes broadcasts an output and the remaining processes either receive or ignore the broadcast according to whether they are listening or not. Consequently any output is autonomous and may be performed in any environment. Reception on the other hand is controlled by the environment in the sense that a process has to continue listening until something is being broadcast.

Clearly CBS is a synchronous calculus. It has been shown in [1] that broadcasting communication may be interpreted in a different kind of synchronous calculus, namely $SCCS$. An interesting question is whether it is also possible to interpret broadcast communication in an asynchronous handshake model like CCS. In [1] it is argued that broadcast communication cannot be interpreted in pure CCS. We believe this is true. On the other hand extensions of CCS may provide the additional control of communication that is needed in order to simulate the "one-to-many" synchronization in CBS.

In this paper we show that by adding a priority choice operator to CCS, any CBS system may be translated to a term in CCS with priority choice. A CBS synchronization must necessarily correspond to a *sequence* of transitions in the CCS model, and the priority choice operator enables us to ensure that such sequences terminate correctly. The translation is correct in the sense that two CBS terms are strongly bisimilar iff their translations are weakly bisimilar. Due to the multiway synchronizations in CBS this is the simplest relation between a term and its translation that we can hope for.

The reader is expected to have some basic knowledge of process calculi like CCS and of operational semantics in the style of Plotkin [6]. We give brief descriptions of the semantics of CBS and CCS with priority choice, but it is probably preferable to consult the references for better descriptions of these two calculi.

<table>
<tr><td>Operator</td><td>Transmit</td><td>Read</td><td>Discard</td></tr>
<tr><td>nil</td><td></td><td></td><td>$nil \xrightarrow{\mu:} nil$</td></tr>
<tr><td>Monitor</td><td></td><td>$a?p \xrightarrow{a?} p$</td><td>$a?p \xrightarrow{\mu:} a?p \; (\mu \neq a)$</td></tr>
<tr><td>Transmit</td><td>$\mu!p \xrightarrow{\mu!} p$</td><td></td><td>$\mu!p \xrightarrow{\nu:} \mu!p$</td></tr>
<tr><td>Choice[1]</td><td colspan="2">$\dfrac{p \xrightarrow{\mu\natural} p'}{p+q \xrightarrow{\mu\natural} p'}$</td><td>$\dfrac{p \xrightarrow{\mu:} p' \quad q \xrightarrow{\mu:} q'}{p+q \xrightarrow{\mu:} p'+q'}$</td></tr>
<tr><td>Recursion</td><td colspan="2">$\dfrac{e\{rec\,x.e/x\} \xrightarrow{\mu\natural} p'}{rec\,x.e \xrightarrow{\mu\natural} p'}$</td><td>$\dfrac{e\{rec\,x.e/x\} \xrightarrow{\mu:} p'}{rec\,x.e \xrightarrow{\mu:} rec\,x.e}$</td></tr>
<tr><td>Parallel</td><td colspan="3">$\dfrac{p \xrightarrow{\mu\sharp_1} p' \quad q \xrightarrow{\mu\sharp_2} q'}{p|q \xrightarrow{\mu(\sharp_1 \bullet \sharp_2)} p'|q'} \; (\sharp_1 \bullet \sharp_2 \neq \perp)$ $\quad \begin{array}{c|ccc} \bullet & ! & ? & : \\ \hline ! & \perp & ! & ! \\ ? & ! & ? & ? \\ : & ! & ? & : \end{array}$</td></tr>
<tr><td>Translate</td><td>$\dfrac{p \xrightarrow{\mu!} p'}{\phi p \xrightarrow{\phi^{\uparrow}\mu!} \phi p'}$</td><td colspan="2">$\dfrac{p \xrightarrow{\phi_{\downarrow}\mu\flat} p'}{\phi p \xrightarrow{\mu\flat} \phi p'}$</td></tr>
</table>

[1] There is a symmetric rule for q.

Table 1: The operational semantics of CBS

2 CBS summary

In this section we present a variant of CBS with channels but without value passing [7] and with the scoping and renaming operators replaced by the new translation operator of [8]. Details on different versions of CBS may be found in [7], [8] and [9].

We define the following sets ranged over by the variables indicated in parenthesis.

A set of names: $\Delta = \{a, b, c, \ldots\}$. A set of messages: $\Delta^{\tau} = \Delta \cup \{\tau\} \quad (\mu, \nu)$.

Let $\sharp$ range over $\{!, ?, :\}$, $\natural$ over $\{!, ?\}$, $\flat$ over $\{?, :\}$, and for $S \subseteq \Delta$ let $S\sharp = \{a\sharp | a \in S\}$.

Let $A = \Delta^{\tau}! \cup \Delta? \cup \Delta^{\tau}: \quad (a, b, c, \ldots)$.

Let ϕ be a pair of functions $\phi^{\uparrow}, \phi_{\downarrow} : \Delta^{\tau} \to \Delta^{\tau}$ satisfying $\phi^{\uparrow}(\tau) = \phi_{\downarrow}(\tau) = \tau$. Note that $\phi^{\uparrow}\mu\sharp$ and $\phi_{\downarrow}\mu\sharp$ means unambiguously $(\phi^{\uparrow}\mu)\sharp$ and $(\phi_{\downarrow}\mu)\sharp$. Assume a set of process variables $\mathcal{V}_{\mathrm{CBS}} = \{x, y, \ldots\}$. The syntax for CBS process expressions is defined by the grammar:

$$e ::= nil \mid x \mid \mu!e \mid a?e \mid e + e \mid rec\,x.e \mid e|e \mid \phi e$$

The operational semantics of the language is defined in table 2. Strong bisimulation is defined in the standard way.

3 CCS with prisum summary

The reader is assumed to be familiar with CCS so we limit ourselves to a brief description of the priority choice operator and the operational semantics for CCS with prisum. Details on adding priority operators to CCS may be found in [2], [3] and [4].

The priority choice constructor $+\rangle$ named *prisum* gives the left agent preference to the right but otherwise behaves like a sum. Intuitively we have that

$$A{+}\rangle B$$

means

Behave as A if you can; if no transitions for A are possible, then behave as B

Now let $CCS^{>}$ denote the new language with priority choice that has CCS as a subset. Let Act be a set of action labels; Act is divided into the set of input labels In, the dual set of output labels Out and the internal action τ. Note that output (input) labels need not necessarily be physical output (input) labels; the division is needed for technical reasons. In what follows, let $\bar{\ }$ be an injective function from Act to Act taking τ to τ and an input (output) label α to its dual output (input) label $\overline{\alpha}$. In general α, β, γ etc. will range over Act and a, b, c etc. will range over In. $CCS^{>}$ expressions then have the following syntax:

$$E \quad ::= \quad X \mid nil \mid \alpha.E \mid E + E \mid E{+}\rangle E \mid (E|E) \mid fix(X = E) \mid E \setminus L \mid E[f]$$

where the symbols from CCS have their usual meaning. Only *guarded* recursions are allowed. For technical reasons we demand that there never be any output labels as possible actions of terms in a priority choice construct; i.e. that output labels are not allowed in priority choice constructs except when guarded by an input prefix.

Before turning to the definition of the transition rules we need some considerations about environments as well as some technical preliminaries. Since all actions in CCS are basically performed in handshakes, one expects the following:

> If the environment of $A{+}\rangle B$ refuses to match any of A's transitions,
> then B doing a transition leading to B' can be allowed.

A refusal set is defined to be a set of input actions, for which it is known that the environment can't perform any of the matching output actions. Since no output labels are allowed as possible actions of a priority construct, the information in a refusal set is all that we need to know about the environment of a term.

Given the informal definition of refusal sets above, it is immediately observable that a transition would be possible under a very large (actually infinite) number of different refusal sets. Consider the term $a.nil{+}\rangle b.nil$. The b may occur if the environment refuses $\{a\}$. It may also occur if the environment refuses $\{a, c\}$, etc. Clearly some kind of simplification is needed. The solution is to use transitions of the form

$$\vdash_R A \xrightarrow{\alpha} A'$$

understood as meaning

> *In an environment which refuses* at least R *(and possibly* more *actions),*
> *and is prepared to match* α*,*
> *the agent* A *can perform* α *to become* A'*.*

Thus, the subscripted refusal set should be the *weakest condition* for a given derivation. In order to allow the b in $a.nil {+\rangle} b.nil$ to occur *all* we need to know is that the environment refuses at least $\{a\}$.

In order to define the transition relation, we need the two auxiliary functions *inActs* and *outActs* respectively giving the set of possible input actions and the set of possible output actions of a term. Note that since recursion is guarded these functions are easily defined by structural induction on the term. We then have the following transition rules of which some have symmetric counterparts:

$$\vdash_\emptyset \alpha.A \xrightarrow{\alpha} A \tag{1}$$

$$\frac{\vdash_R A \xrightarrow{\alpha} A'}{\vdash_R A + B \xrightarrow{\alpha} A'} \tag{2}$$

$$\frac{\vdash_R G_1 \xrightarrow{a} G_1'}{\vdash_R G_1 {+\rangle} G_2 \xrightarrow{a} G_1'} \tag{3}$$

$$\frac{\vdash_R G_2 \xrightarrow{\alpha} G_2',\ \tau \notin inActs(G_1),\ \alpha \notin inActs(G_1)}{\vdash_{R \cup inActs(G_1)} G_1 {+\rangle} G_2 \xrightarrow{\alpha} G_2'} \tag{4}$$

$$\frac{\vdash_R A_0 \xrightarrow{\alpha} A_0',\ R \cap outActs(A_1) = \emptyset}{\vdash_R A_0|A_1 \xrightarrow{\alpha} A_0'|A_1} \tag{5}$$

$$\frac{\vdash_{R_0} A_0 \xrightarrow{\alpha} A_0',\ \vdash_{R_1} A_1 \xrightarrow{\overline{\alpha}} A_1',\ R_1 \cap outActs(A_0) = \emptyset,\ R_0 \cap outActs(A_1) = \emptyset}{\vdash_{R_0 \cup R_1} A_0|A_1 \xrightarrow{\tau} A_0'|A_1'} \tag{6}$$

$$\frac{\vdash_R A\{fix(X = A)/X\} \xrightarrow{\alpha} A'}{\vdash_R fix(X = A) \xrightarrow{\alpha} A'} \tag{7}$$

$$\frac{\vdash_R A \xrightarrow{\alpha} A' \ \ (\alpha \notin L \cup \overline{L})}{\vdash_{R \setminus L} A \setminus L \xrightarrow{\alpha} A' \setminus L} \tag{8}$$

$$\frac{\vdash_R A \xrightarrow{\alpha} A'}{\vdash_{R[f]} A[f] \xrightarrow{f(\alpha)} A'[f]} \tag{9}$$

Below is a short intuitive explanation of the most important rules:

- (3) The highest prioritized agent may always perform its transitions.

- (4) Since *inActs* gives *every* possible action for a guard, the refusal set for G_2 transitions should exactly be extended with $inActs(G_1)$. Since τ is not allowed in refusal sets, $\tau \notin inActs(G_1)$ is a necessary condition. A τ will preempt any lower priority transitions from occurring.
  Furthermore it must be the case that the G_2 action is not itself in the new refusal set. We want that $\vdash_R P \xrightarrow{\alpha} P'$ implies $\alpha \notin R$; an environment cannot both refuse α and be prepared to synchronize on it.

- (5) *outActs* gives exactly *every* possible output action for an agent, and one of the agents in a parallel composition cannot do a transition if the other one violates the refusal set condition. The two sub-agents are part of each other's environments.

- (6) The refusal set for the synchronization is the union of the individual refusal sets; otherwise one of the transitions involved in the synchronization would be impossible.

For transitions with empty refusal sets we will omit the refusal set and merely write $P \xrightarrow{\alpha} P'$. A term is *priority closed* if all transitions in the corresponding transition graph have empty refusal sets. For relabeling of terms that are not priority closed, the relabeling function in general needs to be injective, or else counterintuitive transitions may occur. The definitions of strong and weak bisimulation for $CCS^{>}$ are slightly nonstandard, but on the set of priority closed terms they coincide with standard definitions. Since below we only consider equivalence between priority closed terms, we omit further details and refer the reader to the references.

4 Translating CBS to CCS with Priority Choice

In this section we present a translation taking any CBS term not containing the translation operator to a term in $CCS^{>}$. We have deferred considerations concerning the translation operator to Section 7. Due to the fact that only guarded recursions are allowed in $CCS^{>}$ we do not consider CBS terms that contain unguarded recursions.

4.1 Motivation and preliminaries

As already mentioned, CBS is a synchronous calculus with multiway synchronizations, and CCS with prisum is not. Consequently a single CBS transition necessarily must be translated to a *sequence* of transitions $CCS^{>}$ transitions. Such a sequence once initiated must be completely finished before another sequence is initiated.

One obvious option is to expand all parallels; then we only have a single parallel component and no synchronization problems. On the other hand, such a solution is not very good. In general we would like the translation into $CCS^{>}$ to be compositional. Hence we need a control process in parallel with the rest of the translation system, registering each process synchronizing and realizing when all processes in the system have participated. Furthermore, each parallel component of the system, after registrering with the controller must be prevented from continuing execution until all synchronizations have been made. Otherwise for instance $a?b?nil$ may do $a?$ followed by $b?$ before synchronization on $a?$ is really finished; the interleaving and asynchronous nature of the parallel composition in $CCS^{>}$ gets in the way. Then again, resumption of the execution of each parallel component must be controlled.

Each action of a CBS term, $\tau!$ or otherwise, should correspond to an *observable* action in the translation of that term. Otherwise we cannot distinguish actual CBS occurrences in the translation from the internal synchronizations needed for control. Consequently we want the set of CBS actions A to be a subset of the set of $CCS^{>}$ actions Act.

Only the controller should perform observable actions, since only the controller knows when a synchronization is complete. All control synchronizations should be internalized. Consequently let i be a function from A to Act such that $i(A) \cap A = \emptyset$.

In view of the above, a natural choice for the interpretation of a prefix construction is something like the following:[1]

$$a\natural.P \leadsto a\natural.end.P$$

where $end \in Act$ is a distinguished label indicating that this particular subprocess has already participated in the current multiway synchronization, or rather the corresponding sequence of $CCS^{>}$ transitions. As a motivating example consider:

$$\begin{aligned} a?a?nil &\leadsto a?.end.a?.end.nil \\ a?nil|a?nil &\leadsto a?.end.nil|a?.end.nil \end{aligned}$$

Without the *end* label, there would be no way to distinguish the translations of $a?a?nil$ and $a?nil|a?nil$ respectively, although the two CBS terms have different behaviour. The only other alternative is to expand the parallel before translating, which as already mentioned is highly undesirable.

As a motivating example for the conclusion that we need a separate process controlling the synchronization sequences consider the following:

$$\begin{aligned} a!P|a?Q &\leadsto a!.end.P|a?.end.Q \\ a!P|a?Q|a?nil &\leadsto a!.end.P|a?.end.Q|a?.end.nil \end{aligned}$$

Clearly without detailed analysis of parallel subprocesses we do not in advance know how many components that participate in a particular CBS transition. Since we want our translation to be as compositional and structural as possible, we need a separate process to register *when no more components have an available* $a?$. The connection to refusal sets should be obvious, and in fact the priority choice operator is excellent for this kind of control. Clearly nondeterministic choice is not sufficient since we either run the risk of deadlock or run the risk of premature termination of a sequence.

[1] In the remaining part of this subsection, we have omitted explicit use of the internalization function i.

As a final motivating example consider a situation where one of the parallel components discards a broadcast:

$$a!P|a?Q|b?R \;\rightsquigarrow\; a!.end.P|a?.end.Q|b?.end.R$$

In the synchronous CBS semantics an explicit notion of a discard transition is needed in order to avoid negative premises. But when translating to $CCS^{>}$ as mentioned above we may simply use the priority choice operator to detect the absence of an available $a?$. Hence we need not concern ourselves with discards at all if we use the observation from [7] that the presence of discard transitions does not change the semantics of the language. To be precise, the strong bisimulation equivalence is the same whether we consider discard transitions in its definition or not. Not needing explicit treatment of discards greatly simplifies the translation.

4.2 The translation

We define the translation $T : CBS \rightarrow CCS^{>}$ by:

$$T(P) \;\equiv\; (CON \mid T'(P)) \setminus (i(A) \cup \{end\})$$

CON is the process that we need for controlling the synchronization sequences. It is defined as follows:

$$CON \;\equiv\; fix(X = \sum_{a \in A, \natural \in \{!,?\}} i(a\natural).fix(Y = i(a?).Y +\rangle fix(Z = end.Z +\rangle a\natural.X)))$$

When trying to understand this definition it might be a good idea to reconsider the motivating examples from Section 4.1. A brief explanation of the definition of CON is the following. One of the CBS subprocesses initiates a sequence by synchronizing with CON on $i(a\natural)$. CON proceeds to synchronize with *all* subprocesses having an available $i(a?)$. Note that a subprocess that has already participated in the current sequence has synchronization on *end* as the only possible action. Only when no more subprocesses have available $i(a?)$ actions, corresponding to a situation where all CBS components have participated in the current transition, may the low priority term in the definition of the Y fixpoint proceed. Subsequently the control process "collects" all *end*'s, returning the rest of the system to a state that is the translation of a CBS term. Finally the control process externally signals $a\natural$ in agrement with the fact that now a sequence corresponding to a CBS transition with this label is completed. As already mentioned, for obvious reasons only the control process can know when such a sequence is finished.

Note that no $T'(P)$ can synchronize on $\tau?$, hence we do not need to explicitly exclude such (illegal) transitions. Furthermore note that it is *possible* to begin a synchronization sequence with any enabled internal $a!$ action. Hence, since we are only interested in bisimulation equivalences, we need not concern ourselves with internal output's occurring in the middle of a synchronization sequence. Such occurrences would have forced us to change the final external result of the sequence, and would create unnecessary complications. As mentioned above, only input labels are allowed as possible actions of priority constructs, and we have obeyed this restriction in the construction of CON.

At first glance we need an infinite sum for the definition of CON, which would technically take us outside the language $CCS^{>}$. But since we do not need to consider discards, actually any CBS term has finite sort and consequently a finite sum suffices. Except for perhaps using the sort as described above, the definition of CON is independent of the term that is being translated. Indeed the function $T' : CBS \to CCS^{>}$ represents the actual translation of terms. Below this function is defined inductively in a compositional manner. Having already defined the control system CON, the definition of T' is easy.

Nil

The CBS term nil has no transitions other than discards. Since we ignore discards, we may model nil by nil:

$$T'(nil) \equiv nil$$

Variable

We assume that the sets of variables in the two calculi are essentially the same (i.e. in bijection) and model the CBS variable x by the $CCS^{>}$ variable X:

$$T'(x) \equiv X$$

Receive

We have no discards to consider. The term $a?Q$ is willing to receive on channel a. In the translation this is signalled to the controller by sending an $\overline{i(a?)}$, whereupon the process waits until told by the controller that the current multiway synchronization is ended:

$$T'(a?Q) \equiv \overline{i(a?)}.\overline{end}.T'(Q)$$

Note that since we are forced to use only input labels in the construction of CON, we have to use only output labels in the translation of a receive. Naturally, due to the restriction on internal labels, this has no computational significance.

Transmit

Since we have no discards to consider, the translation of a transmit is analogous to the translation of a receive:

$$T'(a!Q) \equiv \overline{i(a!)}.\overline{end}.T'(Q)$$

Choice

Since we have no discards to consider, we need not be worried about prematurely resolving the choice. We can simply take:

$$T'(P+Q) \equiv T'(P) + T'(Q)$$

Only when the controller is ready to receive one of the possible actions of either P or Q can the sum participate in a $CCS^{>}$ transition. But this corresponds exactly to the process participating in a CBS synchronizations with an action that is not discard. We do not need additional operators such as the "green restriction operator" used in [1].

Parallel

Due to the fact that CON directly controls all multiway synchronizations, parallel may simply be translated directly to parallel:

$$T'(P|Q) \equiv T'(P)|T'(Q)$$

Recursion

Again, since we do not consider discards, we need not be concerned about unfolding the recursion prematurely (or even infinitely many times). We can simply take:

$$T'(rec\, x.P) \equiv fix(X = T'(P))$$

Translation summary

The translation $T : CBS \to CCS^{>}$ is defined by

$$T(P) \;\equiv\; (CON \mid T'(P)) \setminus (i(A) \cup \{end\})$$

where CON is defined by

$$CON \;\equiv\; fix(X = \sum_{a,\natural} a\natural.fix(Y = a?.Y{+}\rangle\tau.(fix(Z = end.Z{+}\rangle\tau.a\natural.X)))$$

and where the function $T' : CBS \to CCS^{>}$ is defined inductively by

$$\begin{array}{rcl}
T'(nil) & \equiv & nil \\
T'(x) & \equiv & X \\
T'(a?Q) & \equiv & \overline{i(a?)}.\overline{end}.T'(Q) \\
T'(a!Q) & \equiv & \overline{i(a!)}.\overline{end}.T'(Q) \\
T'(P+Q) & \equiv & T'(P) + T'(Q) \\
T'(P|Q) & \equiv & T'(P)|T'(Q) \\
T'(recx.P) & \equiv & fix(X = T'(P))
\end{array}$$

In Section 7 this translation is extended to cover relabeling, restriction and/or translation.

5 Examples

In this section, we give a few examples of translations of CBS terms, and briefly consider the relation between transitions of CBS terms and sequences of transitions of their translations.

Consider the term $a!nil|a?nil|b?nil$ which has the transition

$$a!nil|a?nil|b?nil \xrightarrow{a!} nil|nil|b?nil$$

The translation to $CCS^{>}$ gives us the term

$$T(a!nil|a?nil|b?nil) \equiv (CON|\overline{i(a!)}.\overline{end}.nil|\overline{i(a?)}.\overline{end}.nil|\overline{i(b?)}.\overline{end}.nil) \setminus L$$

which has the following sequence of transitions:

$$\begin{array}{rl}
(T(a!nil|a?nil|b?nil)) \setminus L & \xrightarrow{\tau} \\
(\ fix(Y = i(a?).Y+\rangle fix(Z = end.Z+\rangle a!.CON)))| & \\
\overline{end}.nil|\overline{i(a?)}.\overline{end}.nil|\overline{i(b?)}.\overline{end}.nil\ \ ) \setminus L & \xrightarrow{\tau} \\
(\ fix(Y = i(a?).Y+\rangle fix(Z = end.Z+\rangle a!.CON)))| & \\
\overline{end}.nil|\overline{end}.nil|\overline{i(b?)}.\overline{end}.nil\ \ ) \setminus L & \xrightarrow{\tau} \\
(fix(Z = end.Z+\rangle a!.CON)|\overline{end}.nil|\overline{end}.nil|\overline{i(b?)}.\overline{end}.nil) \setminus L & \xrightarrow{\tau} \\
(fix(Z = end.Z+\rangle a!.CON)|nil|\overline{end}.nil|\overline{i(b?)}.\overline{end}.nil) \setminus L & \xrightarrow{\tau} \\
(fix(Z = end.Z+\rangle a!.CON)|nil|nil|\overline{i(b?)}.\overline{end}.nil) \setminus L & \xrightarrow{\tau} \\
(a!.CON|nil|nil|\overline{i(b?)}.\overline{end}.nil) \setminus L & \xrightarrow{a!} \\
(CON|nil|nil|\overline{i(b?)}.\overline{end}.nil) \setminus L &
\end{array}$$

ending in the state

$$(CON|nil|nil|\overline{i(b?)}.\overline{end}.nil) \setminus L \ \equiv\ T(nil|nil|b?nil)$$

Notice how in this example, we *choose* to let the output process be the first to synchronize with CON, giving us a CBS output as result of the synchronization sequence. The interleaving nature of the CCS parallel saves us some considerations concerning (internalized) CBS outputs occurring in the middle of a sequence. Furthermore notice how the completion of the synchronization sequence is forced by the structure of CON. Even though in general any interleaving of the set of inputs from different parallel components taking part in a CBS transition is possible, each such component is forced to take part in the synchronization sequence exactly once with an internalized input action and exactly once with an *end* action.

As an example of the way CON controls choices, consider the term

$$a!nil|(a?nil + b?nil)$$

which has the transition

$$a!nil|(a?nil + b?nil) \xrightarrow{a!} nil|nil$$

The translation to $CCS^{>}$ gives us the term

$$T(a!nil|(a?nil+b?nil)) \equiv (CON|\overline{i(a!)}.\overline{end}.nil|(\overline{i(a?)}.\overline{end}.nil+\overline{i(b?)}.\overline{end}.nil))\setminus L$$

which has the following sequence of transitions:

$$\begin{array}{rl}
T(a!nil|(a?nil+b?nil)) & \xrightarrow{\tau} \\
(\;\; fix(Y = i(a?).Y+\rangle fix(Z = end.Z+\rangle a!.CON)))| & \\
\overline{end}.nil|(\overline{i(a?)}.\overline{end}.nil + \overline{i(b?)}.\overline{end}.nil) \;\;)\setminus L & \xrightarrow{\tau} \\
(fix(Y = i(a?).Y+\rangle fix(Z = end.Z+\rangle a!.CON)))|\overline{end}.nil|\overline{end}.nil)\setminus L & \xrightarrow{\tau} \\
(fix(Z = end.Z+\rangle a!.CON)|\overline{end}.nil|\overline{end}.nil)\setminus L & \xrightarrow{\tau} \\
(fix(Z = end.Z+\rangle a!.CON)|nil|\overline{end}.nil)\setminus L & \xrightarrow{\tau} \\
(fix(Z = end.Z+\rangle a!.CON)|nil|nil)\setminus L & \xrightarrow{\tau} \\
(a!.CON|nil|nil)\setminus L & \xrightarrow{a!} \\
(CON|nil|nil)\setminus L &
\end{array}$$

ending in the state

$$(CON|nil|nil)\setminus L \equiv T(nil|nil)$$

Notice how in this example, the fact that we do not explicitly consider discard actions saves us the trouble of preventing the choice from being prematurely resolved by a discard.

6 Results

In this section only CBS terms that do not contain the translation operator nor an unguarded recursion are considered. Note that the translation of any CBS term is a priority closed $CCS^{>}$ term.

As the main result we prove that two CBS terms are strongly bisimilar iff the $CCS^{>}$ terms resulting from the translation are weakly bisimilar. This is a nice and very simple relationship between terms and their translations, and indeed it is the best that we can hope for. As proof of this just consider

$$(rec\, x.a?x) \;\sim\; (rec\, x.a?x)|(rec\, x.a?x)$$

For the translation we have the following general results:

Proposition 1 *If* $P \xrightarrow{a\natural} P'$ *then* $T(P) \overset{\tau}{\Rightarrow}\xrightarrow{a\natural} T(P')$

Sketch of proof: Clearly due to the construction of CON it is sufficient to prove Lemma 2 below. $\square$

Lemma 2 *If* $P \xrightarrow{a\natural} P'$ *then* $T'(P) \xrightarrow{i(a\natural)} (\xrightarrow{i(a?)})^{*}U(\xrightarrow{end})^{*}T'(P')$ *where* $U \not\xrightarrow{i(a?)}$

Sketch of proof: The proof is done by structural induction on P. Note that every recursion is guarded. The only slightly complicated case is the case for parallel which is treated by dividing into subcases according to the combination of CBS actions for the two parallel components. If an output is one of the components, then internal output synchronization is always chosen as the first transition in the sequence. $\square$

With a proof similar to the proof of Proposition 1, we get:

Proposition 3 *If* $T(P) \stackrel{\tau}{\Rightarrow}\stackrel{a\natural}{\rightarrow} Q$ *then* $\exists P'. Q \equiv T(P') \ \wedge \ P \stackrel{a\natural}{\rightarrow} P'$

We now have the following Corollary:

Corollary 4 $P \stackrel{a\natural}{\rightarrow} P'$ *iff* $T(P) \stackrel{\tau}{\Rightarrow}\stackrel{a\natural}{\rightarrow} T(P')$

Proceeding with the proof of the main result of this Section we have:

Proposition 5 *If* $T(P) \approx T(Q)$ *and* $T(P) \stackrel{\tau}{\Rightarrow}\stackrel{a\natural}{\rightarrow} T(P')$ *then* $\exists Q'. T(Q) \stackrel{\tau}{\Rightarrow}\stackrel{a\natural}{\rightarrow} T(Q') \approx T(P')$

Sketch of proof: The proof uses Lemma 6 below which is a consequence of the construction of CON. It goes as follows. If

$$T(P) \stackrel{\tau}{\Rightarrow}\stackrel{a\natural}{\rightarrow} T(P')$$

then by Lemma 6 we have

$$\exists U. T(P) \stackrel{\tau}{\Rightarrow} U \stackrel{a\natural}{\rightarrow} T(P') \ \wedge \ U \sim a\natural.T(P')$$

By using $T(P) \approx T(Q)$ we get

$$\exists U'. T(P) \stackrel{\tau}{\Rightarrow} U' \approx a\natural.T(P') \approx U$$

Clearly we must have

$$\exists U''. U' \stackrel{\tau}{\Rightarrow} U'' \stackrel{a\natural}{\rightarrow} U'''$$

Where $U'' \approx a\natural.T(P')$. Note that we cannot directly conclude that U''' is weakly bisimilar to $T(P')$. According to Proposition 3 we now have

$$\exists Q'. U''' \equiv T(Q')$$

By Lemma 6 we get $a\natural.T(Q') \approx U'' \approx a\natural.T(P')$ from which we deduce that $T(Q') \approx T(P')$ as desired. $\square$

Lemma 6 *If* $T(P) \stackrel{\tau}{\Rightarrow} U \stackrel{a\natural}{\rightarrow} T(P')$ *then* $U \sim a\natural.T(P')$

We can now prove the following Theorem:

Theorem 7 *The set* $\{\langle P, Q\rangle \mid T(P) \approx T(Q)\}$ *is a* CBS *strong bisimulation.*

Sketch of proof: If $P \xrightarrow{a\natural} P'$ then according to Proposition 1 we get

$$T(P) \overset{\tau}{\Rightarrow}\xrightarrow{a\natural} T(P')$$

From Proposition 5 we get

$$\exists Q'.\, T(Q) \overset{\tau}{\Rightarrow}\xrightarrow{a\natural} T(Q') \approx T(P')$$

But then according to Proposition 3 we get

$$Q \xrightarrow{a\natural} Q'$$

and we are done. □

As a corollary we get:

Corollary 8 *If* $T(P) \approx T(Q)$ *then* $P \sim Q$

The proof of the oppositite implication, if $P \sim Q$ then $T(P) \approx T(Q)$, is simpler. We clearly have the following two results:

Proposition 9 *For any* P*, if* $T(P)$ *has any transitions then* τ *is the only possible action of* $T(P)$*.*

Proposition 10 *For any* P*,* $T(P)$ *has no infinite sequence of* τ *transitions.*

Now with a proof similar to the proof of Proposition 1, using Proposition 10, we get:

Proposition 11 *If* $T(P) \xrightarrow{\tau} Q$ *then there exists* $a, \natural, U, P'$ *such that*

$$T(P) \overset{\tau}{\Rightarrow} U \xrightarrow{a\natural} T(P')$$

and whenever

$$Q \overset{\tau}{\Rightarrow} U' \xrightarrow{\alpha} U''$$

then $U' \equiv U$*,* $U'' \equiv T(P')$ *and* $\alpha = a\natural$

The above Proposition tells us that the first transition of $T(P)$ chooses which CBS synchronization we are simulating, and using this result we get:

Theorem 12 *If* $P \sim Q$ *then* $T(P) \approx T(Q)$

Sketch of proof: The proof can be done either by structural induction or by exhibiting suitable weak bisimulations. In both cases the reasoning below implies the desired result.

Assume that $T(P)$ has some transition. According to Proposition 9 this must be a τ transition. Using Proposition 11 we get:

$$T(P) \overset{U_0}{\rightarrow}\overset{\tau}{\Rightarrow} U \overset{a\natural}{\rightarrow} T(P')$$

and whenever

$$U_0 \overset{\tau}{\Rightarrow} U' \overset{\alpha}{\rightarrow} U''$$

then $U' \equiv U$, $U'' \equiv T(P')$ and $\alpha = a\natural$.
We now have that $P \overset{a\natural}{\rightarrow} P'$ and since $P \sim Q$ we know that there exists Q' such that $Q \overset{a\natural}{\rightarrow} Q' \sim P'$. But then there exists S such that

$$T(Q) \overset{\tau}{\Rightarrow} S \overset{a\natural}{\rightarrow} T(Q')$$

We know that $U \sim a\natural.T(P')$ and that $S \sim a\natural.T(Q')$. Clearly we can choose $T(Q) \overset{\tau}{\Rightarrow} S$ as the match of $T(P) \overset{\tau}{\rightarrow} Q$ in the deduction of weak bisimilarity. $\square$

We have now finished the proof of the main result of this Section:

7 Renaming, restriction and hiding

Originally, CBS had a renaming operator and a restriction/hiding operator (see [7]), but later developments (see eg. [8]) introduce a more general translation operator instead.

We will begin by considering the original renaming and restriction/hiding operators. Let $f : \Delta^{\tau} \rightarrow \Delta^{\tau}$ be a relabeling function and let $L \subset \Delta$. We then have the following additional transition rules:

$$\frac{P \overset{\mu\sharp}{\rightarrow} P'}{P[f] \overset{f(\mu)\sharp}{\rightarrow} P'[f]}$$

$$\frac{P \overset{\mu\sharp}{\rightarrow} P',\ \mu \notin L}{P \setminus L \overset{\mu\sharp}{\rightarrow} P' \setminus L} \quad \frac{P \overset{\mu!}{\rightarrow} P',\ \mu \in L}{P \setminus L \overset{\tau!}{\rightarrow} P' \setminus L} \quad \frac{\mu \in L}{P \setminus L \overset{\mu:}{\rightarrow} P \setminus L}$$

Using the relabeling operator from $CCS^{>}$, the renaming operator from CBS may clearly be translated by extending the definition of T' as follows:

$$T'(P[f]) = T'(P)[f']$$

where f' is f extended to *Act* in the obvious way taking for instance $f'(i(\overline{a\natural}) = i(\overline{f(a)\natural})$.
Note that the relabeling function f' may not be injective, but since $T'(P)$ is priority closed for any P this presents no problems.

Using this extension of the translation from CBS to $CCS^{>}$, the main result of Section 6 clearly still holds. We omit further details.

Restriction is more complicated. Consider the CBS term

$$(a!P|a?Q) \setminus \{a\}$$

which has the transition

$$(a!P|a?Q) \setminus \{a\} \xrightarrow{\tau!} (P|Q) \setminus \{a\}$$

Our problem is that more than one process participates in this internal transition. Hence clearly it is not sufficient just to relabel $a!$ to $\tau!$ in the translation; we need to take care of the $a?$ as well or that action never occurs. Relabeling $a?$ to $\tau?$ is not an option either; just to mention one of the problems there would be no way to distinguish a restricted a input from say a restricted b input.

What we are more or less forced to do is to introduce a local analogue of the control process in order to handle the local multiway synchronization resulting in a $\tau!$ transition. Unfortunately this also forces us to slightly change the definition of CON itself because we need some way of beginning and ending local hidden synchronization sequences. Consider the following version of the control process:

$$CON' \;\equiv\; fix(X = H + \sum i(a\natural).fix(Y = i(a?).Y +\rangle fix(Z = end.Z +\rangle a\natural.X)))$$

where the addition of H inside the first fixpoint is the only change compared to CON and where H is defined as follows:[2]

$$H \;\equiv\; hi_{begin}.(\overline{hi}.\tau!.X + \overline{nohi}.X)$$

Naturally the outermost restriction in the definition of $T(P)$ should now contain hi_{begin}, hi and $nohi$ as well as all internalized labels and end.

The intuition behind the definition of H is that hi_{begin} signals the beginning of (an attempt at) a local hidden synchronization sequence. $nohi$ then indicates that the attempt failed; no local hidden synchronization was performed. hi indicates that a local hidden synchronization has successfully been completed and consequently an external $\tau!$ is sent.

Given the above, we can now define the translation of $P \setminus L$ as follows:

$$T'(P \setminus L) = (LOCCON(L)|T'(P)) \setminus (\{i(a?)|a \in L\} \cup \{i(a!)|a \in L\})$$

where $LOCCON(L)$ is the local analogue of the control process that is used for hidden internal synchronizations on labels in the set L:

$$LOCCON(L)$$
$$\equiv$$
$$fix(X = \overline{hi_{begin}}.((\sum_{a \in L} i(a!).fix(Y = i(a?).Y > fix(Z = end.Z > hi.X))) > nohi.X)$$

[2] "hi" is short for "hide"

Note that inputs are restricted whereas outputs are hidden. Hence $LOCCON(L)$ only needs to consider internal synchronizations that are initiated by an output. Since the restriction operator is static, we need the outermost fixpoint in the definition of $LOCCON(L)$. Clearly after sending a hi_{begin}, $LOCCON(L)$ attempts an internal synchronization on one of the labels in L. If no internal synchronization on a label in L is possible, then $LOCCON(L)$ signals a *nohi*.

Using this extended and slightly changed version of the translation from CBS to $CCS^{>}$, the main result of Section 6 still holds. The proofs need modification due to the new presence of τ cycles in translated terms, but the necessary changes are straighforward. We omit further details.

For the more general CBS translation operator, the relabeling operator of $CCS^{>}$ is inadequate. The way the CBS translation operator treats inputs is really relabeling from the outside and in (contrary to the usual inside-out style). Hence in general using the CBS translation operator corresponds to using a relabeling *relation* and not just a function. To get a simple solution to this problem, we may allow the use of relabeling relations in $CCS^{>}$. Then, using the same techniques as above, we are able to translate any CBS term with the main result of Section 6 still holding.

8 Conclusion

We have presented a translation from the synchronous broadcasting calculus CBS to the asynchronous handshaking calculus CCS with priority choice which "as much as possible" preserves the semantics of terms and which is of a compositional nature.[3]

For the fragment of CBS that does not contain the restriction/hiding operator, the translation is very simple. Even the somewhat lenghty definition of CON should be intuitively clear. The use of priority choice allows us to completely ignore discards in the translation. This avoids many of the problems encountered in [1] and is clearly an advantage.

It is an interesting fact, that the priority choice operator provides the additional control needed to simulate multiway synchronizations. With only nondeterministic choice at hand, such control is not possible; either deadlocks become possible, or we cannot ensure correct termination of sequences. In [4] it is proved that although any priority closed term of $CCS^{>}$ may be translated to a strongly bisimilar term in CCS, this can only be done at the price of expanding all parallels. This strongly supports the claim that no compositional translation from CBS to pure CCS exists. Moreover by noting that the situation does not improve even if it is enough to translate strong bisimilarity to weak bisimilarity, we may in fact prove such a claim.

We see no further problems in generalizations to cope with value passing

[3] Strictly speaking, the presence of the control process CON makes the translation noncompositional, but CON is a fixed expression that is completely independent of the term being translated. It may be viewed as an environment in which the real translation takes place.

and perhaps a prioritized version of CBS. In the future, investigation of the relationship between synchronous and asynchronous calculi may show that priorities indeed provide the missing link between these two different views on concurrency. An important point is that both discards and refusal sets are ways of encoding negative premises.

Currently we are working on using broadcasts to simulate priorities. This is a complex problem, which is solvable only because broadcast may be used for "testing" the capabilities of a terms environment.

References

[1] Uno Holmer. Interpreting Broadcast Communication in SCCS. In *LNCS 715*. Springer Verlag, 1993.

[2] Claus Torp Jensen. CCS with Priorities. Master's thesis, Department of Computer Science, Aarhus University, 1992.

[3] Claus Torp Jensen. The Concurrency Workbench with Priorities. In *LNCS 575*. Springer Verlag, 1992.

[4] Claus Torp Jensen. *Prioritized and Independent Actions in Distributed Computer Systems*. PhD thesis, Department of Computer Science, Aarhus University, 1994.

[5] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[6] G. D. Plotkin. A Structural Approach to Operational Semantics. DAIMI FN 19, Computer Science Department, Aarhus University, 1981.

[7] K. V. S. Prasad. A Calculus of Broadcasting Systems. In *LNCS 493*. Springer Verlag, 1991.

[8] K. V. S. Prasad. A Calculus of Value Broadcasts. In *LNCS 694*. Springer Verlag, 1993.

[9] K. V. S. Prasad. Broadcasting with Priority. Programming Methodology Group Report 75, Chalmers University of Technology, 1993.

Polytypic Programming
— abstract —

Johan Jeuring
Chalmers University of Technology and University of Göteborg
S-412 96 Göteborg, Sweden
email: johanj@cs.chalmers.se

There are many programming problems that can be formulated independent of a specific datatype. For example, the pattern matching problem can be informally specified as follows: given a pattern and a text, find the first occurrence of the pattern in the text. The pattern and the text may be both lists, or they may be both trees, or they may be both multidimensional arrays, etc. Many algorithms solving a programming problem which is originally formulated on the datatype *List* have later been extended to solve 'the same' problem on other datatypes.

I call a function that is defined on arbitrary datatypes a polytypic function, i.e., a polytypic function is a function that is defined on lists, binary trees, and all other 'tree-like' datatypes. A familiar example of a polytypic function is the function `map`, which takes a function `f :: a -> b`, and a value of some datatype `D a` (for example lists over the type `a` or binary trees over the type `a`), and applies function `f` to all elements of type `a` in the value, obtaining a value of type `D b`. Often a polytypic function is also polymorphic, but it need not be. An example of a polytypic function that is not polymorphic is the function that generalises summing a list to arbitrary datatypes.

In this talk I discuss polytypic functions. I combine the polytypic functions that have been defined in the Bird-Meertens calculus, a calculus for transformational programming developed over the last decade, with inductive definitions of natural transformations to build new polytypic functions. Examples of such new polytypic functions are the function `size`, which generalises the function `length` defined on lists to arbitrary datatypes, and the function `pattern_match`, which generalises a version of the pattern-matching algorithm of Knuth, Morris, and Pratt to arbitrary datatypes. Furthermore, I discuss Hollum, an experimental system that supports polytypic programming. Using Hollum it is possible to use functions like `map_Data`, and `pattern_match_Data` for every `data Data` declared in the program. Hollum takes a program written in the functional programming language Gofer, and returns the same program to which instances of polytypic functions are added for each data declared in the original program.

This is joint work with Graham Hutton (Utrecht University) and Oege de Moor (Oxford University).

A Constraint Oriented Proof Methodology based on Modal Transition Systems

Kim G. Larsen*
BRICS†
Aalborg Univ., Denmark,

Bernhard Steffen
FB Math. und Informatik,
Univ. of Passau, Germany,

Carsten Weise‡
LS für Informatik I,
Aachen Univ., Germany

Abstract

In this paper, we present a constraint-oriented state-based proof methodology for concurrent software systems which exploits compositionality and abstraction for the reduction of the verification problem under investigation. Formal basis for this methodology are Modal Transition Systems allowing loose state-based specifications, which can be refined by successively adding constraints. Key concepts of our method are *projective views*, *separation of proof obligations*, *Skolemization* and *abstraction*. The method is even applicable to real time systems.

1 Introduction

The use of formal methods and in particular formal verification of concurrent systems, interactive or fully automatic, is still limited to very specific problem classes. For state-based methods this is mainly due to the state explosion problem: the state graph of a concurrent systems grows exponentially with the number of its parallel components, leading to an unmanageable size for most practically relevant systems. Consequently, several techniques have been developed to tackle

*This author has been partially supported by the European Communities under CONCUR2, BRA 7166.

†Basic Research in Computer Science, Centre of the Danish National Research Foundation.

‡Most of the work of this author was done during a visit to Aalborg University, partially supported by BRICS

this problem. Here we focus on the four main streams and do not discuss the flood of very specific heuristics. Most elegant and ambitious are *compositional* methods (e.g. [ASW94, CLM89, GS90][1]), which due to the nature of parallel compositions are unfortunately rarely applicable. *Partial order* methods try to avoid the state explosion problem by suppressing unnecessary interleavings of actions [GW91, Val93, GP93]. But also these methods, which are extremely successful in special cases, do not work in general. In practice, *Binary Decision Diagram*-based codings of the state graph are successfully applied to an interesting class of systems, see e.g. [Br86, BCMDH90, EFT91]. These codings of the state graph do not explode directly, but they may explode during verification, and it is not yet fully clear when this happens. All these techniques can be accompanied by *abstraction*: depending on the particular property under investigation, systems may be dramatically reduced by suppressing details that are irrelevant for verification, see e.g. [CC77, CGL92, GL93]. Summarizing, all these methods cover very specific cases, and there is no hope for a uniform approach. Thus more application specific approaches are required, extending the practicality of formal methods.

In this paper, we present a constraint-oriented state-based proof methodology for concurrent software systems which exploits compositionality and abstraction for the reduction of the verification problem under investigation. Formal basis for this methodology are Modal Transition Systems (MTS) [LT88] allowing loose state-based specifications, which can be refined by successively adding constraints. In particular, this allows extremely fine-granular specifications, which are characteristic for our approach: each aspect of a system component is specified by a number of independent constraints, one for each parameter configuration. This leads to a usually infinite number of extremely simple constraints which must all be satisfied by a corresponding component implementation. Beside exploiting compositionality in the standard (vertical) fashion, this extreme component decomposition also supports a horizontally compositional approach, which does not only separate proof obligations for subcomponents or subproperties but also for the various parameter instantiations. This is the key for the success of the following three step reduction, which may reduce even a verification problem for infinite state systems to a small number of automatically verifiable problems about finite state systems:

- *Separating the Proof Obligations.* Sections 3 and 4 present a proof principle justifying the separation and specialization of the various proof obligations, which prepare the ground for the subsequent reduction steps.

- *Skolemization.* The separation of the first step leaves us with problems smaller in size but larger in number. Due to the nature of their origin, these

[1]In contrast to the first reference, the subsequent two papers address compositional reduction of systems rather than compositional verification.

problems often fall into a small number of equivalence classes requiring only one prototypical proof each.

- *Abstraction.* After the first two reduction steps there may still be problems with infinite state graphs. However, the extreme specialization of the problem supports the power of abstract interpretation, which finally may reduce all the proof obligations to finite ones.

Our proof methodology is not complete, i.e., there is neither a guarantee for the possibility of a finite state reduction nor a straightforward method for finding the right amount of separation for the success of the succeeding steps or the adequate abstraction for the final verification. Still, as should be clear even from the simple accompanying example in the paper, there is a large class of problems and systems, where the method can be applied quite straightforwardly. Typical examples are systems with limited data dependence, like the one proposed by Leslie Lamport and Manfred Broy for a recent Dagstuhl Seminar as a mean to evaluate methods [BL93]. Of course, the more complex the system structure the more involved will be the required search of appropriate granularity and abstraction.

Whereas complex data dependencies may exclude any possibility of 'horizontal' decomposition, our approach elegantly extends to real time systems, even over a dense time domain. In fact, we will show that this extension does not affect the possibility of a finite state reduction. Even better, the resulting finite state problems can be automatically verified using the EPSILON verification system. All this is illustrated using a simple example of pipelined buffers.

The next section recalls the basic theory of Modal Transition Systems (MTS), which we use for system specification. Section 3 presents our notion of projective views and discusses the first reduction step. The subsequent two sections are devoted to the second and third reduction step, while Section 6 sketches the extension of our method to real time systems over a dense time domain. Finally, Section 7 summarizes our conclusion and directions to future work.

2 Modal Transition Systems

In this section we give a brief introduction to the existing theory of modal transition systems. We assume familiarity with CCS. For more elaborate introductions and proofs we refer the reader to [LT88, HL89, Lar90].

When specifying reactive systems by traditional Process Algebras like e.g. CCS [Mil89], one defines the set of action transitions that can be performed (or observed) in a given system state. In this approach, any valid implementation *must* be able to perform the specified actions, which often constrains the set of possible implementations unnecessarily. One way of improving this situation within the framework of operational specification is to allow specifications where one can explicitly distinguish between transitions that are *admissible* (or allowed) and those that are *required*. This distinction allows a much more flexible specification and a much more generous notion of implementation, and therefore improves the practicality of the operational approach. Technically, this is made precise through the following notion of *modal transition systems*:

Definition 2.1 *A modal transition system is a structure* $\mathcal{S} = (\Sigma, A, \longrightarrow_{\Box}, \longrightarrow_{\Diamond})$, *where* Σ *is a set of states,* A *is a set of actions and* $\longrightarrow_{\Box}, \longrightarrow_{\Diamond} \subseteq \Sigma \times A \times \Sigma$ *are transition relations, satisfying the consistency condition* $\longrightarrow_{\Box} \subseteq \longrightarrow_{\Diamond}$. □

Intuitively, the requirement $\longrightarrow_{\Box} \subseteq \longrightarrow_{\Diamond}$ expresses that anything which is required should also be allowed hence ensuring the consistency of modal specifications. When the relations $\longrightarrow_{\Box}$ and $\longrightarrow_{\Diamond}$ coincide, the above definition reduces to the traditional notion of labelled transition systems.

Syntactically, we represent modal transition systems by means of a slightly extended version of CCS. The only change in the syntax is the introduction of two prefix constructs $a_{\Box}.P$ and $a_{\Diamond}.P$ with the following semantics: $a_{\Diamond}.P \xrightarrow{a}_{\Diamond} P$, $a_{\Box}.P \xrightarrow{a}_{\Box} P$ and $a_{\Box}.P \xrightarrow{a}_{\Diamond} P$. The semantics for the other constructs follow the lines of CCS in the sense that each rule has a version for $\longrightarrow_{\Box}$ and $\longrightarrow_{\Diamond}$ respectively. We will call this version of CCS *modal* CCS.

As usual, we consider a design process as a sequence of *refinement steps* reducing the number of possible implementations. Intuitively, our notion of when a specification S refines another (weaker) specification T is based on the following simple observation. Any behavioural aspect *allowed* by a S should also be allowed by T; and dually, any behavioural aspect which is already guaranteed by the weaker specification T must also be guaranteed by S. Using the derivation relations $\longrightarrow_{\Box}$ and $\longrightarrow_{\Diamond}$ this may be formalized by the following notion of *refinement*:

Definition 2.2 *A refinement* $\mathcal{R}$ *is a binary relation on* Σ *such that whenever* $S\,\mathcal{R}\,T$ *and* $a \in A$ *then the following holds:*

1. *Whenever* $S \xrightarrow{a}_{\Diamond} S'$, *then* $T \xrightarrow{a}_{\Diamond} T'$ *for some* T' *with* $S'\,\mathcal{R}\,T'$,
2. *Whenever* $T \xrightarrow{a}_{\Box} T'$, *then* $S \xrightarrow{a}_{\Box} S'$ *for some* S' *with* $S'\,\mathcal{R}\,T'$.

S is said to be a refinement of T in case (S,T) is contained in some refinement $\mathcal{R}$. *We write* $S \triangleleft T$ *in this case.* □

Note that when we apply the above definition to traditional labelled transition systems (where $\longrightarrow = \longrightarrow_{\Box} = \longrightarrow_{\Diamond}$), we obtain the well–known notion of bisimulation [Par81, Mil89]. Using standard techniques, one straightforwardly establishes that $\triangleleft$ is a preorder preserved by all modal CCS operators.

$\lhd$ allows *loose* specifications. This important property, which can be best explained by looking at the 'weakest' specification $\mathcal{U}$ constantly allowing any action, but never requiring anything to happen. Operationally, $\mathcal{U}$ is completely defined by $\mathcal{U} \xrightarrow{a}_\Diamond \mathcal{U}$ for all actions a. It is easily verified that $S \lhd \mathcal{U}$ for any modal specification S.

Intuitively, S and T are *independent* if they are not contradictory, i.e. any action required by one is not constraint by the other. The following formal definition is due to the fact that for S and T to be *independent* all 'simultaneously'‘reachable processes S' and T' must be indenpendent too:

Definition 2.3 *An independence relation $\mathcal{R}$ is a binary relation on Σ such that whenever $S\,\mathcal{R}\,T$ and $a \in A$ then the following holds:*

1. *Whenever $S \xrightarrow{a}_\Box S'$, there is a unique T' such that $T \xrightarrow{a}_\Diamond T'$ and $S'\,\mathcal{R}\,T'$,*
2. *Whenever $T \xrightarrow{a}_\Box T'$, there is a unique S' such that $S \xrightarrow{a}_\Diamond S'$ and $S'\,\mathcal{R}\,T'$,*
3. *Whenever $S \xrightarrow{a}_\Diamond S'$ and $T \xrightarrow{a}_\Diamond T'$ then $S'\,\mathcal{R}\,T'$.*

S and T are said to be independent *in case (S, T) is contained in some independence relation $\mathcal{R}$.* $\Box$

Note in particular that two specifications are independent if none of them requires any actions. Independence is important, as it allows to define conjunction on modal transition systems by:

$$\frac{S \xrightarrow{a}_\Box S' \quad T \xrightarrow{a}_\Diamond T'}{S \wedge T \xrightarrow{a}_\Box S' \wedge T'} \qquad \frac{S \xrightarrow{a}_\Diamond S' \quad T \xrightarrow{a}_\Box T'}{S \wedge T \xrightarrow{a}_\Box S' \wedge T'}$$

$$\frac{S \xrightarrow{a}_\Diamond S' \quad T \xrightarrow{a}_\Diamond T'}{S \wedge T \xrightarrow{a}_\Diamond S' \wedge T'}$$

Of course, $S \wedge T$ is always a well-defined modal specifications (i.e. any required transition is also allowed), and in fact, for independent arguments S and T it defines their *logical* conjunction:

Theorem 2.4 Let S and T be independent modal specifications. Then $S \wedge T \lhd S$ and $S \wedge T \lhd T$. Moreover, if $R \lhd S$ and $R \lhd T$ then $R \lhd S \wedge T$.

In order to compare specifications at different levels of abstraction, it is important to abstract from transitions resulting from internal communication. The way this is done for modal transition systems follows the lines of traditional labelled transition systems. That is, for a given modal transition system $\mathcal{S} = (\Sigma, A \cup \{\tau\}, \longrightarrow_\Box, \longrightarrow_\Diamond)$ we derive the modal transition system $\mathcal{S}_\varepsilon = (\Sigma, A \cup \{\varepsilon\}, \Longrightarrow_\Box$

, $\Longrightarrow_\Diamond$), where $\overset{\varepsilon}{\Longrightarrow}_\Box$ is the reflexive and transitive closure of $\overset{\tau}{\longrightarrow}_\Box$, and where $T \overset{a}{\Longrightarrow}_\Box T', a \neq \varepsilon$, means that there exist T'', T''' such that

$$T \overset{\varepsilon}{\Longrightarrow}_\Box T'' \overset{a}{\longrightarrow}_\Box T''' \overset{\varepsilon}{\Longrightarrow}_\Box T'$$

The relation $\Longrightarrow_\Diamond$ is defined in a similar manner.

The notion of *weak refinement* can now be introduced as follows: S weakly refines T in $\mathcal{S}$, $S \trianglelefteq T$, iff there exists a refinement relation on $\mathcal{S}_\varepsilon$ containing S and T.

Weak refinement $\trianglelefteq$ essentially enjoys the same pleasant properties as $\triangleleft$: it is a preorder preserved by all modal CCS operators except $+$ [HL89]. Moreover, for ordinary labelled transition systems weak refinement reduces to the usual notion of weak bisimulation ($\approx$).

3 Projective Views

In the following, we present, motivate and clarify our proof methodology by means of a minimal example, which is just sufficient to explain the various phenomena. It should, however, be noted that the method scales up and has been successfully applied to Leslie Lamport's and Manfred Broy's Specification Problem of a recent Dagstuhl Seminar, which aimed at the evaluation of formal methods, see [BL93].

Consider the parallel system in Figure 1. Here two parameterized, disposable component media (supposed to transmit natural numbers) A and B are composed in parallel yielding a pipeline. Informally, the component A is supposed to input a natural number on port a, then output this number on port b after which it will terminate. The behaviour of B is similar. Using modal transition systems,

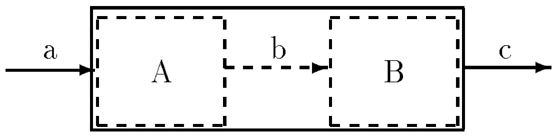


Figure 1: A Pipe Line of Two Disposable Media

the parallel system may be expressed as follows:

$$\Big(\underbrace{a_\Box x.\overline{b_\Box} x}_{A} \mid \underbrace{b_\Box x.\overline{c_\Box} x}_{B}\Big)\backslash\{b\}$$

The behaviour of A and B are given by the two infinite–width transition systems of Figure 2. However, rather than using these direct specifications of A and B we specify the two components behaviour using projective views A_n and B_n; one view for each possible natural number n. The projective view A_n specifies the

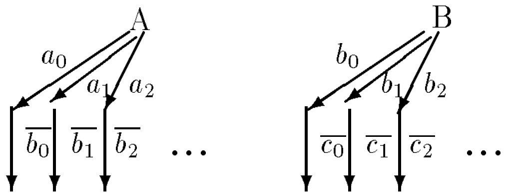
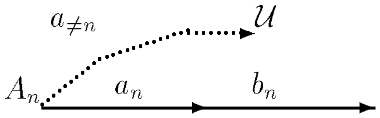


Figure 2: Behaviour of A and B.

constraints on the behaviour of the component A when focusing on transmission of the value n; this constraint can be expressed as the following modal transition system A_n (where we use solid lines for must- and dotted lines for may-transition):



Here $a_{\neq n}$ denotes all labels of the form a_m where $m \neq n$; also $\mathcal{U}$ denotes the universal modal transition system constantly allowing all actions. Note that this 'n-th view' imposes no constraint on the behaviour of A when transporting values different from n. The complete specification of the component A is the conjunction of all projective views[2] A_n. In fact it is easy to establish the following facts:

$$A \trianglelefteq \bigwedge_n A_n \quad \text{and} \quad \bigwedge_n A_n \trianglelefteq A \tag{1}$$

where A refers to the (infinite) transition system of Figure 2. Obviously, we may obtain similar projective views B_n for component B.

Let us now consider the problem of verifying that the overall system $\Big(A \,|\, B\Big)\backslash\{b\}$ is observationally equivalent to the system $C = a_\Box x.\overline{c_\Box} x$ (i.e. a slightly different disposable media). As A, B and C are standard transitions systems, i.e., everything allowed is also required, this problem is equivalent to showing

$$\Big(A \,|\, B\Big)\backslash\{b\} \;\trianglelefteq\; C$$

Thus (1), together with the observation that also C may be expressed as a conjunction of an infinite number of constraints C_n, leave us with the following refinement problem:

$$\Big(\bigwedge_n A_n \,|\, \bigwedge_n B_n\Big)\backslash\{b\} \;\trianglelefteq\; \bigwedge_n C_n \tag{2}$$

[2]Note that all the projective views of A are pairwise independent.

4 Sufficient Proof Condition

Due to the properties of conjunction (cf. Lemma 2.4) the proof of (2) can obviously be reduced to the verification of

$$\Big(\bigwedge_n A_n \mid \bigwedge_n B_n\Big)\backslash\{b\} \;\trianglelefteq\; C_n$$

for each natural n. Thus due to Lemma 2.4 and the fact that $\trianglelefteq$ is preserved by parallel composition and restriction, it suffices to prove

$$\forall n \in N.\ \Big(A_n \mid B_n\Big)\backslash\{b\} \;\trianglelefteq\; C_n \tag{3}$$

There is a general proof principle behind this reduction: in order to conclude:

$$\Big(\bigwedge_{i\in I_1} A_i^1 \mid \ldots \mid \bigwedge_{i\in I_k} A_i^k\Big)\backslash L \;\trianglelefteq\; \bigwedge_{j\in I} C_j$$

it suffices for each $j \in I$ to establish:

$$\Big(\bigwedge_{i\in I_{1,j}} A_i^1 \mid \ldots \mid \bigwedge_{i\in I_{k,j}} A_i^k\Big)\backslash L \;\trianglelefteq\; C_j$$

where $I_{l,j} \subseteq I_l$ for each $l = 1 \ldots k$.

Of course, in general the power of this proof principle strongly depends on a good choice of the $I_{l,j}$, which was trivial in our example.

5 Skolemization and Abstraction

So far we have reduced the overall verification problem of (2) to that of (3). At first sight this doesn't seem much of a reduction as (3) requires a refinement proof to be established for each natural number. Fortunately, these proofs are not really sensitive to the actual value of the natural number n. Letting k be an arbitrary natural number (or a Skolem constant) it suffices to prove:

$$\Big(A_k \mid B_k\Big)\backslash\{b\} \;\trianglelefteq\; C_k \tag{4}$$

in order to infer (3). Thus we are now left with the problem of establishing a *single* refinement. But still, though finite *state* the specifications A_k and B_k both have infinitely many *transitions* (as $a_{\neq k}$ is an inifinite label set). This problem can be overcome using *abstraction* (or *factorization*) with respect to an appropriate equivalence relation.

Definition 5.1 *Let $\mathcal{S} = (\Sigma, S, \longrightarrow_\Box, \longrightarrow_\Diamond)$ be a modal transition system, let $\equiv_\Sigma$ and $\equiv_S$ be equivalence relations on Σ and S, and let $\Sigma^\equiv$ and $S^\equiv$ be the sets of equivalence classes. Then* the factorization of $\mathcal{S}$ *is the modal transition system $\mathcal{S}^\equiv = (\Sigma^\equiv, S^\equiv, \longrightarrow'_\Box, \longrightarrow'_\Diamond)$, where $\longrightarrow'_\Box, \longrightarrow'_\Diamond$ are defined as follows:*

$$\frac{s \xrightarrow{a}_\Box s'}{[s]^\equiv \xrightarrow{[a]^\equiv}{}'_\Box [s']^\equiv} \qquad \frac{s \xrightarrow{a}_\Diamond s'}{[s]^\equiv \xrightarrow{[a]^\equiv}{}'_\Diamond [s']^\equiv}$$

Equivalence relations $\equiv_\Sigma$ and $\equiv_S$ are called compatible *with the modal transition system $\mathcal{S}$ iff for all $a \equiv_\Sigma b, s \equiv_S t, s' \equiv_S t'$:*

$$s \xrightarrow{a}_\Box s' \quad \textit{iff} \quad t \xrightarrow{b}_\Box t' \qquad \textit{and} \qquad s \xrightarrow{a}_\Diamond s' \quad \textit{iff} \quad t \xrightarrow{b}_\Diamond t'$$

For compatible equivalence relations, the following reduction lemma is straightforward:

Lemma 5.2 *Let $S^\equiv$ and $T^\equiv$ be processes in the factorization $\mathcal{S}^\equiv$ of $\mathcal{S}$ with respect to compatible equivalence relations $\equiv_\Sigma$ and $\equiv_S$. Then we have for arbitrary representatives S of $S^\equiv$ and T of $T^\equiv$:*

$$S^\equiv \trianglelefteq T^\equiv \quad \textit{implies} \quad S \trianglelefteq T$$

This Lemma allows us to reduce verification problems for infinite systems to problems for finite systems, as soon as an appropriate factorization can be found.

For our example, let us consider the equivalence relation $\equiv$ defined by $x_k \equiv x_k$ and $x_i \equiv x_j$ whenever $i, j \neq k$, where x ranges over $\{a, b, c\}$. Obviously, $\equiv$ is compatible with the underlying transition system. Thus the verification of (2) can further be reduced to the refinement proof between the finite $\equiv$–abstracted versions of A_k, B_k and C_k

$$\left({A_k}^\equiv \mid {B_k}^\equiv \right) \backslash \{b\} \ \trianglelefteq \ {C_k}^\equiv \tag{5}$$

which can easily be done by means of the automatic verification tool TAV.

6 Specifications with Time

The above example can be extended to deal with real time. For the specification we use Wang Yi's Timed CCS (see [Yi91]) together with modal specifications. For details on these so called *Timed Modal Specifications* see [CGL93]. This method can be used with any totally ordered time domain, while in the following we will assume the positive real numbers.

The passing of time is modelled by a delay action $\varepsilon(d)$, where d is a positive real number. The intuitive meaning of such a delay is that d time units pass until the end of this action. Normal actions are enabled immediately, and can be taken at any time. As an example, the process $a_\Box x.\varepsilon(2).\overline{b_\Box}x$ can execute $a_\Box x$ at any time. Thereafter it must delay for at least two time units before it can engage in $b_\Box x$.

Further we assume *maximal progress*, i.e. a communication must be performed as soon as possible. Putting $a_\Box x.\varepsilon(2).\overline{b_\Box}x$ in parallel with $\overline{a_\Box}x.\varepsilon(3).b_\Box x$ would force the communication via channel a to take place immediately, and the communication via channel b to happen after exactly three time units.

For our specification, the macro $a[l,u]$ is convenient, where a is an action and l,u are real numbers with $l < u$. The intuition is that a process $a[l,u].P$ *may* enable a after l time units and *must* enable a after u time units. In other words, communication via a may be possible after at least l time units, and will be possible at any time after u time units. This macro is defined as $a[l,u].P = (\varepsilon(l).a_\Diamond + \varepsilon(u).a_\Box).P$.

Let d be a fixed real number. Then we specify a timed process $A(d)$, which reads port a and subsequently outputs its input onto port b within d time units, by $a_\Box x.\overline{b}x[0,d]$.Note that this is a timed version of process A. The same construction gives timed versions $B(d)$ and $C(d)$ of B and C.

We are now going to establish that a 'pipeline' with two components with delay d should not be slower than one component with delay $2d$, i.e.

$$\Big(A(d) \mid B(d)\Big) \setminus \{b\} \;\trianglelefteq\; C(2d)$$

The same method as in the untimed case reduces the situation to

$$\Big({A_k}^{\equiv}(d) \mid {B_k}^{\equiv}(d)\Big) \setminus \{b\} \;\trianglelefteq\; {C_k}^{\equiv}(2d)$$

for a Skolem constant k and the equivalence relation of the previous section. Now, given a specific value for d this proof can be carried out using the EPSILON tool (see [CGL93]), which treats real valued timer domains by means of the clock region automaton technique (see [AD94] for details). This technique relies on integer values for all explicit timer constants in the specification, which can be achieved by multiplication with an appropriate constant in most applications. As all timer constants are multiplied by the same constant, this does not affect the principle behaviour of the system. In our example, the obvious choice for this constant is $1/d$, leaving us with the following refinement problem

$$\Big({A_k}^{\equiv}(1) \mid {B_k}^{\equiv}(1)\Big) \setminus \{b\} \;\trianglelefteq\; {C_k}^{\equiv}(2)$$

which can be solved using EPSILON.

Note that this proof indeed covers the statement for any d. Thus even in the presence of real time, the original verification problem is reduced to a very simple, automatically solvable problem.

7 Conclusion and Future Work

We have introduced a new constraint-oriented method for the (automated) verification of concurrent systems. Key concepts of our 'divide and conquer' method are *projective views*, *separation of proof obligations*, *Skolemization* and *abstraction*, which together support a drastic reduction of the complexity of the relevant subproblems. Of course, our proof methodology does neither guarantee the possibility of a finite state reduction nor a straightforward method for finding the right amount of separation or the adequate abstraction. Still, there is a large class of problems and systems, where the method can be applied quite straightforwardly. Typical examples are systems with limited data dependence. Whereas involved data dependencies may exclude any possibility of 'horizontal' decomposition, our approach elegantly extends to real time systems, even over a dense time domain. In fact, the resulting finite state problems can be automatically verified using the EPSILON verification system. All this has been illustrated using a simple example of pipelined buffers. Our experience indicates that our method scales up to practically relevant problems.

Beside further case studies and the search for good heuristics for proof obligation separation and abstraction, we are investigating the limits of tool support during the construction of constraint based specifications and the application of the three reduction steps. Whereas support by graphical interfaces and interactive editors is obvious and partly implemented in the META-Framework, a management system for synthesis, analysis and verification currently developed at the university of Passau, the limits of consistency checking and tool supported search for adequate separation and abstraction are still an interesting open research topic.

References

[ASW94] H. Andersen, C. Stirling, G. Winskel. A Compositional Proof System for the Modal Mu-Calculus. in: Proceedings LICS, 1994.

[AD94] R. Alur, D.L. Dill. A Theory of Timed Automata. in: Theoretical Computer Science Vol. 126, No. 2, April 1994, pp. 183-236.

[BL93] M. Broy, L. Lamport. Specification Problem. Case study for the Dagstuhl Seminar 9439, 1994.

[Br86] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. in: IEEE Transactions on Computation, 35 (8). 1986.

[BCMDH90] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. in: Proceedings LICS'90.

[CGL93] K. Čerāns, J.C. Godesken, K.G. Larsen. Timed Modal Specification - Theory and Tools. in: C. Courcoubetis (Ed.), Proc. 5th Int. Conf. on Computer Aided Verification (CAV '93), Elounda, Greece, June/July 1993. LNCS 697, Springer Berlin 1993, pp. 253–267.

[CGL92] E. Clarke, O. Grumber, D. Long. Model Checking and Abstraction. in: Proceedings XIX POPL'92.

[CLM89] E. Clarke, D. Long, K. McMillan. Compositional Model Checking. in: Proceedings LICS'89.

[CC77] R. Bryant. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction and Approximation of Fixpoints. in: Proceedings POPL'77.

[EFT91] R. Enders, T. Filkorn, D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. in: Proceedings CAV'91, LNCS 575, 1991, pp. 203–213

[GW91] P. Godefroid, P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. in: Proceedings CAV'91, LNCS 575, 1991, pp. 332–342.

[GP93] P. Godefroid, D. Pirottin. Refining Dependencies Improves Partial-Order Verification Methods. in: Proceedings CAV'93, LNCS 697, 1991, pp. 438–449.

[GL93] S. Graf, C. Loiseaux. Program Verification using Compositional Abstraction. in: Proceedings FASE/TAPSOFT'93.

[GS90] S. Graf, B. Steffen. Using Interface Specifications for Compositional Minimization of Finite State Systems. in: Proceedings CAV'90.

[HL89] H. Hüttel and K. Larsen. The use of static constructs in a modal process logic. Proceedings of Logic at Botik'89. LNCS 363, 1989.

[Lar90] K.G. Larsen. Modal specifications. In: Proceedings of Workshop on Automatic Verification Methods for Finite State Systems LNCS 407, 1990.

[LT88] K. Larsen and B. Thomsen. A modal process logic. In: Proceedings LICS'88, 1988.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Par81] D. Park. Concurrency and automata on infinite sequences. In P. Deussen (ed.), 5th GI Conference, LNCS 104, pp. 167–183, 1981.

[Val93] A. Valmari. On-The-Fly Verification with Stubborn Sets. in: C. Courcoubetis (Ed.), Proc. 5th Int. Conf. on Computer Aided Verification (CAV '93), Elounda, Greece, June/July 1993. LNCS 697, Springer Berlin 1993, pp. 397–408.

[Yi91] W. Yi. CCS + Time = an Interleaving Model for Real-Time Systems, Proc.18th Int. Coll. on Automata, Languages and Programming (ICALP), Madrid, July 1991. LNCS 510, Springer New York 1991, pp. 217-228.

Reasoning with Actions

S. B. Lassen
BRICS*, University of Aarhus
email: thales@brics.aau.dk
URL: http://www.daimi.aau.dk/~thales

Abstract Action semantics is a semantic description framework with very good pragmatic properties but a rather weak theory for reasoning about programs. A strong action theory would be of great practical use, however. It would make it possible to reason about the large class of programming languages that can be described in action semantics.

This paper develops the foundations for a richer action theory, by bringing together concepts and techniques from testing theory for processes and from work on operational reasoning about functional programs. Semantic preorders and equivalences in the action semantics setting are studied and a useful operational technique for establishing testing equivalences is presented.

1 Introduction

In this paper we develop a richer theory for reasoning about programs in *action semantics* (AS). Because AS is a general semantic description framework, our work has a great practical scope. A strong action theory would offer techniques for reasoning about programs in any programming language that can be described in AS.

1.1 Program reasoning in action semantics

AS is a denotational-style semantics. An *action semantic description* (ASD) maps program terms to the semantic entities they denote. The semantic entities are called *actions*.

Normally, we shall regard an ASD of a programming language as the formal definition of the language it describes, hence it is trivially adequate. Because ASDs are also compositional, language terms are operationally equivalent if their

*Basic Research in Computer Science, Centre of the Danish National Research Foundation.

denotations, i.e. actions, are equivalent—provided this action equivalence is defined properly.

Therefore, with the right notion of action equivalence, equivalences of programs can be established by showing their actions equivalent. This motivates the undertaking of this paper. In the rest of the paper, we shall examine notions of semantic equivalence of actions and techniques for reasoning about these action equivalences.

1.2 Existing action theory

Current action theory mainly consists of a series of action laws that codify algebraic properties of actions. In [16, App.C] the meaning of actions is defined in a structural operational semantics, a testing equivalence of actions is defined (a sort of MAY equivalence [2]), and a bisimulation technique for establishing equivalences is developed. By means of bisimulation, all action laws can be shown to respect testing equivalence.

Practical applications are few. Action laws are applied in the ACTRESS compiler generator for optimisations [21] but it remains doubtful whether the applicability of present techniques goes beyond simple, local transformations. Especially for reasoning about memory operations neither laws nor bisimulation seem to have much to offer.

1.3 Background for our work

Actions have functional, imperative, as well as concurrent features. There is thus a great range of places to look for inspiration for developing action theory.

Testing and bisimulation are notions from process calculus theory. Following up on the former, we will examine the full range of standard testing preorders and equivalences for actions. But our main source of inspiration comes from work on operational reasoning about impure functional languages.

Ian Mason and Carolyn Talcott have developed powerful operational techniques for deciding program equivalences that involve memory [13, 11]. A drawback of their approach is that it deals with a particular functional language. They have considered an extension to a concurrent ACTOR language [1]. Nevertheless, it is not clear how their techniques apply to other programming languages.

In contrast, if we can develop a strong action theory, it will apply readily to the large class of programming languages that admit an ASD. We shall in the following try to fit actions into Mason and Talcott's framework and generalise their techniques to nondeterminism as it appears in actions.

1.4 Overview

Section 2 introduces AS as well as the actions and action laws to be treated in the rest of the paper. In order to fit actions into Mason and Talcott's framework, Section 3 defines a so-called reduction semantics for action. In Section 4, testing preorders and equivalences are defined and a context lemma is established. Section 5 develops a technique for proving actions test preordered, based on the context lemma, and sketches the resulting inequational and equational action theory. A discussion of future and related work concludes the paper.

2 Action Semantics

Action semantics (AS) is a semantic description framework with very good pragmatic properties [16]. It has been used to formalise a wide range of realistic programming languages, as demonstrated by complete descriptions of Pascal [19] and ANDF [9] as well as of numerous other functional, procedural, and object-oriented languages. Moreover, a number of compiler generators have been based on AS [23, 3, 21, 22].

2.1 Action semantic descriptions

An action semantic description (ASD) consists of compositional semantic functions that map abstract syntax to the universal domain of actions. Suppose we have an imperative language with a syntactic category Statement in the abstract syntax. We can have a semantic function such as:

- execute _ :: Statement → action .

Some example semantic equations:

(1) execute "**skip**" = complete .

(2) execute ⟦ S_1:Statement ";" S_2:Statement ⟧ =
  execute S_1 and then execute S_2 .

(3) execute ⟦ "**loop**" S:Statement ⟧ =
  (unfolding
    (execute S and then unfold))
  trap complete .

(4) execute "**exit**" = escape .

The actions appearing on the right hand sides of the semantic equations will be explained shortly.

2.2 Action notation

Action notation (AN) is the specification language of AS for expressing actions. It is a rich language which expresses common computational notions in a direct way. AN is structured into various *facets* according to the information being processed:

- The **basic** facet is concerned with *control*, no information is processed. Higher-order, exceptional, nondeterministic, and interleaving control flow are expressible.
- The **functional** facet—processes transient data; features unbounded choice.
- The **declarative** facet—deals with bindings; control, data, and binding flow may be combined in various ways.
- The **imperative** facet—operates on a single-threaded storage.
- The **communicative** facet—connects distributed agents which communicate asynchronously.

The semantic equations above employ only the basic facet of AN. For simplicity we shall focus on the basic actions in this paper.

2.3 Basic actions

AN is a combinator based notation. Actions are either *primitives* or compounds built using *combinators.* The outcome of performing an action is

completed, normal termination (subsequent processing continues normally);

escaped, exceptional termination (transfers control to an exception handler);

failed, abnormal termination (aborts current processing); or

divergent, no termination.

The basic primitives are:

complete, escape, and fail, completes, fails, and escapes, respectively.

check _, takes a truth value argument and completes if it is true and fails if it is false.

enact _, invokes the abstraction yielded by its argument—an abstraction is an abstracted (or 'reified') action.

unfold, marks a point of recursion in the body of the unary combinator unfolding (see below).

The basic combinators are:

and then, binary sequential combinator.

and, binary combinator; performs its two subactions in any, arbitrarily interleaved order.

trap, binary combinator; performs its second subaction as an exception handler if the first subaction escapes.

or, binary choice combinator; if one subaction fails, the other is chosen, if neither subaction fails, one is chosen nondeterministically.

unfolding, unary recursion combinator; the primitive unfold in the subaction of unfolding denotes a recursive unwinding of the subaction. A divergent action is most shortly written diverge = unfolding unfold .

A grammar of basic AN can be specified in unified algebras [18] as follows:

- action-primitive = complete **|** escape **|** fail **|**
  check yielder **|** enact yielder **|** unfold .
- action = action-primitive **|** action and then action **|** action and action **|**
  action trap action **|** action or action **|** unfolding action .
- yielder = true **|** false **|** abstraction of action .

The arguments to the parameterised primitives check and enact are called yielders. In the basic facet, yielders are just constants, either truth values or abstractions.

To illustrate how actions give semantics to programming language terms, consider two statements from the example language from before:

(i) **skip** .
(ii) **loop exit** .

The semantic function execute maps them to a primitive and to a compound action, respectively:

(i) complete .
(ii) | unfolding
| | escape and then unfold
trap complete .

The latter escapes inside the body of unfolding, is then caught by trap, and the whole action completes.

Basic actions are not very expressive in themselves. Basic actions can just diverge or terminate in three different ways. In particular, they cannot process data in any way. Only when fused with data processing facets the expressive power of basic actions becomes interesting. Nevertheless, the basic facet suffices to illustrate the main concepts and techniques which we shall present.

2.4 Action laws

There are a series of 'action laws' that characterise actions. The following are a sample of basic action laws:

(1) $\mathsf{check\ true} = \mathsf{complete}$; $\mathsf{check\ false} = \mathsf{fail}$.

(2) $\mathsf{unfolding}\ A = A\ @\ \mathsf{unfolding}\ A$.

(3) $(A_1\ \mathsf{and\ then}\ A_2)\ \mathsf{and\ then}\ A_3 = A_1\ \mathsf{and\ then}\ (A_2\ \mathsf{and\ then}\ A_3)$.

(4) $\mathsf{complete\ and\ then}\ A = A\ \mathsf{and\ then\ complete} = A$.

(5) $\mathsf{escape\ and\ then}\ A = \mathsf{escape}$; $\mathsf{fail\ and\ then}\ A = \mathsf{fail}$.

(6) $(A_1\ \mathsf{trap}\ A_2)\ \mathsf{trap}\ A_3 = A_1\ \mathsf{trap}\ (A_2\ \mathsf{trap}\ A_3)$.

(7) $\mathsf{escape\ trap}\ A = A\ \mathsf{trap\ escape} = A$.

(8) $\mathsf{complete\ trap}\ A = \mathsf{complete}$; $\mathsf{fail\ trap}\ A = \mathsf{fail}$.

Law (1) characterises check. Law (2) states that unfolding is equal to its own unwinding (the operator @ substitutes its second argument for all 'free' occurrences of the primitive unfold in the first argument, $A_1 @ A_2$ = "$A_1[A_2/\mathsf{unfold}]$"). Laws (3)–(5) say that and then is associative, has complete as unit, and is left-absorbed by escape and fail. Laws (6)–(8) are corresponding laws for trap.

As an example application of these laws, the example actions (i)–(ii) above can be shown equal using laws (4), (2), (7).

A catalogue of action laws for all facets of AN may be found in [16, App.B]. In [15], a precursor of AS, such laws served as an algebraic semantics of actions. However, establishing laws sufficient to define full action notation in this way doesn't seem feasible.

3 Reduction semantics

Mason and Talcott operate on the basis of a 'reduction semantics' for their language. A reduction semantics is a small-step operational semantics that represents control and state syntactically.

Our first step towards fitting actions into Mason and Talcott's framework is to define a reduction semantics for basic AN. (A preliminary sketch was presented in [12].) The essential novelty compared to [16, App.C] is that control is represented by means of evaluation contexts.

3.1 Decomposition

An *evaluation context* [6] (or *reduction context* [13]) E is an action with a hole at a legal point of execution in the action. Either just a hole [], a hole to the left of the sequential combinators (*seq* is either of and then , trap), or a hole on either side of the symmetric combinators (*sym* is either of and , or).

- $\mathsf{evaluation\text{-}context} = [\,] \mid \mathsf{evaluation\text{-}context}\ \mathit{seq}\ \mathsf{action}$
  $\mid \mathsf{evaluation\text{-}context}\ \mathit{sym}\ \mathsf{action}$
  $\mid \mathsf{action}\ \mathit{sym}\ \mathsf{evaluation\text{-}context}$.

A *redex* R is any action that can make a transition directly (see the evaluation rules below). $E[R]$ denotes the action obtained by filling R into the hole in E.

In an action $E[R]$, the evaluation context E corresponds to the point of execution in the program, the program pointer, or current continuation. The redex R is the current instruction.

When an action A can be written as an evaluation context E filled with a redex R, viz. $A = E[R]$, we say that A *decomposes* into E, R. In the following example, there are two possible decompositions of A. So decomposition is not unique.

Example 3.1 *Let* $A = \mathsf{check\ true\ or\ (check\ false\ and\ then\ escape)}$.
Then $A = E_1[\mathsf{check\ true}] = E_2[\mathsf{check\ false}]$,
where $E_1 = [\,]\ \mathsf{or\ (check\ false\ and\ then\ escape)}$;
$E_2 = \mathsf{check\ true\ or\ ([\,]\ and\ then\ escape)}$.

3.2 Evaluation

The evaluation rules decompose an action and replace the redex with a reduced (or unwound) action.

$E : \mathsf{evaluation\text{-}context}\ ;\ A, A_1, A_2 : \mathsf{action}\ ;\ Y : \mathsf{yielder} \Rightarrow$

$$
\begin{array}{lll}
(1) & E[\mathsf{check}\ Y] & \mapsto \begin{cases} E[\mathsf{complete}] & \mathit{if}\ \ Y = \mathsf{true} \\ E[\mathsf{fail}] & \mathit{if}\ \ Y = \mathsf{false}\ . \end{cases} \\
(2) & E[\mathsf{enact}\ Y] & \mapsto\ E[A]\ \ \mathit{if}\ \ Y = \mathsf{abstraction\ of}\ A\ . \\
(3) & E[\mathsf{unfolding}\ A] & \mapsto\ E[A\ @\ \mathsf{unfolding}\ A]\ . \\
(4) & E[A_1\ \mathsf{and\ then}\ A_2] & \mapsto \begin{cases} E[A_1] & \mathit{if}\ \ A_1 : \mathsf{escape} \mid \mathsf{fail} \\ E[A_2] & \mathit{if}\ \ A_1 = \mathsf{complete}\ . \end{cases} \\
(5) & E[A_1\ \mathsf{and}\ A_2] & \mapsto \begin{cases} E[A_1] & \mathit{if}\ \ A_1 : \mathsf{escape} \mid \mathsf{fail}\ \ \mathit{or}\ \ A_2 = \mathsf{complete} \\ E[A_2] & \mathit{if}\ \ A_1 = \mathsf{complete}\ \ \mathit{or}\ \ A_2 : \mathsf{escape} \mid \mathsf{fail}\ . \end{cases} \\
(6) & E[A_1\ \mathsf{trap}\ A_2] & \mapsto \begin{cases} E[A_1] & \mathit{if}\ \ A_1 : \mathsf{complete} \mid \mathsf{fail} \\ E[A_2] & \mathit{if}\ \ A_1 = \mathsf{escape}\ . \end{cases} \\
(7) & E[A_1\ \mathsf{or}\ A_2] & \mapsto \begin{cases} E[A_1] & \mathit{if}\ \ A_1 : \mathsf{complete} \mid \mathsf{escape}\ \ \mathit{or}\ \ A_2 = \mathsf{fail} \\ E[A_2] & \mathit{if}\ \ A_1 = \mathsf{fail}\ \ \mathsf{em\ or}\ \ A_2 : \mathsf{complete} \mid \mathsf{escape}\ . \end{cases}
\end{array}
$$

Call an action *terminated* if it is any of complete, escape, fail.

The following useful characterisation of evaluation can be shown. If an action makes a transition and we can compose the action as an evaluation context filled with a non-terminated subaction, then either (i) the transition chooses a redex in our subaction, or else a parallel redex is used which either (ii) doesn't affect our subaction, or (iii) eliminates it.

Lemma 3.1 (choice of execution point) *If* $E[A] \mapsto B'$ *and A is not terminated, then either*

$$\begin{array}{lll} (i) & B' = E[B] & \textit{where } A \mapsto B \ , \\ \textit{or} \ (ii) & B' = E'[A] & \textit{and, for all } A',\ E[A'] \mapsto E'[A'] \ , \\ \textit{or} \ (iii) & E[A'] \mapsto B' & \textit{for all } A' \ . \end{array}$$

4 Testing

In this section, we shall define test preorders and equivalences for actions. We also present a tractable characterisation, based on evaluation contexts.

4.1 Definedness

A *computation* from some action A is a sequence of transitions, $A \mapsto A_1 \mapsto A_2 \mapsto \cdots$ We say that a computation is *successful* if it is finite and the final A_n is either complete or escape. Define two 'definedness' predicates for actions. Action A *may* succeed, $A \downarrow_{\text{MAY}}$, if there exists a successful computation from A. A *must* succeed, $A \downarrow_{\text{MUST}}$, if, moreover, all computations from A are successful.

More formally, let $\mathsf{success} = \mathsf{complete} \mid \mathsf{escape}$, then:

Definition 4.1 (defined) $\downarrow_{\text{MAY}}$ *and* $\downarrow_{\text{MUST}}$ *are the least predicates satisfying*

$$A \downarrow_{\text{MAY}} \quad \textit{if} \quad A \mapsto^* S \ \ \textit{for some} \ \ S{:}\mathsf{success} \ .$$

$$A \downarrow_{\text{MUST}} \quad \textit{if} \quad A \downarrow_{\text{MAY}} \ \ \textit{and} \ \ (\ A \mapsto B \ \ \textit{implies} \ \ B \downarrow_{\text{MUST}}\ ) \ .$$

We will need the following inductive formulation for induction proofs:

$$\begin{array}{lllll} \downarrow_{\text{MAY}} & = & \bigcup_{\lambda<\omega_0} \downarrow^{\lambda}_{\text{MAY}} & \textit{where} \ \downarrow^{0}_{\text{MAY}} & = \ \mathsf{success} \ , \\ & & & \phantom{\textit{where}}\ \downarrow^{\lambda}_{\text{MAY}} & = \ \{\ A \mid \exists\, B \in \bigcup_{\kappa<\lambda} \downarrow^{\kappa}_{\text{MAY}} .\ A \mapsto B\ \} \ . \\ \downarrow_{\text{MUST}} & = & \bigcup_{\lambda<\omega_0} \downarrow^{\lambda}_{\text{MUST}} & \textit{where} \ \downarrow^{0}_{\text{MUST}} & = \ \mathsf{success} \ , \\ & & & \phantom{\textit{where}}\ \downarrow^{\lambda}_{\text{MUST}} & = \ \{\ A \in \downarrow_{\text{MAY}} \mid \forall\, B \ \textit{s.t.}\ A \mapsto B. \\ & & & & \qquad\qquad B \in \bigcup_{\kappa<\lambda} \downarrow^{\kappa}_{\text{MUST}}\ \} \ . \end{array}$$

Basic action notation is finitely branching and by König's lemma we only need to do induction over the finite ordinals (all $\lambda < \omega_0$). If we add the functional facet which introduces countably branching nondeterminism, we must do induction over countable ordinals (all $\lambda < \omega_1$).

4.2 Testing equivalence

Now we can define three testing equivalences. We define them as preorders which induce corresponding equivalences.

Definition 4.2 (test preorders)

$$\begin{array}{rcl} A_1 \sqsubseteq_{\sim \text{MAY}} A_2 & \textit{iff} & \forall\, C.\ C[A_1]\downarrow_{\text{MAY}} \;\Rightarrow\; C[A_2]\downarrow_{\text{MAY}}\ . \\ A_1 \sqsubseteq_{\sim \text{MUST}} A_2 & \textit{iff} & \forall\, C.\ C[A_1]\downarrow_{\text{MUST}} \;\Rightarrow\; C[A_2]\downarrow_{\text{MUST}}\ . \\ A_1 \sqsubseteq_{\sim} A_2 & \textit{iff} & A_1 \sqsubseteq_{\sim \text{MAY}} A_2 \quad \textit{and} \quad A_1 \sqsubseteq_{\sim \text{MUST}} A_2\ . \end{array}$$

C ranges over arbitrary action contexts, actions with any number of holes anywhere (also inside abstractions, unfoldings, etc.) These testing preorders generalise conventional operational preorder to the nondeterministic case.

The induced equivalences are:

$$\begin{array}{rcl} \cong_{\text{MUST}} & = & \sqsubseteq_{\sim \text{MUST}} \cap \sqsupseteq_{\sim \text{MUST}}\ , \\ \cong_{\text{MAY}} & = & \sqsubseteq_{\sim \text{MAY}} \cap \sqsupseteq_{\sim \text{MAY}}\ , \\ \cong & = & \sqsubseteq_{\sim} \cap \sqsupseteq_{\sim}\ . \end{array}$$

Call the three notions of preorder/equivalence MAY, MUST, and TEST, respectively.

These equivalences are, by definition, the weakest equivalences such that, in any context, equivalent subterms are interchangeable without any observable effect. One chooses MAY, MUST, or TEST, dependent on one's view of observation: whether one ignores divergence, regards divergence as catastrophic, or demands equal divergence properties, respectively.

The testing equivalence defined for actions in [16, C.4] corresponds to our MAY equivalence. (Both the notion of test and the notion of observation used there are a little less standard.) We shall instead take the TEST equivalence as basic. It is a stronger equivalence, and thus a 'safe compromise' between the MAY and the MUST view of divergence, and it turns out that the action laws hold even for TEST equivalence.

Having all the test preorders, we can express more properties of actions by a variety of inequational laws. This will be exemplified in Section 5.

4.3 Alternative characterisation

The definition of test preorders is difficult to use to prove actions preordered or equivalent. We need an alternative characterisation which gives a tractable way of determining equivalence. This approach is used in test theory [10] and it is central to Mason and Talcott's operational work. Inspired by the latter we have shown the following 'evaluation' preorders to characterise the test preorders:

Definition 4.3 (evaluation preorders)

$$\begin{array}{rcl} A_1 \lesssim_{\text{MAY}} A_2 & \textit{iff} & \forall\, E.\ E[A_1]\downarrow_{\text{MAY}} \;\Rightarrow\; E[A_2]\downarrow_{\text{MAY}}\ . \\ A_1 \lesssim_{\text{MUST}} A_2 & \textit{iff} & \forall\, E.\ E[A_1]\downarrow_{\text{MUST}} \;\Rightarrow\; E[A_2]\downarrow_{\text{MUST}}\ . \end{array}$$

These preorders only quantify over evaluation contexts, not arbitrary contexts. This turns out to be much more tractable. The following lemma is the key to all reasoning about test preorders and equivalences:

Lemma 4.1 (basic context lemma)

$$A_1 \sqsubsetsim_{\text{MAY}} A_2 \quad \textit{iff} \quad A_1 \lesssim_{\text{MAY}} A_2 \; .$$
$$A_1 \sqsubsetsim_{\text{MUST}} A_2 \quad \textit{iff} \quad A_1 \lesssim_{\text{MUST}} A_2 \; .$$

Proof (outline): Proceed as in the '**ciu**' proof in [11]:

The 'only if's are immediate because evaluation contexts are instances of arbitrary contexts.

The core of the 'if' directions consists of showing the evaluation preorders to be precongruences for basic AN. Then assume $A_1 \lesssim_{\text{MAY}} A_2$ and show $A_1 \sqsubsetsim_{\text{MAY}} A_2$ as follows: First of all, $C[A_1] \lesssim_{\text{MAY}} C[A_2]$, for any C, because $\lesssim_{\text{MAY}}$ is a precongruence. Therefore $C[A_1] \downarrow_{\text{MAY}} \;\Rightarrow\; C[A_2] \downarrow_{\text{MAY}}$. Hence $A_1 \sqsubsetsim_{\text{MAY}} A_2$. Likewise for MUST. □

The formalisation and proof of the basic context lemma have been carefully phrased such that they generalise to extensions of basic AN. We have investigated additions of functional, declarative, and imperative facets, and in all cases natural generalisations carry over. We are studying stronger versions of the basic context lemma which hold for all facets. Hitherto we have shown that in the MAY case (which normally has the simplest characterisation, cf. [10]) we need only consider contexts without occurrences of the or combinator.

The basic facet For basic actions, things are very simple and, in fact, only a small finite number of contexts are needed to distinguish the equivalence classes of the MAY and MUST relations. No more than 2 contexts are necessary to distinguish the MAY equivalence classes:

$$[\,]\ \mathsf{trap\ fail}\ , \quad [\,]\ \mathsf{and\ then\ fail}\ .$$

Only 3 contexts suffice for MUST:

$$([\,]\ \mathsf{or\ complete})\ \mathsf{trap\ fail}\ , \quad [\,]\ , \quad ([\,]\ \mathsf{or\ escape})\ \mathsf{and\ then\ fail}\ .$$

This doesn't generalise to other facets.

5 (In)equational action theory

The advantage of having several testing preorders and equivalences, instead of one canonical equivalence, is a greater expressive power. This section demonstrates how the preorders yield interesting inequational characterisations of actions. Furthermore, we discuss how existing equational theory of actions turns into a TEST equational theory.

5.1 Inequational theory

First some properties of the two test preorders which can be shown using the basic context lemma. They complement the existing action laws and further characterise the nondeterministic behaviour of actions. First define succeed, diverge, and the completely nondeterministic action chaos.

$$\begin{aligned}
&\mathsf{succeed} = \mathsf{complete\ or\ escape}\ .\\
&\mathsf{diverge} = \mathsf{unfolding\ unfold}\ .\\
&\mathsf{chaos} = \left|\ \mathsf{diverge\ and\ fail\ and\ escape}\right.\\
&\qquad\qquad \mathsf{trap\ succeed}\ .
\end{aligned}$$

Below are a sample of inequational laws that hold for actions.

(1) $A_i \underset{\sim}{\sqsubset}_{\mathrm{MAY}} (A_1\ \mathsf{or}\ A_2)$.

(2) $(A_1\ \mathsf{and\ then}\ A_2) \underset{\sim}{\sqsubset}_{\mathrm{MAY}} (A_1\ \mathsf{and}\ A_2)$.

(3) $\bot_{\mathrm{MAY}} \cong_{\mathrm{MAY}} \mathsf{diverge} \cong_{\mathrm{MAY}} \mathsf{fail}$.

(4) $\top_{\mathrm{MAY}} \cong_{\mathrm{MAY}} \mathsf{chaos} \cong_{\mathrm{MAY}} \mathsf{succeed}$.

(5) $(A_1\ \mathsf{and}\ A_2) \underset{\sim}{\sqsubset}_{\mathrm{MUST}} (A_1\ \mathsf{and\ then}\ A_2)$.

(6) $\bot_{\mathrm{MUST}} \cong_{\mathrm{MUST}} \mathsf{diverge} \cong_{\mathrm{MUST}} \mathsf{chaos}$.

As a rough intuition of the $\underset{\sim}{\sqsubset}_{\mathrm{MAY}}$ and $\underset{\sim}{\sqsubset}_{\mathrm{MUST}}$ preorders, they both mean 'more divergent', and both $\underset{\sim}{\sqsupset}_{\mathrm{MAY}}$ and $\underset{\sim}{\sqsubset}_{\mathrm{MUST}}$ mean 'less deterministic'. Thus diverge is both $\bot_{\mathrm{MAY}}$ and $\bot_{\mathrm{MUST}}$, chaos is both $\top_{\mathrm{MAY}}$ and $\bot_{\mathrm{MUST}}$. In this vein, we may read the following lemma may as 'evaluation reduces nondeterminism':

Lemma 5.1 (evaluation) *If* $A_1 \mapsto A_2$, *then* $A_1 \underset{\sim}{\sqsupset}_{\mathrm{MAY}} A_2$ *and* $A_1 \underset{\sim}{\sqsubset}_{\mathrm{MUST}} A_2$.

Proof: By **basic context lemma** (4.1) and induction on the $\downarrow_{\mathrm{MAY}}$ and $\downarrow_{\mathrm{MUST}}$ predicates, using lemma **choice of execution point** (3.1). □

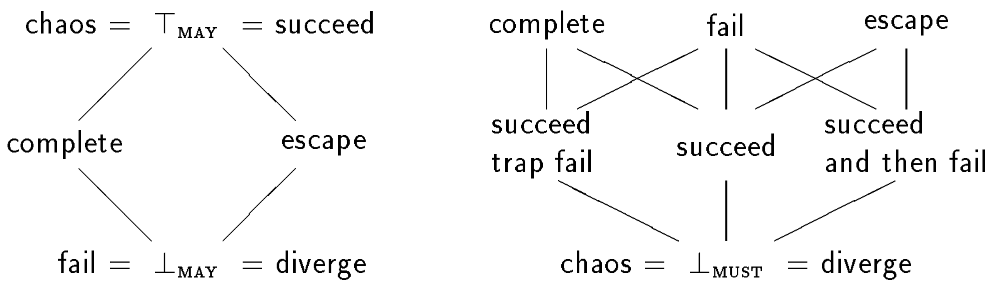
A couple of immediate consequences are:

(7) $\mathsf{complete} \underset{\sim}{\sqsubset}_{\mathrm{MAY}} (\mathsf{complete\ or}\ A) \underset{\sim}{\sqsubset}_{\mathrm{MUST}} \mathsf{complete}$.

(8) $\mathsf{fail} \underset{\sim}{\sqsubset}_{\mathrm{MAY}} (\mathsf{fail\ and}\ A) \underset{\sim}{\sqsubset}_{\mathrm{MUST}} \mathsf{fail}$.

In deterministic settings, when one term can make a transition into another, then the terms are operationally equivalent [7, 13]. This demonstrates how the presence of nondeterminism weakens action theory.

The basic facet There are, in fact, only 4 MAY and 7 MUST equivalence classes for basic actions. The MAY and MUST orderings can be depicted as follows:



5.2 Equational theory

Under certain conditions we may strengthen the evaluation lemma.

Lemma 5.2 (determined evaluation)
If $A_1 \mapsto A_2$ *is the only transition out of* A_1*, then* $A_1 \cong A_2$.

Proof: As for lemma **evaluation** (5.1). □

Some laws from Section 2.4 are direct consequences of this lemma.

(1) check true $\cong$ complete ; check false $\cong$ fail .
(2) unfolding A $\cong$ A @ unfolding A .
(4a) complete and then A $\cong$ A .
(5) escape and then A $\cong$ escape ; fail and then A $\cong$ fail .
(7a) escape trap A $\cong$ A .
(8) complete trap A $\cong$ complete ; fail trap A $\cong$ fail .

By the same proof technique as for the proofs of lemma 5.1 and 5.2, we can show all existing basic action laws TEST equivalent.

6 Conclusion

We have defined test preorders for actions and developed a useful operational proof technique for establishing them, based on a tractable alternative characterisation of the test preorders.

The standard range of test preorders and equivalences allowed us to exhibit inequational and equational laws that give rich characterisations of actions. In particular, we have shown that existing action laws hold for TEST equivalence.

Since basic actions are not very expressive, basic action theory is not very interesting in itself. But we believe that all the theory which we have presented for basic actions generalises to full AN. We have developed this for the main parts of the functional, declarative, and imperative facets—we have defined reduction semantics, proved generalisations of lemma 4.1, and proved existing action laws to hold for TEST equivalence. The proof technique used in the basic facet as well as the evaluation lemma 5.1 and 5.2 carry over to these other facets too. All the inequational and equational basic action laws which we have used as illustrations in this paper remain true when other facets are added.

These extensions are all inspired by [11]. We still have not considered the communicative facet but we believe that [1] should be readily applicable.

6.1 Alternative semantics of actions

Above we have reworked the existing action theory in [16] which addresses full AN, based on a nondeterministic structural operational semantics and bisimulation. Alternative semantic foundations for actions exist but none of these appear extensible to a comprehensive theory of full AN. In [5] a categorical semantics is defined for a typed subset of AN and a number of action laws are proven. For a similar subset of AN, [4] defines a versatile natural semantics framework for reasoning about (contextual) semantic equivalences (among many other things). Palsberg [23] and Moura [20] define natural semantics for larger AN subsets. Moura defines a 'functional' action equivalence but not a full action theory.

6.2 Future work

We shall conclude with some of our plans for future work.

Apart from extending our work to full AN, we would like to employ more of Mason and Talcott's framework, viz. develop the equational theory for actions, as in [14], and develop a program logic for actions, as in [11].

We have a number of applications in mind. First of all it would be interesting whether, given an ASD of Mason and Talcott's functional language, we could prove their results by means of our theory.

Bisimulation Bisimulation has been side-stepped in this paper. In many cases, bisimulation offers the simplest proof technique and it definitely deserves study in our setting. Bisimulation is normally the strongest equivalence considered in semantics and is, generally, included in all other equivalences. Standard weak bisimulation doesn't guarantee MUST equivalence, however, because it is insensitive to 'silent' divergence. In [25] this is remedied and, furthermore, weak bisimulation is formulated in terms of preorders. It would be interesting to define such a divergence sensitive weak bisimulation preorder for actions and show it to be included in all testing preorders along the lines in [16, C.4].

In recent years, bisimulation has been subject to much study in functional settings. Cf. e.g. [8] who also summarises research in this area. This line of work may indicate alternative adaptions of (bi)simulation into action theory.

Full abstraction Typically, ASDs are not *fully abstract* [24]. Actions are, as a rule, more expressive than the language being described. Therefore action contexts can distinguish denotations that are indistinguishable by the less expressive language contexts.

This problem is indirectly addressed by our modular approach. In this paper we investigate only basic actions. We have also considered various other subsets of AN: extensions to the functional, declarative, and imperative facets and also restrictions to deterministic subsets of each of these. The ambition is to develop strong theories for subsets of AN in a structured and modular way which can be put together to sound subtheories for arbitrary subsets of AN. If an ASD is not fully abstract, it is, in general, because it doesn't exploit all of AN. If only a subset of AN is used to express the ASD, this subset may have a stronger theory than the weak, universal action theory. This may be a practical way to narrow the full abstraction gap.

Acknowledgements. The presentation of this work has benefited very much from helpful suggestions by Peter Mosses and Peter Ørbæk.

References

[1] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation, Aug. 1994. To appear in *Journal of Functional Programming*.

[2] R. DeNicola and M. Hennessy. Testing equivalences for processes. *Theoretical Comput. Sci.*, 34:83–133, 1984.

[3] K.-G. Doh and D. Schmidt. Action semantics-directed prototyping. *Comput. Lang.*, 19(4):213–233, 1993.

[4] K.-G. Doh and D. Schmidt. The facets of action semantics: Some principles and applications. In [17], 1994.

[5] S. Even and D. A. Schmidt. Category sorted algebra-based action semantics. *Theoretical Comput. Sci.*, 77:73–96, 1990.

[6] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*. IFIP, 1987.

[7] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Comput. Sci.*, 103:235–271, 1992.

[8] A. D. Gordon. A tutorial on co-induction and functional programming. In *Glasgow Workshop on Functional Programming*, 1994.

[9] B. S. Hansen and J. U. Toft. The formal specification of ANDF, an application of action semantics. In [17], 1994.

[10] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[11] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable typed logic of effects, 1993. To appear in *Information and Computation*.

[12] S. B. Lassen. Design and Semantics of Action Notation. In [17], 1994.

[13] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):297–327, 1991.

[14] I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Comput. Sci.*, 105(2):167–215, 1993.

[15] P. D. Mosses. Abstract semantic algebras! In D. Bjørner, editor, *Formal Description of Programming Concepts II*. IFIP, 1983.

[16] P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[17] P. D. Mosses, editor. *Proceedings of the First International Workshop on Action Semantics (Edinburgh, Scotland, April 1994)*, number NS-94-1 in BRICS Notes Series. Dept. of Computer Science, Univ. of Aarhus, 1994.

[18] P. D. Mosses. Unified algebras and abstract syntax. In *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[19] P. D. Mosses and D. A. Watt. Pascal action semantics, version 0.6. Available by FTP from ftp.daimi.aau.dk in pub/action/pascal, Mar. 1993.

[20] H. Moura. *Action Notation Transformations*. Ph.D. thesis, University of Glasgow, 1993.

[21] H. Moura and D. Watt. Action transformations in the ACTRESS compiler generator. In *CC'94*, volume 786 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[22] P. Ørbæk. OASIS: An Optimizing Action-based Compiler Generator. In *CC'94*, volume 786 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[23] J. Palsberg. *Provably Correct Compiler Generation*. PhD thesis, Dept. of Computer Science, Univ. of Aarhus, 1992.

[24] G. D. Plotkin. LCF considered as a programming language. *Theoretical Comput. Sci.*, 5:223–255, 1977.

[25] D. J. Walker. Bisimulation and divergence. *Information and Computation*, 85(2):202–241, 1990.

Preprocessing by Program Specialization

Karoline Malmkjær and Olivier Danvy
Computer Science Department
Aarhus University *

December 1994

As is well-known, searching for a string in a given text can be linearized in the string by preprocessing the text. Most preprocessing algorithms have been devised either top-down by instantiating abstract properties on words (*e.g.*, Knuth's derivation of the Knuth, Morris, & Pratt linear string-matching algorithm from Cook's theorem [7]), or bottom-up by observing possible regularities in the structure of the text. The present work describes an alternative approach to preprocessing that uses properties of a simple two-arguments matching program to determine structures yielding linear matching. We mechanically deduce a one-argument linear matching program and its tree-like data structures by specializing the two-arguments program with respect to a text.

We consider the usual naïve and quadratic solution: a string occurs in a text if it is a prefix of one of the suffixes of the text.

We use program specialization, or *partial evaluation* [4, 5]: specializing a source program with respect to part of its input yields a specialized version of the source program that, given the rest of the input, yields the same result as the source program, given the whole input. The correctness and the computability of program specialization stem from Kleene's S^m_n-theorem [6].

The naïve quadratic program expects any string and any text and returns the position of that string in that text or -1. Specializing this source program with respect to the text "aba" automatically yields a residual program that expects a string and returns the position of this string in "aba" or -1. This residual program can be expressed graphically by representing a successful test against the character a as the transition $\xrightarrow{a}$, and where each node $\odot$ has two implicit transitions: a success one if the pattern ends there, and a failure if no character matches the explicit transitions. This yields the following tree, which makes it possible to match a string linearly.

*Ny Munkegade, DK-8000 Aarhus C, Denmark. E-mail: (karoline, danvy)@daimi.aau.dk
Supported by the DART project funded by The Danish Research Councils.

$$
\begin{tikzcd}
 & \odot \arrow[r, "b"] & \odot \arrow[r, "a"] & \odot \\
 {} \arrow[r] & \odot \arrow[u, "a"] \arrow[r, "b"'] & \odot \arrow[r, "a"'] & \odot
\end{tikzcd}
$$

This tree is similar to Weiner's prefix trees [8] (a.k.a. position trees [1]), but it contains the full suffixes, whereas prefix trees only contain prefix identifiers. Therefore this tree grows quadratically with the text, whereas prefix trees are only worst-case quadratic.

We can fold it by relaxing the requirement on the source program to return the position of a string in a text: we only want to know *whether* this string occurs in this text. Specializing the corresponding source program with respect to the text "aba" automatically yields a residual program that expects a string and determines whether this string occurs in "aba" by traversing the string linearly. In this case, further compaction is possible: the tree can be folded into a directed acyclic word graph [2].

$$
\begin{tikzcd}
 & \odot \arrow[dr, "b"] & & \\
 {} \arrow[r] & \odot \arrow[u, "a"] \arrow[r, "b"'] & \odot \arrow[r, "a"'] & \odot
\end{tikzcd}
$$

This approach generalizes the derivation of the Knuth, Morris, & Pratt linear string-matching program by partial evaluation of a naïve and quadradic string-matching program [3].

Acknowledgements

Originally, this work was partially supported by the Commission for Educational Exchange Between Denmark and the United States of America, the Danish Research Academy, and the American-Scandinavian Foundation.

The diagrams were drawn with Kristoffer Rose's XY-pic package.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[3] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989.

[4] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[5] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[6] Stephen C. Kleene. *Introduction to Metamathematics*. D. van Nostrand, Princeton, New Jersey, 1952.

[7] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[8] Peter Weiner. Linear pattern matching algorithms. In *IEEE Symposium on Switching and Automata Theory*, pages 1–11, New York, 1973.

Introduction to ALF — an Interactive Proof Editor

Lena Magnusson

Chalmers Technical University
Inst. for Computer Science
Göteborg, Sweden
E-mail: `lena@cs.chalmers.se`

We will give a description of incremental type checking, which is the main tool for doing interactive proof development in ALF.

ALF is based on Martin-Löf's type theory, which is a logical framework in the sense that the language can be extended with new definitions. The language is a functional language with dependent function types, which can be extended with data types, functions and constants as new definitions, in the spirit of traditional functional languages. The dependent function type gives us a richer type system, in which propositions and specifications of programs can be represented. Thus, the relation

$$e : T$$

can denote a program e of type T or a proof e of proposition T. This relation is decidable and the type (proof) checking algorithm is the core of the proof editor ALF. However, the objective of ALF is to interactively build proof terms or programs, in a flexible and safe way. For this purpose we have extended the language with the notion of placeholders (meta variables) which denotes sub-terms intended to be filled in. Hence, an incomplete proof term is a term containing placeholders, and proof refinement corresponds to successively replacing a placeholder by a (possibly incomplete) sub-term.

The type checking algorithm is extended to handle incomplete terms which gives as result a unification problem, i.e. a set of equations and typing restrictions of the placeholders occurring in the incomplete term. This set corresponds to the requirements which must be satisfied in future refinements. The unification problem can not always be solved, due to higher order placeholders, and unsolved equations are left as constraints restricting future instantiations of the placeholders. Correctness of a refinement is established by type checking the refined (incomplete) proof term. Thus, proof refinement is reduced to type checking incomplete proof terms.

Static and Dynamic Processor Allocation for Higher-Order Concurrent Languages*

Hanne Riis Nielson, Flemming Nielson

Computer Science Department, Aarhus University, Denmark.
e-mail:{hrnielson,fnielson}@daimi.aau.dk

Abstract. Starting from the process algebra for Concurrent ML we develop two program analyses that facilitate the intelligent placement of processes on processors. Both analyses are obtained by augmenting an inference system for counting the number of channels created, the number of input and output operations performed, and the number of processes spawned by the execution of a Concurrent ML program. One analysis provides information useful for making a static decision about processor allocation; to this end it accumulates the communication cost for all processes with the same label. The other analysis provides information useful for making a dynamic decision about processor allocation; to this end it determines the maximum communication cost among processes with the same label. We prove the soundness of the inference system and the two analyses and demonstrate how to implement them; the latter amounts to transforming the syntax-directed inference problems to instances of syntax-free equation solving problems.

1 Introduction

Higher-order concurrent languages as CML [16] and FACILE [5] offer primitives for the dynamic creation of processes and channels. A distributed implementation of these languages immediately raises the problem of processor allocation. The efficiency of the implementation will depend upon how well the *network configuration* matches the *communication topology* of the program – and here it is important which processes reside on which processors. When deciding this it will be useful to know:

- Which channels will be used by the process for input and output operations and how many times will the operations be performed?
- Which channels and processes will be created by the process and how many instances will be generated?

As an example, two processes that frequently communicate with one another should be allocated on processors in the network so as to ensure a low communication overhead.

*This is a preliminary version of a paper accepted for TAPSOFT'95; the full version appears as DAIMI-PB 483 and is also available on `http://www.daimi.aau.dk/~bra8130/LOMAPS.html`

In CML and FACILE processes and channels are created dynamically and this leads naturally to a distinction between two different processor allocation schemes:

- *Static processor allocation*: At compile-time it is decided where all instances of a process will reside at run-time.
- *Dynamic processor allocation*: At run-time it is decided where the individual instances of a process will reside.

The first scheme is the simpler one and it is used in the current distributed implementation of FACILE; finer grain control over parallelism may be achieved using the second scheme [18].

What has been accomplished. In this paper we present *analyses providing information for static and dynamic processor allocation* of CML programs. We shall follow the approach of [13] and develop the analyses in two stages:

- Extract the communication behaviour of the CML program.
- Analyse the behaviour.

The two analyses only differ in the second stage.

The *first stage* follows [13] in developing a type and behaviour inference system for expressing the communication capabilities of programs in CML. This formulation takes full account of the polymorphism present in ML and algorithms for the automatic extraction of behaviours from CML programs are developed in [14, 1]. As was already indicated in [12] the behaviours may be regarded as terms in a process algebra (like CCS or CSP); however the process algebra of behaviours is specifically designed so as to capture those aspects of communication that are relevant for the efficient implementation of programs in CML.

The *second stage* of the two analyses are developed in detail in the present paper. To prepare for this we first develop an analysis that uses simple ideas from abstract interpretation to count for *each behaviour* the number of channels created, the number of input and output operations performed and the number of processes spawned. To provide information for static and dynamic processor allocation we then differentiate the information with respect to *labels* associated with the `fork` operations of the CML program; these labels will identify all instances of a given process and *for each label* we count the number of channels created, the number of input and output operations performed and the number of processes spawned. The central observation is now that for the *static* allocation scheme we *accumulate* the requirements of the individual instances whereas for the *dynamic* allocation scheme we take the *maximum* of the individual instance requirements.

In this paper we prove the *correctness* of the second stage of the analysis. The analyses are specified as inference systems and the correctness proof is based on a structural operational semantics for behaviours and an appropriate abstraction of the non-negative natural numbers. The correctness of the complete analysis then follows from the subject reduction result of [13] that allows us to *lift* safety (as opposed to liveness) results from the behaviours to safety results for CML programs.

We also address the *implementation* of the second stage of the analysis. Here the idea is to transform the problem as specified by the syntax-directed inference system into a syntax-free equation solving problem where standard techniques from data flow analysis can be used to obtain fast implementations. (As already mentioned the implementation of the first stage is the topic of [14, 1].)

Comparison with other work. First we want to stress that our approach to processor allocation is that of *static program analysis* rather than, say, heuristics based on profiling as is often found in the literature on implementation of concurrent languages.

In the literature there are only few program analyses for combined functional and concurrent languages. An extension of SML with Linda communication primitives is studied in [3] and, based on the corresponding process algebra, an analysis is presented that provides useful information for the placement of processes on a finite number of processors. A functional language with communication via shared variables is studied in [9] and its communication patterns are analysed, again with the goal of producing useful information for processor (and storage) allocation. Also a couple of program analyses have been developed for concurrent languages with an imperative facet. The papers [4, 8, 15] all present reachability analyses for concurrent programs with a statically determined communication topology; only [15] shows how this restriction can be lifted to allow communication in the style of the π-calculus. Finally, [11] presents an analysis determining the number of communications on each channel connecting two processes in a CSP-like language.

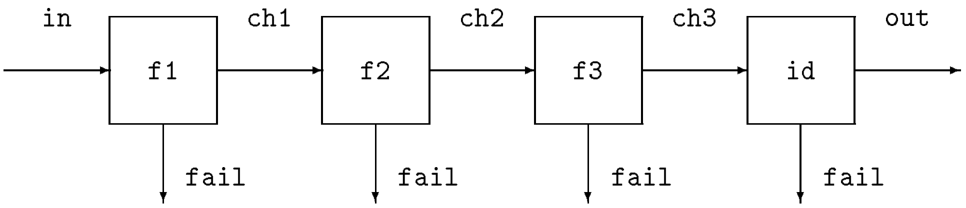
As mentioned our analysis is specified in two stages. The *first stage* is formalised in [13, 14]; similar considerations were carried out by Havelund and Larsen leading to a comparable process algebra [6] but with *no* formal study of the link to CML *nor* with any algorithm for automatically extracting behaviours. The same overall idea is present in [3] but again with *no* formal study of the link between the process algebra and the programming language.

The *second stage* of the analysis extracts much more detailed information from the behaviours and this leads to a much more complex notion of correctness than in [13]. Furthermore, the analysis is parameterised on the choice of value space thereby incorporating ideas from abstract interpretation.

2 Behaviours

Full details of the syntax of CML are not necessary for the developments of the present paper. It will suffice to introduce a running example and to use it to motivate the process algebra of CML.

Example 2.1 *Suppose we want to define a program* `pipe [f1,f2,f3] in out` *that constructs a pipeline of processes: the sequence of inputs is taken over channel* `in`*, the sequence of outputs is produced over channel* `out` *and the functions* `f1`*,* `f2`*,* `f3` *(and the identity function* `id` *defined by* `fn x => x`*) are applied in turn. To achieve concurrency we want separate processes for each of the functions* `f1`*,* `f2`*,* `f3` *(and* `id`*). This system might be depicted graphically as follows:*



Here `ch1`*,* `ch2`*, and* `ch3` *are new internal channels for interconnecting the processes; and* `fail` *is a channel over which failure of operation may be reported.*

Taking the second process as an example it may be created by the CML expression `node f2 ch1 ch2` *where the function* `node` *is given by*

```
fn f => fn in => fn out =>
forkπ (rec loop d =>
      sync (choose [wrap (receive in,
                          fn x => sync (send (out, f x));
                                  loop d),
                    send(fail,())]))
```

Here `f` *is the function to be applied,* `in` *is the input channel and* `out` *is the output channel. The function* $\texttt{fork}_\pi$ *creates a new process labelled* π *that performs as described by the recursive function* `loop` *that takes the dummy parameter* `d`*. In each recursive call the function may either report failure by* `send(fail,())` *or it may perform one step of the processing: receive the input by means of* `receive in`*, take the value* `x` *received and transmit the modified value* `f x` *by means of* `send(out,f x)` *after which the process repeats itself by means of* `loop d`*. The primitive* `choose` *allows to perform an unspecified choice between the two communication possibilities and* `wrap` *allows to modify a communication by postprocessing the value received or transmitted. The* `sync` *primitive enforces synchronisation at the right points and we refer to [16] for a discussion of the language design issues involved in this; once we have arrived at the process algebra such considerations will be of little importance to us.*

The overall construction of the network of processes is then the task of the `pipe` *function defined by*

```
rec pipe fs => fn in => fn out =>
if isnil fs
then node (fn x => x) in out
else let ch = channel ()
     in (node (hd fs) in ch; pipe (tl fs) ch out)
```

Here `fs` *is the list of functions to be applied,* `in` *is the input channel, and* `out` *is the output channel. If the list of functions is empty we connect* `in` *and* `out` *by means of a process that applies the identity function; otherwise we create a new internal channel by means of* `channel ()` *and then we create the process for the first function in the list and then recurse on the remainder of the list.*

The process algebra of CML [13] allows to give succinct representations of the communications taking place in CML programs. The terms of the process algebra are called *behaviours*, denoted $b \in \mathbf{Beh}$, and are given by

$$b \quad ::= \quad \epsilon \mid L!t \mid L?t \mid t \ \mathrm{CHAN}_L \mid \beta \mid \mathrm{FORK}_L \ b \mid b_1; b_2 \mid b_1 + b_2 \mid \mathrm{REC}\beta.\, b$$

where $L \subseteq \mathbf{Labels}$ is a non-empty and finite set of program labels. The behaviour ϵ is associated with the pure functional computations of CML. The behaviours $L!t$ and $L?t$ are associated with sending and receiving values of type t over channels with label in L, the behaviour $t \ \mathrm{CHAN}_L$ is associated with creating a new channel with label in L and over which values of type t can be communicated, and the behaviour $\mathrm{FORK}_L \ b$ is associated with creating a new process with behaviour b and with label in L. Together these behaviours constitute the *atomic behaviours*, denoted $p \in \mathbf{ABeh}$, as may be expressed by setting

$$p \quad ::= \quad \epsilon \mid L!t \mid L?t \mid t \ \mathrm{CHAN}_L \mid \mathrm{FORK}_L \ b$$

Finally, behaviours may be composed by sequencing (as in $b_1; b_2$) and internal choice (as in $b_1 + b_2$) and we use *behaviour variables* together with an explicit REC construct to express recursive behaviours.

The structure of the types, denoted $t \in \mathbf{Typ}$, shall be of little concern to us in this paper and we shall therefore leave it mostly unspecified (but see [13]); however, we need to state that $\alpha \ \mathtt{chan}_L$ is the type of a channel with label in L over which elements of type α may be communicated. Since types might conceivably contain behaviours the notion of free variables needs to be replaced by a notion of exposed variables: we shall say that a behaviour variable β is *exposed* in a behaviour b if it has a free occurrence that is not a subterm of any type mentioned in b.

Example 2.2 *Assuming that* `fail` *is a channel of type* $\mathtt{unit\ chan}_L$ *the type inference system of [13] can be used to prove that* `pipe` *has type*

$$(\alpha \ \to^{\beta} \alpha) \ \mathtt{list} \to^{\epsilon} \alpha \ \mathtt{chan}_{L_1} \to^{\epsilon} \alpha \ \mathtt{chan}_{L_2} \to^{b} \mathtt{unit}$$

where b is

$$\begin{array}{l} \mathrm{REC}\beta'.(\mathrm{FORK}_\pi(\mathrm{REC}\beta''.(L_1?\alpha; \epsilon; L_2!\alpha; \beta'' + L!\mathtt{unit})) \\ \qquad + \alpha \ \mathrm{CHAN}_{L_1}; \mathrm{FORK}_\pi(\mathrm{REC}\beta''.(L_1?\alpha; \beta; L_2!\alpha; \beta'' + L!\mathtt{unit})); \beta') \end{array}$$

Thus the behaviour expresses directly that the `pipe` *function is recursively defined and that it either spawns a single process or creates a channel, spawns a process and recurses. The spawned processes will all be recursive and they will either report failure over a channel in L and terminate, or else input over a channel in L_1, do something (as expressed by ϵ and β), output over a channel in L_2 and recurse.*

The *semantics* of behaviours is defined by a transition relation of the form

$$PB \Longrightarrow^{a}_{ps} PB'$$

where PB and PB' are mappings from process identifiers to *closed* behaviours and the special symbol $\surd$ denoting termination. Furthermore, a is an action that takes place and ps is a list of the processes that take part in the action. The *actions* rather closely correspond to atomic behaviours and are given by

$$a ::= \epsilon \mid L!t?L \mid t\ \text{CHAN}_L \mid \text{FORK}_L\ b$$

If the transition $PB \Longrightarrow^{a}_{ps} PB'$ has $a = \epsilon$ this means that *one* of the behaviours in PB performed some internal computation that *did not* involve communication; in other words it performed the atomic behaviour ϵ. If $a = L!t?L$ this means that *two* disctinct behaviours performed a communication: one performed the atomic behaviour $L!t$ and the other the atomic behaviour $L?t$. Finally if $a = \text{CHAN}_L$ or $a = \text{FORK}_L$ this means that *one* of the behaviours in PB allocated a new channel or forked a new process. Since we have covered all possibilities of atomic behaviours we have also covered all possibilities of actions. We refer to [13] for the precise details of the semantics as these are of little importance for the development of the analyses.

3 Value Spaces

In the analyses we want to predict the number of times certain events may happen. The precision as well as the complexity of the analyses will depend upon how we count so we shall parameterise the formulation of the analyses on our notion of counting.

This amounts to abstracting the non-negative integers $\mathbf{N}$ by a complete lattice $(\mathbf{Abs}, \sqsubseteq)$. As usual we write $\bot$ for the least element, $\top$ for the greatest element, $\bigsqcup$ and $\sqcup$ for least upper bounds by a function and $\sqcap$ for greatest lower bounds. The abstraction is expressed

$\mathcal{R} : \mathbf{N} \rightarrow_m \mathbf{Abs}$ that is strict (has $\mathcal{R}(0) = \bot$) and monotone (has $\mathcal{R}(n_1) \sqsubseteq \mathcal{R}(n_2)$ whenever $n_1 \leq n_2$); hence the ordering on the natural numbers is reflected in the abstract values. Three elements of $\mathbf{Abs}$ are of particular interest and we shall introduce special syntax for them:

$$\text{O} = \mathcal{R}(0) = \bot \qquad \text{I} = \mathcal{R}(1) \qquad \text{M} = \top$$

We cannot expect our notion of counting to be precisely reflected by $\mathbf{Abs}$; indeed it is likely that we shall allow to identify for example $\mathcal{R}(2)$ and $\mathcal{R}(3)$ and perhaps even $\mathcal{R}(1)$ and $\mathcal{R}(2)$. However, we shall ensure throughout that no identifications involve $\mathcal{R}(0)$ by demanding that $\mathcal{R}^{-1}(\text{O}) = \{0\}$ so that O really represents "did not happen".

We shall be interested in two binary operations on the non-negative integers. One is the operation of *maximum*: $\max\{n_1, n_2\}$ is the larger of n_1 and n_2. In $\mathbf{Abs}$ we shall use the binary least upper bound operation to express the maximum operation. Indeed $\mathcal{R}(\max\{n_1, n_2\}) = \mathcal{R}(n_1) \sqcup \mathcal{R}(n_2)$ holds by monotonicity of $\mathcal{R}$ as do the laws $n_1 \sqsubseteq n_1 \sqcup n_2$, $n_2 \sqsubseteq n_1 \sqcup n_2$ and $n \sqcup n = n$. As a consequence $n_1 \sqcup n_2 = \text{O}$ iff both n_1 and n_2 equal O.

The other operation is *addition*: $n_1 + n_2$ is the sum of n_1 and n_2. In **Abs** we shall have to define a function $\oplus$ and demand that (**Abs**, $\oplus$, O) is an Abelian *monoid* with $\oplus$ monotone. This ensures that we have the associative law $n_1 \oplus (n_2 \oplus n_3) = (n_1 \oplus n_2) \oplus n_3$, the absorption laws $n \oplus \text{O} = \text{O} \oplus n = n$, the commutative law $n_1 \oplus n_2 = n_2 \oplus n_1$ and by monotonicity we have also the laws $n_1 \sqsubseteq n_1 \oplus n_2$ and $n_2 \sqsubseteq n_1 \oplus n_2$. As a consequence $n_1 \oplus n_2 = \text{O}$ iff both n_1 and n_2 equal O. To ensure that $\oplus$ models addition on the integers we impose the condition $\forall n_1, n_2.\ \mathcal{R}(n_1 + n_2) \sqsubseteq \mathcal{R}(n_1) \oplus \mathcal{R}(n_2)$ that is common in abstract interpretation.

Definition 3.1 *A* value space *is a structure* (**Abs**, $\sqsubseteq$, O, I, M, $\oplus$, $\mathcal{R}$) *as detailed above. It is an* atomic value space *if* I *is an atom (that is* $\text{O} \sqsubseteq n \sqsubseteq \text{I}$ *implies that* $\text{O} = n$ *or* $\text{I} = n$*).*

Example 3.2 *One possibility is to use* **A3** $= \{\text{O}, \text{I}, \text{M}\}$ *and define* $\sqsubseteq$ *by* $\text{O} \sqsubseteq \text{I} \sqsubseteq \text{M}$*. The abstraction function* $\mathcal{R}$ *will then map 0 to* O*, 1 to* I *and all other numbers to* M*. The operations* $\sqcup$ *and* $\oplus$ *can then be given by the following tables:*

$\sqcup$	O	I	M
O	O	I	M
I	I	I	M
M	M	M	M

$\oplus$	O	I	M
O	O	I	M
I	I	M	M
M	M	M	M

This defines an atomic value space.

For two value spaces (**Abs**$'$, $\sqsubseteq'$, O', I', M', $\oplus'$, $\mathcal{R}'$) and (**Abs**$''$, $\sqsubseteq''$, O'', I'', M'', $\oplus''$, $\mathcal{R}''$) we may construct their *cartesian product* (**Abs**, $\sqsubseteq$, O, I, M, $\oplus$, $\mathcal{R}$) by setting **Abs** $=$ **Abs**$'$ $\times$ **Abs**$''$ and by defining $\sqsubseteq$, O, I, M, $\oplus$ and $\mathcal{R}$ componentwise. This defines a value space but it is not atomic even if **Abs**$'$ and **Abs**$''$ both are. As a consequence $\text{I} = (\text{I}', \text{I}'')$ will be of no concern to us; instead we use (O', I'') and (I', O'') as appropriate.

For a value space (**Abs**$'$, $\sqsubseteq'$, O', I', M', $\oplus'$, $\mathcal{R}'$) and a non-empty set E of *events* we may construct the *indexed value space* (or function space) (**Abs**, $\sqsubseteq$, O, I, M, $\oplus$, $\mathcal{R}$) by setting **Abs** $= E \to$ **Abs**$'$ (the set of total functions from E to **Abs**$'$) and by defining $\sqsubseteq$, O, I, M, $\oplus$ and $\mathcal{R}$ componentwise. This defines a value space that is almost never atomic; as a consequence $\text{I} = \lambda e.\text{I}'$ will be of no concern to us.

For indexed value spaces we may represent ($f \in E \to$ **Abs**) by ($\text{rep}(f) \in E \hookrightarrow$ **Abs**$\backslash\{\text{O}\}$) where $E \hookrightarrow$ **Abs**$\backslash\{\text{O}\}$ denotes the set of partial functions from E to **Abs**$\backslash\{\text{O}\}$; here $\text{rep}(f)$ maps e to n iff $f(e) = n$ and $n \neq \text{O}$. In practice we want to restrict E to be a finite set in order to obtain finite representations; we write ($f \in E \to_f$ **Abs**) to indicate that f is O on all but a finite number of arguments so that such a representation is possible.

4 Counting the Behaviours

For a given behaviour b and value space **Abs** we may ask the following four questions:

$$benv \vdash \epsilon : [\,]$$

$$benv \vdash L!t : [L \mapsto (\mathrm{o},\mathrm{o},\mathrm{i},\mathrm{o})] \qquad benv \vdash L?t : [L \mapsto (\mathrm{o},\mathrm{i},\mathrm{o},\mathrm{o})]$$

$$benv \vdash t \ \textsc{chan}_L : [L \mapsto (\mathrm{i},\mathrm{o},\mathrm{o},\mathrm{o})] \qquad \frac{benv \vdash b : A}{benv \vdash \textsc{fork}_L\ b : [L \mapsto (\mathrm{o},\mathrm{o},\mathrm{o},\mathrm{i})] \oplus A}$$

$$\frac{benv \vdash b_1 : A_1 \quad benv \vdash b_2 : A_2}{benv \vdash b_1; b_2 : A_1 \oplus A_2} \qquad \frac{benv \vdash b_1 : A_1 \quad benv \vdash b_2 : A_2}{benv \vdash b_1 + b_2 : A_1 \sqcup A_2}$$

$$\frac{benv[\beta \mapsto A] \vdash b : A}{benv \vdash \textsc{rec}\ \beta.\ b : A} \qquad benv \vdash \beta : A \quad \text{if } benv(\beta) = A$$

Table 1: Analysis of behaviours

how many times are channels labelled by L created?

how many times do channels labelled by L participate in input?

how many times do channels labelled by L participate in output?

and how many times are processes labelled by L generated?

To answer these questions we define an inference system with formulae

$$benv \vdash b : A$$

where **LabSet** $= \mathcal{P}_f(\textbf{Labels})$ is the set of finite and non-empty subsets of **Labels** and $A \in \textbf{LabSet} \to_f \textbf{Abs}$ records the required information.

In this section we shall define the inference system for answering all four questions simultaneously. Hence we let **Abs** be the four-fold cartesian product $\mathbf{Ab}^4$ of an atomic value space **Ab**; we shall leave the formulation parameterised on the choice of **Ab** but a useful candidate is the three-element value space **A3** of Example 3.2 and this will be the choice in all examples.

The idea is that $A(L) = (n_c, n_i, n_o, n_f)$ means that channels labelled by L are created at most n_c times, that channels labelled by L participate in at most n_i input operations, that channels labelled by L participate in at most n_o output operations, and that processes labelled by L are generated at most n_f times. The behaviour environment *benv* then associates each behaviour variable with an element of **LabSet** $\to_f$ **Abs**.

The analysis is defined in Table 1. We use $[\,]$ as a shorthand for $\lambda L.(\mathrm{o},\mathrm{o},\mathrm{o},\mathrm{o})$ and $[L \mapsto \vec{n}]$ as a shorthand for $\lambda L'.\left\{\begin{array}{ll}(\mathrm{o},\mathrm{o},\mathrm{o},\mathrm{o}) & \text{if } L' \neq L \\ \vec{n} & \text{if } L' = L\end{array}\right\}$. Note that i denotes the designated "one"-element in each copy of **Ab** since it is the atoms $(\mathrm{i},\mathrm{o},\mathrm{o},\mathrm{o})$, $(\mathrm{o},\mathrm{i},\mathrm{o},\mathrm{o})$, $(\mathrm{o},\mathrm{o},\mathrm{i},\mathrm{o})$, and $(\mathrm{o},\mathrm{o},\mathrm{o},\mathrm{i})$ that are useful for increasing the count. In the rule for $\textsc{fork}_L$ we are deliberately incorporating the effects of the forked process; to avoid doing so simply remove the "$\oplus A$" component. The rules for sequencing, choice, and behaviour variables are straightforward given the developments of the previous section.

Note that the rule for recursion expresses a fixed point property and so allows some slackness; it would be inelegant to specify a least (or greatest) fixed point property whereas a post-fixed point[1] could easily be accomodated by incorporating a notion of subsumption into the rule. We decided not to incorporate a general subsumption rule and to aim for specifying as unique results as the rule for recursion allows.

Example 4.1 *For the* `pipe` *function of Examples 2.1 and 2.2 the analysis will give the following information (read "*M*" as "many"):*

L_1:	M *channels created and* M *inputs performed*
L_2:	M *outputs performed*
L:	M *outputs performed*
π:	M *processes created*

While this is evidently correct it also seems pretty uninformative; yet we shall see that this simple analysis suffices for developing more informative analyses for static and dynamic processor allocation.

To formally express the correctness of the analysis we need a few definitions. Given a list X of actions define:

$$\mathrm{COUNT}(X) = \lambda L.(CC(X,L), CI(X,L), CO(X,L), CF(X,L))$$

$CC(X,L)$:	the number of elements of the form t $\mathbf{CHAN}_L$ in X,
$CI(X,L)$:	the number of elements of the form $L'!t?L$ in X,
$CO(X,L)$:	the number of elements of the form $L!t?L'$ in X, and
$CF(X,L)$:	the number of elements of the form $\mathbf{FORK}_L$ b in X.

The formal version of our explanations above about the intentions with the analysis then amounts to the following soundness result:

Theorem 4.2 *If* $\emptyset \vdash b : A$ *and* $[pi_0 \mapsto b] \Longrightarrow^{a_1}_{ps_1} \cdots \Longrightarrow^{a_k}_{ps_k} PB$ *then we have*

$$\mathcal{R}^*(COUNT[a_1, \cdots, a_k]) \sqsubseteq A.$$

where $\mathcal{R}^*(C)(L) = (\mathcal{R}(c), \mathcal{R}(i), \mathcal{R}(o), \mathcal{R}(f))$ *if* $C(L) = (c, i, o, f)$.

5 Implementation

It is well-known that compositional specifications of program analyses (whether as abstract interpretations or annotated type systems) are not the most efficient way of obtaining the actual solutions. We therefore demonstrate how the inference problem may be transformed to an equation solving problem that is independent of the syntax of our process algebra and where standard algorithmic techniques may be applied. This approach also carries over to the inference systems for processor allocation developed subsequently.

The first step is to generate the set of equations. To show that this does not affect the set of solutions we shall be careful to avoid undesirable "cross-over" between equations generated from disjoint syntactic components of the behaviour. One possible cause for such "cross-over" is that behaviour variables

[1] We take a post-fixed point of a function f to be an argument n such that $f(n) \sqsubseteq n$.

$\mathcal{E}[\![B : \varpi : \epsilon]\!] = \{\langle\varpi\rangle = [\,]\}$
$\mathcal{E}[\![B : \varpi : L!t]\!] = \{\langle\varpi\rangle = [L \mapsto (\textsc{o},\textsc{o},\textsc{i},\textsc{o})]\}$
$\mathcal{E}[\![B : \varpi : L?t]\!] = \{\langle\varpi\rangle = [L \mapsto (\textsc{o},\textsc{i},\textsc{o},\textsc{o})]\}$
$\mathcal{E}[\![B : \varpi : t\ \textbf{CHAN}_L]\!] = \{\langle\varpi\rangle = [L \mapsto (\textsc{i},\textsc{o},\textsc{o},\textsc{o})]\}$
$\mathcal{E}[\![B : \varpi : \textbf{FORK}_L\ b]\!] = \{\langle\varpi\rangle = [L \mapsto (\textsc{o},\textsc{o},\textsc{o},\textsc{i})] \oplus \langle\varpi 1\rangle\} \cup \mathcal{E}[\![B : \varpi 1 : b]\!]$
$\mathcal{E}[\![B : \varpi : b_1; b_2]\!] = \{\langle\varpi\rangle = \langle\varpi 1\rangle \oplus \langle\varpi 2\rangle\} \cup \mathcal{E}[\![B : \varpi 1 : b_1]\!] \cup \mathcal{E}[\![B : \varpi 2 : b_2]\!]$
$\mathcal{E}[\![B : \varpi : b_1 + b_2]\!] = \{\langle\varpi\rangle = \langle\varpi 1\rangle \sqcup \langle\varpi 2\rangle\} \cup \mathcal{E}[\![B : \varpi 1 : b_1]\!] \cup \mathcal{E}[\![B : \varpi 2 : b_2]\!]$
$\mathcal{E}[\![B : \varpi : \beta]\!] = \{\langle\varpi\rangle = \langle\beta\rangle\}$
$\mathcal{E}[\![B : \varpi : \textbf{REC}\ \beta.\ b]\!] = \mathrm{CLOSE}^{\varpi}_{\beta}(\ \{\langle\varpi\rangle = \langle\varpi 1\rangle,\ \langle\varpi\rangle = \langle\beta\rangle\} \cup \mathcal{E}[\![B : \varpi 1 : b]\!]\ )$

Table 2: Constructing the equation system

may be bound in more than one **REC**; one classical solution to this is to require that the overall behaviour be alpha-renamed such that this does not occur; the solution we adopt avoids this requirement by suitable modification of the equation system. Another possible cause for "cross-over" is that disjoint syntactic components of the overall behaviour may nonetheless have components that syntactically appear the same; we avoid this problem by the standard use of tree-addresses (denoted ϖ).

The function $\mathcal{E}$ for generating the equations for the overall behaviour B achieves this by the call $\mathcal{E}[\![B : \varepsilon : b]\!]$ where ε denotes the empty tree-address. In general $B : \varpi : b$ indicates that the subtree of B rooted at ϖ is of the form b and the result of $\mathcal{E}[\![B : \varpi : b]\!]$ is the set of equations produced for b. The formal definition is given in Table 2.

The key idea is that $\mathcal{E}[\![B : \varpi : b]\!]$ operates with *flow variables* of the form $\langle\varpi'\rangle$ and $\langle\beta'\rangle$. We maintain the invariant that all ϖ' occurring in $\mathcal{E}[\![B : \varpi : b]\!]$ are (possibly empty) prolongations of ϖ and that all β' occurring in $\mathcal{E}[\![B : \varpi : b]\!]$ are exposed in b. To maintain this invariant in the case of recursion we define

$$\mathrm{CLOSE}^{\varpi}_{\beta}(\mathbf{E}) = \{\ (L[\langle\varpi\rangle/\langle\beta\rangle] = R[\langle\varpi\rangle/\langle\beta\rangle])\ \mid\ (L = R) \in \mathbf{E}\}$$

(although it would actually suffice to apply the substitution $[\langle\varpi\rangle/\langle\beta\rangle]$ on the righthand sides of equations and it would be correct to remove the trivial equation produced).

Terms of the equations are formal terms over the flow variables (that range over the complete lattice $\mathbf{LabSet} \to \mathbf{Abs}$), the operations $\oplus$ and $\sqcup$ and the constants (that are elements of the complete lattice $\mathbf{LabSet} \to \mathbf{Abs}$). Thus all terms are monotonic in their free flow variables. A *solution* to a set $\mathbf{E}$ of equations is a partial function σ from flow variables to $\mathbf{LabSet} \to \mathbf{Abs}$ such that all flow variables in $\mathbf{E}$ are in the domain of σ and such that all equations $(L = R)$ of $\mathbf{E}$ have $\sigma(L) = \sigma(R)$ where σ is extended to formal terms in the obvious way. We write $\sigma \models \mathbf{E}$ whenever this is the case.

Theorem 5.1 $[\,] \vdash b : A$ *iff* $\exists\sigma.\ \sigma \models \mathcal{E}[\![b : \varepsilon : b]\!] \wedge \sigma(\langle\varepsilon\rangle) = A.$

Corollary 5.2 *The least (or greatest) A such that $[\,] \vdash b : A$ is $\sigma(\langle\varepsilon\rangle)$ for the least (or greatest) σ such that $\sigma \models \mathcal{E}[\![b : \varepsilon : b]\!]$.*

We have now transformed our inference problem to a form where the standard algorithmic techniques can be exploited. These include simplifications of the equation system, partitioning the equation system into strongly connected components processed in (reverse) topological order, widening to ensure convergence when **Abs** does not have finite height etc.; a good overview of useful techniques may be found in [2, 7, 10, 17]. Also the flow variables may be decomposed to families of flow variables over simpler value spaces.

6 Static Processor Allocation

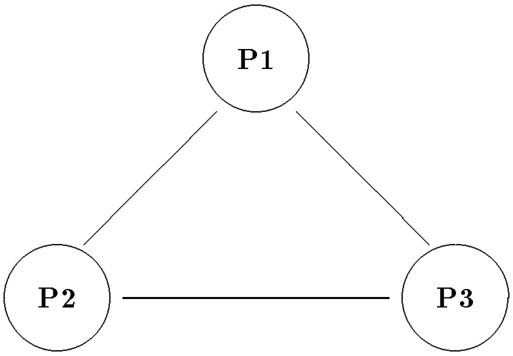
The idea behind the static processor allocation is that all processes with the *same* label will be placed on the *same* processor and we would therefore like to know what requirements this puts on the processor. To obtain such information we shall extend the simple counting analysis of Section 4 to associate information with the *process labels* mentioned in a given behaviour b. For each process label L_a we therefore ask the four questions of Section 4 accumulating the *total* information for all processes with label L_a: how many times are channels labelled by L created, how many times do channels labelled by L participate in input, how many times do channels labelled by L participate in output, and how many times are processes labelled by L generated?

Example 6.1 *Let us return to the* `pipe` *function of Examples 2.1 and 2.2 and suppose that we want to perform* static processor allocation*. This means that* all *instances of the processes labelled π will reside on the* same *processor. The analysis should therefore estimate the* total *requirements of these processes as follows:*

main:	L_1:	M *channels created*	π:	L_1:	M *inputs performed*
	π:	M *processes created*		L_2:	M *outputs performed*
				L:	M *outputs performed*

Note that even though each process labelled by π can only communicate once over L we can generate many such processes and their combined behaviour is to communicate many times over L. It follows from this analysis that the main program does not in itself communicate over L_2 or L and that the processes do not by themselves spawn new processes.

Now suppose we have a network of processors that may be explained graphically as follows:



One way to place our processes is to place the main program on **P1** *and all the processes labelled* π *on* **P2**. *This requires support for multitasking on* **P2** *and for multiplexing (over* L_1*) on* **P1** *and* **P2**.

The analysis (specified in Table 3) is obtained by modifying the inference system of Section 4 to have formulae

$$benv \vdash b : A \ \& \ P$$

where $A \in \mathbf{LabSet} \to_f \mathbf{Abs}$ as before and the new ingredient is

$$P : \mathbf{LabSet} \to_f (\mathbf{LabSet} \to_f \mathbf{Abs})$$

The idea is that if some process is labelled L_a then $P(L_a)$ describes the *total* requirements of all processes labelled by L_a. The behaviour environment *benv* is an extension of that of Section 4 in that it associates pairs $A \ \& \ P$ with the behaviour variables. Note that in the rule for $\texttt{FORK}_L$ we have removed the "$\oplus A$" component from the local effect; instead it is incorporated in the global effect for L.

To express the correctness of the analysis we need to keep track of the relationship between the process identifiers and the associated labels. So let *penv* be a mapping from process identifiers to elements L_a of **LabSet**. We shall say that *penv* respects the derivation sequence $PB \Longrightarrow^{a_1}_{ps_1} \dots \Longrightarrow^{a_k}_{ps_k} PB'$ if whenever (a_i, ps_i) have the form $(\texttt{FORK}_L\ b, (pi_1, pi_2))$ then $penv(pi_2) = L$; this ensures that the newly created process (pi_2) indeed has a label (in L) as reported by the semantics.

We can now redefine the function COUNT of Section 4. Given a list $\mathcal{X}$ of pairs of actions and lists of process identifiers define

$$\mathrm{COUNT}^{penv}(\mathcal{X}) = \lambda L_a.\lambda L.(CC_{L_a}(\mathcal{X}, L), CI_{L_a}(\mathcal{X}, L), CO_{L_a}(\mathcal{X}, L), CF_{L_a}(\mathcal{X}, L))$$

$CC_{L_a}(\mathcal{X}, L)$: the number of elements of the form $(t\ \texttt{CHAN}_L, pi)$ in $\mathcal{X}$ where $penv(pi) = L_a$,
$CI_{L_a}(\mathcal{X}, L)$: the number of elements of the form $(L'!t?L, (pi', pi))$ in $\mathcal{X}$, where $penv(pi) = L_a$,
$CO_{L_a}(\mathcal{X}, L)$: the number of elements of the form $(L!t?L', (pi, pi'))$ in $\mathcal{X}$, where $penv(pi) = L_a$, and
$CF_{L_a}(\mathcal{X}, L)$: the number of elements of the form $(\texttt{FORK}_L b, (pi, pi'))$ in $\mathcal{X}$ where $penv(pi) = L_a$.

Soundness of the analysis then amounts to:

$$benv \vdash \epsilon : [\,] \;\&\; [\,]$$

$$benv \vdash L!t : [L \mapsto (\mathrm{O},\mathrm{O},\mathrm{I},\mathrm{O})] \;\&\; [\,]$$

$$benv \vdash L?t : [L \mapsto (\mathrm{O},\mathrm{I},\mathrm{O},\mathrm{O})] \;\&\; [\,]$$

$$benv \vdash t \;\mathtt{CHAN}_L : [L \mapsto (\mathrm{I},\mathrm{O},\mathrm{O},\mathrm{O})] \;\&\; [\,]$$

$$\frac{benv \vdash b : A \;\&\; P}{benv \vdash \mathtt{FORK}_L\; b : [L \mapsto (\mathrm{O},\mathrm{O},\mathrm{O},\mathrm{I})] \;\&\; ([L \mapsto A] \oplus P)}$$

$$\frac{benv \vdash b_1 : A_1 \;\&\; P_1 \quad benv \vdash b_2 : A_2 \;\&\; P_2}{benv \vdash b_1; b_2 : A_1 \oplus A_2 \;\&\; P_1 \oplus P_2}$$

$$\frac{benv \vdash b_1 : A_1 \;\&\; P_1 \quad benv \vdash b_2 : A_2 \;\&\; P_2}{benv \vdash b_1 + b_2 : A_1 \sqcup A_2 \;\&\; P_1 \sqcup P_2}$$

$$\frac{benv[\beta \mapsto A \;\&\; P] \vdash b : A \;\&\; P}{benv \vdash \mathtt{REC}\; \beta.\, b : A \;\&\; P}$$

$$benv \vdash \beta : A \;\&\; P \qquad \text{if } benv(\beta) = A \;\&\; P$$

Table 3: Analysis for static process allocation

Theorem 6.2 *Assume that* $\emptyset \vdash b : A \;\&\; P$ *and* $[pi_0 \mapsto b] \Longrightarrow^{a_1}_{ps_1} \cdots \Longrightarrow^{a_k}_{ps_k} PB$ *and let penv be a mapping from process identifiers to elements of* **LabSet** *respecting the above derivation sequence and such that* $penv(pi_0) = L_0$*. We then have*

$$\mathcal{R}^*(COUNT^{penv}[(a_1, ps_1), \cdots, (a_k, ps_k)]) \sqsubseteq (P \oplus [L_0 \mapsto A])$$

where $\mathcal{R}^*(C)(L_a)(L) = (\mathcal{R}(c), \mathcal{R}(i), \mathcal{R}(o), \mathcal{R}(f))$ *if* $C(L_a)(L) = (c, i, o, f)$.

Note that the lefthand side of the inequality counts the number of operations for all processes whose labels is given (by L_a); hence our information is useful for static processor allocation.

To obtain an efficient implementation of the analysis it is once more profitable to generate an equation system. This is hardly any different from the approach of Section 5 except that by now there is even greater scope for decomposing the flow variables into families of flow variables over simpler value spaces.

7 Dynamic Processor Allocation

The idea behind the dynamic processor allocation is that the decision of how to place processes on processors is taken dynamically. Again we will be interested in knowing which requirements this puts on the processor but in contrast to the previous section we are only concerned with a single process rather than all processes with a given label. We shall now modify the analysis of Section 6 to associate *worst-case* information with the *process labels* rather than accumulating the total information. For each process label L_a we therefore ask the four

$$\frac{benv \vdash b : A \ \& \ P}{benv \vdash \texttt{FORK}_L\ b : [L \mapsto (\text{O}, \text{O}, \text{O}, \text{I})] \ \& \ ([L \mapsto A] \sqcup P)}$$

$$\frac{benv \vdash b_1 : A_1 \ \& \ P_1 \quad benv \vdash b_2 : A_2 \ \& \ P_2}{benv \vdash b_1; b_2 : A_1 \oplus A_2 \ \& \ P_1 \sqcup P_2}$$

$$\frac{benv[\beta \mapsto A] \vdash b : A \ \& \ P}{benv \vdash \texttt{REC}\ \beta.\ b : A \ \& \ P}$$

$$benv \vdash \beta : A \ \& \ [\,] \qquad \text{if } benv(\beta) = A$$

Table 4: Analysis for dynamic process allocation

questions of Section 4 taking the *maximum* information over all processes with label L_a: how many times are channels labelled by L created, how many times do channels labelled by L participate in input, how many times do channels labelled by L participate in output, and how many times are processes labelled by L generated?

Example 7.1 *Let us return to the* `pipe` *function of Examples 2.1 and 2.2 and suppose that we want to perform* dynamic processor allocation. *This means that all the processes labelled* π need not *reside on the same processor. The analysis should therefore estimate the* maximal *requirements of the instances of these processes as follows:*

main:	L_1:	M *channels created*	π:	L_1:	M *inputs performed*
	π:	M *processes created*		L_2:	M *outputs performed*
				L:	I *output performed*

Note that now we do record that each individual process labelled by π *actually only communicates over* L *at most once.*

Returning to the processor network of Example 6.1 we may allocate the main program on **P1** *and the remaining processes on* **P2** *and* **P3** *(and possibly* **P1** *as well): say* `f1` *and* `f3` *on* **P2** *and* `f2` *and* `id` *on* **P3**. *Facilities for multitasking are needed on* **P2** *and* **P3** *and facilities for multiplexing on all of* **P1**, **P2** *and* **P3**.

The inference system still has formulae

$$benv \vdash b : A \ \& \ P$$

where A and P are as in Section 6 and now $benv$ is as in Section 4: it does not incorporate the P component[2]. Most of the axioms and rules are as in Table 3; the modifications are listed in Table 4.

A difference from Section 6 is that now we need to keep track of the individual process identifiers. We therefore redefine the function COUNT^{penv} as follows:

[2]It could be as in Section 6 as well because we now combine P components using $\sqcup$ rather than $\oplus$.

$\mathrm{COUNT}^{penv}(\mathcal{X}) = \lambda L_a.\lambda L.((CC_{PI}(\mathcal{X},L), CI_{PI}(\mathcal{X},L), CO_{PI}(\mathcal{X},L), CF_{PI}(\mathcal{X},L))$
where $PI = penv^{-1}(L_a))$

$CC_{PI}(\mathcal{X},L)$:	the maximum over all $pi \in PI$ of the number of elements of the form $(t\ \mathbf{CHAN}_L, pi)$ in $\mathcal{X}$,
$CI_{PI}(\mathcal{X},L)$:	the maximum over all $pi \in PI$ of the number of elements of the form $(L'!t?L, (pi', pi))$ in $\mathcal{X}$,
$CO_{PI}(\mathcal{X},L)$:	the maximum over all $pi \in PI$ of the number of elements of the form $(L!t?L', (pi, pi'))$ in $\mathcal{X}$, and
$CF_{PI}(\mathcal{X},L)$:	the maximum over all $pi \in PI$ of the number of elements of the form $(\mathbf{FORK}_L b, (pi, pi'))$ in $\mathcal{X}$.

Soundness of the analysis then amounts to:

Theorem 7.2 *Assume that* $\emptyset \vdash b : A\ \&\ P$ *and* $[pi_0 \mapsto b] \Longrightarrow^{a_1}_{ps_1} \cdots \Longrightarrow^{a_k}_{ps_k} PB$ *and let penv be a mapping from process identifiers to elements of* **LabSet** *respecting the above derivation sequence and such that* $penv(pi_0) = L_0$*. We then have*

$$\mathcal{R}^*(COUNT^{penv}[(a_1, ps_1), \cdots, (a_k, ps_k)]) \sqsubseteq (P \sqcup [L_0 \mapsto A])$$

where $\mathcal{R}^*$ *is as in Theorem 6.2.*

Note that the lefthand side of the inequality gives the maximum number of operations over all processes with a given label; hence our information is useful for dynamic processor allocation.

To obtain an efficient implementation of the analysis it is once more profitable to generate an equation system and the remarks at the end of the previous section still apply.

8 Conclusion

The specifications of the analyses for static and dynamic allocation have much in common; the major difference of course being that for static processor allocation we *accumulate* the total numbers whereas for dynamic processor allocation we calculate the *maximum*; a minor difference being that for the static analysis it was crucial to let behaviour environments include the P component whereas for the dynamic analysis this was hardly of any importance.

This difference in approach is reminiscent of the difference between the formulation of MFP-style and MOP-style analyses: in the former the effects of paths (corresponding to process identifiers with the same label set) are merged along the way whereas in the latter the paths (corresponding to the process identifiers) have to be kept separate and their effects can only be merged when the propagation of effects has taken place.

Acknowledgements. We would like to thank Torben Amtoft for many interesting discussions. This research has been funded in part by the LOMAPS (ESPRIT BRA) and DART (Danish Science Research Council) projects.

References

[1] T.Amtoft, F.Nielson, H.R.Nielson: Type and behaviour reconstruction for higher-order concurrent programs. This proceedings.

[2] J.Cai, R.Paige: Program Derivation by Fixed Point Computation. *Science of Computer Programming* **11**, pp. 197–261, 1989.

[3] R. Cridlig, E.Goubault: Semantics and analysis of Linda-based languages. *Proc. Static Analysis*, Springer Lecture Notes in Computer Science **724**, 1993.

[4] C.E.McDowell: A practical algorithm for static analysis of parallel programs. *Journal of parallel and distributed computing* **6**, 1989.

[5] A.Giacalone, P.Mishra, S.Prasad: Operational and Algebraic Semantics for Facile: a Symmetric Integration of Concurrent and Functional Programming. *Proc. ICALP'90*, Springer Lecture Notes in Computer Science **443**, 1990.

[6] K.Havelund, K.G.Larsen: The Fork Calculus. *Proc. ICALP'93*, Springer Lecture Notes in Computer Science **700**, 1993.

[7] M.S.Hecht: *Flow Analysis of Computer Programs*, North-Holland, 1977.

[8] Y.-C.Hung, G.-H.Chen: Reverse reachability analysis: a new technique for deadlock detection on communicating finite state machines. *Software — Practice and Experience* **23**, 1993.

[9] S.Jagannathan, S.Week: Analysing stores and references in a parallel symbolic language. *Proc. L&FP*, 1994.

[10] M.Jourdan, D.Parigot: Techniques for Improving Grammar Flow Analysis. *Proc. ESOP'90*, Springer Lecture Notes in Computer Science **432**, pp. 240–255, 1990.

[11] N. Mercouroff: An algorithm for analysing communicating processes. *Proc. of MFPS*, Springer Lecture Notes in Computer Science **598**, 1992.

[12] F.Nielson, H.R.Nielson: From CML to Process Algebras. *Proc. CONCUR'93*, Springer Lecture Notes in Computer Science **715**, 1993.

[13] H.R.Nielson, F.Nielson: Higher-Order Concurrent Programs with Finite Communication Topology. *Proc. POPL'94*, pp. 84–97, ACM Press, 1994.

[14] F.Nielson, H.R.Nielson: Constraints for Polymorphic Behaviours for Concurrent ML. *Proc. CCL'94*, Springer Lecture Notes in Computer Science **845**, 1994.

[15] J.H.Reif, S.A.Smolka: Dataflow analysis of distributed communicating processes. *International Journal of Parallel Programs* **19**, 1990.

[16] J.R.Reppy: Concurrent ML: Design, Application and Semantics. Springer Lecture Notes in Computer Science **693**, pp. 165–198, 1993.

[17] R.Tarjan: Iterative Algorithms for Global Flow Analysis. In J.Traub (ed.), *Algorithms and Complexity*, pp. 91–102, Academic Press, 1976.

[18] B.Thomsen. Personal communication, May 1994.

Termination of order-sorted rewriting

Peter C. Ølveczky
Department of Informatics
University of Oslo
E-mail: peterol@ifi.uio.no

Abstract

We present a method for proving termination of order-sorted rewrite systems by transforming an order-sorted rewrite system into an unsorted one such that termination of the latter implies termination of the order-sorted system. The method is inspired by ideas of Gnaedig and Ganzinger and contains as special cases the method presented in [Gn92a] and the method that simply ignores sort information.

1 Introduction

Order-sorted specifications, i. e. many-sorted specifications where the set of sorts is partially ordered, has been introduced to provide for a more powerful type concept, allowing us to express partiality of functions, error handling and subtype inheritance [GM88]. In languages like OBJ3 [GW88], order-sorted specifications have an operational semantics based on *rewriting*.

One of the most important properties a rewrite system may have is *termination*, which means that no infinite computation can take place. Termination is especially important in completion procedures, and last but not least, proving termination of order-sorted rewrite systems ensures termination of programs written in languages like OBJ3.

In general it is undecidable whether a rewrite system terminates, but numerous techniques (e. g. based on the lexicographic path ordering [KL80, Der87]), have been developed that with varying degrees of success may be used to prove termination of *unsorted* rewrite systems. These techniques have been used to prove termination of order-sorted systems by simply ignoring sort information [CH93, Gan91, GKK90, Gn92b, Wal92]. However, this approach is not good enough since many order-sorted systems terminate where the corresponding unsorted system would not terminate. This is illustrated by the following example, taken from [Gn92a] where the problem of order-sorted termination was first addressed.

Example 1 This system defines a predicate is_even over the integers. The idea is to define it over the positive numbers first and to extend it to the negative numbers by using the operation opp.

sorts: $Zero$, $NzNeg$, Neg, $NzPos$, Pos, Int, $Bool$
$Zero \leq Neg \leq Int$
$Zero \leq Pos \leq Int$
$NzNeg \leq Neg$
$NzPos \leq Pos$
funcs: $0 : Zero$
$s : Pos \rightarrow NzPos$
$p : Neg \rightarrow NzNeg$
$true : Bool$
$false : Bool$
$is\_even : Int \rightarrow Bool$
$opp : NzNeg \rightarrow NzPos$
rules: $is\_even(0) \longrightarrow true$
$is\_even(s(0)) \longrightarrow false$
$(\forall x : Pos)\ is\_even(s(s(x))) \longrightarrow is\_even(x)$
$(\forall y : NzNeg)\ is\_even(y) \longrightarrow is\_even(opp(y))$
$opp(p(0)) \longrightarrow s(0)$
$(\forall y : NzNeg)\ opp(p(y)) \longrightarrow s(opp(y))$

The system is terminating, the normal form of any ground term $is\_even(t)$ being either $true$ or $false$. But if sorts were ignored, the system would not terminate since the fourth rule would generate the infinite rewrite sequence

$$is\_even(y) \longrightarrow is\_even(opp(y)) is\_even(opp(opp(y))) \longrightarrow \cdots$$

However, the system does terminate because $opp(y : NzNeg)$ is of sort $NzPos$, so y cannot be instantiated with $opp(t)$ for any term t.

In Ganzinger [Gan91], where an order-sorted rewrite system is transformed into a many-sorted equational rewrite system, each function declaration $f : w \rightarrow s$ induces a (many-sorted) function symbol $f_{w \rightarrow s}$. This idea was adopted by Gnaedig [Gn92a] where these labelled function symbols are used to prove termination of order-sorted rewrite systems where the signature is *minimal*, i. e. if $f : w \rightarrow s$ and $f : w' \rightarrow s'$ are two function declarations then w and w' must be incomparable with respect to $\leq$. Termination can then be proven if $l' \succ_{lpo} r'$ for every rule $l \longrightarrow r$, t' being t where each function symbol f is replaced with the "corresponding" symbol $f_{w \rightarrow s}$. Unfortunately, not many signatures are minimal, and even for those that are minimal, this approach is not very strong (e. g. this method cannot show termination of the above system even though the signature is minimal).

We will also pursue the idea of labelling each function symbol in a term, but for our purpose the best result is obtained when a function symbol is labelled with the smallest sorts of its arguments. In the previous example the fourth rule would be labelled

$$is\_even_{NzNeg}(y) \longrightarrow is\_even_{NzPos}(opp_{NzNeg}(y))$$

The symbols is_even_{NzNeg} and is_even_{NzPos} may then be treated as distinct symbols in e. g. the lexicographic path ordering making the inequality

$$is\_even_{NzNeg}(y) \succ_{lpo} is\_even_{NzPos}(opp_{NzNeg}(y))$$

valid with precedence $is\_even_{NzNeg} \succ_F is\_even_{NzPos}$ and $is\_even_{NzNeg} \succ_F opp_{NzNeg}$.

We will use such labelling to transform an order-sorted rewrite system $\mathcal{R}$ into an unsorted rewrite system R' such that $\mathcal{R}$ terminates whenever R' does. We can thus use unsorted termination orderings to prove that R', and hence $\mathcal{R}$ terminates. However, special features of termination orderings imply that the best result after applying such an ordering is not guaranteed when the transformation R' is the most accurate. We will keep this in mind and will therefore exemplify our method with the lexicographic path ordering.

This paper is an extended abstract of [Ølv94] where a more thorough investigation on the subject of order-sorted termination as well as the correctness proofs for claims in this paper can be found.

2 Basic notions

2.1 Order-sorted rewriting

We will only introduce the syntactic aspects of order-sorted algebra and refer to [GM88] and [Wal92] for semantics of order-sorted specifications. In this paper, notions and notations mainly follow [GM88] and [GKK90].

Order-sorted algebra is based on a subsort relation, a partial order $\leq$ on a set S of sorts. We write $s' < s$ when $s' \leq s$ and $s' \neq s$ and $\geq$ for the symmetric relation of $\leq$. The subsort relation $\leq$ can be extended to n-tuples of sorts (often denoted w) where $w = \langle s_1, \ldots, s_n \rangle \leq \langle s'_1, \ldots, s'_1 \rangle = w'$ iff $s_i \leq s'_i$ for $i = 1..n$ and $w < w'$ iff $w \leq w'$ and $w \neq w'$. We write $w \lessdot w'$ when $w < w'$ and w and w' only differs in ith position, i. e. $w = \langle s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n \rangle$, $w' = \langle s_1, \ldots, s_{i-1}, s'_i, s_{i+1}, \ldots, s_n \rangle$ and $s_i < s'_i$.

A function declaration has the form $f : s_1, \ldots, s_n \rightarrow s$ where each s_i is a sort symbol. If $n = 0$ then f is called a constant and the declaration is written $f : s$.

An order-sorted signature $\langle S, \leq, \Sigma \rangle$, usually denoted just Σ, consists of a partially ordered sort set $\langle S, \leq \rangle$ and a set Σ of function declarations. In this paper, all signatures are assumed to be finite, i. e. to contain a finite number of sorts and declarations.

In the rest of this paper we assume that $\langle S, \leq, \Sigma \rangle$ is an order-sorted signature. Given a set $X = \{X_s \mid s \in S\}$ of disjoint sets, the set of terms of sort s, denoted $\mathcal{T}(\Sigma, X)_s$, is the least set such that $X_{s'} \subseteq \mathcal{T}(\Sigma, X)_s$ if $s' \leq s$, and $f(s_1, \ldots, s_n) \in \mathcal{T}(\Sigma, X)_s$ if there exists a declaration $f : s_1, \ldots, s_n \rightarrow s'$ such that t_i is a term of sort s_i for $i = 1..n$ and $s' \leq s$. The set $\mathcal{T}(\Sigma, X)$ is defined as $\mathcal{T}(\Sigma, X) = \{\mathcal{T}(\Sigma, X)_s \mid s \in S\}$ and an element of any set $\mathcal{T}(\Sigma, X)_s$ is called a (well-formed) term. Given a set X of variables that contains an infinite number of variables of each sort, we will assume that every term t in $\mathcal{T}(\Sigma, X)$ has a unique (and computable) least sort, denoted $LS(t)$. A ground term is a term that does not contain any variables, and we will assume that there exists a ground term of every sort s in the signature.

A term may be viewed as an ordered tree in the obvious way. A *position* in a term can be seen as a finite sequence of natural numbers, identifying a path from the root of the term. The node in the term where the path ends is the actual position. The subterm of t at position p, written $t|_p$, is the subterm of t which has its root symbol at position p. The result of replacing the subterm of t at position p by the term u is written $t[u]_p$. If p and q are positions we write $p.q$ to denote the concatenation of p and q.

A *substitution* $\sigma = \{\sigma_s : X_s \rightarrow \mathcal{T}(\Sigma, Y)_s \mid s \in S\}$ is a mapping from variables to terms. We shall use lowercase Greek letters to denote substitutions. The application of a substitution σ to a term t consists of simultaneously replacing the variables occurring in t by a term according to the mapping. The result of this application is written $t\sigma$.

A *specialization*, often written μ, is a substitution $\mu : X \rightarrow Y$ that maps a variable x of sort s to a variable x of sort s' less than or equal to s, i. e. $(x\!:\!s)\mu = x\!:\!s'$ where $s' \leq s$. In a finite signature, a finite set of variables has a finite set of specializations.

An order-sorted *rewrite rule* is a directed equation $(\forall X)\ \ l \longrightarrow r$ where l and r are terms in $\mathcal{T}(\Sigma, X)$. A major difference between unsorted and order-sorted rewriting lies in the fact that a term t cannot always be rewritten if the left-hand-side of a rule matches a subterm of t. The reason for this is either that rewriting could result in a non-well-formed term or because of the definition of congruence in order-sorted algebras where $f(a) \approx f(b)$ may not be valid even though $a \approx b$ and both $f(a)$ and $f(b)$ are well-formed terms. Therefore, given a *rewrite system*, i. e. a set of rewrite rules, $\mathcal{R}$, a term $t \in \mathcal{T}(\Sigma, X)$ *rewrites* to t' with a rule $(\forall Y)\ l \longrightarrow r$ in $\mathcal{R}$ at position p, denoted $t \xrightarrow[\mathcal{R}]{} t'$ if

1. there is a substitution $\sigma : Y \rightarrow \mathcal{T}(\Sigma, X)$ such that $l\sigma = t|_p$ and $t' = t[r\sigma]_p$ and
2. there is a sort s such that, for x a variable of sort s, $t[x]_p$ is a well-formed term and $l\sigma, r\sigma \in \mathcal{T}(\Sigma, X)_s$.

We write $t \xrightarrow[\mathcal{R}]{+} t'$ when t rewrites to t' in one or more steps. We will assume that every rule $l \longrightarrow r$ is *sort-decreasing*, i. e. $LS(l\sigma) \geq LS(r\sigma)$ for every substitution σ. Sort-decreasingness ensures 1) that condition 2 in the above definition of rewriting is satisfied when the first condition is fulfilled, 2) that the critical pair lemma is valid and 3) that Birkhoff's theorem is valid for confluent systems [GKK90]. Sort-decreasingness may however be too restrictive for many purposes; we will therefore indicate how the method presented can be extended to handle non-sort-decreasing systems as well.

A rewrite system $\mathcal{R}$ is *terminating* if there does not exist any infinite rewrite sequence

$$t_1 \xrightarrow[\mathcal{R}]{} t_2 \xrightarrow[\mathcal{R}]{} t_3 \xrightarrow[\mathcal{R}]{} \cdots$$

2.2 Termination of unsorted rewriting

Unsorted rewriting is the special case of order-sorted rewriting where the signature has only one sort. Since an (unsorted) rewrite system can model a Turing machine, termination of rewriting is in general undecidable (see for instance [Der87]). Many techniques, mostly so-called *reduction* or *termination orderings*, have been developed that sometimes may be used to prove termination of unsorted rewrite systems.

A *quasi-ordering* $\succsim$ on a set S is a reflexive and transitive relation on S. The symmetric relation to $\succsim$ is denoted $\precsim$. The strict part $\succ$ of $\succsim$ and the induced equivalence relation $\approx$ are defined $\succ = (\succsim \setminus \precsim)$ and $\approx = (\succsim \cap \precsim)$ respectively.

We will use the *lexicographic path ordering* to exemplify our method. Assume that F is a set of function symbols and X is a set of variables. Let $\succsim_F$ be a quasi-ordering on $F \cup X$ such that a variable (seen as a constant in the ordering) is incomparable in $\succ_F$ with any other symbol. The lexicographic ordering $\succ_{lpo}$ on the set of $T(F, X)$ of terms is then defined as follows:

$$\begin{array}{lrl} & \multicolumn{2}{c}{t = f(t_1, \ldots, t_m) \succ_{lpo} g(t'_1, \ldots, t'_n) = t'} \\ & \multicolumn{2}{c}{\text{iff}} \\ \text{either} & & t_i \succ_{lpo} t' \quad \text{or} \quad t_i \approx_{lpo} t' \\ \text{or} & f \succ_F g & \text{and} \quad t \succ_{lpo} t'_i \text{ for all } i \in 1..n \\ \text{or} & f \approx_F g & \text{and} \quad \langle t_1, \ldots, t_m \rangle \succ_{lpo}^{lex} \langle t'_1, \ldots, t'_n \rangle \\ & & \text{and} \quad t \succ_{lpo} t'_i \quad \text{for} \quad i \geq 2 \end{array}$$

where $t \approx_{lpo} t'$ if t and t' are equal up to replacement of $\approx_F$ -equal symbols.

A finite rewrite system R is terminating if $l \succ_{lpo} r$ for each rule $l \longrightarrow r$ in R.

The reader is referred to [Der87] for an introduction to unsorted termination.

3 Termination of order-sorted rewriting

In this section we propose a technique for proving termination of an order-sorted rewrite system $\mathcal{R}$ by transforming it into an unsorted system R'.

Definition 1 *Let Σ be an order-sorted signature. Its* translation *into a corresponding unsorted signature (i. e. a set of function symbols) F is defined*

$$F = \{f_w \mid f : w' \rightarrow s \text{ declaration in } \Sigma \text{ for any } s \text{ and } w \leq w'\}$$

A term t in $\mathcal{T}(\Sigma, X)$ is represented in $T(F, X)$ by the term $\lfloor t \rfloor$ where each function symbol in t is labelled with the least sorts of its arguments:

Definition 2 *The mapping $\lfloor\ \rfloor : \mathcal{T}(\Sigma, X) \rightarrow T(F, X)$ where F is the translation of Σ is defined inductively as follows:*

- $\lfloor x : s \rfloor = x$ *for a variable* $x \in X_s$.
- $\lfloor f(t_1, \ldots, t_n) \rfloor = f_{LS(t_1),\ldots,LS(t_n)}(\lfloor t_1 \rfloor, \ldots, \lfloor t_n \rfloor)$

We will use $\lfloor\ \rfloor$ to make a *reflection* of an order-sorted rewrite system $\mathcal{R}$:

Definition 3 *An unsorted rewrite system R'* reflects *an order-sorted rewrite system $\mathcal{R}$ iff*

$$t \xrightarrow[\mathcal{R}]{} t' \Longrightarrow \lfloor t \rfloor \xrightarrow[R']{+} \lfloor t' \rfloor$$

for all terms $t, t' \in \mathcal{T}(\Sigma, X)$.

Clearly, $\mathcal{R}$ terminates whenever its reflection does.

Let us now define R'. Letting $R' = \{\lfloor l \rfloor \longrightarrow \lfloor r \rfloor \mid l \longrightarrow r \in \mathcal{R}\}$ is not sufficient, R' must at least contain a rule $\lfloor l\mu \rfloor \longrightarrow \lfloor r\mu \rfloor$ for every specialization μ of each rule $l \longrightarrow r$ in $\mathcal{R}$. Unfortunately, the following example shows that not even this definition is sufficient to make R' a reflection of an order-sorted system $\mathcal{R}$.

Example 2 Let a rewrite system $\mathcal{R}$ be given

sorts: s, s_1, s_2
$s_1 \leq s$
$s_2 \leq s$
funcs: $f : s \rightarrow s$
$g : s \rightarrow s$
$g : s_1 \rightarrow s_2$
$a : s$
$b : s_1$
rule: $a \longrightarrow b$

The term $f(g(a))$ rewrites to $f(g(b))$ in $\mathcal{R}$ but $\lfloor f(g(a)) \rfloor = f_s(g_s(a))$ and $\lfloor f(g(b)) \rfloor = f_{s_2}(g_{s_1}(b))$ so we need

$$f_s(g_s(a)) \xrightarrow[R']{+} f_{s_2}(g_{s_1}(b))$$

which is not true if R' only contains the rule $a \longrightarrow b$.

In the above example all the (unsorted) function symbols have changed even though the order-sorted rewriting occurred at position 1.1. We call such a "side effect" of a rewrite step a *function symbol jump* or just a *F-jump*:

Definition 4 *Assume that a term t is rewritten to t' in an order-sorted rewrite system and that the rule is applied at position p in t. There is a* function symbol jump*, often just called an F*-jump*, from f_w to $f_{w'}$ if there is a position q in t such that $\lfloor t \rfloor|_q = f_w(\ldots)$ and $\lfloor t' \rfloor|_q = f_{w'}(\ldots)$ and there is no position r such that $q = p.r$*

In the previous example we had two F-jumps, from f_s to f_{s_2} and from g_s to g_{s_1} respectively. Sort-decreasingness and monotonicity properties of order-sorted signatures imply that $w' \lessdot w$ when an F-jump from f_w to $f_{w'}$ occurs. To handle this, a so-called *extra rule*

$$f_w(x_1, \ldots, x_n) \longrightarrow f_{w'}(x_1, \ldots, x_n)$$

is added to R' for all symbols $f_w, f_{w'}$ in F when $w' \lessdot w$. In example 2, R' would have contained the extra rules $f_s(x) \longrightarrow f_{s_1}(x)$, $f_s(x) \longrightarrow f_{s_2}(x)$, $g_s(x) \longrightarrow g_{s_1}(x)$ and $g_s(x) \longrightarrow g_{s_2}(x)$. Together with a rule $a \longrightarrow b$, R' is a reflection of $\mathcal{R}$ where the rewrite sequence $f_s(g_s(a)) \xrightarrow[R']{} f_s(g_s(b)) \xrightarrow[R']{} f_s(g_{s_1}(b)) \xrightarrow[R']{} f_{s_2}(g_{s_1}(b))$ is the desired $\lfloor f(g(a)) \rfloor \xrightarrow[R']{+} \lfloor f(g(b)) \rfloor$.

A reflection R' can then be defined

$$\begin{aligned} R' \;=\; & \{ \lfloor l\mu \rfloor \longrightarrow \lfloor r\mu \rfloor \mid l \longrightarrow r \in \mathcal{R},\ \mu \text{ specialization } \} \bigcup \\ & \{ f_w(x_1, \ldots, x_n) \longrightarrow f_{w'}(x_1, \ldots, x_n) \mid f_w \in F, w' \lessdot w \} \end{aligned}$$

Using the lexicographic path ordering, the extra rules are decreasing iff $f_w \succ_F f_{w'}$ in the precedence when $w' \lessdot w$. The relation $\succ_F$ being transitive, this means that $f_w \succ_F f_{w'}$ for all w, w' where $w' < w$. However, we can allow a precedence $f_w \approx_F f_{w'}$ by replacing each function symbol f_w in R' with a canonical element of the class $[f_w]_{\approx_F}$ with respect to f and omitting the obvious extra rules.

The first version of our method is thus described by the following theorem:

Theorem 5 *A finite, sort-decreasing order-sorted rewrite system $\mathcal{R}$ terminates if*

$$\lfloor l\mu \rfloor \succ_{lpo} \lfloor r\mu \rfloor$$

for each rule $l \longrightarrow r$ in $\mathcal{R}$ and each specialization μ of the variables in the rule, provided that the precedence $\succsim_F$ satisfies

$$f_w \succsim_F f_{w'}$$

for all symbols $f_w, f_{w'} \in F$ where $w' < w$.

This method, which can be applied to other termination orderings in the same way, can be totally automated and contains as special cases the method published in [Gn92a] and the method that ignores sort information (by letting $f_w \approx_F f_{w'}$ in the precedence for all function symbols f in Σ and labels w and w'). The main disadvantages of the method are:

1. Loss in efficiency, even though investigating each specialization seems inherent to most problems in order-sorted rewriting.

2. R' contains a rule $f_w(x_1, \ldots, x_n) \longrightarrow f_{w'}(x_1, \ldots, x_n)$ whenever $w' \lessdot w$ even in systems where an F-jump from f_w to $f_{w'}$ cannot occur. If, for instance, all specializations of all rules are sort-preserving, *no* F-jump can take place, and therefore all restrictions on the precedence $\succsim_F$ are superfluous. This problem is addressed in section 4.2.

3. A possible F-jump form f_w to $f_{w'}$ can be handled in more elaborate ways than just adding a rule $f_w(x_1, \ldots, x_n) \longrightarrow f_{w'}(x_1, \ldots, x_n)$. This opportunity is dealt with in the next section.

4. The method requires sort-decreasing rules. In section 4.2 we will indicate how it can be extended to non-sort-decreasing systems as well.

An example

Let us see how the method can be applied to the following system, which cannot be shown to terminate by ignoring sort information:

Example 3 Let a rewrite system $\mathcal{R}$ be

sorts: s', s
$s' \leq s$
funcs: $f : s \to s$
$g : s \to s$
$a : s$
$b : s'$
rules: $(\forall x : s')\ f(x) \to g(x)$
$g(a) \to f(a)$

Since $s' < s$ the precedence $\succsim_F$ must satisfy $g_s \succsim_F g_{s'}$ and $f_s \succsim_F f_{s'}$ and

$$\begin{array}{rcl} \lfloor f(x:s') \rfloor = f_{s'}(x) & \succ_{lpo} & g_{s'}(x) = \lfloor g(x) \rfloor \\ \lfloor g(a) \rfloor = g_s(a) & \succ_{lpo} & f_s(a) = \lfloor f(a) \rfloor \end{array}$$

which are all satisfied with the precedence

$$g_s \succ_F f_s \succ_F f_{s'} \succ_F g_{s'}$$

4 Extensions of the method

4.1 A more elaborate treatment of possible F-jumps

A possible F-jump from f_w to $f_{w'}$ is handled with an extra rule $f_w(x_1, \ldots, x_n) \longrightarrow f_{w'}(x_1, \ldots, x_n)$ in R' (or by having $f_w \approx_F f_{w'}$), forcing the precedence to satisfy $f_w \succsim_F f_{w'}$. The following example shows a weakness of this approach:

Example 4 Let the signature Σ and the terminating rewrite system $\mathcal{R}$ be given

sorts: s', s
$s' \leq s$
funcs: $f : s \to s$
$g : s' \to s$
$h : s \to s$
$h : s' \to s'$
$a : s$
$b : s'$
rules: $(\forall x : s')\ f(x) \longrightarrow f(g(x))$
$a \longrightarrow b$

To have the desired inequality

$$\lfloor f(x:s')\rfloor = f_{s'}(x) \succ_{lpo} f_s(g_{s'}(x)) = \lfloor f(g(x))\rfloor$$

we need $f_{s'} \succ_F f_s$, but our method requires $f_s \gtrsim_F f_{s'}$ since $s' \leq s$. Note also that an F-jump from f_s to $f_{s'}$ may occur because of the second rule.

We may have an F-jump from $f_{s_1,..,s_i,..,s_n}$ to $f_{s_1,..,s'_i,..,s_n}$ if f is "applied" to an instance $l\sigma \longrightarrow r\sigma$ of a rule where $LS(l\sigma) = s_i$ and $LS(r\sigma) = s'_i$ or if there is a function symbol g in Σ such that $LS(g(\ldots l\sigma \ldots)) = s_i$ and $LS(g(\ldots r\sigma \ldots)) = s'_i$. In example 4, we have F-jumps from f_s to $f_{s'}$ because f is "applied" to the second rule ($f(a) \xrightarrow[\mathcal{R}]{} f(b)$) and because h "changes" value sort (e. g. $f(h(a)) \xrightarrow[\mathcal{R}]{} f(h(b))$, $f(h(h(a))) \xrightarrow[\mathcal{R}]{} f(h(h(b)))$, ...).

An extra rule

$$f_{s_1,..,s_i,..,s_n}(x_1, \ldots, x_i, \ldots, x_n) \longrightarrow f_{s_1,..,s'_i,..,s_n}(x_1, \ldots, x_i, \ldots, x_n)$$

may therefore be replaced by a rule

$$f_{s_1,..,s_i,..,s_n}(x_1, \ldots, \lfloor l\mu \rfloor, \ldots, x_n) \longrightarrow f_{s_1,..,s'_i,..,s_n}(x_1, \ldots, \lfloor r\mu \rfloor, \ldots, x_n)$$

for every rule specialization $l\mu \longrightarrow r\mu$ where $LS(l\mu) = s_i$ and $LS(r\mu) = s'_i$ and with a rule

$$f_{s_1,..,s_i,..,s_n}(x_1, \ldots, g_w(y_1, \ldots, y_m), \ldots, x_n) \longrightarrow f_{s_1,..,s'_i,..,s_n}(x_1, \ldots, g_{w'}(y_1, \ldots, y_m), \ldots, x_n)$$

for each pair $g_w, g_{w'}$ of function symbols in F where $w' \lessdot w$, $LS(g(\overline{x}:w)) = s_i$ and $LS(g(\overline{y}:w')) = s'_i$.

Unfortunately, we do not have

$$f_{s_1,..,s_i,..,s_n}(x_1, \ldots, \lfloor l\mu \rfloor, \ldots, x_n) \succ_{lpo} f_{s_1,..,s'_i,..,s_n}(x_1, \ldots, \lfloor r\mu \rfloor, \ldots, x_n)$$

or

$$f_{s_1,..,s_i,..,s_n}(x_1, \ldots, g_w(y_1, \ldots, y_m), \ldots, x_n) \succ_{lpo} f_{s_1,..,s'_i,..,s_n}(x_1, \ldots, g_{w'}(y_1, \ldots, y_m), \ldots, x_n)$$

if the arity of f is greater than 1 and $f_{s_1,..,s_i,..,s_n} \not\gtrsim_F f_{s_1,..,s'_i,..,s_n}$. Using any path ordering, all function symbols that violate the restriction $w' < w \Longrightarrow f_w \gtrsim_F f_{w'}$ must be unary. Other orderings, such as the Knuth-Bendix-ordering [KB70], do not have this restriction. For the lexicographic path ordering, this method can be described by the following theorem:

Theorem 6 *Assume that Σ is a finite order-sorted signature and $\mathcal{R}$ a finite set of sort-decreasing rewrite rules. $\mathcal{R}$ terminates if the following conditions are satisfied:*

1. *For any non-unary function symbol g in Σ*

$$g_w \in F \wedge w' < w \Longrightarrow g_w \succsim_F g_{w'}$$

2. *For every rule $l \longrightarrow r$ in $\mathcal{R}$ and every specialization μ*

$$\lfloor l\mu \rfloor \succ_{lpo} \lfloor r\mu \rfloor$$

3. *For any pair $f_s, f_{s'}$ in F where $s' < s$ and $f_s \not\succsim_F f_{s'}$ we have*

   - *$f_s(\lfloor l\mu \rfloor) \succ_{lpo} f_{s'}(\lfloor r\mu \rfloor)$, that is $\lfloor l\mu \rfloor \succ_{lpo} f_{s'}(\lfloor r\mu \rfloor)$, for every rule specialization $l\mu \longrightarrow r\mu$ where $LS(l\mu) = s$ and $LS(r\mu) = s'$.*
   - *$f_s(g_w(x_1, \ldots, x_n)) \succ_{lpo} f_{s'}(g_{w'}(x_1, \ldots, x_n))$, that is $g_w \succ_F f_{s'}$ and $g_w \succ_F g_{w'}$, for each pair $g_w, g_{w'}$ in F where $w' \lessdot w$, $LS(g(\overline{x} : w)) = s$ and $LS(g(\overline{y} : w')) = s'$.*

Although the method becomes quite complicated with this version it can be fully automated and can be used to prove termination of the system in example 4.

Example 4 (cont.) We need $f_{s'} \succ_F f_s$ to obtain the inequality $f_{s'}(x) \succ_{lpo} f_s(g_s(x))$. Therefore, the inequalities $f_s(a) \succ_{lpo} f_{s'}(b)$ and $f_s(h_s(x)) \succ_{lpo} f_{s'}(h_{s'}(x))$ must also hold. The precedence

$$a \succsim_F h_s \succ_F f_{s'} \succ_F f_s \succsim_F g_s \succsim_F g_{s'} \succsim_F b \succsim_F h_{s'}$$

satisfies all the requirements of method 2 and hence termination of the system is established.

The process described above can then be repeated such that an extra rule

$$f_s(g_w(x)) \longrightarrow f_{s'}(g_{s'}(x))$$

in R' is replaced by a set of rules

$$f_s(g_w(\lfloor l\mu \rfloor)) \longrightarrow f_{s'}(g_{s'}(\lfloor r\mu \rfloor))$$

and a set of rules

$$f_s(g_w(h_{s_1}(x))) \longrightarrow f_{s'}(g_{s'}(h_{s_2}(x)))$$

for all necessary rule specializations and pairs of F-symbols. Then, the rule $f_s(g_w(h_{s_1}(x))) \longrightarrow f_{s'}(g_{s'}(h_{s_2}(x)))$ may be substituted in the same way, and this process can continue. However, the process may not terminate. Assume for instance that the system in example 4 is extended and that we need a precedence $h_{s'} \succ_F h_s$. Then, a (possible) extra rule $h_s(x) \longrightarrow h_{s'}(x)$ is replaced by $h_s(a) \longrightarrow h_{s'}(b)$ and $h_s(h_s(x)) \longrightarrow h_{s'}(h_{s'}(x))$. The latter is not decreasing in the ordering and is therefore replaced by $h_s(h_s(a)) \longrightarrow h_{s'}(h_{s'}(b))$ and $h_s(h_s(h_s(x))) \longrightarrow h_{s'}(h_{s'}(h_{s'}(x)))$ which is replaced by ... and so on.

It may happen (at least for the path orderings) that the number of iterations necessary to prove termination is bounded by the depth of the terms in the rules in $\mathcal{R}$.

4.2 Possible F-jumps

Until now, we have assumed (more or less) that an F-jump from $f_{s_1,..,s_i,..,s_n}$ to $f_{s_1,..,s'_i,..,s_n}$ can occur whenever $s'_i < s_i$ and have concentrated on how to handle such a possible jump. However, in some systems an F-jump from $f_{s_1,..,s_i,..,s_n}$ to $f_{s_1,..,s'_i,..,s_n}$ may not occur. Our method can be strengthened by computing the set of F-jumps that actually can take place. A more thorough analysis also has the advantage that the proposed method can be used on non-sort-decreasing systems as well, which is exemplified by the following system:

Example 5 Let a terminating rewrite system be given

sorts: s, s_1, s_2, s_3, s_4
$s_1 \leq s$
$s_2 \leq s$
$s_3 \leq s$
$s_4 \leq s$
funcs: $f : s \rightarrow s$
$g : s \rightarrow s$
$g : s_3 \rightarrow s_1$
$g : s_4 \rightarrow s_2$
$h : s_1 \rightarrow s_2$
$a : s_3$
$b : s_4$
rules: $a \longrightarrow b$
$(\forall x : s_1)\ f(x) \longrightarrow f(h(x))$

The second rule implies that an unsorted analysis cannot prove termination of the system which is non-sort-decreasing by the first rule. An analysis of the system gives the set of possible F-jumps as follows:

g_{s_3} to g_{s_4}	$(g(a) \xrightarrow[\mathcal{R}]{} g(b))$
g_{s_1} to g_{s_2}	$(g(g(a)) \xrightarrow[\mathcal{R}]{} g(g(b)))$
f_{s_3} to f_{s_4}	$(f(a) \xrightarrow[\mathcal{R}]{} f(b))$
f_{s_1} to f_{s_2}	$(f(g(a)) \xrightarrow[\mathcal{R}]{} f(g(b)))$

In a first approach, each possible F-jump from f_{w_1} to f_{w_2} imply a restriction $f_{w_1} \succsim_F f_{w_2}$ using the lexicographic path ordering. In this example we would have the restrictions $g_{s_3} \succsim_F g_{s_4}$, $g_{s_1} \succsim_F g_{s_2}$, $f_{s_3} \succsim_F f_{s_4}$, and $f_{s_1} \succsim_F f_{s_2}$ which combined with $f_{s_1} \succ_F f_{s_2}$, $f_{s_1} \succ_F h_{s_1}$ and $a \succ_F b$ proves termination of the system.

A complete analysis with respect to possible F-jumps is quite tedious but it *is* decidable whether a F-jump from f_{w_1} to f_{w_2} may take place. Any possible F-jump can then be handled in the way described in the previous section.

4.3 Other improvements

The method presented here can be improved in many ways, e. g. in finding more suitable orderings which tackle the specific problems with the method, making the method more efficient. Another improvement, also mentioned in [Gn92a], is to "preprocess" the rewrite system *before* applying the technique presented here. For instance, a rule $(\forall x:s)\ l\longrightarrow r$ can be replaced by two rules $(\forall x:s_1)\ l\longrightarrow r$ and $(\forall x:s_2)\ l\longrightarrow r$ provided that there is no ground term of sort s which is not a term of either sort s_1 or sort s_2.

5 Conclusion

The method presented is an extension of the works of Gnaedig [Gn92a] and Ganzinger [Gan91]. The latter is not concerned with termination when transforming an order-sorted system into a many-sorted one for completion purposes, but introduces the idea of labelling function symbols and adding equations/rules to handle F-jumps. Gnaedig [Gn92a] requires, in addition to sort-decreasingness, that the signatures are minimal. That approach has the advantage of being more efficient, but our method has been shown to be significantly stronger.

The technique presented is only one of many approaches to the problem of proving termination of order-sorted rewriting. A totally different idea we have been looking at, is to handle variables in better ways in the termination orderings, always achieving the crucial $t \succ t' \Longrightarrow t\sigma \succ t'\sigma$. In unsorted orderings, a variable is not comparable to any term not containing the same variable. This restriction can sometimes be loosened by letting a variable of sort s be comparable to some terms that are not of sort s. In [Ølv94] we propose an extension to the lexicographic path ordering based on this principle, thereby being able to prove termination of a system which cannot be shown to terminate with the method given in this paper.

Acknowledgements: I would like to thank Olav Lysne for comments on the full version [Ølv94] of this paper.

References

[CH93] H. Chen and J. Hsiang. *Order-Sorted Equational Specification and Completion.* Unpublished note, Department of Computer Science, State University of New York at Stony Brook, 1993.

[Der82] N. Dershowitz. *Orderings for term-rewriting systems.* Theoretical Computer Science 17(3):279-301, 1982.

[Der87] N. Dershowitz. *Termination of rewriting.* Journal of Symbolic Computation, 3:69-116, 1987.

[DJ90] N. Dershowitz and J.P. Jouannaud. *Rewrite systems.* Handbook of Theoretical Computer Science, chapter 15, North-Holland, 1990.

[Gan91] H. Ganzinger. *Order-sorted completion: the many-sorted way.* Theoretical Computer Science 89(1991) 3-32.

[GKK90] I. Gnaedig, C. Kirchner and H. Kirchner. *Equational Completion in Order-Sorted Algebras.* Theoretical Computer Science 72:169-202, 1990.

[GM88] J. A. Goguen and J. Meseguer. *Order-Sorted Algebra I.* Center for the Study of Language and Information, Stanford University, 1988.

[Gn92a] I. Gnaedig. *Termination of order-sorted rewriting.* Proc. Algebraic and Logic Programming. Third International Conference, Springer-Verlag, 1992.

[Gn92b] I. Gnaedig. *ELIOS-OBJ: theorem proving in a specification language.* Proc. ESOP '92, 4th European Symposion on Programming, Springer-Verlag, 1992.

[GW88] J. A. Goguen and T. Winkler. *Introducing OBJ3.* Tech. report, SRI International, Computer Science Lab., 1988.

[KB70] D. Knuth and P. Bendix. *Simple word problems in universal algebras.* J. Leech, editor, Computational Problems in Abstract Algebra, pages 263-297, Pergamon Press, Oxford, 1970.

[KL80] S. Kamin and J.-J. Lévy. *Two generalizations of the recursive path ordering.* Unpublished note, Dept. of Computer Science, University of Illinois, Urbana, IL.

[Wal92] U. Waldmann. *Semantics of order-sorted specifications.* Theoretical Computer Science 94(1992) 1-35.

[Ølv94] P. C. Ølveczky. *Terminering av typeordnet omskrivning.* Masters thesis, Dept. of Informatics, University of Oslo, 1994. In Norwegian.

A Type System Equivalent to Flow Analysis*

Jens Palsberg† Patrick O'Keefe‡

Abstract

Flow-based safety analysis of higher-order languages has been studied by Shivers, and Palsberg and Schwartzbach. Open until now is the problem of finding a type system that accepts exactly the same programs as safety analysis.

In this paper we prove that Amadio and Cardelli's type system with subtyping and recursive types accepts the same programs as a certain safety analysis. The proof involves mappings from types to flow information and back. As a result, we obtain an inference algorithm for the type system, thereby solving an open problem.

1 Introduction

1.1 Background

Many program analyses for higher-order languages are based on *flow analysis*, also known as *closure analysis*. Examples include various analyses in the Standard ML of New Jersey compiler [3], and the binding-time analyses for Scheme in the partial evaluators Schism [5] and Similix [4]. Such analyses have the advantage that they can be applied to *untyped* languages. This is in contrast to more traditional abstract interpretations which use types when defining the abstract domains.

Recently, it has become popular to define program analyses for typed languages by annotating the types with information about program behavior [8, 2]. This has lead to clear specifications of a range of analyses, and often such an analysis can be efficiently computed by a straightforward extension of a known type inference algorithm.

*Extended abstract of a paper in Proc. POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages.

†Computer Science Department, Aarhus University, DK-8000 Aarhus C, Denmark. E-mail: palsberg@daimi.aau.dk.

‡151 Coolidge Avenue #211, Watertown, MA 02172, USA. E-mail: pmo@world.std.com.

The precision of a type-based analysis depends on the expressiveness of the underlying type system. Similarly, the precision of a flow-based analysis depends on the expressiveness of the underlying flow analysis. In this paper we address an instance of the following fundamental question:

> **Fundamental question.** What type-based analysis computes the same information as a given flow-based analysis?

We consider the case of flow-based *safety* analysis, that is, an analysis which collects type information from for example constants and applications of primitive operations. Such an analysis was first presented in 1991 by Shivers [13] who called it type *recovery*. Later, Palsberg and Schwartzbach [11, 10] proved that on the basis of the collected information, one can define a predicate which accepts only programs which cannot go wrong. They called this *safety* analysis. They also proved that their safety analysis accepts more programs than simple type inference.

In this paper, we consider the following instance of the above question:

> Which type system accepts the same programs as safety analysis?

The particular safety analysis we consider is defined in Section 3.

Our thesis is that the type system that answers this specific question will in many cases also be the answer to the fundamental question.

1.2 Our result

We prove that a natural type system with subtyping and recursive types accepts the same programs as safety analysis. The proof involves mappings from types to flow information and back.

The type system has been studied by Amadio and Cardelli [1], and an $O(n^2)$ algorithm for deciding the subtyping relation has been presented by Kozen, Palsberg, Schwartzbach, [6]. Open until now is the question of type inference. As a corollary of our result we get a type inference algorithm which works by first doing safety analysis and then mapping the flow information to types.

The set of types can be presented by the following grammar:

$$t ::= t_1 \rightarrow t_2 \mid \mathsf{Int} \mid v \mid \mu v.t \mid \top \mid \bot$$

The type system contains the following components: the binary function type constructor $\rightarrow$, the constant type Int, the possibility for creating recursive types, and two more constant types $\top$, and $\bot$. Moreover, there is a subtype relation, written $\leq$. In contrast, safety analysis uses an abstract domain containing sets of syntactic occurrences of abstractions and the constant Int.

In slogan-form, our result reads:

$$\textit{Flow analysis + Safety checks} =$$
$$\textit{Simple types + Recursive types} + \top + \bot + \textit{Subtyping}$$

Each component of the type system captures a facet of flow analysis:

- The function type constructor $\rightarrow$ corresponds to a set of abstractions. Intuitively, a function type is less concrete than a set of abstractions. Indeed, the other components of the type system are essential to make it accept the same programs as the safety analysis.
- The constant Int is used for the same purpose in both systems. For simplicity, we do not consider other base types, or product and sum constructors, etc. Such constructs can be handled by techniques that are similar to the ones we will present.
- Recursive types are needed in order that safety analysis accepts all programs that do not contain constants.
- The constant $\top$ corresponds to the largest possible set of flow information. This type is needed for variables which can hold both a function and a base value. Intuitively, a program with such a variable should be type incorrect. However, the flow-based analysis may detect that this variable is only passed around but never actually used. For the type system to have that capability, $\top$ is required.
- The constant $\bot$ corresponds to the empty set of flow information. This type is needed for variables which are used both as a function and as a base value. Intuitively, a program that uses a variable in both these ways should be type incorrect. However, the flow-based analysis may detect that this part of the program will never be executed. For the type system to have that capability, $\bot$ is required.
- Subtyping is needed to capture flow of information. Intuitively, if information flows from A to B, then the type of A will be a subtype of the type of B.

Palsberg and Schwartzbach [10, 11] proved that the system *without* $\bot$ accepts at most as many programs as safety analysis. In this paper we present the type system which accepts exactly the same programs as safety analysis. This may be seen as a natural culmination of the previous results.

1.3 Examples

Our example language is a λ-calculus, generated by the following grammar:

$$E ::= x \mid \lambda x.E \mid E_1 E_2 \mid \mathsf{0} \mid \mathsf{succ}\ E$$

Programs that yield a run-time error include $(\mathsf{0}\ x)$, $\mathsf{succ}(\lambda x.x)$, and $(\mathsf{succ}\ \mathsf{0})(x)$, because $\mathsf{0}$ is not a function, succ cannot be applied to functions, and $(\mathsf{succ}\ \mathsf{0})$ is not a function. These programs are not typable and they are rejected by safety analysis. Some programs can be typed in the type system without the use of $\bot$ and $\top$, for example

$$\lambda x.xx \ : \ \mu\alpha.\alpha \to \alpha \ ,$$

where $E : t$ means "E has type t". Some programs require the use of $\top$, for example

$$(\lambda f.(\lambda x.fI)(f\mathsf{0}))I \ : \ \top \ ,$$

where $I = \lambda x.x$. Note that $\top$ is the only type of $(\lambda f.(\lambda x.fI)(f\mathsf{0}))I$. Some programs require the use of $\bot$, for example

$$\lambda x.x(\mathsf{succ}\ x) \ : \ \bot \to t \quad \text{for any } t.$$

Both type inference and safety analysis can be phrased as solving a system of constraints, derived from the program text. We will now present the constraint systems for the last of the above examples. For notational convenience, we give each of the two occurrences of x a label so that the λ-term reads $\lambda x.x_1(\mathsf{succ}\ x_2)$. For brevity, let $E = \lambda x.x_1(\mathsf{succ}\ x_2)$. The constraint system for type inference looks as follows:

$$\begin{array}{rcl}
x \to [\![x_1(\mathsf{succ}\ x_2)]\!] & \leq & [\![E]\!] \\
[\![x_1]\!] & \leq & [\![\mathsf{succ}\ x_2]\!] \to [\![x_1(\mathsf{succ}\ x_2)]\!] \\
x & \leq & [\![x_1]\!] \\
x & \leq & [\![x_2]\!] \\
\mathsf{Int} & \leq & [\![\mathsf{succ}\ x_2]\!] \\
[\![x_2]\!] & \leq & \mathsf{Int}
\end{array}$$

Here, the symbols x, $[\![x_1]\!]$, $[\![x_2]\!]$, $[\![\mathsf{succ}\ x_2]\!]$, $[\![x_1(\mathsf{succ}\ x_2)]\!]$, $[\![E]\!]$ are type variables. Solving this constraint system yields that the possible types for the λ-term $\lambda x.x(\mathsf{succ}\ x)$ are $\top$ and $\bot \to t$ for any type t. Among these, $\bot \to \bot$ is a least type. In general, however, such a constraint system need not have a least solution.

The constraint system for safety analysis looks as follows:

$$\begin{array}{rcl}
\{E\} & \subseteq & [\![E]\!] \\
[\![x_1]\!] & \subseteq & \{E\} \\
x & \subseteq & [\![x_1]\!] \\
x & \subseteq & [\![x_2]\!] \\
\{E\} \subseteq [\![x_1]\!] & \Rightarrow & [\![\mathsf{succ}\ x_2]\!] \subseteq x \\
\{E\} \subseteq [\![x_1]\!] & \Rightarrow & [\![x_1(\mathsf{succ}\ x_2)]\!] \subseteq [\![x_1(\mathsf{succ}\ x_2)]\!] \\
\{\mathsf{Int}\} & \subseteq & [\![\mathsf{succ}\ x_2]\!] \\
[\![x_2]\!] & \subseteq & \{\mathsf{Int}\}
\end{array}$$

If such a constraint system is solvable, then it has a least solution. This particular constraint system is indeed solvable, and the least solution is the mapping φ, where

$$\begin{aligned}
&\varphi([\![E]\!]) = \{E\} \\
&\varphi([\![\mathsf{succ}\ x_2]\!]) = \{\mathsf{Int}\} \\
&\varphi([\![x_1(\mathsf{succ}\ x_2)]\!]) = \varphi(x) = \varphi([\![x_1]\!]) = \varphi([\![x_2]\!]) = \emptyset
\end{aligned}$$

In the following two sections we present the type system and the safety analysis, and in Section 4 we prove that they accept the same programs.

2 The type system

2.1 Types

Definition 1 Let $\Sigma = \{\rightarrow, \mathsf{Int}, \bot, \top\}$ be the ranked alphabet where $\rightarrow$ is binary and $\mathsf{Int}, \bot, \top$ are nullary. A *type* is a regular tree over Σ. A *path* from the root of such a tree is a string over $\{0, 1\}$, where 0 indicates "left subtree" and 1 indicates "right subtree". □

Definition 2 We represent a type by a *term*, that is, a partial function

$$t : \{0,1\}^* \rightarrow \Sigma$$

with domain $\mathcal{D}(t)$ where t maps each path from the root of the type to the symbol at the end of the path. The set of all such terms is denoted T_Σ. □

Following [6], we finitely represent a term by a so-called *term automaton*, as follows.

Definition 3 A *term automaton* over Σ is a tuple

$$\mathcal{M} \;=\; (Q,\ \Sigma,\ q_0,\ \delta,\ \ell)$$

where:

- Q is a finite set of *states*,
- $q_0 \in Q$ is the *start state*,
- $\delta : Q \times \{0,1\} \rightarrow Q$ is a partial function called the *transition function*, and
- $\ell : Q \rightarrow \Sigma$ is a (total) *labeling function*,

such that for any state $q \in Q$, if $\ell(q) \in \{\rightarrow\}$ then

$$\{i \mid \delta(q,i) \text{ is defined}\} \quad = \quad \{0,1\}$$

and if $\ell(q) \in \{\mathsf{Int}, \bot, \top\}$ then

$$\{i \mid \delta(q,i) \text{ is defined}\} \quad = \quad \emptyset \ .$$

The partial function δ extends naturally to a partial function

$$\widehat{\delta} : Q \times \{0,1\}^* \rightarrow Q$$

inductively as follows:

$$\begin{aligned} \widehat{\delta}(q,\epsilon) &= q \\ \widehat{\delta}(q,\alpha i) &= \delta(\widehat{\delta}(q,\alpha),i) \ , \text{ for } i \in \{0,1\}. \end{aligned}$$

The term represented by $\mathcal{M}$ is the term

$$t_{\mathcal{M}} \quad = \quad \lambda\alpha.\ell(\widehat{\delta}(q_0,\alpha)) \ .$$

□

Intuitively, $t_{\mathcal{M}}(\alpha)$ is determined by starting in the start state q_0 and scanning the input α, following transitions of $\mathcal{M}$ as far as possible. If it is not possible to scan all of α because some i-transition along the way does not exist, then $t_{\mathcal{M}}(\alpha)$ is undefined. If on the other hand $\mathcal{M}$ scans the entire input α and ends up in state q, then $t_{\mathcal{M}}(\alpha) = \ell(q)$.

Types are ordered by the subtype relation $\leq$, as follows.

Definition 4 The *parity* of $\alpha \in \{0,1\}^*$ is the number mod 2 of 0's in α. The parity of α is denoted $\pi\alpha$. A string α is said to be *even* if $\pi\alpha = 0$ and *odd* if $\pi\alpha = 1$. Let $\leq_0$ be the partial order on Σ given by

$$\begin{array}{lll} \bot \leq_0 \rightarrow & \text{and} & \rightarrow \leq_0 \top \text{ and} \\ \bot \leq_0 \mathsf{Int} & \text{and} & \mathsf{Int} \leq_0 \top \end{array}$$

and let $\leq_1$ be its reverse

$$\begin{array}{lll} \top \leq_1 \rightarrow & \text{and} & \rightarrow \leq_1 \bot \text{ and} \\ \top \leq_1 \mathsf{Int} & \text{and} & \mathsf{Int} \leq_1 \bot \end{array}$$

For $s,t \in T_\Sigma$, define $s \leq t$ if $s(\alpha) \leq_{\pi\alpha} t(\alpha)$ for all $\alpha \in \mathcal{D}(s) \cap \mathcal{D}(t)$. □

Kozen, Palsberg, and Schwartzbach [6] showed that the relation $\leq$ is equivalent to the order defined by Amadio and Cardelli [1]. The relation $\leq$ is a partial order, and if $s \rightarrow t \leq s' \rightarrow t'$, then $s' \leq s$ and $t \leq t'$ [1, 6].

2.2 Type rules

If E is a λ-term, t is a type, and A is a type environment, *i.e.* a partial function assigning types to variables, then the judgement

$$A \vdash E : t$$

means that E has the type t in the environment A. Formally, this holds when the judgement is derivable using the following six rules:

$$A \vdash \mathsf{0} : \mathsf{Int} \tag{1}$$

$$\frac{A \vdash E : \mathsf{Int}}{A \vdash \mathsf{succ}\ E : \mathsf{Int}} \tag{2}$$

$$A \vdash x : t \quad (\text{provided } A(x) = t) \tag{3}$$

$$\frac{A[x \leftarrow s] \vdash E : t}{A \vdash \lambda x.E : s \rightarrow t} \tag{4}$$

$$\frac{A \vdash E : s \rightarrow t \quad A \vdash F : s}{A \vdash EF : t} \tag{5}$$

$$\frac{A \vdash E : s \quad s \leq t}{A \vdash E : t} \tag{6}$$

The first five rules are the usual rules for simple types and the last rule is the rule of *subsumption*.

The type system has the subject reduction property, that is, if $A \vdash E : t$ is derivable and E β-reduces to E', then $A \vdash E' : t$ is derivable. This is proved by straightforward induction on the structure of the derivation of $A \vdash E : t$.

2.3 Constraints

Given a λ-term E, the type inference problem can be rephrased in terms of solving a system of type constraints. Assume that E has been α-converted so that all bound variables are distinct. Let X_E be the set of λ-variables x occurring in E, and let Y_E be a set of variables disjoint from X_E consisting of one variable $[\![F]\!]$ for each occurrence of a subterm F of E. (The notation $[\![F]\!]$ is ambiguous because there may be more than one occurrence of F in E. However, it will always be clear from context which occurrence is meant.) We generate the following system of inequalities over $X_E \cup Y_E$:

- for every occurrence in E of a subterm of the form $\mathsf{0}$, the inequality

  $$\mathsf{Int} \ \leq \ [\![\mathsf{0}]\!] \ ;$$

- for every occurrence in E of a subterm of the form $\mathsf{succ}\ F$, the two inequalities

$$\begin{array}{rcl} \mathsf{Int} & \leq & [\![\mathsf{succ}\ F]\!] \\ [\![F]\!] & \leq & \mathsf{Int}\ ; \end{array}$$

- for every occurrence in E of a subterm of the form $\lambda x.F$, the inequality

$$(x \rightarrow [\![F]\!])_{\lambda x.F} \quad \leq \quad [\![\lambda x.F]\!]\ ;$$

- for every occurrence in E of a subterm of the form GH, the inequality

$$[\![G]\!] \quad \leq \quad ([\![H]\!] \rightarrow [\![GH]\!])_{GH}\ ;$$

- for every occurrence in E of a λ-variable x, the inequality

$$x \quad \leq \quad [\![x]\!]\ .$$

The subscripts are present to ease notation in Section 4.1; they have no semantic impact and will be explicitly written only in Section 4.1.

Denote by $T(E)$ the system of constraints generated from E in this fashion. The solutions of $T(E)$ over T_Σ correspond to the possible type annotations of E in a sense made precise by Theorem 5.

Let A be a type environment assigning a type to each λ-variable occurring freely in E. If ψ is a function assigning a type to each variable in $X_E \cup Y_E$, we say that ψ *extends* A if A and ψ agree on the domain of A.

Theorem 5 *The judgement $A \vdash E : t$ is derivable if and only if there exists a solution ψ of $T(E)$ extending A such that $\psi([\![E]\!]) = t$. In particular, if E is closed, then E is typable with type t if and only if there exists a solution ψ of $T(E)$ such that $\psi([\![E]\!]) = t$.*

Proof. Similar to the proof of Theorem 2.1 in the journal version of [7], in outline as follows. Given a solution of the constraint system, it is straightforward to construct a derivation of $A \vdash M{:}t$. Conversely, observe that if $A \vdash M{:}t$ is derivable, then there exists a derivation of $A \vdash M{:}t$ such that each use of one of the ordinary rules is followed by exactly one use of the subsumption rule. The approach in for example [14, 11] then gives a set of inequalities of the desired form. $\square$

3 The safety analysis

Following [10, 11], we will use a flow analysis as a basis for a safety analysis. Given a λ-term E, assume that E has been α-converted so that all bound variables are distinct. The set $\mathsf{Abs}(E)$ is the set of subterms of E of the form $\lambda x.F$. The set $\mathsf{Cl}(E)$ is the powerset of $\mathsf{Abs}(E) \cup \{\mathsf{Int}\}$. Safety analysis of a λ-term E can be phrased as solving the following system of constraints over $X_E \cup Y_E$ where type variables range over $\mathsf{Cl}(E)$.

- For every occurrence in E of a subterm of the form $\mathsf{0}$, the constraint

$$\{\mathsf{Int}\} \subseteq [\![\mathsf{0}]\!] \ ;$$

- for every occurrence in E of a subterm of the form $\mathsf{succ}\ F$, the two constraints

$$\begin{aligned} \{\mathsf{Int}\} &\subseteq [\![\mathsf{succ}\ F]\!] \\ [\![F]\!] &\subseteq \{\mathsf{Int}\} \end{aligned}$$

  where the latter provides a safety check;

- for every occurrence in E of a subterm of the form $\lambda x.F$, the constraint

$$(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![\lambda x.F]\!] \ ;$$

- for every occurrence in E of a subterm of the form GH, the constraint

$$[\![G]\!] \subseteq (\mathsf{Abs}(E))_{GH} \ ;$$

  which provides a safety check;

- for every occurrence in E of a λ-variable x, the constraint

$$x \subseteq [\![x]\!] \ ;$$

- for every occurrence in E of a subterm of the form $\lambda x.F$, and for every occurrence in E of a subterm of the form GH, the constraints

$$\begin{aligned} (\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!] &\Rightarrow [\![H]\!] \subseteq x \\ (\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!] &\Rightarrow [\![F]\!] \subseteq [\![GH]\!] \ . \end{aligned}$$

Again, the subscripts are present to ease notation in Section 4.1; they have no semantic impact and will be explicitly written only in Section 4.1.

Denote by $C(E)$ the system of constraints generated from E in this fashion. A solution of $C(E)$ assigns an element of $\mathsf{Cl}(E)$ to each type variable such that all constraints are satisfied. Solutions are ordered by variable-wise set inclusion. See [10, 12] for a cubic time algorithm that given E computes the least solution of $C(E)$ or decides that none exists. See [9] for a proof of the following subject reduction property. If E β-reduces to E', and $C(E)$ has solution φ, then $C(E')$ also has solution φ.

4 Equivalence

4.1 Deductive Closures

We now introduce two auxiliary constraint systems called $\overline{C}(E)$ and $\overline{T}(E)$. They may be thought of as "deductive closures" of $C(E)$ and $T(E)$. We then show that they are isomorphic (Theorem 9).

Definition 6 For every λ-term E, define $\overline{C}(E)$ to be the smallest set such that:

- The non-conditional constraints of $C(E)$ are members of $\overline{C}(E)$.
- If a constraint $c \Rightarrow K$ is in $C(E)$ and c is in $\overline{C}(E)$, then K is in $\overline{C}(E)$.
- For $s \in X_E \cup Y_E$, if $r \subseteq s$ and $s \subseteq t$ both are in $\overline{C}(E)$, then $r \subseteq t$ is in $\overline{C}(E)$.

Notice that every constraint in $\overline{C}(E)$ is of the form $W \subseteq W'$, where W is of the forms V, $\{\mathsf{Int}\}$, or $(\{\lambda x.F\})_{\lambda x.F}$, and where W' is of the forms V, $\{\mathsf{Int}\}$, or $(\mathsf{Abs}(E))_{GH}$, for $V \in X_E \cup Y_E$. □

Definition 7 For every λ-term E, define $\overline{T}(E)$ to be the smallest set such that:

- $T(E) \subseteq \overline{T}(E)$.
- If $(s \to t)_{\lambda x.F} \leq (s' \to t')_{GH}$ is in $\overline{T}(E)$, then $s' \leq s$ and $t \leq t'$ are in $\overline{T}(E)$.
- For $s \in X_E \cup Y_E$, if $r \leq s$ and $s \leq t$ both are in $\overline{T}(E)$, then $r \leq t$ is in $\overline{T}(E)$.

Notice that every constraint in $\overline{T}(E)$ is of the form $W \leq W'$ where W is of the forms V, Int, or $(V \to V')_{\lambda x.F}$, and where W' is of the form V, Int, or $(V \to V')_{GH}$, for $V, V' \in X_E \cup Y_E$. □

We will now present the definition of two functions $\mathcal{I}$ and $\mathcal{J}$, one from $\overline{C}(E)$ to $\overline{T}(E)$ and one from $\overline{T}(E)$ to $\overline{C}(E)$. After the definition we prove that they are well-defined and each others inverses.

Definition 8 The functions

$$\begin{aligned}\mathcal{I} &: \overline{C}(E) \to \overline{T}(E) \\ \mathcal{J} &: \overline{T}(E) \to \overline{C}(E)\end{aligned}$$

are defined as follows.

$$\begin{aligned}\mathcal{I}(W \subseteq W') &= (\mathcal{L}_{\mathcal{I}}(W) \leq \mathcal{L}_{\mathcal{I}}(W')) \\ \mathcal{J}(W \leq W') &= (\mathcal{L}_{\mathcal{J}}(W) \subseteq \mathcal{L}_{\mathcal{J}}(W'))\end{aligned}$$

where the functions $\mathcal{L}_{\mathcal{I}}$ and $\mathcal{L}_{\mathcal{J}}$ are:

$$\mathcal{L}_{\mathcal{I}}(W) = \begin{cases} W & \text{if } W \in X_E \cup Y_E \\ \mathsf{Int} & \text{if } W = \{\mathsf{Int}\} \\ (x \to [\![F]\!])_{\lambda x.F} & \text{if } W = (\{\lambda x.F\})_{\lambda x.F} \\ ([\![H]\!] \to [\![GH]\!])_{GH} & \text{if } W = (\mathsf{Abs}(E))_{GH} \end{cases}$$

$$\mathcal{L}_{\mathcal{J}}(W) = \begin{cases} W & \text{if } W \in X_E \cup Y_E \\ \{\mathsf{Int}\} & \text{if } W = \mathsf{Int} \\ (\{\lambda x.F\})_{\lambda x.F} & \text{if } W = (x \to [\![F]\!])_{\lambda x.F} \\ (\mathsf{Abs}(E))_{GH} & \text{if } W = ([\![H]\!] \to [\![GH]\!])_{GH} \end{cases}$$

□

Theorem 9 *The sets $\overline{C}(E)$ and $\overline{T}(E)$ are isomorphic, and $\mathcal{I}$ and $\mathcal{J}$ are bijections and each others inverses.*

Proof. If $\mathcal{I}$ and $\mathcal{J}$ are well-defined, then clearly they are inverses of each other and thus bijections, so $\overline{C}(E)$ and $\overline{T}(E)$ are isomorphic.

First we show that $\mathcal{I}$ is well-defined. We proceed by induction on the construction of $\overline{C}(E)$. In the base case, consider the non-conditional constraints of $C(E)$ and observe that for those we have:

$\overline{C}(E)$	$\overline{T}(E)$
$\{\mathsf{Int}\} \subseteq [\![0]\!]$	$\mathsf{Int} \leq [\![0]\!]$
$\{\mathsf{Int}\} \subseteq [\![\mathsf{succ}\ F]\!]$	$\mathsf{Int} \leq [\![\mathsf{succ}\ F]\!]$
$[\![F]\!] \subseteq \{\mathsf{Int}\}$	$[\![F]\!] \leq \mathsf{Int}$
$(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![\lambda x.F]\!]$	$(x \to [\![F]\!])_{\lambda x.F} \leq [\![\lambda x.F]\!]$
$[\![G]\!] \subseteq (\mathsf{Abs}(E))_{GH}$	$[\![G]\!] \leq ([\![H]\!] \to [\![GH]\!])_{GH}$
$x \subseteq [\![x]\!]$	$x \leq [\![x]\!]$

It follows that the lemma holds in the base case.

In the induction step, consider first the constraints

$$(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!] \;\Rightarrow\; [\![H]\!] \subseteq x$$
$$(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!] \;\Rightarrow\; [\![F]\!] \subseteq [\![GH]\!]$$

in $C(E)$ and suppose $(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!]$ is in $\overline{C}(E)$. By the induction hypothesis, $(x \to [\![F]\!])_{\lambda x.F} \leq [\![G]\!]$ is in $\overline{T}(E)$. Moreover, $[\![G]\!] \leq ([\![H]\!] \to [\![GH]\!])_{GH}$ is in $T(E)$ and thus also in $\overline{T}(E)$. Hence, $(x \to [\![F]\!])_{\lambda x.F} \leq ([\![H]\!] \to [\![GH]\!])_{GH}$ is in $\overline{T}(E)$, so also $[\![H]\!] \leq x$ and $[\![F]\!] \leq [\![GH]\!]$ are in $\overline{T}(E)$.

Consider then $r \subseteq s$ and $s \subseteq t$ in $\overline{C}(E)$, and suppose $s \in X_E \cup Y_E$. By the induction hypothesis, $\mathcal{L}_{\mathcal{I}}(r) \leq \mathcal{L}_{\mathcal{I}}(s)$ and $\mathcal{L}_{\mathcal{I}}(s) \leq \mathcal{L}_{\mathcal{I}}(t)$ are in $\overline{T}(E)$. From $s \in X_E \cup Y_E$ we get $\mathcal{L}_{\mathcal{I}}(s) = s$, so $\mathcal{L}_{\mathcal{I}}(r) \leq \mathcal{L}_{\mathcal{I}}(t)$ is in $\overline{T}(E)$.

Then we show that $\mathcal{J}$ is well-defined. We proceed by induction on the construction of $\overline{T}(E)$. In the base case, consider the constraints of $T(E)$. Using the same table as above we observe that $\mathcal{J}$ is well-defined on all these constraints.

In the induction step, consider first $(x \to [\![F]\!])_{\lambda x.F} \leq ([\![H]\!] \to [\![GH]\!])_{GH}$ in $\overline{T}(E)$. It is sufficient to prove that $\mathcal{L}_{\mathcal{J}}([\![H]\!]) \subseteq \mathcal{L}_{\mathcal{J}}(x)$ and $\mathcal{L}_{\mathcal{J}}([\![F]\!]) \subseteq \mathcal{L}_{\mathcal{J}}([\![GH]\!])$ are in $\overline{C}(E)$, or equivalently, that $[\![H]\!] \subseteq x$ and $[\![F]\!] \subseteq [\![GH]\!]$ are in $\overline{C}(E)$. In $C(E)$ we have

$$
\begin{aligned}
(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!] \quad &\Rightarrow \quad [\![H]\!] \subseteq x \\
(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!] \quad &\Rightarrow \quad [\![F]\!] \subseteq [\![GH]\!] \ .
\end{aligned}
$$

Moreover, $(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!]$ is in $\overline{C}(E)$. Hence, $[\![H]\!] \subseteq x$ and $[\![F]\!] \subseteq [\![GH]\!]$ are in $\overline{C}(E)$.

Consider then $r \leq s$ and $s \leq t$ in $\overline{T}(E)$, and suppose $s \in X_E \cup Y_E$. By the induction hypothesis, $\mathcal{L}_{\mathcal{J}}(r) \subseteq \mathcal{L}_{\mathcal{J}}(s)$ and $\mathcal{L}_{\mathcal{J}}(s) \subseteq \mathcal{L}_{\mathcal{J}}(t)$ are in $\overline{C}(E)$. From $s \in X_E \cup Y_E$ we get $\mathcal{L}_{\mathcal{J}}(s) = s$, so $\mathcal{L}_{\mathcal{J}}(r) \subseteq \mathcal{L}_{\mathcal{J}}(t)$ is in $\overline{C}(E)$. □

4.2 The equivalence proof

Definition 10 For every λ-term E, let $\mathsf{Tmap}(E)$ be the set of total functions from $X_E \cup Y_E$ to T_Σ and let $\mathsf{Cmap}(E)$ be the set of total functions from $X_E \cup Y_E$ to $\mathsf{Cl}(E)$. □

The following construction is the key to mapping flow information to types.

Definition 11 For every λ-term E, $\varphi \in \mathsf{Cmap}(E)$, and $q_0 \in \mathsf{Cl}(E)$, define the term automaton $\mathcal{A}(E, \varphi, q_0)$ as follows:

$$\mathcal{A}(E, \varphi, q_0) \quad = \quad (\mathsf{Cl}(E),\ \Sigma,\ q_0,\ \delta,\ \ell)$$

where:

- $\delta(\{\lambda x_1.E_1, \ldots, \lambda x_n.E_n\}, 0) = \bigcap_{i=1}^{n} \varphi(x_i)$
  for $n > 0$
- $\delta(\{\lambda x_1.E_1, \ldots, \lambda x_n.E_n\}, 1) = \bigcup_{i=1}^{n} \varphi[\![E_i]\!]$
  for $n > 0$
- $$\ell(q) = \begin{cases} \bot & \text{if } q = \emptyset \\ \mathsf{Int} & \text{if } q = \{\mathsf{Int}\} \\ \to & \text{if } q \subseteq \mathsf{Abs}(E) \wedge q \neq \emptyset \\ \top & \text{otherwise} \end{cases}$$

□

Lemma 12 *Suppose* $\varphi \in \mathsf{Cmap}(E)$ *and* $S_1, S_2 \in \mathsf{Cl}(E)$. *If* $S_1 \subseteq S_2$, *then* $t_{\mathcal{A}(E,\varphi,S_1)} \leq t_{\mathcal{A}(E,\varphi,S_2)}$.

Proof. Define the orderings $\subseteq_0$, $\subseteq_1$ on $\mathsf{Cl}(E)$ such that $\subseteq_0$ equals $\subseteq$ and $\subseteq_1$ equals $\supseteq$. The desired conclusion follows immediately from the property that if $\alpha \in \mathcal{D}(t_{\mathcal{A}(E,\varphi,S_1)}) \cap \mathcal{D}(t_{\mathcal{A}(E,\varphi,S_2)})$, then $\widehat{\delta}(S_1, \alpha) \subseteq_{\pi\alpha} \widehat{\delta}(S_2, \alpha)$. This property is proved by straightforward induction on the length of α. $\square$

We can now prove that the type system and the safety analysis accept the same programs.

Theorem 13 *For every λ-term E, the following seven conditions are equivalent:*

1. *$C(E)$ is solvable.*
2. *$T(E)$ is solvable.*
3. *$\overline{T}(E)$ is solvable.*
4. *$\overline{C}(E)$ is solvable.*
5. *$\overline{C}(E)$ does not contain constraints of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$.*
6. *$\overline{T}(E)$ does not contain constraints of the forms $\mathsf{Int} \leq V \to V'$ or $V \to V' \leq \mathsf{Int}$, where $V, V' \in X_E \cup Y_E$.*
7. *The function*

   $$\lambda V.\{\ k \mid \ \textit{the constraint } \{k\} \subseteq V \ \textit{ is in } \overline{C}(E)\ \}$$

   is the least solution of $C(E)$.

Proof. Given a λ-term E, notice that by the isomorphism of Theorem 9, (5) $\Leftrightarrow$ (6). To show the remaining equivalences, we proceed by proving the implications:

$$(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (4) \Rightarrow (5) \Rightarrow (7) \Rightarrow (1)$$

To prove (1) $\Rightarrow$ (2), suppose $C(E)$ has solution $\varphi \in \mathsf{Cmap}(E)$. Let f be the function $\lambda S.t_{\mathcal{A}(E,\varphi,S)}$ and define $\psi \in \mathsf{Tmap}(E)$ by $\psi = f \circ \varphi$. We will show that $T(E)$ has solution ψ. We consider each of the constraints in turn. The cases of the constraints generated from subterms of the forms $\mathsf{0}$, $\mathsf{succ}\ E$, x are immediate, by using Lemma 12. Consider then $\lambda x.F$ and the constraint $x \to [\![F]\!] \leq [\![\lambda x.F]\!]$. By Lemma 12 we get

$$\psi(x) \to \psi([\![F]\!]) = f(\{\lambda x.F\}) \leq \psi([\![\lambda x.F]\!])\ .$$

Consider then GH and the constraint $[\![G]\!] \leq [\![H]\!] \to [\![GH]\!]$. We know that $\varphi([\![G]\!]) \subseteq \mathsf{Abs}(E)$ so there are two cases. Suppose first that $\varphi([\![G]\!]) = \emptyset$. We then have $\psi([\![G]\!]) = \bot \leq \psi([\![H]\!]) \to \psi([\![GH]\!])$. Consider then the case where $\varphi([\![G]\!]) = \{\lambda x_1.E_1, \ldots, \lambda x_n.E_n\}$, for $n > 0$. We then have that $\varphi([\![H]\!]) \subseteq \varphi([\![x_i]\!])$ and $\varphi([\![E_i]\!]) \subseteq \varphi([\![GH]\!])$ for $i \in \{1, \ldots, n\}$. Thus, $\varphi([\![H]\!]) \subseteq \bigcap_{i=1}^n \varphi([\![x_i]\!])$ and $\bigcup_{i=1}^n \varphi([\![E_i]\!]) \subseteq \varphi([\![GH]\!])$. So, by Lemma 12,

$$\begin{aligned}
\psi([\![G]\!]) &= f(\varphi([\![G]\!])) \\
&= f(\bigcap_{i=1}^{n} \varphi([\![x_i]\!])) \to f(\bigcup_{i=1}^{n} \varphi([\![E_i]\!])) \\
&\leq f(\varphi([\![H]\!])) \to f(\varphi([\![GH]\!])) \\
&= \psi([\![H]\!]) \to \psi([\![GH]\!])
\end{aligned}$$

To prove (2) $\Rightarrow$ (3), suppose $T(E)$ has solution $\psi \in \mathsf{Tmap}(E)$. It is sufficient to show that $\overline{T}(E)$ has solution ψ, and this can be proved by straightforward induction on the construction of $\overline{T}(E)$.

To prove (3) $\Rightarrow$ (4), suppose $\overline{T}(E)$ has solution $\psi \in \mathsf{Tmap}(E)$. Define $\overline{\varphi} \in \mathsf{Cmap}(E)$ as follows:

$$\overline{\varphi}(V) = \begin{cases} \emptyset & \text{if } (\psi(V))(\epsilon) = \bot \\ \{\mathsf{Int}\} & \text{if } (\psi(V))(\epsilon) = \mathsf{Int} \\ \mathsf{Abs}(E) & \text{if } (\psi(V))(\epsilon) = \to \\ \mathsf{Abs}(E) \cup \{\mathsf{Int}\} & \text{if } (\psi(V))(\epsilon) = \top \end{cases}$$

We will show that $\overline{C}(E)$ has solution $\overline{\varphi}$. To see this, let $W \subseteq Z$ be a constraint in $\overline{C}(E)$. If it is of the forms $\{\mathsf{Int}\} \subseteq \{\mathsf{Int}\}$ or $\{\lambda x.F\} \subseteq \mathsf{Abs}(E)$, then it is solvable by all functions, including $\overline{\varphi}$. For the remaining cases, notice that by Theorem 9, $\mathcal{L}_{\mathcal{I}}(W) \leq \mathcal{L}_{\mathcal{I}}(Z)$ is in $\overline{T}(E)$ and thus it has solution ψ. This means that $W \subseteq Z$ cannot be of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$. Suppose then that $W \subseteq Z$ is of one of the remaining forms, that is, $\{\mathsf{Int}\} \subseteq V$, $V \subseteq V'$, $V \subseteq \{\mathsf{Int}\}$, $\{\lambda x.F\} \subseteq V$, $V \subseteq \mathsf{Abs}(E)$, where $V, V' \in X_E \cup Y_E$. We will treat just the first of them, the others are similar. For a constraint of the form $\{\mathsf{Int}\} \subseteq V$, it follows that $\mathsf{Int} \leq V$ is in $\overline{T}(E)$. Since $\overline{T}(E)$ has solution ψ we get that $(\psi(V))(\epsilon) \in \{\mathsf{Int}, \top\}$. Thus, $\overline{\varphi}(V)$ is either $\{\mathsf{Int}\}$ or $\mathsf{Abs}(E) \cup \{\mathsf{Int}\}$, and hence $\{\mathsf{Int}\} \subseteq V$ has solution $\overline{\varphi}$.

To prove (4) $\Rightarrow$ (5), observe that constraints of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$ are not solvable.

To prove (5) $\Rightarrow$ (7), suppose $\overline{C}(E)$ does not contain constraints of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$. Define

$$\varphi' = \lambda V.\{\ k \mid \text{ the constraint } \{k\} \subseteq V \text{ is in } \overline{C}(E)\ \}$$

We proceed in four steps, as follows.

- First we show that φ' is a solution of $\overline{C}(E)$. We consider in turn each of the seven possible forms of constraints in $\overline{C}(E)$. Constraints of the forms

$\{\mathsf{Int}\} \subseteq \{\mathsf{Int}\}$ and $\{\lambda x.F\} \subseteq \mathsf{Abs}(E)$ have any solution, including φ'. We are thus left with constraints of the forms $\{\mathsf{Int}\} \subseteq V$, $V \subseteq V'$, $V \subseteq \{\mathsf{Int}\}$, $\{\lambda x.F\} \subseteq V$, $V \subseteq \mathsf{Abs}(E)$, where $V, V' \in X_E \cup Y_E$. We will treat just the first three, since case four is similar to case one and since case five is similar to case three. For a constraint of the form $\{\mathsf{Int}\} \subseteq V$, notice that $\mathsf{Int} \in \varphi'(V)$, so the constraint has solution φ'. For a constraint of the form $V \subseteq V'$, suppose $k \in \varphi'(V)$. Then the constraint $\{k\} \subseteq V$ is in $\overline{C}(E)$, and hence the constraint $\{k\} \subseteq V'$ is also in $\overline{C}(E)$. It follows that $k \in \varphi'(V')$. For a constraint of the form $V \subseteq \{\mathsf{Int}\}$, suppose it does not have solution φ'. Hence, there exist $k \in \varphi'(V)$ such that $k \neq \mathsf{Int}$. It follows that the constraint $\{k\} \subseteq V$ is in $\overline{C}(E)$, and hence the constraint $\{k\} \subseteq \{\mathsf{Int}\}$ is also in $\overline{C}(E)$, a contradiction.

- Next we show that φ' is the least solution of $\overline{C}(E)$. To do this, let φ be any solution of $\overline{C}(E)$ and suppose $V \in X_E \cup Y_E$. It is sufficient to prove that $\varphi'(V) \subseteq \varphi(V)$. Suppose $k \in \varphi'(V)$. Then the constraint $\{k\} \subseteq V$ is in $\overline{C}(E)$. Since φ is a solution of $\overline{C}(E)$, $k \in \varphi(V)$.

- Next we show that φ' is a solution of $C(E)$. Consider first the non-conditional constraints of $C(E)$. Since these constraints are also members of $\overline{C}(E)$, they have solution φ'. Consider then $\{\lambda x.F\} \subseteq V \Rightarrow K$ in $C(E)$ and suppose $\{\lambda x.F\} \subseteq V$ has solution φ'. Then by the definition of φ', we have that $\{\lambda x.F\} \subseteq V$ is in $\overline{C}(E)$, so also K is in $\overline{C}(E)$, and hence K has solution φ'.

- Finally we show that φ' is the least solution of $C(E)$. To do this, let φ be any solution of $C(E)$. Then φ is also a solution of $\overline{C}(E)$, as can be proved by straightforward induction on the construction of $\overline{C}(E)$. Since φ' is the least solution of $\overline{C}(E)$, φ' is smaller than φ.

To prove (7) $\Rightarrow$ (1), simply notice that since $C(E)$ has a solution, it is solvable. $\square$

Corollary 14 *The type system accepts the same programs as the safety analysis.*

4.3 Algorithms

As corollaries of Theorem 13 we get two cubic time algorithms. Given a λ-term E, first observe that both $\overline{C}(E)$ and $\overline{T}(E)$ can be computed in time $O(n^3)$ where n is the size of E. We can then easily answer the following two questions:

- Question **(safety)**: Is E accepted by safety analysis?
  Algorithm: Check that $\overline{C}(E)$ does not contain constraint of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$.

- Question **(type inference)**: Is E typable? If so, what is an annotation of it?
  Algorithm: Use the safety checking algorithm. If E turns out to be typable, we get an annotation by first calculating the two functions

  $$\varphi' = \lambda V.\{\ k \mid \text{ the constraint } \{k\} \subseteq V \text{ is in } \overline{C}(E)\ \}$$

  and

  $$f = \lambda S.t_{\mathcal{A}(E,\varphi,S)}$$

  and then forming the composition

  $$\psi = f \circ \varphi' \ .$$

  This function ψ is a solution of $T(E)$.

The question of type inference has been open until now. In contrast, it is well-known that flow analysis in the style discussed in this paper can be computed in time $O(n^3)$.

Acknowledgements

The authors thank Mitchell Wand for encouragement and helpful discussions. The results of this paper were obtained while the first author was at Northeastern University, Boston; he is currently hosted by BRICS, Basic Research in Computer Science, Centre of the Danish National Research Foundation.

References

[1] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Also in Proc. POPL'91.

[2] Torben Amtoft. Minimal thunkification. In *Proc. WSA'93*, pages 218–229, 1993.

[3] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1–3):3–34, December 1991.

[5] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proc. PEPM'93, Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, 1993.

[6] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*. To appear. Also in Proc. POPL'93, Twentieth Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 419–428, Charleston, South Carolina, January 1993.

[7] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994. Also in Proc. FOCS'92, 33rd IEEE Symposium on Foundations of Computer Science, pages 363–371, Pittsburgh, Pennsylvania, October 1992.

[8] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 260–272, 1989.

[9] Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*. To appear. Also in Proc. CAAP'94, Colloquium on Trees in Algebra and Programming, Springer-Verlag (*LNCS* 787), pages 276–290, Edinburgh, Scotland, April 1994.

[10] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*. To appear.

[11] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference for partial types. *Information Processing Letters*, 43:175–180, 1992.

[12] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[13] Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 47–87. MIT Press, 1991.

[14] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, 1991.

Specifying and Verifying Parametric Processes

Wiesław Pawłowski* Paweł Pączkowski† Stefan Sokołowski*

December 1994

Abstract

A framework in which processes parametrized with other processes can be specified, defined and verified is introduced. Higher order process-parameters are allowed. The formalism resembles typed lambda calculus built on top of a process algebra, where specifications play the role of types. A proof system for deriving judgements "parametric process meets a specification" is given.

1 Introduction

A typical approach to software development is that of decomposition of large tasks into smaller subtasks. The decomposition principle, when formalized in terms of a *specification system*, by which, following [Sokołowski 94], we mean a triple

$$(\mathit{Constr}, \mathit{Spec}, \mathbf{sat})$$

consisting of a set of constructions *Constr* (programs, processes, etc.), a set of specifications *Spec*, and satisfaction relation **sat** relating constructions to specifications, amounts to the requirement that a task of providing a construction c that satisfies a given specification S (denoted c **sat** S) can be decomposed as follows:

1. provide sub-constructions $c_1, \ldots, c_n$ that satisfy some sub-specifications $S_1, \ldots, S_n$ into which the original specification S is decomposed, and

2. provide a method of combining $c_1, \ldots, c_n$ so as to obtain such c that c **sat** S.

In [Sokołowski 94], a point of view was taken, that the method mentioned in point 2 should be a *construction* itself, i.e. belong to *Constr*. Adopting this point of view we can rephrase 2 as

*Institute of Computer Science, Polish Academy of Sciences, ul. Abrahama 18, 81-825 Sopot, Poland, e-mail: {W.Pawlowski, S.Sokolowski}@IPIPAN.Gda.PL; supported by ICS PAN, KBN grant PB-1312/P3/92/02 and CRIT IC 1010/II.

†Department of Computing Science, Chalmers University of Technology, 412-96 Göteborg, Sweden, and Institute of Mathematics, University of Gdańsk, ul. Wita Stwosza 57, 80-952 Gdańsk, Poland, e-mail: pawel@cs.chalmers.se, matpmp@halina.univ.gda.pl; supported by KBN grant PB-1312/P3/92/02, CRIT IC 1010/II and ESPRIT BRA CONCUR2.

$2'$. provide a parametric construction f such that

$$f \ \mathbf{sat} \ S_1 \to S_2 \to \cdots \to S_n \to S$$

The "functional" specification $S_1 \to S_2 \to \cdots \to S_n \to S$, which is assumed to belong to *Spec*, is intended to specify precisely those parametric constructions which yield objects satisfying S when applied to sub-constructions satisfying $S_1, \ldots, S_n$. Thus, if the decomposition steps 1 and $2'$ are realized then $fc_1 \ldots c_n$ defines c we were looking for.

When the sub-constructions are again decomposed into smaller bits the parameters become parametric themselves and we have to deal with higher order parameters. In [Sokołowski 94] a more elaborate decomposition schema than the one presented above is considered: dependent products rather than functional specifications are allowed, but here we will confine ourselves to a simple setup.

The purpose of this work is to apply the methodology sketched above in the context of concurrent process specification and verification. As a starting point we adopt a process algebra and logic for specifying processes proposed in [Olderog 91]. Olderog extends a process algebra by introducing mixed terms, where a specification can become a part of a process term. This enables transformation based development of processes [Olderog 91, Olderog 91a]. We consider a different option and introduce parametric processes that contain process variables. We allow higher order parametricity: process variables can represent also parametric processes. Technically speaking, Olderogs's process algebra is extended with abstraction and application primitives. This leads to a formalism resembling a typed lambda calculus built on top of a process algebra, where specifications play the role of types. Mixed terms can be viewed as a special case of parametric processes.

We provide a proof system for deriving judgements "a (parametric) process meets a specification". To this end, we needed to reformulate the proof system for nonparametric processes that was given in [Olderog 91]. Derivations in that proof system use mixed terms. We provide a direct proof system for nonparametric processes, which does not appeal to mixed terms and which is shown to be equivalent to that of [Olderog 91]. The direct proof system is extended to the parametric case.

The concept of parametric processes (but only first order ones) has already appeared in [Milner 89]. However, this method of constructing CCS processes is not reflected in their verification because the bisimulation proof technique used in this reference requires a considerable effort in decomposing large verification tasks into smaller ones (see e.g. [Larsen and Milner 86]).

The formalism we develop allows decomposition steps of the form 1 and $2'$ that involve higher-order processes as parameters and helps one to do a systematic book-keeping of process dependencies, which is useful in higher-order constructions. We view our approach as an alternative to process development methods using transformation [Olderog 91] or refinement [Holmström 89] techniques.

The paper is organized as follows. In Section 2 we present a process algebra of nonparametric processes and a logic for specifying them that appear in

[Olderog 91]. A new, direct proof system for nonparametric processes is given in Section 3. In Section 4 we define parametric processes and extend the proof system to the parametric case. Section 5 contains an example.

2 Nonparametric processes and specifications

The class of processes that we consider and a logic for specifying their properties are taken from [Olderog 91]. We briefly present the main notions, referring to [Olderog 91] for a comprehensive exposition.

2.1 Processes

Let *Comm* be an infinite set of *communications*. Together with a special symbol τ representing a hidden or internal activity of a process it will constitute the set of *actions*.

In order to define a syntactic representation of processes we start with a description of the set of *recursive terms Rec*. Let $\mathcal{X}$ be a (countable) set of (process) *identifiers*. We assume that $\mathcal{X}$ is partitioned into sets $\mathcal{X}_A$ of *identifiers with alphabet* A, where $A \subseteq Comm$ is finite. In the definition below P and Q range over recursive terms, X ranges over identifiers, and A over finite sets of communications. The set *Rec* is defined as follows:

$$
\begin{array}{lcll}
P & ::= & \mathrm{stop}_A & \text{(deadlock)} \\
 & \mid & a.P & \text{(prefix)} \\
 & \mid & P + Q & \text{(choice)} \\
 & \mid & P \parallel Q & \text{(parallel composition)} \\
 & \mid & P[b/a] & \text{(renaming)} \\
 & \mid & P \backslash a & \text{(hiding)} \\
 & \mid & X & \text{(identifier)} \\
 & \mid & \mu X.P & \text{(recursion)}
\end{array}
$$

Iterated applications of renaming and hiding will be abbreviated as follows:

- $P[\bar{b}/\bar{a}]$ will stand for $P[b_1/a_1]\ldots[b_n/a_n]$, where $\bar{b} = b_1, \ldots, b_n, \bar{a} = a_1, \ldots, a_n$
- $P\backslash A$ will denote $P\backslash a_1 \ldots \backslash a_n$, where $A = \{a_1, \ldots a_n\}$.

Definition 1 A recursive term P is *communication guarded* if there exists a set of communications A such that for every recursive subterm $\mu X.Q$ of P, for every free occurrence of X in Q

(1) X lies within a subterm of Q of the form $a.R$

(2) X does not lie within a subterm of Q of the form $R[b/a]$ or $R\backslash a$, where $a \in A$ and $b \notin A$.

If the condition (2) above is dropped, we say that P is *action guarded* (this notion is usually given a separate, slightly simpler definition).

To define process terms one more notion is needed — a notion of a *(communication) alphabet*. An alphabet of a recursive term P (denoted by $\alpha(P)$) is a set of communications defined inductively as follows:

$$\begin{array}{lcl}
\alpha(\mathrm{stop}_A) & = & A \\
\alpha(a.P) & = & \{a\} \cup \alpha(P) \\
\alpha(P+Q) & = & \alpha(P) \cup \alpha(Q) \\
\alpha(P \parallel Q) & = & \alpha(P) \cup \alpha(Q) \\
\alpha(P[b/a]) & = & (\alpha(P) - \{a\}) \cup \{b\} \\
\alpha(P\backslash a) & = & (\alpha(P) - \{a\}) \\
\alpha(X) & = & A \text{ if } X \in \mathcal{X}_A \\
\alpha(\mu X.P) & = & \alpha(X) \cup \alpha(P)
\end{array}$$

Definition 2 We say that a recursive term P is a *process term* (or simply, a process) if it satisfies the following conditions:

- P is action guarded
- every subterm $a.Q$ of P satisfies $a \in \alpha(Q)$
- every subterm $Q + R$ of P satisfies $\alpha(Q) = \alpha(R)$
- every subterm $\mu X.Q$ of P satisfies $\alpha(X) = \alpha(Q)$

The set of all process terms will be denoted by *Proc*. A process term is *closed* if it does not contain free process identifiers. The set of all closed process terms will be denoted by *CProc*.

The intuitive meaning of process terms is as follows:

- stop_A denotes a process which neither engages in any communication nor in any internal action.
- $a.P$ denotes a process which first communicates a and then behaves like P.
- $P + Q$ denotes a process which behaves like P or like Q depending on whether its first action was an action of P or of Q. If the first action could be performed by P as well as by Q the choice made is nondeterministic.
- $P \parallel Q$ denotes a parallel composition of P and Q which behaves as P and Q working independently except for synchronization on actions belonging to the intersection of their communication alphabets.
- $P[b/a]$ denotes a process which behaves like P, but with all actions a changed to b.
- $P\backslash a$ denotes a process which behaves like P, but with all actions a hidden, i.e. changed to τ.
- $\mu X.P$ behaves like P, but with every occurrence of X inside P denoting a "recursive call" to $\mu X.P$.

The above intuition can be made precise in several different ways. One possible choice is so called *readiness semantics.* It describes process behaviour in terms of pairs $(tr, \mathcal{F})$ where tr is finite sequence of communications called *trace*, and $\mathcal{F}$, called *ready set*, is a set of actions in which a process is ready to engage after "performing" the trace tr. Besides the ready pairs the semantic domain contains a distinguished element τ representing initial *unstability* of a process, and "divergent" ready pairs of the form $(tr, \uparrow)$ which represent behaviours where a process after performing trace tr enters an infinite loop.

Thus, the set of *process information* over A is defined as:

$$\text{Info}_{\mathcal{R},A} = \{\tau\} \cup A^* \times \mathcal{P}(A) \cup A^* \times \{\uparrow\}$$

and the semantic domain of the readiness semantics is defined as follows:

$$\text{DOM}_{\mathcal{R}} = \bigcup_{A \subseteq Comm} \text{DOM}_{\mathcal{R},A}$$

where A is a finite subset of $Comm$ and

$$\text{DOM}_{\mathcal{R},A} = \{(A, \Gamma) | \Gamma \subseteq \text{Info}_{\mathcal{R},A}\}.$$

The readiness semantics for closed process terms is a function:

$$\mathcal{R}^*[\![\cdot]\!] : CProc \to \text{DOM}_{\mathcal{R}}$$

which can be given an elegant denotational definition.

There is a natural partial ordering on the semantic domain $\text{DOM}_{\mathcal{R}}$:

$$(A_1, R_1) \sqsubseteq (A_2, R_2) \text{ if } A_1 = A_2 \text{ and } R_1 \supseteq R_2$$

Intuitively, if $\mathcal{R}^*[\![P]\!] \sqsubseteq \mathcal{R}^*[\![Q]\!]$ then the process denoted by Q is more *controllable* than the one denoted by P, i.e. it is more deterministic, less divergent and more stable.

2.2 Readiness specifications

Processes are specified by formulas of many-sorted predicate logic, called *readiness logic*, which has sorts of communications, traces (finite sequences of communications), ready sets (finite subsets of communications), natural numbers and logical values. In this report we give just a minimal necessary information on readiness logic. For our purposes it is sufficient to remark that readiness logic allows one to define sets of ready pairs in a direct manner and is fairly expressive, for example, regular sets over $Comm$ can be expressed in it.

A *readiness specification* is a formula of readiness logic, in which at most the distinguished trace variable h and the distinguished set variable F are free, and which satisfies some natural syntactic restrictions that guarantee a monotonicity property explained below. Since a specification has just two free variables h and F, the notion of truth of a specification with respect to the evaluation $(tr, \mathcal{F})$ of h and F can be defined and will be denoted

$$(tr, \mathcal{F}) \models S \tag{1}$$

If (1) holds we say that S is satisfied by a ready pair $(tr, \mathcal{F})$. Thus, a specification S determines a set of ready pairs which satisfy it.

Moreover, with every specification S its alphabet, denoted by $\alpha(S)$, can be associated. This is a consequence of yet another syntactic restriction on readiness formulas that every occurrence of the free variable h lies in the scope of *projection* $\downarrow_A$ onto some finite set of communications A. What it means semantically is that S defines a set of ready pairs $(tr, \mathcal{F})$, where only the projection of tr onto $\alpha(S)$ is defined and the actions outside $\alpha(S)$ do not affect satisfiability of S by $(tr, \mathcal{F})$.

The *monotonicity* property of readiness specifications that was mentioned above is defined as follows: for an arbitrary readiness specification S and for all traces tr and ready sets $\mathcal{F}$ and $\mathcal{G}$

$$(tr, \mathcal{F}) \models S \text{ and } \mathcal{F} \subseteq \mathcal{G} \text{ imply } (tr, \mathcal{G}) \models S$$

The set of specifications will be denoted by *Spec*. The *readiness semantics of readiness specifications* is defined as follows:

$$\mathcal{R}^*[\![S]\!] = (\alpha(S), \{(tr, \mathcal{F}) | tr \in \alpha(S)^* \wedge \mathcal{F} \subseteq \alpha(S) \wedge (tr, \mathcal{F}) \models S\}).$$

2.3 Satisfaction relation

Since processes as well as specifications are interpreted semantically over readiness domain, the satisfaction relation has a simple semantic definition:

$$P \textbf{ sat } S \text{ iff } \mathcal{R}^*[\![P]\!] \sqsupseteq \mathcal{R}^*[\![S]\!]$$

Intuitively, it means that

- The process P and the specification S have the same alphabets i.e. $\alpha(P) = \alpha(S)$.
- *Safety condition.* If P may engage in a trace tr then there must exist a ready set $\mathcal{F}$ such that $(tr, \mathcal{F}) \models S$.
- *Liveness condition.* After performing a trace tr the process P must engage in one of the ready sets $\mathcal{F}$ such that $(tr, \mathcal{F}) \models S$.
- *Stability condition.* The process P is stable immediately.

2.4 Proof system for sat via mixed terms

The inference system for **sat** defined in [Olderog 91] makes use of a notion of *mixed terms*. Mixed terms are, as the name suggests, a mixture of readiness specifications and process terms and are formally defined by allowing specifications in recursive terms

$$P ::= S \mid \mathrm{stop}_A \mid a.P \mid \ldots$$

and generalizing the definitions of alphabets and processes in a straightforward way.

Since processes and specifications are interpreted in the same semantic domain, there is no difficulty in extending readiness semantics to mixed terms. The satisfaction relation on mixed terms, denoted by $\equiv>$, is defined as

$$M_1 \equiv> M_2 \text{ iff } \mathcal{R}^*[\![M_1]\!] \sqsupseteq \mathcal{R}^*[\![M_2]\!],$$

where M_1, M_2 are mixed terms. The relation **sat** can be viewed as a special case of $\equiv>$ restricted to the pairs $P \equiv> S$ where P is a process and S is a specification.

In [Olderog 91] a proof system for $\equiv>$ is given, which obviously allows us to derive assertions of the form P **sat** S, but mixed terms will be used in the derivations. The inference rules of the proof system for $\equiv>$ are presented in Table 1. In these rules, P, Q, R range over closed mixed terms, S and T — over specifications. Curly braces are used to denote syntactic substitutions.

3 Proof system for sat

The first step towards introducing parametricity into the formalism proposed by Olderog is to provide a proof system for **sat**, which does not appeal to mixed terms. In Table 2 we propose such a proof system, whose rules bear a clear resemblance to Olderog's proof system for $\equiv>$. (From now on, the symbol $\equiv>$ will refer to the syntactic notion of derivability in the proof system of Table 1.)

The judgements of the proof system for **sat** have the form

$$\Gamma \vdash P \textbf{ sat } S, \tag{2}$$

where $P \in \mathit{Proc}$, $S \in \mathit{Spec}$ and Γ is a *set of assumptions*,

$$\Gamma = \{X_1 \textbf{ sat } S_1, \ldots, X_n \textbf{ sat } S_n\}.$$

The sets of assumptions will be needed to handle recursion. The special case of (2), with $\Gamma = \emptyset$ will be shown to coincide with Olderog's notion of $P \equiv> S$.

The theorems below show the relative soundness and completeness of the proposed proof system for **sat** with respect to the proof system for $\equiv>$.

To state the theorems we need the following auxiliary notation: for a process term P and $\Gamma = \{X_1 \textbf{ sat } S_1, \ldots, X_n \textbf{ sat } S_n\}$, by $P\{\Gamma\}$ we shall denote a mixed term $P\{S_1/X_1, \ldots, S_n/X_n\}$, obtained from P by replacing free occurrences of variables $X_1, \ldots, X_n$ by specifications $S_1, \ldots, S_n$.

To distinguish the rules of the proof systems for $\equiv>$ and for **sat**, we add index $_{\equiv>}$ to the names of the respective rules of the proof system for $\equiv>$.

Theorem 3 (Relative soundness of proof system for **sat**) *If* $\Gamma \vdash P \textbf{ sat } S$ *then* $P\{\Gamma\} \equiv> S$.

Proof. Follows by by induction on the length of the derivation of $\Gamma \vdash X \textbf{ sat } S$. Note, that some of the assumptions of proof rules in Table 1 are replaced by different but equivalent versions in Table 2. Such equivalent pairs of assumptions are collected in Lemma 4 below. □

(Reflexivity) $\quad P \equiv> P$

(Transitivity) $\quad \dfrac{P \equiv> Q \qquad Q \equiv> R}{P \equiv> R}$

(Context) $\quad \dfrac{Q \equiv> R}{M\{Q/X\} \equiv> M\{R/X\}}$

where M is a mixed term, M$\{Q/X\}$ and M$\{R/X\}$ are closed mixed terms

(Consequence) $\quad \dfrac{\models S \ \Rightarrow\ T}{P \equiv> T}\ ,\quad$ where $\alpha(S) = \alpha(T)$

(Deadlock) $\quad stop_A \equiv> h{\downarrow}_A \leq \varepsilon$

(Prefix) $\quad \dfrac{(\varepsilon, \{a\}) \models S}{a.S\{a.h/h\} \equiv> S}\ ,\quad$ where $a \in \alpha(S)$

(Choice) $\quad S + T \equiv> \varphi_+(S,T)$

where $\varphi_+(S,T) \equiv (h{\downarrow}_A = \varepsilon \ \Rightarrow\ S \wedge T)\ \wedge\ (h{\downarrow}_A \neq \varepsilon \ \Rightarrow\ S \vee T)$
and $\alpha(S) = \alpha(T)$

(Parallel) $\quad S \parallel T \equiv> \varphi_{\parallel}(S,T)$

where $\varphi_{\parallel}(S,T) \equiv \exists G\ \exists H\ (S\{G/F\} \wedge T\{H/F\} \wedge$
$G \cap H \subseteq F \wedge (G \cup H) - A \subseteq F)$
and $A = \alpha(S) = \alpha(T)$

(Renaming) $\quad S[\bar{b}/\bar{a}] \equiv> \varphi_{[]}(S, \bar{b}, \bar{a})$

where $\varphi_{[]}(S, \bar{b}, \bar{a}) \equiv \exists t\ \exists G\ (S\{t/h, G/F\} \wedge t[\bar{b}/\bar{a}] = h{\downarrow}_A \wedge$
$G[\bar{b}/\bar{a}] \subseteq F)$
and $A = \alpha(S)\{\bar{b}/\bar{a}\}$

(Hiding) $$\dfrac{\begin{array}{l}\models S\{F - B/F\} \ \Rightarrow\ T \\ \forall b \in B\ b \models \neg\exists F\ S \\ \forall tr \in \alpha(S)^*\ \exists n \geq 0\ \forall b_1, \ldots, b_n \in B\ tr\, b_1 \ldots b_n \models \neg\exists F\ S\end{array}}{S \setminus B \equiv> T}$$

where $\alpha(T) = \alpha(S) - B$ and $\alpha(S) = A$

(Recursion) $\quad \dfrac{P\{S/X\} \equiv> S}{\mu X.P \equiv> S}\ ,$

where $\mu X.P$ is a communication guarded closed mixed term and $\alpha(S) = \alpha(X)$

Table 1.

(Consequence) $$\frac{\begin{array}{lll} & \models & S \Rightarrow T \\ \Gamma & \vdash & P \textbf{ sat } S \end{array}}{\begin{array}{lll}\Gamma & \vdash & P \textbf{ sat } T\end{array}}, \quad \text{where } \alpha(S) = \alpha(T)$$

(Deadlock) $$\Gamma \;\vdash\; \mathit{stop}_A \textbf{ sat } h{\downarrow}_A \le \varepsilon$$

(Prefix) $$\frac{\begin{array}{lll} & \models & S\{\varepsilon/h, \{a\}/F\} \\ \Gamma & \vdash & P \textbf{ sat } S\{ah/h\} \end{array}}{\begin{array}{lll}\Gamma & \vdash & a.P \textbf{ sat } S\end{array}}, \quad \text{where } a \in \alpha(S)$$

(Choice) $$\frac{\begin{array}{lll} \Gamma & \vdash & P \textbf{ sat } S \\ \Gamma & \vdash & Q \textbf{ sat } T \end{array}}{\begin{array}{lll}\Gamma & \vdash & P + Q \textbf{ sat } \varphi_+(S,T)\end{array}}, \quad \text{where } \alpha(S) = \alpha(T)$$

(Parallel) $$\frac{\begin{array}{lll} \Gamma & \vdash & P \textbf{ sat } S \\ \Gamma & \vdash & Q \textbf{ sat } T \end{array}}{\begin{array}{lll}\Gamma & \vdash & P \parallel Q \textbf{ sat } \varphi_{\parallel}(S,T)\end{array}}$$

(Renaming) $$\frac{\Gamma \;\vdash\; P \textbf{ sat } S}{\Gamma \;\vdash\; P[\bar{b}/\bar{a}] \textbf{ sat } \varphi_{[\,]}(S, \bar{b}, \bar{a})}$$

(Hiding) $$\frac{\begin{array}{lll} & \models & S\{F - B/F\} \Rightarrow T \\ & \models & S \Rightarrow \forall b \in B \; h{\downarrow}_A \neq b \\ & \models & S \Rightarrow \exists n \forall b_1 \ldots b_n \in B^* \; \neg \exists F \; S\{h\, b_1 \ldots b_n/h\} \\ \Gamma & \vdash & P \textbf{ sat } S \end{array}}{\begin{array}{lll}\Gamma & \vdash & P \backslash B \textbf{ sat } T\end{array}},$$

where $\alpha(T) = \alpha(S) - B$ and $\alpha(S) = A$

(Recursion) $$\frac{\Gamma,\; X \textbf{ sat } S \;\vdash\; P \textbf{ sat } S}{\Gamma \;\vdash\; \mu X.P \textbf{ sat } S},$$

where $\mu X.P$ is communication guarded and $\alpha(S) = \alpha(X)$

(Assumption) $$\Gamma,\; X \textbf{ sat } S \;\vdash\; X \textbf{ sat } S, \quad \text{where } \alpha(S) = \alpha(X)$$

Table 2.

Lemma 4

(a) $\models S\{\varepsilon/h, \{a\}/F\}$ *if and only if* $(\varepsilon, \{a\}) \models S$

(b) $\models S \;\Rightarrow\; \forall b \in B\ h{\downarrow_A} \neq b$ *if and only if* $\forall b \in B\ b \models \neg\exists F\ S$

(c)
$$\models \exists n \forall b_1 \ldots b_n \in B^* \ \neg\exists F\ S\{hb_1 \ldots b_n/h\}$$
if and only if
$$\forall tr \in \alpha(S)^*\ \exists n \geq 0\ \forall b_1, \ldots, b_n \in B\ \ tr\, b_1 \ldots b_n \models \neg\exists F\ S$$

Corollary 5 *Proof system for* **sat** *is sound, that is,*

$$\textit{if} \quad \vdash P \textbf{ sat } S \quad \textit{then} \quad \mathcal{R}^*[\![S]\!] \sqsubseteq \mathcal{R}^*[\![P]\!]$$

Proof. Follows from Theorem 3 and soundness of proof system for $\equiv>$. □

The converse of Theorem 3 also holds.

Theorem 6 (Relative Completeness) *Let P be a process and Γ a set of assumptions such that $P\{\Gamma\}$ is a closed term. If $P\{\Gamma\} \equiv> S$ for a specification S then $\Gamma \vdash P$ **sat** S.*

Proof. The proof is long and is omitted due to space limitations. □

We end this section with the following useful observation

Proposition 7 *If for a recursive term P a judgement $\Gamma \vdash P$ **sat** S can be derived for some S and Γ, then $P \in Proc$, that is, P satisfies the restrictions of Definition 2.*

Proof. Follows by induction on the derivation of $\Gamma \vdash P$ **sat** S. □

4 Parametric Processes and Specifications

The triple
$$(\mathit{CProc}, \mathit{Spec}, \textbf{sat})$$
can be viewed as a specification system, in the sense of [Sokołowski 94]. We lift it to the parametric case by extending the process algebra with abstraction and application primitives, in a manner resembling the typed lambda calculus, where types are replaced with specifications.

First we introduce a syntactic class *PSpec* of *parametric specifications* ranged over by σ and defined by the following abstract syntax:

$$\sigma ::= S \mid \sigma \rightarrow \sigma$$

Intuitively, specification $\sigma_1 \rightarrow \sigma_2$ describes a set of parametric processes, each of which when given a process satisfying (possibly parametric) specification σ_1 as an argument yields a process that satisfies specification σ_2.

The syntax of recursive terms is extended by adding two new clauses

$$P ::= \ \ldots \mid \lambda X : \sigma \,.\, P \mid PP$$

We consider the recursive terms up to α conversion, where λ as well as μ are variable binding operators.

The set of *parametric processes* will be defined as a subset of so extended recursive terms. In the case of nonparametric processes, syntactic restrictions enumerated in Definition 2 were used to distinguish the set *Proc* of processes in the set of (nonparametric) recursive terms. However, as it has been noted in Proposition 7, these restrictions are encoded also in proof system for **sat**. In the parametric case, it is more convenient to adopt a counterpart of Proposition 7 as a definition rather than separately formulate syntactic restrictions that define the desired set of parametric processes. This is the approach we are going to take, but first we need to refine some notions.

The notions of alphabet of a process and alphabet of a specification have to be extended to the parametric case. So far the alphabets were just finite subsets of *Comm*, the set of communications. Alphabets of parametric processes (and specifications), we shall call them *hyper-alphabets* in the sequel, have to be more complex objects, indicating the alphabets of possible arguments of a process. We define the set $\mathcal{H}$ of all hyper-alphabets as follows:

$$\begin{array}{lcl} \mathcal{H}_0 & = & \text{set of finite subsets of } Comm \\ \mathcal{H}_{n+1} & = & \{(A,B) \mid A,B \in \mathcal{H}_n\} \cup \mathcal{H}_n \\ \mathcal{H} & = & \bigcup_{n=0}^{\infty} \mathcal{H}_n \end{array}$$

Now, the hyper-alphabet $\alpha^*(\sigma)$ of a specification σ is defined recursively by the following clauses:

$$\begin{array}{ll} \alpha^*(\sigma) = \alpha(S), & \text{if } \sigma = S \\ \alpha^*(\sigma) = (\alpha^*(\sigma_1), \alpha^*(\sigma_2)), & \text{if } \sigma = \sigma_1 \to \sigma_2 \end{array}$$

The hyper-alphabet $\alpha^*(P)$ of a recursive term P is defined by the clauses given below. If it is not possible to associate an alphabet with a term using these defining clauses, $\alpha^*(P)$ is considered to be undefined. Just as in the non-parametric case we assume that every process variable X has an associated hyper-alphabet $\alpha^*(X)$.

$$\begin{array}{lcll} \alpha^*(X) & = & \text{the hyper-alphabet associated with } X & \\ \alpha^*(stop : A) & = & A & \\ \alpha^*(a\,.\,P) & = & \alpha^*(P) \cup \{a\} & \text{if } \alpha^*(P) \in \mathcal{H}_0 \\ \alpha^*(P+Q) & = & \alpha^*(P) \cup \alpha^*(Q) & \text{if } \alpha^*(P), \alpha^*(Q) \in \mathcal{H}_0 \\ \alpha^*(P \| Q) & = & \alpha^*(P) \cup \alpha^*(Q) & \text{if } \alpha^*(P), \alpha^*(Q) \in \mathcal{H}_0 \\ \alpha^*(P[b/a]) & = & (\alpha^*(P) - \{a\}) \cup \{b\} & \text{if } \alpha^*(P) \in \mathcal{H}_0 \\ \alpha^*(P \backslash a) & = & (\alpha^*(P) - \{a\}) & \text{if } \alpha^*(P) \in \mathcal{H}_0 \\ \alpha^*(\mu X\,.\,P) & = & \alpha^*(P) \cup \alpha^*(X) & \text{if } \alpha^*(P) \in \mathcal{H}_0 \\ \alpha^*(\lambda X : \sigma\,.\,P) & = & (\alpha^*(X), \alpha^*(P)) & \text{if } \alpha^*(X) = \alpha^*(\sigma) \\ \alpha^*(PQ) & = & B & \text{if } \alpha^*(P) = (A,B) \\ & & & \quad \text{and } \alpha^*(Q) = A \end{array}$$

Note that $\alpha^*(P) = \alpha(P)$ for $P \in Proc$ and that alphabets of terms such as $(\lambda X : \sigma.P) + Q$, where a process algebra operator $+$ s applied to a higher order term, are not defined.

In the sequel we will consider only those recursive terms, whose hyper-alphabets are defined, and the notion of recursive terms will refer to such terms from now on. The set of so understood recursive terms can be stratified into two subsets: the subset of terms whose alphabet is in $\mathcal{H}_0$ and to which process constructors $+$, $a.$, $\|$, $\backslash$, $[\,]$ and μ can be applied, and the subset of higher order terms, on which only abstractions and applications can be performed. Note however, that an application PQ, where P is a higher order term, can belong to the former subset and process algebra operators can be applied to it.

The syntactic notion of communication guardedness also needs to be re-defined. Apart from conditions (1) and (2), we add the following point to Definition 1:

(3) X does not lie within a subterm of Q of the form RQ

Now we can formulate the proof system for the satisfaction relation **sat** extended to the parametric case. The judgements will have the form

$$\Gamma \;\vdash\; P \textbf{ sat } \sigma$$

where P is a (parametric) recursive term and the set of assumptions Γ is extended in the obvious way to contain assumptions of the form X **sat** σ.

The proof system contains all the rules collected in Table 2, which were used for nonparametric processes, but this time we assume that S and T range over nonparametric specifications, P and Q range over parametric recursive terms, Γ is extended as explained above, and the side conditions concerning alphabets refer to hyper-alphabets, i.e. α is replaced with α^*. The side condition of communication guardedness in Recursion rule refers now to the modified version of this notion.

Moreover, the proof system of Table 2 is extended with the following rules, which are essentially λ-introduction and λ-elimination rules of the proof system for Church typed λ calculus (see for example [Barendregt 92]), plus a modified Assumption rule, which replaces the Assumption rule from Table 2.

$$\text{(Assumption)} \qquad \Gamma,\, X \textbf{ sat } \sigma \;\vdash\; X \textbf{ sat } \sigma\;, \qquad \text{where } \alpha^*(X) = \alpha^*(\sigma)$$

$$\text{(Abstraction)} \qquad \frac{\Gamma,\, X \textbf{ sat } \sigma \;\vdash\; P \textbf{ sat } \rho}{\Gamma \;\vdash\; \lambda X:\sigma\,.\,P \textbf{ sat } \sigma \to \rho}\,, \qquad \text{where } \alpha^*(X) = \alpha^*(\sigma)$$

$$\text{(Application)} \qquad \frac{\Gamma \;\vdash\; P \textbf{ sat } \sigma \to \rho \qquad \Gamma \;\vdash\; Q \textbf{ sat } \sigma}{\Gamma \;\vdash\; PQ \textbf{ sat } \rho}$$

Definition 8 The set of *parametric processes PProc* consists of those recursive terms P, for which $\Gamma \;\vdash\; P$ **sat** σ can be derived for some Γ and σ. *CPProc*, the set of *closed parametric processes*, is defined as the set of those parametric processes that have no free variables.

Note that the above definitions and Proposition 7 imply that $Proc \subseteq PProc$ and $CProc \subseteq CPProc$.

We have just defined all components of a parametric specification system

$$(CPProc, PSpec, \mathbf{sat})$$

In this report we do not provide semantical account of parametric processes. This can be done by extending the readiness denotational semantics so that parametric processes are interpreted as process-valued functions on processes. Such a construction, which is an adaptation of a general procedure described in [Sokołowski 94], will be reported separately together with soundness proof of the proof system for **sat** . Here, we just show that the proof system respects β reductions.

Since parametric processes can be seen as lambda terms built on top of non-parametric processes, the notion of β-reduction can be defined in a standard manner:

$$(\lambda X : \sigma \,.\, P)\, Q \xrightarrow{\beta} P\{Q/X\}$$

We will use the same arrow $\xrightarrow{\beta}$ to denote β-reducibility in possibly many steps.

The next proposition ensures that **sat** is preserved by β reductions.

Proposition 9 *If* $\Gamma \vdash P$ **sat** σ *and* $P \xrightarrow{\beta} P'$ *then* $\Gamma \vdash P'$ **sat** σ.

Proof. By induction on the derivation of $\Gamma \vdash P$ **sat** σ. $\Box$

Corollary 10 *If* $P \in PProc$ *and* $P \xrightarrow{\beta} P'$ *then* $P' \in PProc$.

In [Sokołowski 94] a special case of *shallow* parametricity is distinguished. A shallow n-parameter process has the form

$$\lambda X_1 : S_1 \ldots \lambda X_n : S_n \,.\, P \tag{3}$$

This class of parametric processes corresponds to mixed terms of [Olderog 91]. The process term shown above can be syntactically translated into a mixed term

$$P\{S_1/X_1, \ldots, S_n/X_n\}$$

Conversely, a mixed term P can be translated into a process having the form (3) by replacing every occurrence of any specification S_i appearing in P with a fresh variable X_i.

The following proposition shows that there is an exact match between the proof systems for $\equiv>$ on mixed terms and the proof system for **sat** on the restricted class of shallowly parametric processes.

Proposition 11 $\vdash \lambda X_1 : S_1 \ldots \lambda X_n : S_n \,.\, P$ **sat** $S_1 \to \cdots \to S_n \to S$ *if and only if* $P\{S_1/X_1, \ldots, S_n/X_n\} \equiv> S$.

Proof. It can be show by induction that

$$\vdash \lambda X_1 : S_1 \ldots \lambda X_n : S_n \,.\, P \ \mathbf{sat}\ S_1 \to \cdots \to S_n \to S$$

if and only if

$$S_1 \ \mathbf{sat}\ X_1, \ldots, S_n \ \mathbf{sat}\ X_n \vdash P \ \mathbf{sat}\ S.$$

By Theorems 3 and 6 this is equivalent to $P\{S_1/X_1, \ldots, S_n/X_n\} \equiv> S$. $\Box$

5 Example

We present here small example to illustrate the use of the proposed formalism: a task of constructing a counter of capacity 2. *CNT2* below is a readiness specification of such a counter. Upon communication *up* (*dn*) the value stored in the counter is increased (decreased) by 1. The symbol $up\#h$ denotes the number of *up* communications in trace h.

$$\begin{aligned} \mathit{CNT2} \;\equiv\; & (0 \leq up\#h - dn\#h \leq 2) \wedge (up\#h - dn\#h < 2 \;\Rightarrow\; up \in F) \wedge \\ & (up\#h - dn\#h > 0 \;\Rightarrow\; dn \in F) \end{aligned}$$

The construction of process *cnt2* such that

$$\vdash \mathit{cnt2}\ \mathbf{sat}\ \mathit{CNT2}$$

can be decomposed into constructing a counter of capacity 1, *cnt1*, and a parametric process *mk-cnt2* that constructs a counter of capacity 2 given a counter of capacity 1:

$$\vdash \mathit{mk\text{-}cnt2}\ \mathbf{sat}\ \mathit{CNT1} \rightarrow \mathit{CNT2}$$

$$\vdash \mathit{cnt1}\ \mathbf{sat}\ \mathit{CNT1}$$

CNT1 is a specification of a counter of capacity 1:

$$\begin{aligned} \mathit{CNT1} \;\equiv\; & (0 \leq up\#h - dn\#h \leq 1) \wedge (up\#h - dn\#h = 0 \;\Rightarrow\; up \in F) \wedge \\ & (up\#h - dn\#h = 1 \;\Rightarrow\; dn \in F) \end{aligned}$$

Finally, the processes that realize the specifications can be defined as follows:

$$\begin{aligned} \mathit{mk\text{-}cnt2} &\equiv \lambda X : \mathit{CNT1} . (X[lk/up] \| X[lk/dn]) \backslash lk \\ \mathit{cnt1} &\equiv \mu X . up.dn.X \\ \mathit{cnt2} &\equiv \mathit{mk\text{-}cnt2}\ \mathit{cnt1} \end{aligned}$$

Using the proof system for parametric processes we can derive

$$\vdash \mathit{cnt2}\ \mathbf{sat}\ \mathit{CNT2}$$

as well as the remaining satisfaction assertions above.

Acknowledgement

We would like to thank Thorsten Altenkirch, Kārlis Černās and Uno Holmer for helpful comments on this work.

References

[Barendregt 92] H.P. Barendregt, *Lambda Calculi with Types*, in: Handbook of Logic in Computer Science, vol. 2 (S. Abramsky, Dov M. Gabbay, T. S. E. Maibaum, Eds.) pp. 117–309, Calderon Press, Oxford 1992.

[Holmström 89] S. Holmström, A refinement calculus for specification in Hennessy-Milner logic with recursion, Formal Aspects of Computing, vol. 1 (3). pp. 242–272, 1989.

[Larsen and Milner 86] K. Larsen and R. Milner, A Complete Protocol Verification using Relativised Bisimulations, R 86-12, Institute of Electrical Systems, Aalborg University Centre, 1986.

[Milner 89] R. Milner, *Communication and Concurency*, Prentice Hall, 1989.

[Olderog 91] E.-R. Olderog, *Nets, Terms and Formulas*, Cambridge University Press, Cambridge 1991.

[Olderog 91a] E.-R. Olderog, Towards a design calculus for communicating programs, in: Proc. CONCUR '91, LNCS, Springer, 1991.

[Sokołowski 94] S. Sokołowski, The GDM approach to specifications and their realizations. Part I: Specification systems. Technical Report, Gdańsk, 1994.

On the Formal Derivation of a FEAL Microprocessor

R. Rukšėnas* K. Sere† Y. Zhao*

Abstract

We present an outline of a method for *formal* derivation of asynchronous VLSI circuits. The proposed method focuses on transformational style of the design and it uses techniques familiar from the construction of parallel programs. *Refinement calculus* and *action systems* are used as a framework for the design process. As a case study we look at the derivation of an asynchronous encryption/decryption microprocessor.

1 Introduction

The paper describes ongoing work on exploring a methodology for *formal* derivation of asynchronous delay-insensitive VLSI circuits within the *refinement calculus* and the *action system* framework. It is aimed to be used in the design of *application-specific* circuits. The basic idea is to apply techniques familiar from the construction of parallel programs to VLSI design. This approach was originally taken by Martin [9] who has developed a methodology for designing asynchronous VLSI circuits as concurrent programs within the CSP-framework. Using his method he has specified and derived a number of nontrivial circuits. Delay-insensitive, asynchronous chips were then produced from the concurrent programs that can be seen as equivalent to the parallel composition of action systems that we derive here. However, the method of Martin is semiformal and the transformations carried out during the design process are not formally proved to be correct. We want to provide a completely formal basis for Martin's method.

An *action system* is a parallel or distributed program where parallel activity is described in terms of events, so called actions. Several actions can be executed in parallel, as long as the actions do not share any variables. A recent extension of the action system framework, adding procedure declarations to action systems [5], gives us a very general mechanism for communication between action systems. The action systems formalism was proposed by Back and Kurki-Suonio [3].

The *refinement calculus* is a formalization of the stepwise refinement method of program construction. It was originally proposed by Back [1] and it has been later studied and extended by several researchers, see [11, 12] among others.

*Åbo Akademi University, Department of Computer Science, FIN-20520 Turku, Finland, emails: {rruksena,zyi}@aton.abo.fi

†University of Kuopio, Department of Computer Science and Applied Mathematics, FIN-70211 Kuopio, Finland, email: Kaisa.Sere@uku.fi, fax: +358-71-162595

Originally, the refinement calculus was designed as a framework for systematic derivation of sequential programs only. Back and Sere [4, 13] extended the refinement calculus for the design of action systems and hence, it was possible to handle *parallel algorithms* within the calculus. Back [2] made yet another extension to the calculus showing how *reactive programs* could be derived in a stepwise manner within it relying heavily on the work done in data refinement.

Action systems and the refinement calculus approach has already proved to be suitable for VLSI circuit specification and design on logical behaviour level [6]. Now we want to explore its applicability in the design on rather low (signal) level.

The refinement calculus has been also formalised within a mechanical theorem prover by Back and von Wright [7]. This gives us the tool to mechanise the design process and prove correctness of the derived circuit. A somewhat related method and formalism is developed in [14], but the emphasis is put on the verification of and formal models for delay-insensitive circuits.

As a case study we look at the design of an asynchronous FEAL (Fast Encipherment Algorithm) processor [10]. We start from the sequential algorithm which describes logical behaviour of the encryption processor. Our goal is to identify and isolate the basic functional components of the circuit into action systems of their own. The component action systems are joint together in a parallel composition, where they interact with each other using remote procedure calls. Synchronous communication between different componnets of the circuit is achieved through the procedures mechanism [6].

Overview of the paper In section 2, we briefly describe the action systems formalism. In section 3, we describe how action systems are composed into parallel systems. We also briefly mention the refinement calculus. In section 4, we describe the the target processor and give an initial specification for it as a sequential program. In section 5, this specification is stepwise turned into a parallel composition of action systems, where each action system represents one basic functional component of the circuit. In section 6, we introduce real parallelism into our system by creating 3-level pipeline. Finally in section 7, we conclude with some remarks on the proposed method and further steps in the derivation process.

2 Action systems

An *action system* (with procedures) is a statement of the form

$$\begin{array}{ll} \mathcal{A} :: & \textbf{var } v; \textbf{proc } w \bullet \\ & |[\ \textbf{var } x_1, \ldots, x_h := x_10, \ldots, x_h0; \textbf{proc } p_1 = P_1;\ \ldots; p_n = P_n; \\ & \quad \textbf{do } A_1 \ [\!]\ \ldots\ [\!]\ A_m \textbf{ od} \\ &]|: z \end{array}$$

The identifiers $x_1, \ldots, x_h$ are the variables declared in $\mathcal{A}$ and initialized to $x_10, \ldots, x_h0$, $p_1, \ldots, p_n$ are the *procedure headers*, and P_i is the *procedure body* of p_i, $i = 1, \ldots, n$. Within the loop, $A_1, \ldots, A_m$ are the *actions* of $\mathcal{A}$. Finally,

z, v and w are pairwise distinct lists of identifiers. The list z is the *import list*, indicating which variables and procedures are referenced, but not declared in $\mathcal{A}$. The lists v and w are the *export lists*, indicating which variables and procedures declared in $\mathcal{A}$ are accessible from other action systems. Procedure bodies and actions can be arbitrary statements, and may contain procedure calls.

The *guard* of a program statement S is the condition gS, defined by

$$gS \;=\; \neg \mathrm{wp}(S, \mathit{false})$$

where wp is the *weakest precondition* predicate transformer of Dijkstra [8]. Both procedure bodies and actions will in general be *guarded commands*, i.e., statements of the form

$$C = g \rightarrow S,$$

where g is a boolean condition and S is a program statement. In this case, the guard of C is $g \wedge \neg \mathrm{wp}(S, \mathit{false})$.

Enabledness of an action Let $p = (b \rightarrow T)$ be a procedure and let $a \rightarrow S; p$ be an action that calls on p. Then the enabledness of this action is determined by the value of the action guard

$$g(a \rightarrow S;\, p) \;=\; a \wedge g(S;\, p)$$

If a procedure or action contains a call to a procedure that is not declared in the action system, then the behavior of the action system will depend on the way in which the procedures are declared in some other action system, which constitutes the environment of the action system as will be described below.

3 Composing and refining action systems

Consider two action systems,

$$\begin{array}{ll} \mathcal{A} :: & \mathbf{var}\ v;\ \mathbf{proc}\ r\ \bullet \\ & |[\ \ \mathbf{var}\ x := x0;\ \mathbf{proc}\ p_1 = P_1;\ \ldots;\ p_n = P_n; \\ & \quad\ \mathbf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_m\ \mathbf{od} \\ &]|\!:\ z \end{array}$$

and

$$\begin{array}{ll} \mathcal{B} :: & \mathbf{var}\ w;\ \mathbf{proc}\ s\ \bullet \\ & |[\ \ \mathbf{var}\ y := y0;\ \mathbf{proc}\ q_1 = Q_1;\ \ldots;\ q_l = Q_l; \\ & \quad\ \mathbf{do}\ B_1\ [\!]\ \ldots\ [\!]\ B_k\ \mathbf{od} \\ &]|\!:\ u \end{array}$$

where $x \cap y = \emptyset$, $v \cap w = \emptyset$, and $r \cap s = \emptyset$. Furthermore, the local procedures declared in the two action systems are required to be distinct.

We define the *parallel composition* $\mathcal{A} \parallel \mathcal{B}$ of $\mathcal{A}$ and $\mathcal{B}$ to be the action system $\mathcal{C}$

$$
\begin{array}{ll}
\mathcal{C} :: & \textbf{var } b; \textbf{ proc } c \bullet \\
& |[\ \textbf{var } x, y := x0, y0; \\
& \quad \textbf{proc } p_1 = P_1;\ \ldots; p_n = P_n;\ q_1 = Q_1;\ \ldots; q_l = Q_l; \\
& \quad \textbf{do } A_1 \ [\!]\ \ldots\ [\!]\ A_m\ [\!]\ B_1\ [\!]\ \ldots\ [\!]\ B_k \textbf{ od} \\
&]|: a
\end{array}
$$

where $a = z \cup u - (v \cup r \cup w \cup s), b = v \cup w, c = r \cup s$.

Thus, parallel composition will combine the state spaces of the two constituent action systems, merging the global variables and global procedures and keeping the local variables distinct. The imported identifiers denote those global variables and/or procedures that are not declared in either $\mathcal{A}$ or $\mathcal{B}$. The exported identifiers are the variables and/or procedures declared global in $\mathcal{A}$ or $\mathcal{B}$. The procedure declarations and the actions in the parallel composition consists of the procedure declarations and and actions in the original systems.

Enabledness We permit procedure bodies to have guards that are not identically true. Hence, it is possible that an action which is enabled, calls a procedure in another action system, which then turns out not to be enabled in the state in which it is called. This situation is then the same as if the calling action had not been enabled at all, and had therefore never initiated the call. In other words, the enabledness of an action is determined by the enabledness of the whole statement that is invoked when the action is executed, including enabledness of all procedures that might be invoked.

Decomposing action systems Given an action system

$$
\begin{array}{ll}
\mathcal{C} :: & \textbf{var } u; \textbf{ proc } s \bullet \\
& |[\ \textbf{var } v := v0; \\
& \quad \textbf{do } C_1 \ [\!]\ \ldots\ [\!]\ C_n \textbf{ od} \\
&]|: z
\end{array}
$$

we can *decompose* it into smaller action systems by parallel composition. This means that we split the variables, actions and procedures of $\mathcal{C}$ into disjoint sets so that

$$
\begin{array}{ll}
\mathcal{C} = & \textbf{var } u; \textbf{ proc } s \bullet \\
& |[\ \textbf{var } w := w0; \textbf{ proc } r = R; \\
& \quad \mathcal{A} \parallel \mathcal{B} \\
&]|: z
\end{array}
$$

where

$$
\begin{array}{ll}
\mathcal{A} :: & \textbf{var } a_2; \textbf{ proc } a_3 \bullet \\
& |[\ \textbf{var } x := x0; \textbf{ proc } p = P; \\
& \quad \textbf{do } A_1 \ [\!]\ \ldots\ [\!]\ A_m \textbf{ od} \\
&]|: a_1
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{B} :: & \textbf{var } b_2; \textbf{ proc } b_3 \bullet \\
& |[\ \textbf{var } y := y0; \textbf{ proc } q = Q; \\
& \quad \textbf{do } B_1 \ [\!]\ \ldots\ [\!]\ B_k \textbf{ od} \\
&]|: b_1
\end{array}
$$

The reactive components $\mathcal{A}$ and $\mathcal{B}$ interact with each other via the global variables and procedures included in the lists a_2, a_3, b_2, b_3.

Hiding and revealing Let

$$\textbf{var } v_1, v_2; \textbf{ proc } v_3, v_4 \bullet \mathcal{A} : z$$

be an action system of the form above, where z denotes the import list and v_1, v_2, v_3, v_4 denote the export lists. We can *hide* some of the exported global variables (v_2) and procedure names (v_4) by removing them from the export list,

$$\mathcal{A}' = \textbf{var } v_1; \textbf{ proc } v_3 \bullet \mathcal{A} : z.$$

Hiding the variables v_2 and procedure names v_4 makes them inaccessible from other actions outside $\mathcal{A}'$ in a parallel composition. Hiding thus has an effect only on the variables and procedures in the export list. The opposite operation, *revealing*, is also useful.

In connection with the parallel composition below we will follow the following convention. Let **var** a_1; **proc** $a_2 \bullet \mathcal{A} : a_3$ and **var** b_1; **proc** $b_2 \bullet \mathcal{B} : b_3$ be two action systems. Then their parallel composition is the action system

$$\textbf{var } a_1 \cup b_1; \textbf{ proc } a_2 \cup b_2 \bullet \mathcal{A} \parallel \mathcal{B} : c$$

where $c = a_3 \cup b_3 - (a_1 \cup a_2 \cup b_1 \cup b_2)$ according to the definition above. Hence, the parallel composition exports all the variables and procedures exported by either $\mathcal{A}$ or $\mathcal{B}$. Sometimes there is no need to export all these identifiers, i.e., when they are exclusively accessed by the two component action systems $\mathcal{A}$ and $\mathcal{B}$. This effect is achieved with the following construct that turns out to be extremely useful later:

$$\textbf{var } v; \textbf{ proc } p \bullet |[\, \mathcal{A} \parallel \mathcal{B} \,]| : c$$

Here the identifiers v and p are as follows: $v \subseteq a_1 \cup b_1$ and $p \subseteq a_2 \cup b_2$.

Refining action systems Most of the steps we will carry out within the microprocessor derivation are purely syntactic decomposition steps. There are, however, a couple of steps where a higher level action system is refined into an other action system. These steps are formally carried out within the refinement calculus, where we consider action systems as ordinary statements, i.e., as initialized iteration statements.

The refinement calculus is based on the following definition. Let S and S' be two statements. Then S is correctly *refined* by S', denoted $S \leq S'$, if for any postcondition Q

$$\mathrm{wp}(S, Q) \Rightarrow \mathrm{wp}(S', Q).$$

4 FEAL algorithm and initial specification of the processor

Algorithm As a case study we look at the design of an asynchronous encryption/decryption (FEAL) processor that implements a special case of the so called FEAL-N algorithm [10]. The algorithm takes plaintext (64-bit word) as an input and produces encrypted text of the same length. While encrypting it uses 16-bit keys that are computed from the original one given by the user. We here assume that there are available (when needed) $N + 8$ keys in the so called key memory.

Encryption of input word T is given by the following set of equations:

$$L_0.R_{-1} = K_{N..N+3} \oplus T \tag{1}$$
$$R_0 = R_{-1} \oplus L_0 \tag{2}$$

for $i \in 1..N$:

$$R_i = L_{i-1} \oplus F(R_{i-1}, K_{i-1}) \tag{3}$$
$$L_i = R_{i-1} \tag{4}$$

$$L_{N+1} = R_N \oplus L_N \tag{5}$$
$$C = K_{N+4..N+7} \oplus R_N.L_{N+1} \tag{6}$$

Here C stands for the output (encrypted text). A concatenation of two words W_1 and W_2 is writen as $W_1.W_2$, while exclusive-or (for W_1, W_2 of the equal length) is denoted as $W_1 \oplus W_2$. The word under encryption is then represented as a concatenation $L.R$ of the left and right part (both 32-bit long). Furthermore, let $K_{m..n}$ be short for $K_m.K_{m+1}...K_n$ with K_j being j-th key. We assume that function F is available and do not consider its implementation. Decryption algorithm has the same structure only the keys are used in different order.

The parameter N is the number of rounds in the main encryption loop (Equations 3, 4). We consider the case with $N = 8$. Some extra processing of the word is done before the loop (Equations 1, 2) and after (Equations 5, 6) it. So the algorithm consists of three logical parts.

Structure of the microprocessor We make the design decision to implement each logical part of the algorithm by a separate functional component in the target microprocessor. When processing one word the components are activated one after the other without any possibility for simultaneous activity. However, the processor is supposed to work with a sequence of words. In this case concurrent activity of all its parts is possible. The solution is to have a 3-level pipeline system. To keep input of the second and third components unchanged until the processing of data has been completed and result passed to the next component we introduce two registers (see Figure 1). All components of the circuit communicates through the indicated channels.

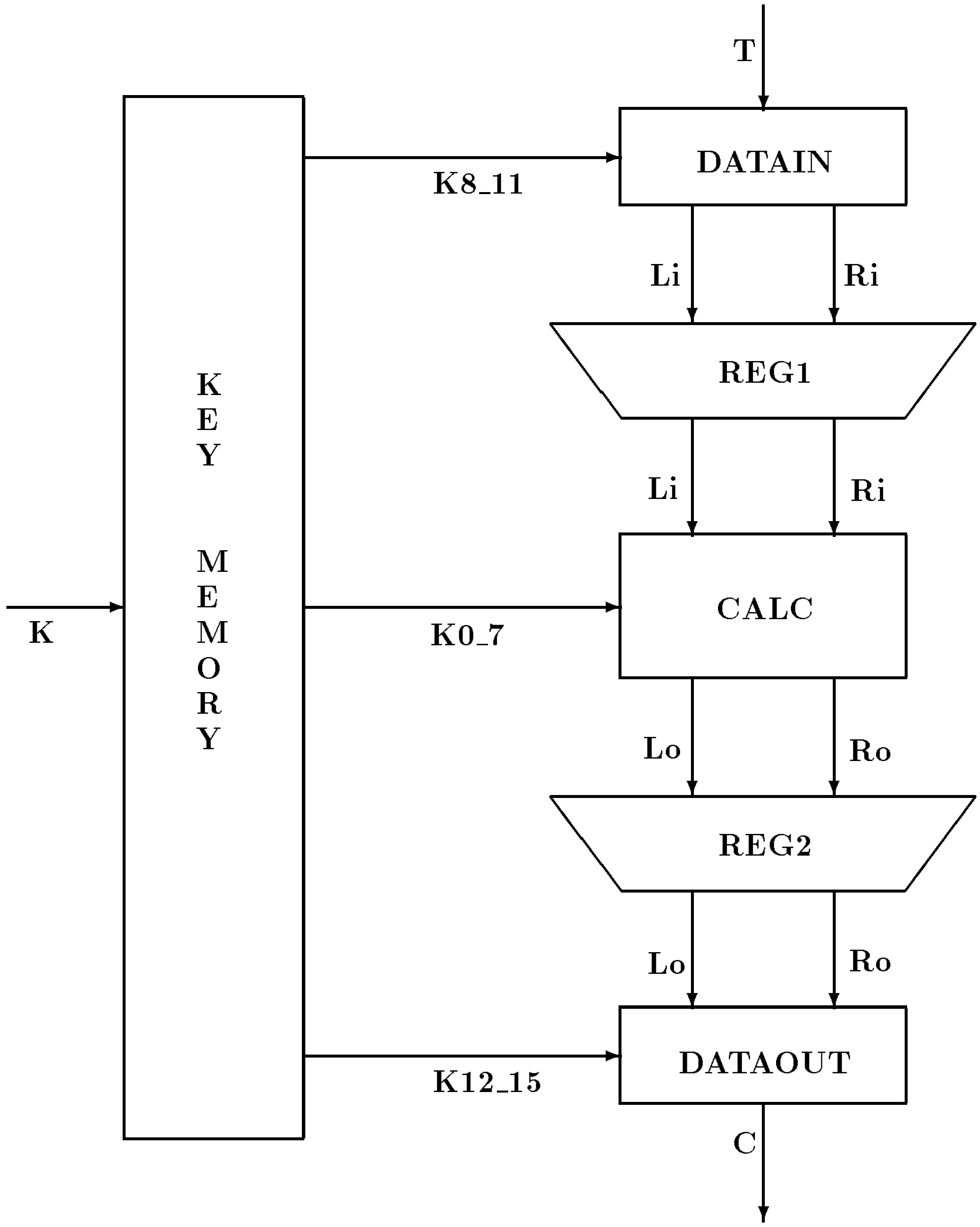


Figure 1: Structure of the microprocessor

We start from the sequential action system that implements above algorithm in the procedure $FEAL$:

$\mathcal{F}0$:: **proc** $FEAL$ $\bullet$
  |[**var** L_i,R_i , R_o,L_o , T ;
   proc $FEAL$(t)=
     ($T:=t$;
      |[**var** $R_{\underline{i}},L_{\underline{o}},L[j],R[j]$ **for** $j=0,\ldots,8$;
       $L_i.R_{\underline{i}}:= K8..11 \oplus T$;
       $R_i:= L_i \oplus R_{\underline{i}}$;
       $L[0],R[0]:= L_i,R_i$;
       [$L[j],R[j]:= R[j\text{-}1]$, $L[j-1] \oplus F(R[j\text{-}1],K0..7[16j\text{-}15..16j])$]
          for $j=0,\ldots,8$;
       $R_o,L_o := R[8]$, $L[8]$;
       $L_{\underline{o}}:= R_o \oplus L_o$;
       $C:= K12..15 \oplus R_o.L_{\underline{o}}$
      ]|)
  ]|:C , K

In our target program each component of the processor is represented as an action system of its own. Parallel composition of them models the entire system where all elements can be active at the same time. Communication on the channels (synchronization of the action systems) is expressed as a remote procedure calls. We do not consider a derivation of any particular component of the processor here, instead we look more carefully at the task of isolating the basic functional parts and modelling communication through the channels on the low (signal) level.

5 Decomposition into parallel action systems

As described above, the algorithm consists of three parts. We therefore start by making three procedures to represent them:

$\mathcal{F}1$:: **proc** $FEAL$ $\bullet$
  |[**var** L_i,R_i , R_o,L_o , T ;
   proc $INXOR$(**var** l_i,r_i)=(|[**var** $R_{\underline{i}}$; $l_i.R_{\underline{i}}:= K8..11 \oplus T$; $r_i:= l_i \oplus R_{\underline{i}}$]|) ;
   proc $CIPHER$(l_i,r_i,**var** r_o,l_o)=
     (|[**var** $L[j],R[j]$ **for** $j=0,\ldots,8$;
       $L[0],R[0]:= l_i,r_i$;
       [$L[j],R[j]:= R[j\text{-}1]$, $L[j-1] \oplus F(R[j\text{-}1],K0..7[16j\text{-}15..16j])$]
          for $j=0,\ldots,8$;
       r_o , $l_o := R[8]$, $L[8]$
      ]|) ;
   proc $OUTXOR$(r_o , l_o)=(|[**var** $L_{\underline{o}}$; $L_{\underline{o}}:= r_o \oplus l_o$; $C:= K12..15 \oplus r_o.L_{\underline{o}}$]|) ;
   proc $FEAL(t)$=
     ($T:= t$; $INXOR(L_i,R_i)$; $CIPHER(L_i,R_i,R_o,L_o)$; $OUTXOR(R_o,L_o)$)
  ]|:C , K

We have that $\mathcal{F}0 \leq \mathcal{F}1$.

Now the encryption itself is done in procedure *CIPHER*, while *INXOR* performs preparatory processing of the word and *OUTXOR* ends whole process.

Parallelize F There is no parallel activity possible in the above action system. Hence, we proceed by creating a parallel composition of action systems which contains actions that can be executed simultaneously.

First we split the procedure *FEAL* into three parts by creating new procedures *DIN*, *CALC* and *DOUT*. *DIN* isolates procedure *INXOR* from the rest while *CALC* and *DOUT* separate procedures *CIPHER* and *OUTXOR*. More formally this can be represented as the following sequence of refinements:

$(T{:=}\ t\ ;\ INXOR(L_i\ ,R_i);\ CIPHER(L_i,R_i,R_o,L_o);\ OUTXOR(R_o,L_o))$
$\leq$
proc $DIN(l_i,r_i){=}(INXOR(l_i,r_i)\ ;\ CIPHER(L_i,R_i,R_o,L_o);\ OUTXOR(R_o,L_o))\ ;$
$(T{:=}\ t\ ;\ DIN(L_i,R_i))$
$\leq$
proc $DIN(l_i,r_i){=}(INXOR\ (l_i,r_i)\ ;\ CALC(l_i,r_i))\ ;$
proc $CALC(l_i,r_i){=}(CIPHER(l_i,r_i,R_o,L_o)\ ;\ OUTXOR(R_o,L_o))\ ;$
$(T{:=}\ t\ ;\ DIN(L_i,R_i))$
$\leq$
proc $DIN(l_i,r_i){=}(INXOR(l_i,r_i)\ ;\ CALC(l_i,r_i))\ ;$
proc $CALC(l_i,r_i){=}(CIPHER(l_i,r_i,R_o,L_o)\ ;\ DOUT(R_o,L_o))\ ;$
proc $DOUT(r_o\ ,\ l_o){=}(OUTXOR(r_o\ ,\ l_o))\ ;$
$(T{:=}\ t\ ;\ DIN(L_i,R_i))$

Now a new action system is as below:

$\mathcal{F}2$:: **proc** *FEAL* $\bullet$
$[\![$ **var** $L_i,R_i\ ,\ R_o,L_o\ ,\ T\ ;$
proc $INXOR($**var** $l_i,r_i){=}(|[$ **var** $R_{\underline{i}}\ ;\ l_i.R_{\underline{i}}{:=}\ K8..11 \oplus T\ ;\ r_i{:=}\ l_i \oplus R_{\underline{i}}\ ]|)\ ;$
proc $CIPHER(l_i,r_i,$**var** $r_o,l_o){=}$
$(|[$**var** $L[j],R[j]$ **for** $j{=}0,\ldots,8\ ;$
$L[0],R[0]{:=}\ l_i,r_i\ ;$
$[\ L[j],R[j]{:=}\ R[j\text{-}1]\ ,\ L[j-1] \oplus F(R[j\text{-}1],K0..7[16j\text{-}15..16j])\ ]$
for $j{=}0,\ldots,8\ ;$
$r_o\ ,\ l_o\ {:=}R[8]\ ,\ L[8]$
$]|\ )\ ;$
proc $OUTXOR(r_o\ ,\ l_o){=}(|[$ **var** $L_{\underline{o}}\ ;\ L_{\underline{o}}{:=}\ r_o \oplus l_o\ ;\ C{:=}\ K12..15 \oplus r_o.L_{\underline{o}}\ ]|)\ ;$
proc $DIN(l_i,r_i){=}(INXOR(l_i,r_i)\ ;\ CALC(l_i,r_i))\ ;$
proc $CALC(l_i,r_i){=}(CIPHER(l_i,r_i,R_o,L_o)\ ;\ DOUT(R_o,L_o))\ ;$
proc $DOUT(r_o\ ,\ l_o){=}(OUTXOR(r_o\ ,\ l_o));$
proc $FEAL(t){=}(\ T{:=}\ t\ ;\ DIN(L_i,R_i))$
$]\!]{:}C\ ,\ K$

We have also proved the refinement relation $\mathcal{F}1 \leq \mathcal{F}2$.

Next we can relax the atomicity constraints as follows. First we add and initialize two new variables *in* and *ca*:

```
𝓕3:: proc FEAL •
  |[ var Li,Ri , Ro,Lo , T ; ic , co ∈ boolean ;
    proc INXOR(var li,ri)=(|[ var R_i ; li.R_i:= K8..11⊕T ; ri:= li⊕R_i ]|) ;
    proc CIPHER(li,ri,var ro,lo)=
          (|[var L[j],R[j] for j=0,...,8 ;
             L[0],R[0]:= li,ri ;
             [ L[j],R[j]:= R[j-1] , L[j − 1]⊕F(R[j-1],K0..7[16j-15..16j]) ]
                    for j=0,...,8 ;
            ro , lo :=R[8] , L[8]
           ]| ) ;
    proc OUTXOR(ro , lo)=(|[ var L_o ; L_o:= ro⊕lo ; C:= K12..15⊕ro.L_o ]|) ;
    proc DIN(li,ri)=( ¬in → INXOR(li,ri) ; in:= true ) ;
    proc CALC(li,ri)=( ¬ca → CIPHER(li,ri,Ro,Lo) ; ca:= true ) ;
    proc DOUT(ro , lo)=(OUTXOR(ro , lo)) ;
    proc FEAL(t)=(T:= t ; DIN(Li,Ri)) ;
    in , ca :=false , false ;
     do
        in → CALC(li,ri) ; in:= false
      [] ca → DOUT(Ro,Lo) ; ca:= false
    od
  ]|:K , C
```

We have that $\mathcal{F}3 \leq \mathcal{F}3$.

Both *DIN* and *CALC* can be viewed as consisting of two parts. The first one execute procedures *INXOR* and *CIPHER* correspondingly. Then the control is returned to the caller. Later on the procedures *CALC* and *DOUT* are called from separate actions that can be enabled simultaneously. The effect of these refinements is as if we would have introduced an explicit **return** statement into our language:

```
𝓕4:: proc FEAL •
  |[ var Li,Ri , Ro,Lo , T ;
    proc INXOR(var li,ri)=(|[ var R_i ; li.R_i:= K8..11⊕T ; ri:= li⊕R_i ]|) ;
    proc CIPHER(li,ri,var ro,lo)=
          (|[var L[j],R[j] for j=0,...,8 ;
             L[0],R[0]:= li,ri ;
             [ L[j],R[j]:= R[j-1] , L[j − 1]⊕F(R[j-1],K0..7[16j-15..16j]) ]
                    for j=0,...,8 ;
            ro , lo :=R[8] , L[8]
           ]| ) ;
    proc OUTXOR(ro , lo)=(|[ var L_o ; L_o:= ro⊕lo ; C:= K12..15⊕ro.L_o ]|) ;
    proc DIN(li,ri)=(INXOR(li,ri) ; return ; CALC(li,ri)) ;
    proc CALC(li,ri)=(CIPHER(li,ri,Ro,Lo) ; return ; DOUT(Ro,Lo)) ;
    proc DOUT(ro , lo)=(OUTXOR(ro,lo));
    proc FEAL(t)=(T:= t ; DIN(Li,Ri))
  ]|:C , K
```

Hence we have that $\mathcal{F}3 = \mathcal{F}4$.

The effect of carrying out $\mathcal{F}3$ is the same as that of $\mathcal{F}4$. The **return** statement is merely syntactic sugaring.

Separation of DATAIN, CALC and DATAOUT Let us now decompose the action system $\mathcal{F}4$ into a parallel composition of three action systems $\mathcal{D}atain$, $\mathcal{C}alc$ and $\mathcal{D}ataout$:

$$\mathcal{F}4 \leq \mathcal{D}atain \parallel \mathcal{C}alc \parallel \mathcal{D}ataout$$

where new action systems are defined as below:

$\mathcal{D}atain$:: **proc** *FEAL* •
  |[**var** L_i,R_i , T ;
    proc *INXOR*=(|[**var** $R_{\underline{i}}$; $L_i.R_{\underline{i}}$:= *K8..11*$\oplus T$; R_i:= $L_i \oplus R_{\underline{i}}$]|) ;
    proc *DIN*(l_i,r_i)=(L_i,R_i:= l_i,r_i ; *INXOR* ; **return** ; *CALC*(L_i,R_i));
    proc *FEAL*(t)=(T:= t ; *DIN*(L_i,R_i))
  ]|:K , *CALC*

$\mathcal{C}alc$:: **proc** *CALC* •
  |[**var** L_i,R_i,R_o,L_o ;
    proc *CIPHER*=
        (|[**var** $L[j]$,$R[j]$ **for** j=$0,\ldots,8$;
          $L[0]$,$R[0]$:= L_i,R_i ;
          [$L[j]$,$R[j]$:= $R[j\text{-}1]$, $L[j-1] \oplus F(R[j\text{-}1], K0..7[16j\text{-}15..16j])$]
              for j=$0,\ldots,8$;
          R_o,L_o :=$R[8]$, $L[8]$
        ]|) ;
    proc *CALC*(l_i,r_i)=(L_i,R_i:= l_i,r_i ; *CIPHER* ; **return** ; *DOUT*(R_o , L_o))
  ]|:K , *DOUT*

$\mathcal{D}ataout$:: **proc** *DOUT* •
  |[**var** R_o , L_o ;
    proc *OUTXOR*=(|[**var** $L_{\underline{o}}$; $L_{\underline{o}}$:= $R_o \oplus L_o$; C:= *K12..15*$\oplus R_o.L_{\underline{o}}$]|) ;
    proc *DOUT*(r_o , l_o)=(R_o , L_o:= r_o , l_o ; *OUTXOR*)
  ]|:C , K

Now the systems communicate through the global procedures *CALC* and *DOUT*. We have also removed the parameters (l_i,r_i) and (r_o,l_o) from procedures *INXOR*, *CIPHER* and *OUTXOR* and added some extra assignments into procedures *DIN*, *CALC* and *DOUT* to play the role of the now unnecessary parameters.

6 Registers

From three action systems above only $\mathcal{D}atain$ and $\mathcal{D}ataout$ can be executed in parallel. To make all components executable simultaneously, we add two registers between the action systems. New procedures *REG1* and *REG2* are created to keep intermediate results of the computation unchanged while they are used by the next component.

First we refine $\mathcal{D}atain$ to $\mathcal{D}atain'$:

$$
\begin{array}{l}
\ (L_i,R_i{:=}\ l_i,r_i\ ;\ \mathit{INXOR}\ ;\ \mathbf{return}\ ;\ \mathit{CALC}(L_i,R_i)) \\
\leq \\
\ \mathbf{proc}\ \mathit{REG1}(l_i,r_i){=}(\|[\ \mathbf{var}\ L_{\underline{i}},R_{\underline{i}}\ ;\ L_{\underline{i}},R_{\underline{i}}{:=}\ l_i,r_i\ ;\ \mathbf{return}\ ;\ \mathit{CALC}(L_{\underline{i}},R_{\underline{i}})\ ]\|)\ ; \\
\ (L_i,R_i{:=}\ l_i,r_i\ ;\ \mathit{INXOR}\ ;\ \mathbf{return}\ ;\ \mathit{REG1}(L_i,R_i))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{D}atain'{::}\ \mathbf{proc}\ \mathit{FEAL}\ \bullet \\
\quad \|[\ \mathbf{var}\ L_i,R_i\ ,\ T\ ; \\
\qquad \mathbf{proc}\ \mathit{INXOR}{=}(\|[\ \mathbf{var}\ \ R_{\underline{i}};\ L_i.R_{\underline{i}}{:=}\ K8..11{\oplus}T;\ R_i{:=}\ L_i{\oplus}R_{\underline{i}}\ ]\|)\ ; \\
\qquad \mathbf{proc}\ \mathit{FEAL}(t){=}(\ T{:=}\ t\ ;\ \mathit{DIN}(L_i,R_i))\ ; \\
\qquad \mathbf{proc}\ \mathit{DIN}(l_i,r_i){=}(L_i,R_i{:=}\ l_i,r_i\ ;\ \mathit{INXOR}\ ;\ \mathbf{return}\ ;\ \mathit{REG1}(L_i,R_i))\ ; \\
\qquad \mathbf{proc}\ \mathit{REG1}(l_i,r_i){=} \\
\qquad\qquad (\|[\ \ \mathbf{var}\ L_{\underline{i}},R_{\underline{i}}\ ;\ L_{\underline{i}},R_{\underline{i}}{:=}\ l_i,r_i\ ;\ \mathbf{return}\ ;\ \mathit{CALC}(L_{\underline{i}},R_{\underline{i}})\ \ ]\|); \\
\qquad \mathbf{proc}\ \mathit{FEAL}(t){=}(\ T{:=}\ t\ ;\ \mathit{DIN}(L_i,R_i)) \\
\quad]\|{:}K\ ,\ \mathit{CALC}
\end{array}
$$

Then $\mathcal{C}alc$ is refined by $\mathcal{C}alc'$:

$$
\begin{array}{l}
\ (L_i,R_i{:=}\ l_i,r_i\ ;\ \mathit{CIPHER}\ ;\ \mathbf{return}\ ;\ \mathit{DOUT}(R_o\ ,\ L_o)) \\
\leq \\
\ \mathbf{proc}\ \mathit{REG2}(r_o,l_o){=}(\|[\ \mathbf{var}\ R_{\underline{o}},L_{\underline{o}}\ ;\ R_{\underline{o}},L_{\underline{o}}{:=}\ R_o,L_o\ ;\ \mathbf{return}\ ;\ \mathit{DOUT}(R_{\underline{o}},L_{\underline{o}})\ ]\|)\ ; \\
\ (L_i,R_i{:=}\ l_i,r_i\ ;\ \mathit{CIPHER}\ ;\ \mathbf{return}\ ;\ \mathit{REG2}(R_o\ ,\ L_o))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{C}alc'{::}\ \mathbf{proc}\ \mathit{CALC}\ \bullet \\
\quad \|[\ \mathbf{var}\ L_i,R_i\ ,\ R_o,L_o\ ; \\
\qquad \mathbf{proc}\ \mathit{CIPHER}{=} \\
\qquad\qquad (\|[\mathbf{var}\ L[j],R[j]\ \mathbf{for}\ j{=}0,\ldots,8\ ; \\
\qquad\qquad\quad L[0],R[0]{:=}\ L_i,R_i\ ; \\
\qquad\qquad\quad [\ L[j],R[j]{:=}\ R[j\text{-}1]\ ,\ L[j-1]{\oplus}F(R[j\text{-}1],K0..7[16j\text{-}15..16j])\ ] \\
\qquad\qquad\qquad\quad \mathbf{for}\ j{=}0,\ldots,8\ ; \\
\qquad\qquad\quad R_o,L_o\ {:=}R[8]\ ,\ L[8] \\
\qquad\qquad\ ]\|\ )\ ; \\
\qquad \mathbf{proc}\ \mathit{CALC}(l_i,r_i){=}(L_i,R_i{:=}\ l_i,r_i\ ;\ \mathit{CIPHER}\ ;\ \mathbf{return}\ ;\ \mathit{REG2}(R_o\ ,\ L_o))\ ; \\
\qquad \mathbf{proc}\ \mathit{REG2}(r_o,l_o){=} \\
\qquad\qquad (\|[\ \ \mathbf{var}\ R_{\underline{o}},L_{\underline{o}}\ ;\ R_{\underline{o}},L_{\underline{o}}{:=}\ R_o,L_o\ ;\ \mathbf{return}\ ;\ \mathit{DOUT}(R_{\underline{o}},L_{\underline{o}})\ \ ]\|) \\
\quad]\|{:}K\ ,\ \mathit{DOUT}
\end{array}
$$

Finally, we split $\mathcal{D}atain'$ into two components $\mathcal{D}atain''$ and $\mathcal{R}eg1$ as follows:

$$
\begin{array}{l}
\mathcal{D}atain''{::}\ \mathbf{proc}\ \mathit{FEAL}\ \bullet \\
\quad \|[\ \mathbf{var}\ L_i,R_i\ ,\ T\ ; \\
\qquad \mathbf{proc}\ \mathit{INXOR}{=}(\|[\ \mathbf{var}\ \ R_{\underline{i}};\ L_i.R_{\underline{i}}{:=}\ K8..11{\oplus}T;\ R_i{:=}\ L_i{\oplus}R_{\underline{i}}\ ]\|)\ ; \\
\qquad \mathbf{proc}\ \mathit{DIN}(l_i,r_i){=}(L_i,R_i{:=}\ l_i,r_i\ ;\ \mathit{INXOR}\ ;\ \mathbf{return}\ ;\ \mathit{REG1}(L_i,R_i)); \\
\qquad \mathbf{proc}\ \mathit{FEAL}(t){=}(\ T{:=}\ t\ ;\ \mathit{DIN}(L_i,R_i)) \\
\quad]\|{:}K\ ,\ \mathit{REG1}
\end{array}
$$

$\mathcal{R}eg1$:: **proc** *REG1* $\bullet$
$\|[$ **proc** $REG1(l_i,r_i)=$
$(\|[$ **var** $L_{\underline{i}},R_{\underline{i}}$; $L_{\underline{i}},R_{\underline{i}}:= l_i,r_i$; **return** ; $CALC(L_{\underline{i}},R_{\underline{i}})$ $]\|)$
$]\|:K$, $CALC$

Similarly, $\mathcal{C}alc'$ is divided into $\mathcal{C}alc''$ and $\mathcal{R}eg2$:

$\mathcal{C}alc''$:: **proc** *CALC* $\bullet$
$\|[$ **var** L_i,R_i , R_o,L_o ;
proc *CIPHER*=
$(\|[$**var** $L[j],R[j]$ **for** $j=0,\ldots,8$;
$L[0],R[0]:= L_i,R_i$;
[$L[j],R[j]:= R[j\text{-}1]$, $L[j-1]\oplus F(R[j\text{-}1],K0..7[16j\text{-}15..16j])$]
for $j=0,\ldots,8$;
$R_o,L_o := R[8]$, $L[8]$
$]\|$) ;
proc $CALC(l_i,r_i)=(L_i,R_i:= l_i,r_i$; $CIPHER$; **return** ; $REG2(R_o$, $L_o))$
$]\|:K$, $REG2$

$\mathcal{R}eg2$:: **proc** *REG2* $\bullet$
$\|[$ **proc** $REG2(r_o,l_o)=$
$(\|[$ **var** $R_{\underline{o}},L_{\underline{o}}$; $R_{\underline{o}},L_{\underline{o}}:= R_o,L_o$; **return** ; $DOUT(R_{\underline{o}},L_{\underline{o}})$ $]\|)$
$]\|:K$, $DOUT$

As a result, we have derived the action system that models the processor in Figure 1:

$\mathcal{F}5$:: **proc** *FEAL* $\bullet$ $\|[\mathcal{D}atain'' \parallel \mathcal{R}eg1 \parallel \mathcal{C}alc'' \parallel \mathcal{R}eg2 \parallel \mathcal{D}ataout]\|$: C , K

We have also established the refinement relation between initial and final programs:

$$\mathcal{F}0 \leq \mathcal{F}5$$

7 Concluding remarks

The final parallel composition of action systems is a refinement of the initial sequential program. We isolated so far the basic functional components of the processor and introduced two registers thus creating 3-level pipeline system. All manipulations with the program were completely formal and based on the refinement calculus. To achieve this result we mostly used the method of parallel decomposition of an action system. Communication through the channels was modelled by remote procedure calls. Some steps involved a refinement of the atomicity of the system so they were true refinements and required proofs.

Next step in the derivation would be a refinement of the communication between action systems (components of the processor). We intend to implement it by so called handshaking protocol. Then an isolation of control and data parts of the circuit could follow.

References

[1] R. J. R. Back. *On the Correctness of Refinement Steps in Program Development.* PhD thesis, Department of Computer Science, University of Helsinki, Helsinki, Finland, 1978. Report A–1978–4.

[2] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.–P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Proceedings. 1989*, volume 430 of *Lecture Notes in Computer Science.* Springer–Verlag, 1990.

[3] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proc. of the 2nd ACM SIGACT–SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[4] R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming* 13, pages 133–180, 1989.

[5] R. J. R. Back and K. Sere. Action systems with synchronous communication. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi. Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference*, pages 107–126, 1994.

[6] R. J. R. Back and K. Sere. Specification of a Microprocessor. Reports on computer science and mathematics 148, Åbo Akademi, 1994.

[7] R. J. R. Back and J. von Wright. Refinement concepts formalised in higher-order logic. *Formal Aspects of Computing* 2, pages 247–272, 1990.

[8] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.

[9] A. J. Martin. Synthesis of Asynchronous VLSI Circuits. *CalTech*, Technical Report, 1993.

[10] S. Miyaguchi and A. Shimizu. Fast Data Encipherment Algorithm. *EUROCRYPT '87*, pages 267–278, 1987.

[11] C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[12] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

[13] K. Sere *Stepwise Refinement of Parallel Algorithms.* PhD thesis, Department of Computer Science, Åbo Akademi University, Turku, Finland, 1990.

[14] J. Staunstrup and M. R. Greenstreet. *Synchronized Transitions.* IFIP WG 10.5, Summer school on Formal Methods for VLSI Design, Lecture Notes 1990.

Strictness and Totality Analysis

Kirsten Lackner Solberg*
Hanne Riis Nielson and Flemming Nielson
Computer Science Dept.
Aarhus University, Denmark
e-mail: {kls,hrn,fn}@daimi.aau.dk

A full version of the paper can be found in *Proceedings of SAS'94*, LNCS 864 pages 408 – 422, 1994

Strictness analysis has proved useful in the implementation of lazy functional languages as Miranda, Lazy ML and Haskell: when a function is strict it is safe to evaluate its argument before performing the function call. Totality analysis is equally useful but has not be adopted so widely: if the argument to a function is known to terminate then it is safe to evaluate it before performing the function call.

In this talk we present an inference system for performing strictness *and* totality analysis. We restrict our attention to a simply typed lambda-calculus with constants and a fixpoint operator. The inference system is an extension of the usual type system in that we introduce three annotations on types t:

- $!^{\mathbf{b}}$t: the value has type t and it definitely $\bot$,
- $!^{\mathbf{n}}$t: the value has type t and is definitely *not* $\bot$, and
- $!^{\top}$t: the value has type t and it can be any value.

Annotated types can be constructed using the function type constructor and (top-level) conjunction. As an example a function may have the annotated type $(!^{\mathbf{n}}\mathtt{Int} \to !^{\mathbf{n}}\mathtt{Int}) \wedge (!^{\mathbf{b}}\mathtt{Int} \to !^{\mathbf{b}}\mathtt{Int})$ which means that given a terminating argument the function will definitely terminate and given a non-terminating argument it will definitely not terminate. Thus we capture the strictness as well as the totality of the function. Strictness and totality information can also be combined as in $(!^{\mathbf{b}}\mathtt{Int} \to !^{\mathbf{n}}\mathtt{Int} \to !^{\mathbf{n}}\mathtt{Int}) \wedge (!^{\mathbf{n}}\mathtt{Int} \to !^{\mathbf{b}}\mathtt{Int} \to !^{\mathbf{n}}\mathtt{Int})$ $\wedge\ (!^{\mathbf{b}}\mathtt{Int} \to !^{\mathbf{b}}\mathtt{Int} \to !^{\mathbf{b}}\mathtt{Int})$ which will be the annotated type of McCarthy's ambiguity operator.

We give examples of its use and prove the correctness with respect to a natural-style operational semantics.

*Dept. of Math. and Computer Science, Odense University, Denmark

Backward Refinement for Verifying Distributed Algorithms

K. Sere* M. Waldén**

We present a new verification method for distributed algorithms. The basic idea is that an algorithm to be verified is stepwise transformed into a high level specification through a number of correctness-preserving steps. At each step some mechanism of the algorithm is identified and abstracted away while the basic computation in the original algorithm is preserved. In this way the algorithm becomes more coarse-grained. Only the essential parts of the algorithm are then left for final verification.

The method is formalized within the *refinement calculus* [1] using *superposition refinement* [2] in a backward direction.

The idea is as follows. We verify an algorithm through a number of backward refinement steps. Each step can be verified within the refinement calculus using the superposition refinement rule. The correctness of the final algorithm is then easily verified, thereby establishing the correctness of the original algorithm. An extensive case study is described in [4]. An additional contribution of the backward refinement method is that the algorithm will be described as consisting of some basic computation and a number of mechanisms added on top of this.

Our method is closely related to the reduction method of Lipton [3]. In contrast to Lipton, the method presented here is based on a formal calculus, the refinement calculus, for reasoning about programs. The main purpose of the refinement calculus is to provide a basis for the stepwise refinement approach to program construction. Our work shows how this calculus can be used to verify an algorithm.

References

[1] R. J. R. Back. *On the correctness of refinement in program development.* Ph.D. thesis, Report A-1978-4, Department of computer science, University of Helsinki, Finland, 1978.

[2] R. J. R. Back and K. Sere. *Superposition refinement of Reactive Systems.* Series A–144, Reports on Computer Science and Mathematics, Åbo Akademi University, Finland, 1993.

[3] R. J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of th ACM 18*, No 12, pages 717–721, 1975.

[4] K. Sere and M. Waldén. *Verification of a Distributed Algorithm due to Chu.* Manuscript, Department of Computer Science, Åbo Akademi University, Turku, Finland, 1994. Abstract presented at the 13th Symposium on Principles of Distributed Computing (PODC'94), Los Angeles, USA.

*University of Kuopio, Department of Computer Science and Applied Mathematics, P.O.Box 1627, SF-70211 Kuopio, Finland, e-mail: `Kaisa.Sere@uku.fi`

**Åbo Akademi University, Department of Computer Science, SF-20520 Turku, Finland, e-mail: `mwalden@abo.fi`

NONCLAUSAL RESOLUTION SYSTEM FOR BRANCHING TEMPORAL LOGIC

Jūratė Sakalauskaitė
Institute of Mathematics and Informatics
Akademijos 4, 2600 Vilnius, Lithuania
e-mail: jurate.sakalauskaite @mlats.mii.lt

Abstract. We present a proof system for branching propositional temporal logic. The system is based on nonclausal resolution. The system is proved to be complete. The proof of completeness uses tableau construction for the logic.

1. Introduction

Temporal logic is an appropriate formalism to reason about concurrent systems. We consider here the branching propositional temporal logic BPTL, i.e. in underlying model of the logic any instant of time may split into different possible futures. Branching time logic allows to reason about different possible futures. BPTL is a subsystem of branching time logic introduced in [BPM]. The language of BPTL contains the usual propositional connectives (say $\wedge, \vee, \neg, \supset$) and temporal modalities. Time is assumed discrete and branching. In BPTL if u and v range over formulas then

- $\bigcirc\ u$ means "u is true in each next state";
- $\circledcirc\ u$ means "u is true in some next state; in other words $\circledcirc u \equiv \neg\bigcirc\neg u$;
- $\Box\ u$ means "u is always true (from now on)";
- $\Diamond\ u$ means "u is eventually true"; in other words $\Diamond u \equiv \neg\Box\neg u$.

For BPTL a Hilbert style proof system can be obtained as a subsystem of Hilbert style proof system presented in [BPM].

In this paper we present nonclausal resolution proof system for BPTL. Nonclausal resolution has the advantage over the classical clausal resolution of not requiring formulas to be in clause form. The proof of completeness uses tableau construction for BPTL extracted from [BPM]. The idea to use tableau construction to prove completeness of nonclausal resolution proof system is adopted from [AM].

In section 2 we present syntax and semantics of BPTL. In section 3 we introduce a nonclausal resolution system $\mathbb{R}$ and explore soundness issues. In section 4 we present the proof of completeness of $\mathbb{R}$. Section 5 contains some concluding remarks.

2. Syntax and semantics of BPTL

Formulas are defined as usual with the help of propositional connectives $\wedge, \vee, \supset$ and temporal modalities $\bigcirc, \circledcirc, \Box, \Diamond$.

A model for BPTL is a triple (S, P, R), where S is a set of states, P is an assignment of propositional letters to states and R is a binary relation on states such that for each $s \in S$ there is $t \in S$ $(s,t) \in R$. For a propositional letter a and a state $s \in S$, $a \in P(s)$ iff a is true in s. We extend the interpretation over the model to all formulas in BPTL as follows, where $b = (s = s_0, s_1, \ldots, s_i, \ldots)$ is an infinite path through the model such that $s_i R s_{i+1}$.

1. $s \models a$ iff $a \in P(s)$ for a atomic;
2. $s \models \neg p$ iff $s \not\models p$;
3. $s \models p \vee q$ iff $s \models p$ or $s \models q$;
4. $s \models \Box p$ iff $\forall b\ \forall t\ (t \in b$ implies $t \models p)$;
5. $s \models \bigcirc p$ iff $\forall t\ (sRt$ implies $t \models p)$;
6. $s \models \Diamond p$ iff $\exists b\ \exists t\ (t \in b$ implies $t \models p)$;
7. $s \models \circledcirc p$ iff $\exists t\ (sRt$ implies $t \models p)$.

p is satisfiable iff $s \models p$ for some model M and some state s in M. p is valid iff $s \models p$ for each model M and each state s in M.

3. The resolution system for BPTL

In this section we describe a nonclausal resolution proof system $\mathbb{R}$ for BPTL.

3.1 Preliminaries

The sequence of formulas $S_0, \ldots, S_n$ is a *derivation* of S_n from S_0 if, for all i, S_{i+1} is obtained from S_i by rules of the system. We refer to the S_i as a derivation step. In the special case, where $S_n = false$, the sequence $S_0, \ldots, S_n$ is a *refutation* of S_0. A *proof* of the formula w is a refutation of $\neg w$. We write $\vdash w$ to denote that formula w has a proof within $\mathbb{R}$.

The resolution system in this paper consists of *simplification rules* and *deduction rules*. We also use the *recursive-call mechanism* which enables to apply rules to subformulas of a proof step.

We define the set of *conjuncts* of a formula u as follows: u is a conjunct of u, and if u is of the form $u_1 \wedge u_2$, then conjuncts of u_1 and the conjuncts of u_2 are conjuncts of u.

- *Simplification rules*

These rules simplify formulas, or put them in forms where the other rules can be applied. They are all of the form

$$u_1, \ldots, u_m \Rightarrow v.$$

The rule $u_1, \ldots, u_m \Rightarrow v$ can be applied to a formula S_i, if the formulas $u_1, \ldots, u_m$ occur as conjuncts in S_i. Then in order to obtain corresponding derivation step we delete one occurrence of each u_i and add the derived formula v to the conjunction.

- *Deduction rules.*

The deduction rules have a form

$$u_1, \ldots, u_m \longmapsto D.$$

The rule $u_1, \ldots, u_m \longmapsto v$ can be applied to a formula S_i, if the formulas $u_1, \ldots, u_m$ occur as conjuncts in S_i. Then, in order to obtain corresponding derivation step, add the derived formula v to the conjunction.

• *Polarity.*

An occurrence of a subformula has *positive polarity* in a formula if it is embedded in the scope of an even number of explicit or implicit $\neg$'s.

• *The recursive-call mechanism.*

The recursive-call mechanism for BPTL is: if there is a derivation of v from u then we may take S_{i+1} to be S_i, with a positive occurrence of u replaced with v.

Usually, we do not mention the recursive-call mechanism in our proofs. Instead we mention the simplification and deduction rules that enables us to apply.

3.2 Soundness For our proof notion to be meaningful we require that rules be sound, i.e., that they maintain satisfiability: if S_0 is satisfiable, then S_n is satisfiable.

The following lemma is helpful in proving soundness of the system.

Lemma 3.1. If $u \supset v$ is valid and w_1 is the result of replacing in w one positive occurrence of u with v, then $w \supset w_1$ is valid.

Proof. By induction on complexity of w.

Each of simplification rules and deduction rules of $\mathbb{R}$ has the property that

$$(u_1 \wedge \ldots \wedge u_m) \supset v$$

is valid.

This guarantees the soundness of $\mathbb{R}$:

Theorem 3.2(soundness) Assume that $(u_1 \wedge \ldots \wedge u_m) \supset v$ is valid for all simplification rules and all deduction rules. If there is a derivation of S_n from S_0 and S_0 is satisfiable, then S_n is satisfiable.

Proof. By induction on the depth of recursive calls in the derivation.

3.3 rules of the system $\mathbb{R}$

3.3.1 Simplification rules

• *True-false reduction rules*

These rules include

$$\begin{array}{ll} \square\ \text{true} \Rightarrow \text{true} & \Diamond\ \text{true} \Rightarrow \text{true} \\ \square\ \text{false} \Rightarrow \text{false} & \Diamond\ \text{false} \Rightarrow \text{false} \\ \bigcirc\ \text{true} \Rightarrow \text{true} & \odot\ \text{true} \Rightarrow \text{true} \\ \bigcirc\ \text{false} \Rightarrow \text{false} & \odot\ \text{false} \Rightarrow \text{false} \end{array}$$

and the regular true-false reduction rules in propositional logic such as false $\wedge u \Rightarrow$false, etc. .

• *Weakening rules*

$$u, v \Rightarrow u, \quad u, v \Rightarrow v.$$

- *Negation rules*

$$\neg\Box u \Rightarrow \Diamond\neg u, \qquad \neg\Diamond u \Rightarrow \Box\neg u,$$
$$\neg\bigcirc u \Rightarrow \odot\neg u, \qquad \neg\odot u \Rightarrow \bigcirc\neg u.$$
$$\neg(u \vee v) \Rightarrow (\neg u \wedge \neg v),$$
$$\neg(u \wedge v) \Rightarrow (\neg u \vee \neg v),$$
$$\neg\neg u \Rightarrow u.$$

Similar rules are added for the other connectives.

3.3.2 Deduction rules

•*Resolution rule*

To state this rule , we introduce some substitution notation. We write $u < v >$ to indicate that the formula v occurs in u, and then $u < w >$ denotes the result of replacing *exactly one* occurrence of v with w in u. We write $u < v_1, \ldots, v_n >$ to indicate that each of $v_1, \ldots, v_n$ occurs in u ,and then $u < w >$ denotes the result of sequentially replacing exactly one occurrence of each of $v_1, \ldots, v_n$ with w in u.

The nonclausal resolution rule for propositional logic is

$$A < u >, B < u > \longmapsto A < \text{true} > \vee B < \text{false} > .$$

That is, if the formulas $A < u, \ldots, u >$ and $B < u, \ldots, u >$ have a common subformula u, then we can derive the resolvent $A < \text{true} > \vee B < \text{false} >$. This is obtained by replacing certain occurrences of u in $A < u, \ldots, u >$ with true and certain occurrences of u in $B < u, \ldots, u >$ with false, and taking the disjunction of the results. (Here "certain occurrences" means "one or more occurrences").

As noted in [AM] this rule does not carry over to linear propositional temporal logic .It does not carry to branching propositional temporal logic as well. The reason is that while u occurs in both A and B, it need not denote the same truth value in all its occurrences; intuitively, each occurrence of u may refer to different instants of time.

The resolution rule is sound in $\mathbb{R}$ under the following *restriction:* the occurrences of u in A or B that are substituted by true or false, respectively, are all in the scope of the same number of $\bigcirc$'s and $\odot$'s and not in the scope of any $\Box$ or $\Diamond$ in A or B; and all these occurrences of u may be except one are in the scope of only $\bigcirc$'s, each of which occurs with positive polarity.

For example, consider the formulas

$$A: \ \bigcirc \odot \Box p \wedge \Diamond\Box p \quad \text{and}$$
$$B: \ \bigcirc\bigcirc\Box p \wedge \odot\bigcirc\Box p.$$

Taking u to be $\Box p$ the rule allows us to derive the resolvent

$$(\bigcirc \circledcirc \text{ true} \wedge \Diamond\Box p) \vee (\bigcirc\bigcirc\text{false} \wedge \circledcirc\bigcirc\Box p).$$

We only substituted true or false for those occurrences in the scope of $\bigcirc\circledcirc$ and $\bigcirc\bigcirc$. We cannot substitute the other occurrence of $\Box p$ in A, since it is in the scope of $\Diamond$. Also, we cannot substitute the other occurrence of $\Box p$ in B, since we substituted $\Box p$ in $\bigcirc \circledcirc \Box p$ in A, which is in the scope of $\bigcirc\circledcirc$, but not in the scope of $\bigcirc\bigcirc$.

- *Modality rules*

  These are rules to handle formulas in the scope of $\Box$, $\Diamond$, $\bigcirc$, $\circledcirc$.
  - $\Box$ rule: $\Box u \longmapsto u \wedge \bigcirc\Box u$;
  - $\Diamond$ rule: $\Diamond u \longmapsto u \vee \circledcirc\Diamond u$;
  - $\bigcirc\bigcirc$ rule: $\bigcirc u, \bigcirc v \longmapsto \bigcirc(u \wedge v)$;
  - $\circledcirc\bigcirc$ rule: $\circledcirc u, \bigcirc v \longmapsto \circledcirc(u \wedge v)$.

•*Induction rule*

if $\vdash \neg(w \wedge u)$, then

$$w, \Diamond u \longmapsto \Diamond\big(\neg u \wedge \circledcirc(u \wedge \neg w)\big).$$

- *Distribution rule*

$$u, v_1 \vee \ldots \vee v_n \longmapsto (u \wedge v_1) \vee \ldots \vee (u \wedge v_n).$$

One can write similar rules for the other propositional connectives, e.g. $\supset$.

3.4 An example. Let us prove the validity of the formula

$$(p \wedge \Box(p \supset \bigcirc p)) \supset \Box p.$$

In other words , we refute

$$\neg(\neg(p \wedge \Box(p \supset \bigcirc p)) \vee \Box p.$$

The negation rules yield

$$(p \wedge \Box(p \supset \bigcirc p)) \wedge \Diamond\neg p.$$

Since $\vdash \neg(p \wedge \neg p)$, we apply the induction rule and weakening to derive

$$\Diamond(p \wedge \circledcirc\neg p) \wedge \Box(p \supset \bigcirc p).$$

Since $\vdash \neg(\Box(p \supset \bigcirc p) \wedge (p \wedge \circledcirc\neg p))$ (as we check later), as above we derive

$$\Diamond(\neg(p \wedge \circledcirc\neg p) \wedge \circledcirc((p \wedge \circledcirc\neg p) \wedge \neg\Box(p \supset \bigcirc p)) \wedge \Box(p \supset \bigcirc p).$$

By weakening this yields

$$\Diamond \odot (\neg\Box(p \supset \bigcirc p)) \wedge \Box(p \supset \bigcirc p).$$

Since $\vdash \neg(\odot(\neg\Box(p \supset \bigcirc p)) \wedge \Box(p \supset \bigcirc p))$ (as we check later), as above we derive

$$\Diamond(\neg(\odot(\neg\Box(p \supset \bigcirc p))) \wedge \odot \odot \neg\Box(p \supset \bigcirc p)).$$

By $\Box$ rule and weakening this yields

$$\Diamond(\neg(\odot(\neg\bigcirc\Box(p \supset \bigcirc p))) \wedge \odot \odot \neg\Box(p \supset \bigcirc p)).$$

By negation rule and weakening this yields

$$\Diamond(\bigcirc\bigcirc\Box(p \supset \bigcirc p) \wedge \odot \odot \neg\Box(p \supset \bigcirc p)).$$

We apply the resolution rule (taking $A = \bigcirc\bigcirc\Box(p \supset \bigcirc p)))$, $B = \odot\odot\neg\Box(p \supset \bigcirc p)$ and $u = \Box(p \supset \bigcirc p)$ to derive

$$\Diamond\bigcirc\bigcirc false \vee \odot \odot \neg true,$$

and then by simplification and weakening

$$\Diamond false.$$

And then

$$false.$$

We still have to show a) $\vdash \Box(p \supset \bigcirc p) \wedge (p \wedge \odot\neg p)$ and b) $\vdash \neg(\odot(\neg\Box(p \supset \bigcirc p)) \wedge \Box(p \supset \bigcirc p))$ to justify the application of the induction rule in this proof. We show a). b) is left to the reader. We apply $\Box$ rule and weakening to derive

$$(p \supset \bigcirc p) \wedge (p \wedge \odot\neg p).$$

By distribution rule and weakening this yields

$$(\neg p \wedge p \wedge \odot\neg p) \vee (\bigcirc p \wedge p \wedge \odot\neg p).$$

We apply twice the resolution rule to derive

$$\neg true \vee (false \wedge \odot\neg p) \vee (\bigcirc false \vee (p \wedge \odot\neg true)).$$

Negation rules and simplifications yield

$$false.$$

4.Completeness of $\mathbb{R}$

In this section we prove that the resolution system $\mathbb{R}$ is complete.

Theorem 4.1(completeness) Every valid BPTL formula is provable in the resolution system $\mathbb{R}$.

Proof. Our strategy as in [AM] is to show that any tableau proof can be transformed into a resolution proof. In other words, if the tableau decision procedure finds $\neg u$ of BPTL unsatisfiable, then $\neg u$ has a refutation. If n is a node of the tableau then U_n denotes the set of formulas at n. We prove a more general fact:

for every node n in the tableau for u ,if n is eliminated, then $\wedge U_n$ has the refutation.

The proof is by complete induction on the stage at which nodes are eliminated by the tableau decision procedure.(In other words ,the rank of a node is the stage at which it is deleted).

We recall the definition of a tableau construction for u and the tableau decision procedure ([BPM]). Formulas are classified as α-formulas and β -formulas

α	α_1	α_2
p$\wedge$q	p	q
$\Box$p	p	$\bigcirc\Box$p
β	β_1	β_2
p$\vee$q	p	q
$\Diamond p$	p	$\odot\Diamond$p.

Tableau construction and decision procedure. Let u be BPTL formula. Let n_0 be the root of T and let $U_{n_0} = \{\neg u, \Box \odot \text{true}\}$. T, the tableau for $\neg u$, is constructed inductively by applying the following rules to nodes which are leaves of T.

R_α: if $\alpha \in U_n$, then create a son n_1 of n and define

$$U_{n_1} = U_n \cup \{\alpha_1, \alpha_2\}.$$

R_β: if $\beta \in U_n$, then create two sons $n_1,\ n_2$ of n and define

$$U_{n_i} = U_n \cup \{\beta_i\}, \quad i = 1, 2.$$

$R_\odot$: let $V_n = \{\odot p_1, \ldots, \odot p_k, \bigcirc q_1, \ldots \bigcirc q_m\}$ be the set of next time formulas in parent n. Create k sons $n_i,\ i = 1, \ldots, k$ of n and define

$$U_{n_i} = \{p_i, q_1, \ldots, q_m\}.$$

Notation. If the $R_\alpha/R_\beta/R_\odot$-rule was applied at node n, the n is called $\alpha/\beta/\odot$-node. Root and sons of $\odot$-node are called *pre-states*. $\odot$-node is called a *state*.

The construction of T is made finite by introducing the following two termination rules:

1. if $p \in U_n$ and $\neg p \in U_n$, then this node is called closed and is not expanded further.

2. if a node m is about to be created as a son of an $\circledcirc$-node n and there is an ancestor n' of n such that n' is an immediate descendant of a $\circledcirc$-node and $U_{n'} = U_m$, then do not create m but instead connect n to n' with "feedback" edge.

A node of T is eliminated in the following three cases:

1) it contains a proposition and its negation;
2) it is α- or β-node and all of its descendants have been eliminated; it is $\circledcirc$-node and one of its descendants have been eliminated;
3) if the node is a pre-state and contains $\Diamond v$ and on no path from the node do v occurs.

$\neg u$ is found unsatisfiable if and only if all nodes of the tableau have been eliminated.

Consider a node n. Assume that if the node m has already been eliminated then $\wedge U_m$ has a refutation, and suppose that n is eliminated. To prove completeness we show that U_n is refuted in R. We analyse three cases; they correspond to the three possible reasons why n can be eliminated.

Lemma 4.1. 1) If U_n contains p and $\neg p$ then $\wedge U_n$ can be refuted in R; 2) if n is $\alpha/\beta/\circledcirc$-node and $\wedge U_{n_1}$ is refuted in $\mathbb{R}/\wedge U_{n_1}$ and $\wedge U_{n_2}$ are refuted in $\mathbb{R}/\wedge U_{n_i}$ for some i is refuted in $\mathbb{R}$ then $\wedge U_n$ can be refuted in $\mathbb{R}$.

Proof. In case 1) $\wedge U_n = p \wedge \neg p \wedge \phi$ for some formula ϕ. We derive $false$ by resolution on p and simplifications.

We consider 2).

• Let U_{n_1} be created from U_n by α-rule. Then $p \wedge q \in U_n$ and $U_{n_1} = U_n \cup \{p, q\}$ or $\Box p \in U_n$ and $U_{n_1} = U_n \cup \{p, \bigcirc\Box p\}$. Assume, there is a refutation of $\wedge U_{n_1}$ to show that there is a refutation of $\wedge U_n$. In both cases the refutation of $\wedge U_{n_1}$ can be extended to the refutation of U_n.

• Let U_{n_1} and U_{n_2} be obtained from U_n by β-rule. Then $p \vee q \in U_n$ or $\Diamond p \in U_n$. In the first case $(\wedge U_{n_1}) \vee (\wedge U_{n_2})$ is obtained from $\wedge U_n$ by the distribution rule . In the second case $(\wedge U_1) \vee (\wedge U_{n_2})$ is obtained from $\wedge U_n$ by applying $\Diamond$-rule and next the distribution rule. Assume there is a refutation of $\wedge U_{n_1}$ and $\wedge U_{n_2}$, to show there is a refutation of $\wedge U_n$. In both cases the refutation of U_{n_1} and U_{n_2} can be extended to the refutation of U_n.

• Let U_{n_i} be created from U_n by $\circledcirc$-rule. Then $\wedge U_n = \circledcirc p_1 \wedge \ldots \wedge \circledcirc p_k \wedge \bigcirc q_1 \wedge \ldots \wedge \bigcirc q_l \wedge v_1 \wedge \ldots \wedge v_m$, where v_i's are not of the form $\bigcirc w$ or $\circledcirc w$ and $\wedge U_{n_i} = p_i \wedge q_1 \wedge \ldots \wedge q_l$. Assume there is a refutation of $\wedge U_{n_i}$ in $\mathbb{R}$. The following lines are refutation of $\wedge U_n$ in $\mathbb{R}$.

1.	$\circledcirc p_1 \wedge \ldots \circledcirc p_k \wedge \bigcirc q_1 \wedge \ldots \wedge \bigcirc q_l \wedge v_1 \wedge \ldots \wedge v_m$	$\wedge U_n$
2.	$\circledcirc p_i \wedge \bigcirc q_1 \wedge \ldots \wedge \bigcirc q_l$	by weakening from 1.
3.	$\circledcirc p_i \wedge \bigcirc(q_1 \wedge \ldots \wedge q_l)$	by $\bigcirc\bigcirc$-rule and weakening from 2.
4.	$\circledcirc(p_i \wedge q_1 \wedge \ldots \wedge q_l)$	by $\circledcirc\bigcirc$-rule and weakening from 3.
	$\vdots$	
S.	$\circledcirc$ false	by assumption on

U_{n_i}.

S+1. false — by simplification from S.

Lemma 4.2. If a node n is a prestate which contains $\Diamond v$ and on no path from the node do v occurs, then $\wedge U_n$ can be refuted in R.

To prove lemma 4.2 we prove some auxiliary lemmas.

Let $[n]^*$ be the set of prestates accessible from n by choosing at $\circledcirc$-node the son corresponding to $\circledcirc\Diamond v$. Then $\Diamond v \in U_n$ and on no path from $u \in [n]^*$ do v occurs. Let Y_u, $u \in n \cup [n]^*$, denote a part of prestate u obtained by erasing $\bigcirc$'s in a state preceding u. We call Y_u *universal next time part of* u. We will want to construct universal next time parts of prestates in $[n]^*$. More precisely, we get w_i's that say what universal next time parts of prestates at depth i counting from n will be true then. We call w_i the i-th fringe. For example, if $U_n = \{\Box(\bigcirc q \vee \circledcirc q_1), \Diamond v\}$, then $w_1 = \bigcirc(q \wedge \Box(\bigcirc q \vee \circledcirc q_1)) \vee \bigcirc\Box(\bigcirc q \vee \circledcirc q_1)$, where $(q \wedge \Box(\bigcirc q \vee \circledcirc q_1), \Box(\bigcirc q \vee \circledcirc q_1)$, are universal next time parts of prestates at depth 1 in $[n]^*$

Lemma 4.2.1. Given a prestate n $\mathbb{R}$ can derive formulas w_i built from the universal next time parts p_i^l of prestates from $[n]^*$ at depth i from Y_n with $\vee$ and $\bigcirc$. Furthermore, each p_i^l occurs in the scope of exactly i $\bigcirc$'s.

Proof. Put $w_o = \wedge Y_n$. We show how to obtain w_{i+1} from w_i. Let Y_t be a universal next time part of a prestate $t \in [n]^*$ in w_i at depth i. Then $U_t = Y_t \cup \Diamond v$.

Let $[t]$ be the states accessible from t by α, β nodes which include $\circledcirc\Diamond v$. We will show that we derive $\bigvee_{u\in[t]} \wedge(U_u - \{\circledcirc\Diamond v\})$ from $Y_t = \wedge(U_t - \{\Diamond v\})$ in $\mathbb{R}$. In fact applying $\Box-, \Diamond$-rules and distribution rule we derive $\bigvee_{u\in[t]^+}(\wedge U_u)$ from U_t, where $[t]^+$ is all states accessible from t by α, β nodes, i.e. $[t]^+$ is obtained from $[t]$ by adding states which include v. But these states are eliminated, so refuted by induction. Thus we derive $\bigvee_{u\in[t]}(\wedge U_u)$ from $\wedge U_t$. Hence we derive $\bigvee_{u\in[t]}(U_u - \{\circledcirc\Diamond v\})$ from $\wedge(U_t - \Diamond v)$. By weakening we drop from U_u all the formulas other than prefixed with $\bigcirc$. So we derive $\bigvee_{u\in[t]}(\bigwedge_{p\in Y_u} \bigcirc p)$. Apply $\bigcirc\bigcirc$-rule to pull $\bigcirc$'s out of conjunction. We derive $\bigvee_{u\in[t]}(\bigcirc \bigwedge_{p\in Y_u} p)$. Applying this procedure to all universal next time parts at depth i from w_i we derive w_{i+1}.

Lemma 4.2.2. Given a prestate $t \in n \cup [n]^*$ $\wedge Y_t \wedge v$ can be refuted in $\mathbb{R}$.

Proof. $U_t = Y_t \cup \{\Diamond v\}$ as mentioned above. Without loss of generality we can assume that β rule was applied to $\Diamond v$ at t to construct sons t_1, t_2. Then $U_{t_1} = \{U_t - \{\Diamond v\}\} \cup \{v\} = Y_t \cup \{v\}$. But t_1 have been eliminated; otherwise we have path with v from n. Thus U_{t_1} is refuted in $\mathbb{R}$ by induction.

Lemma 4.2.3. For any $i \geqslant 0 \vdash \neg(w_i \wedge \circledcirc^i v)$.

Proof. For $i = 0$ distribute v by the distribution rule and apply lemma 4.2.2. For $i > 0$ distribute $\circledcirc^i v$ by the distribution rule. Next use $\circledcirc\bigcirc$- rule to push v

inwards. Then we obtain conjunctions of universal next time parts of prestates in the depth i from $n \cup [n]^*$ with v. Applying lemma 4.2.2 we obtain the refutation of this conjunctions. Then applying reduction rules false $\vee A \to A$, $\circledcirc$ false $\to$ false we obtain refutation of $w_i \wedge \circledcirc^i v$.

The lemma 4.2.4 is the main lemma of completeness proof.

Lemma 4.2.4. If for all $i \vdash \urcorner(w_i \wedge \circledcirc^i v)$, then $\vdash \urcorner \wedge U_n$.

Proof. $\wedge U_n = \wedge Y_n \wedge \Diamond v$ by assumption. Assume $\vdash \urcorner(w_i \wedge \circledcirc^i v)$, for all$i$. We put $w_0 = \wedge Y_n$. From w_0 and $\Diamond v$, by the induction rule and weakening we derive

$$\Diamond \circledcirc (v \wedge \urcorner w_0)$$

since $\vdash \urcorner(w_0 \wedge v)$ by assumption.
By lemma 4.2.1 we can derive w_1. Furthermore, $\vdash \urcorner(w_1 \wedge \circledcirc v)$ by assumption. Thus $\vdash \urcorner(w_1 \wedge \circledcirc(v \wedge \urcorner w_0))$. This by the induction rule yields

$$\Diamond \circledcirc (\circledcirc(v \wedge \urcorner w_0) \wedge \urcorner w_1).$$

In general, we can get all w_i's and we have that $\vdash \urcorner(w_i \wedge \circledcirc^i v)$, by assumption. Successive applications of the induction rule will give

$$\Diamond \circledcirc (\circledcirc(\ldots \circledcirc (v \wedge \urcorner w_0) \wedge \ldots) \wedge \urcorner w_{m-1}) \wedge \urcorner w_m)$$

for any m.

We weaken this to

$$\Diamond \circledcirc (\circledcirc(\ldots \circledcirc (\urcorner w_0) \wedge \ldots) \wedge \urcorner w_{m-1}) \wedge \urcorner w_m)$$

Call this formula Ψ_m and define also Ω_m by $\Psi_m = \Diamond \Omega_m$.

The finite model property tells us that there are only finitely many universal next time parts generated from w_0. Thus for some s we have all universal next time parts generated from w_0 in $\mathbb{R}$. We check that $\vdash \urcorner(w_0 \wedge \Omega_s)$. Take $s+1$-th fringe w_{s+1} of w_0. Each universal next time part Y in w_{s+1} is in the scope of $s+1$ of $\bigcirc$'s and it is denied in Ω_s. Furthermore $\urcorner Y$ is in the scope of $s+1$ of $\circledcirc$'s in Ω_s. Applying the resolution rule we obtain false derived from $w_0 \wedge \Omega_s$, i.e. $w_o \wedge \Omega_s$ has the refutation in $\mathbb{R}$. Thus we can apply the induction rule (and weakening) once more and get

$$\Diamond(\urcorner \Omega_s \wedge \circledcirc \Omega_s).$$

Call this formula ϕ. ϕ says that at each point in depth $s+1$ we are in one of universal next time prestates reachable from w_0 and that there are the next instant we are in none of them. Of course, this cannot be the case; in fact, we can refute $\urcorner \Omega_s \wedge \circledcirc \Omega_s$: we derive the first fringe of all universal next time parts of prestates in $\urcorner \Omega_s$ and check that universal next time parts of prestates in these fringes were already in Ω_s. Thus by the resolution rule we derive $\Diamond$ false and hence false.

Proof of lemma 4.2. From lemma 4.2.3 and lemma 4.2.4 we obtain that $\wedge U_n$ can be refuted in R.

5. Concluding remarks

We have presented a nonclausal resolution approach to theorem proving in BPTL with temporal modalities $\bigcirc, \odot, \Box, \Diamond$. We show that presented system is complete. We expect to generalize this approach to get nonclausal resolution system for branching temporal logic with modality *Until*.

References

[AM] (1990) Abadi M. and Z.Manna, Nonclausal deduction in first order temporal logic, Journal of ACM 37, 2, 279-317.

[BPM] (1983) Ben-Ari M., A.Pnueli and Z.Manna, The temporal logic of branching time, Acta Informatica 20, 207–226.

Composed Reduction Systems

David Sands*
DIKU, University of Copenhagen†

Abstract

This paper studies *composed reduction systems*: a system of programs built up from the reduction relations of some reduction system, by means of parallel and sequential composition operators. The trace-based compositional semantics of composed reduction systems is considered, and a new graph-representation is introduced as an alternative basis for the study of compositional semantics, refinement, true concurrency (in the case of composed rewriting systems) and program logics.

1 Introduction

Reduction systems are simply sets equipped with some collection of binary "rewrite" relations. A reduction systems can be thought of as an abstract view of computation, embodying the fundamental computational concepts of *iteration*, *termination*, and *nontermination*. Computation is the process of repeatedly rewriting, beginning with some object of the set, and termination corresponds to obtaining an object which cannot be rewritten further; nontermination is the ability to rewrite indefinitely.

Since reduction systems have little structure, there are relatively few properties one can state about theses systems, although unique-termination ("Church-Rosser") properties of reduction systems have been studied by eg. Rosen [Ros73], Hindley [Hin69] Staples [Sta74].

In this paper we consider systems ("programs") whose basic components are the reduction relations of some reduction system. These systems, which we call *composed reduction systems*, are built by composing reduction relations with two natural composition operators: *parallel* and *sequential* composition. Composed reduction systems are not necessarily reduction systems, but they possess a notion of a "reduction step", and a corresponding notion of termination.

- Parallel composition allows arbitrary interleaving of reduction steps. In the simplest case, the parallel composition of two reduction relations corresponds to the union of these relations. Parallel composition terminates when, simultaneously, both sub-systems have terminated.

*This work was partially funded by ESPRIT BRA 9102, "Coordination"
†Universitetsparken 1, 2100 København Ø, DENMARK. e-mail: dave@diku.dk

- Sequential composition, on the other hand, takes us outside the realm of reduction systems (over the given set). The sequential composition of two reduction relations is the system which behaves like the first reduction relation, until termination of the first system, after which it behaves like the second system. The sequentially composed system is said to terminate when the second sub-system has terminated.

Note, then, that the sequential composition of two reduction relations (the simplest case) is not the relational composition of these relations. Composed reduction systems over a given reduction system are built from arbitrary sequential and parallel compositions of reduction relations.

In this paper we study the semantics of composed reduction systems, expressed in terms of its constituent reduction relations. We focus on a comparison relation for programs which partially orders programs on the basis of their "input-output" behaviours, and is also a precongruence with respect to program construction.

In the first part of the paper (Section 2) we consider a standard compositional semantics based on "reactive traces" (sequences of object-pairs) derived from the SOS-like rules which give the operational semantics for composed reduction systems. We outline some of the program laws that can be obtained, and consider the relationship to an alternative form of parallel composition.

In the second part of the paper (Section 3), we define a static graph representation for programs, and argue that it forms a better basis (than the transition traces) for the study of:

- compositional semantics, since it is higher-level than the transition traces;
- refinement laws, since the graphs can also be defined compositionally;
- concurrency, since "concurrently active" reduction relations are explicit in the representation, and
- program logics, since logics for the underlying reduction systems can be used to reasoning about paths through the graph.

Related Work and Applications This work grew out of the study of composition of specific kind of reduction system, namely programs in the Gamma model [BM93], which can be thought of as conditional associative-commutative string rewriting. The composition operators for Gamma were introduced in [HMS92], and the compositional semantics and laws were studied in [San93a][San93b]. The development of section 2 is a direct (and straightforward) adaptation of [San93a][San93b] to this more general setting. The graph representation in section 3 is new, and is particularly relevant from the point of view of composed Gamma programs. [1]

The techniques given here may also be interesting when applied to other concrete reduction systems. In particular we have in mind *rewriting systems*

[1] For example, through this representation have discovered additional laws for Gamma programs.

in which the objects rewritten have some structure (eg. trees, graphs, strings), and the reduction relation is specified by rules for rewriting a substructure, in terms of purely *local* conditions. For such systems (eg. the usual notion of term rewriting [Klo92] [DJ89]) there is a natural (implicit) notion of concurrency, viz. disjoint parts of a substructure can be rewritten asynchronously, and hence concurrently. This view of rewriting as a natural vehicle for concurrency and parallel programming is central to Meseguer's approach [Mes92][MW91]; the composition operators studied here also make sense in that setting.

Another form of reduction system, where one could reasonably employ the composition operators studied here, is the guarded iteration statement from [Dij76], also known as *action systems* [Bac89a][Bac89b]. Action systems are nondeterministic **do-od** programs consisting of a collection of guarded atomic actions, which are executed nondeterministicly so long as some guard remains true. In their uninitialised form, guarded iteration statements can be thought of as reduction systems over program states. The method of parallel execution is to allow actions involving disjoint program variables to be executed in parallel, which is consistent with the rewriting viewpoint above. In [Bac89b] Back studies compositional notions of refinement for action systems with respect to a meta-linguistic parallel composition operator. [2] The parallel composition studied here is strictly more general since it permits parallel composition of sequentially composed systems.

2 Operational and Compositional Semantics

In this section we give the operational semantics of composed reduction systems built from basic reduction relations, parallel and sequential composition.

In what follows, we assume some reduction system $\langle \mathbf{U}, \{\rightarrow_{\mathbf{r}}\}_{\mathbf{r}\in\mathcal{R}}\rangle$, where $\mathbf{U}$ is a set, with typical elements $M, N, M_1 \ldots$. We will sometimes refer to the elements of $\mathbf{U}$ as *states*. The reduction relations, $\{\rightarrow_{\mathbf{r}}\}_{\mathbf{r}\in\mathcal{R}}$ are just binary relations on states.We will think of the elements of the indexing set $\mathcal{R}$, ranged over by $\mathbf{r}, \mathbf{r}_2 \ldots$, as the basic units of our composed reduction systems. Somewhat improperly, for more concrete examples we will think of $\mathcal{R}$ as the set of *representations* of the corresponding reduction relation.

With respect to some $\mathbf{r}$, we say that

- M reduces to N if $M \rightarrow_{\mathbf{r}} N$ (ie. $(M, N) \in \rightarrow_{\mathbf{r}}$);
- M converges immediately, written $M{\downarrow}^{\mathbf{r}}$, if $\neg\exists N.\, M \rightarrow_{\mathbf{r}} N$.

For the moment we consider composed reduction systems, ranged over by P, Q, P_1, Q_1 etc, given by the following grammar:

$$P ::= \mathbf{r} \mid P\,;Q \mid P \mathbin{\#} Q$$

Henceforth we will use the terms "composed reduction system" and "program" synonymously.

[2] UNITY [CM88] has a similar composition operator, called *union*, but UNITY is not a reduction system in the same sense because the notion of termination for UNITY is that of *stability*—reaching a fixed-point—rather than inactivity.

2.1 SOS semantics

Because of the presence of sequential composition, programs cannot be viewed as reduction systems over $\mathbf{U}$, since the program is not a static entity. To define the semantics for these programs we define a single step transition relation between *configurations*. The configurations are program-state pairs, written $\langle P, M\rangle$. The final result of a computation is given by an immediate-convergence predicate, $\downarrow$, on configurations. Single step reduction and immediate-convergence is given by SOS-style rules in figure 1. It is easily verified that immediate con-

$$\frac{M \rightarrow_{\mathbf{r}} N}{\langle \mathbf{r}, M\rangle \rightarrow \langle \mathbf{r}, N\rangle} \qquad \frac{M\downarrow^{\mathbf{r}}}{\langle \mathbf{r}, M\rangle\downarrow}$$

$$\frac{\langle P, M\rangle \rightarrow \langle P', M'\rangle}{\langle P\ ;\ Q, M\rangle \rightarrow \langle P\ ;\ Q, M'\rangle} \qquad \frac{\langle P, M\rangle\downarrow}{\langle P\ ;\ Q, M\rangle \rightarrow \langle Q, M\rangle}$$

$$\frac{\langle P, M\rangle \rightarrow \langle P', M'\rangle}{\langle P \mathbin{+\!\!\!\!+} Q, M\rangle \rightarrow \langle P' \mathbin{+\!\!\!\!+} Q, M'\rangle} \quad \frac{\langle Q, M\rangle \rightarrow \langle Q', M'\rangle}{\langle P \mathbin{+\!\!\!\!+} Q, M\rangle \rightarrow \langle P \mathbin{+\!\!\!\!+} Q', M'\rangle} \quad \frac{\langle P, M\rangle\downarrow \quad \langle Q, M\rangle\downarrow}{\langle P \mathbin{+\!\!\!\!+} Q, M\rangle\downarrow}$$

Figure 1: Structural Operational Semantics of composed reduction systems

vergence of a configuration corresponds to the absence of any transitions for that configuration. In other words, $\langle P, M\rangle \rightarrow \langle Q, N\rangle$ for some $\langle Q, N\rangle$ if and only if $\neg(\langle P, M\rangle\downarrow)$.

Let $\rightarrow^*$ denote the transitive, reflexive closure of $\rightarrow$. By a small abuse of the notation, we will write $\langle P, M\rangle \rightarrow^* N$ to mean that there exists some $\langle Q, N\rangle$ such that $\langle P, M\rangle \rightarrow^* \langle Q, N\rangle$ and $\langle Q, N\rangle\downarrow$.

2.2 Behavioural orderings

In this paper we will focus on the relational (input-output) behaviours of a program. A number of "refinement" orderings on programs arise from the various natural ways to compare programs on the basis of their input-output (or relational) behaviour. One possible "behaviour" which we should consider significant is the possibility of nontermination for a given input. Non-termination, or "divergence" is a predicate on program configurations:

Definition 1 *P may diverge on M, $\langle P, M\rangle\uparrow$, if there exist $\{\langle P_i, M_i\rangle\}_{i\in\omega}$ such that $\langle P_0, M_0\rangle = \langle P, M\rangle$ and $\langle P_i, M_i\rangle \rightarrow \langle P_{i+1}, M_{i+1}\rangle$.*

It is convenient to abstract the possible relational behaviours of a program as a set of possible input-output pairs. This includes the possibility of non-termination, which we represent as a possible "output" using symbol '$\perp$' ($\notin \mathbf{U}$):

Definition 2 *The behaviours of a program P are given by*

$$\begin{array}{rcl} \mathcal{B}(P) & = & \{(M, N) \mid \langle P, M\rangle \rightarrow^* N\} \\ & \cup & \{(M, \perp) \mid \langle P, M\rangle\uparrow\} \end{array}$$

Note that for every P, M, either $(M, N) \in \mathcal{B}(P)$ for some N, or $(M, \perp) \in \mathcal{B}(P)$ (or both). There are a variety of orderings on programs obtained by comparing their behaviours: the *partial correctness* ordering ignores divergent behaviours; the *lower* and *upper* orderings are formed by considering the associated discrete power-domain orderings on $\mathbf{U}_\perp$. In this study we only consider the *strong correctness* ordering, which attaches the same significance to nonterminating computations as to the terminating ones. The strong correctness ordering is defined to be the largest (pre)congruence which satisfies

$$P \sqsubseteq_o Q \Rightarrow \mathcal{B}(P) \subseteq \mathcal{B}(Q)$$

This is given directly by the following:

Definition 3 *Let* $\mathbf{C}$ *range over program contexts. We define* strong precongruence *($\sqsubseteq_o$) and* strong congruence*($\equiv_o$) respectively by:*

$$\begin{array}{lcl} P \sqsubseteq_o Q & \iff & \forall \mathbf{C}. \mathcal{B}(\mathbf{C}[P]) \subseteq \mathcal{B}(\mathbf{C}[Q]) \\ P \equiv_o Q & \iff & P \sqsubseteq_o Q \;\&\; Q \sqsubseteq_o P \end{array}$$

2.3 Laws

In this section we present a number of the basic laws of strong precongruence, and show the relationship to an alternative definition of parallel composition. Let Δ denote the empty reduction relation, satisfying $\forall M. M{\downarrow}^{\Delta}$. For example, in conditional rewriting systems, this could be represented by a reduction rule with the condition *false*.

Proposition 4

1.	$P \,;\, (Q \,;\, R) \equiv_o (P \,;\, Q) \,;\, R$	5.	$\Delta \mathbin{+\!\!\!\!+} P \equiv_o P \mathbin{+\!\!\!\!+} \Delta \equiv_o P$
2.	$P \mathbin{+\!\!\!\!+} (Q \mathbin{+\!\!\!\!+} R) \equiv_o (P \mathbin{+\!\!\!\!+} Q) \mathbin{+\!\!\!\!+} R$	6.	$P \sqsubseteq_o P \,;\, \Delta$
3.	$P \mathbin{+\!\!\!\!+} Q \equiv_o Q \mathbin{+\!\!\!\!+} P$	7.	$P \equiv_o \Delta \,;\, P$
4.	$Q \,;\, (P_1 \mathbin{+\!\!\!\!+} P_2) \sqsubseteq_o (Q \,;\, P_1) \mathbin{+\!\!\!\!+} P_2$	8.	$P \sqsubseteq_o P \mathbin{+\!\!\!\!+} P$

These are just a few of the laws of strong precongruence. In fact, almost all of the partial correctness laws of composed Gamma programs [San93b] hold for these more general composed reduction systems. Note in particular that law 6 cannot be strengthened to an equality, ie.

$$P \,;\, \Delta \not\equiv_o P$$

The intuition for this is that Δ acts as a de-synchroniser for parallel composition: P must synchronise with its context in order to terminate, but with $P \,;\, \Delta$, P is allowed to terminate autonomously, leaving Δ to synchronise with its context—which it is trivially always able to do. As an example consider the following two term rewrite rules, where a and b are constants: $a \to b$ and $b \to a$. It is easily seen that $\langle (a \to b) \mathbin{+\!\!\!\!+} (b \to a), a \rangle$ can never terminate, but

$$\langle (P_1 \,;\, \Delta) \mathbin{+\!\!\!\!+} P_2, a \rangle \to^* a$$

and so $(P_1 \,;\, \Delta) \mathbin{+\!\!\!\!+} P_2 \not\sqsubseteq_o P_1 \mathbin{+\!\!\!\!+} P_2$.

An Alternative Parallel Composition

There is a natural alternative form of parallel program composition, which does not require that the two programs terminate synchronously. Extend the syntax of the language with $P ::= P_1 \,\|\, P_2$, and add the operational rules:

$$\frac{\langle Q, M\rangle\!\downarrow}{\langle P \,\|\, Q, M\rangle \to \langle P, M\rangle} \qquad \frac{\langle P, M\rangle\!\downarrow}{\langle P \,\|\, Q, M\rangle \to \langle Q, M\rangle}$$

The expected associativity and commutativity properties also hold for $\|$. Some relationships with ; and $\#$ can be summarised in the diagram in figure 2, where the arrows ($\to$) depict the ordering $\sqsubseteq_o$. To give some intuition to the fact in

$$\begin{tikzcd}[column sep=small]
 & & (P\,;\Delta) \# (Q\,;\Delta) \equiv_o (P\,;\Delta) \,\|\, (Q\,;\Delta) & & \\
 & & P \,\|\, Q \arrow[u] & & \\
 & (P\,;\Delta) \# Q \arrow[ur] & & P \# (Q\,;\Delta) \arrow[ul] & \\
P\,;Q \arrow[ur] & & P \# Q \arrow[ul] \arrow[ur] & & Q\,;P \arrow[ul]
\end{tikzcd}$$

Figure 2: Relationships between Compositions

this diagram, consider the increasing chain of "behaviours":

$$P\,;Q \sqsubseteq_o (P\,;\Delta) \# Q \sqsubseteq_o P \,\|\, Q \sqsubseteq_o (P\,;\Delta) \# (Q\,;\Delta)$$

For $P \,\|\, Q$ to terminate, either P (or some derivative thereof) terminates followed by Q, or vice-versa, and either P or Q is left to synchronise its termination with the context. The system $(P\,;\Delta) \# Q$ has fewer behaviours because although reductions from P are potentially concurrent with reductions from Q, P must always terminate first. $(P\,;\Delta) \# (Q\,;\Delta)$, on the other hand, exhibits more behaviours since P and Q are "concurrent", can terminate autonomously, and neither of them is required to synchronise its termination with the context.

2.4 Trace Semantics

We can characterise $\sqsubseteq_o$ (in order to *prove* the laws of the previous section) by finding a compositional semantics which is consistent (ie. sound) with respect to the behaviours. Clearly the *behaviours* of a program will not suffice as its denotation. As is well-known from the study state-based concurrency, it insufficient to use sequences of states as a means of distinguishing programs. The solution we adopt follows a simple approach to modelling shared-state (interleaving) concurrency via sequences of state-pairs[3] (eg. sequences of "moves"[Abr79];

[3]The set of all such sequences for a program can be thought of as an "unraveling" of the program's *resumption semantics* [Plo76][HP79]. This "unraveling" leads to a mathematically simpler domain (no powerdomains) which is more amenable to further refinements than resumptions.

"abstract paths" of [Par79]). Following the terminology of [Bro93], we will use the term *transition traces*, or simply *traces* to refer to this kind of sequence. In these models, a pair of states in the trace of a program represents an atomic computation step of the program; adjacent pairs in any given sequence model a possible interference by some other process executing in parallel with the program.

The transition traces have a straightforward operational specification:

Definition 5 *The transition traces of a program, $\mathsf{T}[\![P]\!]$, are the finite and infinite sequences of state-pairs, given by:*

$$\begin{array}{rcl}
\mathsf{T}[\![P]\!] & = & \{(M_0, N_0)(M_1, N_1)\ldots(M_k, N_k)| \\
& & \langle P, M_0\rangle \rightarrow \langle P_1, N_0\rangle \ \& \\
& & \langle P_1, M_1\rangle \rightarrow \langle P_2, N_1\rangle \ \& \ldots \& \ \langle P_k, M_k\rangle\downarrow, \& \ M_k = N_k\} \\
& \cup & \\
& & \{(M_0, N_0)(M_1, N_1)\ldots(M_i, N_i)\ldots| \\
& & \langle P, M_0\rangle \rightarrow \langle P_1, N_0\rangle \ \& \ \langle P_i, M_i\rangle \rightarrow \langle P_{i+1}, N_i\rangle, i \geq 1\}
\end{array}$$

The intuition behind the use of transition traces is that each transition trace

$$(M_0, N_0)(M_1, N_1)\ldots \in \mathsf{T}[\![P]\!]$$

represents computation steps of program P in some context; starting with state M_0, each of the pairs (M_i, N_i) represents computation steps performed by (derivatives of) P and the adjacent states N_{i-1}, M_i represent possible interfering computation steps performed by the "context". If the trace is finite then the last step corresponds to the termination "step" for a derivative of P.

Stuttering

Clearly the behaviours of a program are obtainable from its transition traces, by considering the "chained" traces of the form: $(M_0, M_1)(M_1, M_2)\ldots$. Transition traces are adequate for giving a compositional semantics to composed reduction systems, by interpreting sequential composition as (set-wise) trace concatenation, and parallel composition as interleaving (with the proviso that interleaved finite traces must agree on their last elements).

However, the transition traces distinguish between programs which compute at different "speeds". For example, considering the empty action system $DO \quad OD$, then the transition traces of $DO \quad OD$ are different from those of $(DO \quad OD)$; $(DO \quad OD)$ [4] The key to obtaining a better level of abstraction is to equate processes which only vary by "uninteresting" steps. This is the "stuttering equivalence" well-known from Lamport's work on temporal logics for concurrent systems [Lam89]. Closure under stuttering equivalence has been used by de Boer et al [dBKPR91], and by Brookes [Bro93] to provide fully abstract semantics for languages with shared-state and parallel composition.

Following Brookes [Bro93] we define a closure operation for sets of transition traces:

[4] In order to obtain full abstraction for a while-language with parallel composition, Hennessy and Plotkin added a co-routine command, which is able to distinguish these programs.

Definition 6 *Let ϵ denote the empty sequence. Let α range over finite sequences of state pairs, and β range over finite or infinite sequences. A set T of finite and infinite traces is closed under left-stuttering*[5] *and absorption if it satisfies the following two conditions*

$$\textbf{left-stuttering}\frac{\alpha\beta \in T, \beta \neq \epsilon}{\alpha(M,M)\beta \in T} \qquad \textbf{absorption}\frac{\alpha(M,N)(N,M')\beta \in T}{\alpha(M,M')\beta \in T}$$

Let $\ddagger T$ denote the left-stuttering and absorption closure (henceforth just closure) of a set T.

In [dBKPR91] a slightly different closure operation is used, in which only stuttered steps can be absorbed. With respect to the above closure conditions, the difference is that in the clause for absorption we should also require that either $M = N$ or $N = M'$. This leads to a coarser abstraction for specific reduction systems; for example, the composed string-rewriting systems: $(1, 1 \rightarrow 2, 2) \mathbin{+\!\!\!+} (1 \rightarrow 2)$ and $(1 \rightarrow 2)$ have different traces under the stuttering-closure operation of [dBKPR91], but are the same under $\ddagger$.

Clearly the behaviours are also derivable from $\ddagger\mathsf{T}[\![P]\!]$, and what is more, $\ddagger\mathsf{T}[\![P]\!]$ can be specified compositionally (using monotonic operators) which gives the following:

Proposition 7 $\ddagger\mathsf{T}[\![P]\!] \subseteq \ddagger\mathsf{T}[\![Q]\!] \Longrightarrow P \sqsubseteq_o Q$

In the appendix we give the compositional definition of transition traces.

For a specific collection of reduction relations over some given universe, the transition traces may not be *fully abstract*. In other words, we cannot reverse the implication in the above proposition. For a specific example where full abstraction fails, see [San93a]. Even if we allow all reduction relations over a given universe it is unclear as to whether the transition traces are fully abstract.

3 Graph Representation

In this section we outline a static graph-representation for composed reduction systems, and argue that it forms a better basis for the study of compositional semantics, refinement, true concurrency and program logics.

The graph representation we will develop is something like a finite, acyclic control-flow graph, where each node corresponds to a simple form of loop. A node carries a set of reduction relations which are (con)currently active; an edge represents an internal termination step, where the child node may inherit some reductions from the parent but adds some new active reductions. From the viewpoint of the observational semantics, we will identify the graph with its set of complete paths.

The idea is best illustrated with an example. Consider a program consisting of four reduction relations:

$$(\mathbf{r}_1 \,;\, \mathbf{r}_2) \mathbin{+\!\!\!+} (\mathbf{r}_3 \,;\, \mathbf{r}_4)$$

[5] Notice that we say *left*-stuttering to reflect that the context is not permitted to change the state after the termination of the program. In this way each transition trace of a program only charts interactions with its context up to the point of the programs termination.

Initially $\mathbf{r}_1$ and $\mathbf{r}_3$ are *active* and thereby able to contribute to the reduction steps. At some time, $\mathbf{r}_1$ or $\mathbf{r}_3$ may be able to terminate. Suppose $\mathbf{r}_1$ terminates first; then $\mathbf{r}_2$ and $\mathbf{r}_3$ become active. Symmetrically, if $\mathbf{r}_3$ terminates first then $\mathbf{r}_1$ and $\mathbf{r}_4$ become active. Continuing in this way we construct the graph for this program:

$$\begin{tikzcd} & \{\mathbf{r}_1, \mathbf{r}_3\} \arrow[dl, "\{\mathbf{r}_3\}"'] \arrow[dr, "\{\mathbf{r}_1\}"] & \\ \{\mathbf{r}_1, \mathbf{r}_4\} \arrow[dr, "\{\mathbf{r}_1\}"'] & & \{\mathbf{r}_2, \mathbf{r}_3\} \arrow[dl, "\{\mathbf{r}_3\}"] \\ & \{\mathbf{r}_2, \mathbf{r}_4\} & \end{tikzcd}$$

The operational semantics of such graphs should be transparent: control begins at the root-node of the graph, and each node is labeled with a set of concurrently active reduction-relations; each arc is labeled with a set of reduction relations which must converge with respect to the current state for the control to be allowed move along that edge.

3.1 From SOS rules to Graph Representation

The graph representation will be constructed from two "abstract interpretations" of the one-step evaluation relation. Consider any possible one-step reduction on configurations:

$$\langle P, M \rangle \rightarrow \langle P', M' \rangle$$

From inspection of the rules it is clear that either:

1. $P \neq P'$ and $M = M'$, or
2. $P = P'$ and $M \rightarrow_{\mathbf{r}} M'$ for some reduction $\mathbf{r}$ in P.

In the terminology of [HMS92], we call a transition of the first kind as a *passive* step, and one of the second kind as an *active* step. The passive step corresponds to some internal termination step in which the left-operand of a sequential composition is discarded. The convergence of a configuration can similarly be considered to be a passive step. An active step corresponds to a reduction step on the state-component of a configuration.

We construct the graph representation of a given program by separately *abstracting*:

1. the passive steps, which will give us the arcs in the graph, and
2. the active steps which will tell us what reductions are contained in the nodes.

Abstract Passive Steps We abstract the passive steps performable by a program via a (labeled) transition system with judgements of the form $P \overset{R}{\rightsquigarrow} Q$, where R is a set of reductions. As an auxiliary, we define a notion of convergence for programs which is an abstraction of the convergence predicate for configurations. The abstract convergence predicate is trivial: a program can converge only if it does not contain any sequential compositions. Let $[P]$ denote the set of reduction relations that comprise the program P. Figure 3 defines the rules, closely following the form of the rules of figure 1.

$$\frac{}{\mathbf{r}\downarrow}$$

$$\frac{P \overset{R}{\rightsquigarrow} P'}{P \,;\, Q \overset{R}{\rightsquigarrow} P' \,;\, Q} \qquad \frac{P\downarrow}{P \,;\, Q \overset{[P]}{\rightsquigarrow} Q}$$

$$\frac{P \overset{R}{\rightsquigarrow} P'}{P \parallel Q \overset{R}{\rightsquigarrow} P' \parallel Q} \qquad \frac{Q \overset{R}{\rightsquigarrow} Q'}{P \parallel Q \overset{R}{\rightsquigarrow} P \parallel Q'} \qquad \frac{P\downarrow \quad Q\downarrow}{P \parallel Q\downarrow}$$

Figure 3: Abstract Passive Steps

Active Region We abstract the active steps of a program simply by saying which reductions in the program are immediately applicable. The immediately-applicable reductions are just those which are not "guarded" by a sequential composition on their left. The *active region* of a program P, written $\lceil P \rceil$, is defined inductively by:

$$\begin{aligned} \lceil \mathbf{r} \rceil &= \{\mathbf{r}\} \\ \lceil P \,;\, Q \rceil &= \lceil P \rceil \\ \lceil P \parallel Q \rceil &= \lceil P \rceil \cup \lceil Q \rceil \end{aligned}$$

The following proposition states the precise relationship between the above abstractions and the transition relation of the structural operational semantics:

Proposition 8 *For all composed reduction systems P over some universe $\mathbf{U}$, and for all $M, N \in \mathbf{U}$,*
$\langle P, M \rangle \rightarrow \langle Q, N \rangle$ *if and only if either*

1. $M = N$ *and* $P \overset{R}{\rightsquigarrow} Q$ *for some* R *such that for all* $\mathbf{r} \in R$, $M\downarrow^{\mathbf{r}}$, *or*

2. $P = Q$, *and there exits some* $\mathbf{r} \in \lceil P \rceil$ *such that* $M \rightarrow_{\mathbf{r}} N$.

The graph form will be constructed by combining the passive steps with the active regions. We note the following facts about the passive steps.

- Passive steps are normalising: ie. there are no infinite chains of the form $P \overset{R_1}{\rightsquigarrow} P_1 \overset{R_2}{\rightsquigarrow} P_2 \overset{R_3}{\rightsquigarrow} \cdots$, since the size of the programs are strictly decreasing with each passive step.

- For any P, the number of R and Q such that $P \overset{R}{\rightsquigarrow} Q$ is finite.

In fact, $\rightsquigarrow$ satisfies a "strong diamond property", namely that if $P \overset{R_1}{\rightsquigarrow} P_1$ and $P \overset{R_2}{\rightsquigarrow} P_2$ with $P_1 \neq P_2$, then there is a Q such that $P_1 \overset{R_2}{\rightsquigarrow} Q$ and $P_2 \overset{R_1}{\rightsquigarrow} Q$.

The graph form of a program P is rooted directed finite acyclic graph formed by (i) forming the passive-graph according to the $\rightsquigarrow$ relation, and then (ii) mapping the function $\lceil \_ \rceil$ over the nodes of the passive-graph to extract their active reductions. So, for example, taking the program $(\mathbf{r}_1 ; \mathbf{r}_2) \mathbin{\#} (\mathbf{r}_3 ; \mathbf{r}_4)$ we (i) construct the passive graph:

$$\begin{tikzcd}
& (\mathbf{r}_1 ; \mathbf{r}_2) \mathbin{\#} (\mathbf{r}_3 ; \mathbf{r}_4) \arrow[dl, "\{\mathbf{r}_3\}"'] \arrow[dr, "\{\mathbf{r}_1\}"] & \\
(\mathbf{r}_1 ; \mathbf{r}_2) \mathbin{\#} \mathbf{r}_4 \arrow[dr, "\{\mathbf{r}_1\}"'] & & \mathbf{r}_2 \mathbin{\#} (\mathbf{r}_3 ; \mathbf{r}_4) \arrow[dl, "\{\mathbf{r}_3\}"] \\
& \mathbf{r}_2 \mathbin{\#} \mathbf{r}_4 &
\end{tikzcd}$$

and (ii) abstract the active region from each node to obtain:

$$\begin{tikzcd}
& \{\mathbf{r}_1, \mathbf{r}_3\} \arrow[dl, "\{\mathbf{r}_3\}"'] \arrow[dr, "\{\mathbf{r}_1\}"] & \\
\{\mathbf{r}_1, \mathbf{r}_4\} \arrow[dr, "\{\mathbf{r}_1\}"'] & & \{\mathbf{r}_2, \mathbf{r}_3\} \arrow[dl, "\{\mathbf{r}_3\}"] \\
& \{\mathbf{r}_2, \mathbf{r}_4\} &
\end{tikzcd}$$

3.2 Reasoning from Graphs

It should be clear from Proposition 8 that the transition traces of a program can be constructed from its graph. In fact, from the point of view of giving the operational semantics a program P, we can use the *tree* corresponding to the graph.

We will show how the graph representation can be used to reason about strong equivalence and strong approximation between programs. For the purpose of the behaviours (or transition traces) of programs, only need the set of complete paths through the graph.

Let $\mathbf{paths}(P)$ denote the complete (and necessarily finite) paths in the graph of P. So the graph of a program $\mathbf{r}_1 ; (\mathbf{r}_2 \mathbin{\#} \mathbf{r}_3)$ is just $\{\mathbf{r}_1\} \xrightarrow{\{\mathbf{r}_1\}} \{\mathbf{r}_2, \mathbf{r}_3\}$, and so the program has just a single path, $\langle \{\mathbf{r}_1\} \, \{\mathbf{r}_1\} \, \{\mathbf{r}_2, \mathbf{r}_3\} \rangle$

The domain of paths (ranged over by p_1, p_2 etc.) are the finite, nonempty odd-length sequences of sets of reduction relations. Writing concatenation of sequences by juxtaposition, if R, R_1, R_2 etc. range over sets of reduction relations, then a path is either a sequence of length one, $\langle R \rangle$, or a sequence of the form $\langle R_1, R_2 \rangle p$ for some path p. Alternatively we will denote a path by $\langle n_1 a_1 n_2 a_2 \ldots a_{k-1} n_k \rangle$ where the n_i (nodes) and a_i (arcs) are again sets of reduction relations.

The paths of a composed reduction system can be defined directly by induction on the passive steps:

Definition 9 *The paths of a program P,* $\mathbf{paths}(P)$*, is the least set of nonempty sequences of sets of reduction relations, such that:*

- *if* $P\downarrow$ *then* $[P] \in \mathbf{paths}(P)$
- *if* $P \overset{R}{\rightsquigarrow} P'$ *and* $p' \in \mathbf{paths}(P')$ *then* $\langle \lceil P \rceil, R \rangle p' \in \mathbf{paths}(P)$.

Now, in turn, the transition traces of a program can be defined in terms of its paths. This is given in the appendix.

The first implication of this is that if two programs have equivalent paths, then they must be strongly equivalent. We conjecture a tighter relationship, namely:

Conjecture 10 $\mathbf{paths}(P_1) = \mathbf{paths}(P_2)$ *if and only if* $(P_1 \equiv_o P_2)$ *is provable from the equational theory generated by the laws:*

(i) $\mathbf{r} \equiv_o \mathbf{r} \,\#\, \mathbf{r}$ $\quad (ii)$ $\#$ *is associative and commutative* $\quad (iii)$; *is associative.*

In fact, other than a few laws for the desynchroniser Δ, we have not found any other strong equivalences (than those derivable from the above). The inequational theory for $\sqsubseteq_o$ is, however, much richer. But proving inequalities from the transition traces (so far the only method we have) is rather tedious. Now we consider how to reason about $\sqsubseteq_o$ by building comparison relations on path-sets.

3.3 Path Comparisons

Each "node" represent the possible reductions possible at that node. The reductions on each "edge" represent the termination condition—a set of reductions which must be inapplicable for control to transfer along that edge. Comparing two paths of the same length, one path describes a broader range of behaviours than another, if it has at least as many reductions at each corresponding node (the odd elements of the sequence) but no more reductions on each edge (the even elements of the sequence). This leads to the following:

Definition 11 (Path Inclusion)
Two paths of equal length,
$p = \langle n_1, a_1, n_2 \ldots a_{k-1}, n_k \rangle$ *and* $p' = \langle n'_1, a'_1, n'_2 \ldots a'_{k-1}, n'_k \rangle$,
are in the path-inclusion ordering, written $p \leq p'$, *if*

1. $n_k = n'_k$,
2. $n_i \subseteq n'_i$, *for all* $i < k$, *and*
3. $a'_i \subseteq a_i$, *for all* $i \leq k$.

The path inclusion ordering is defined on composed reduction systems as:
$P \leq Q$ *if and only if for all* $p \in \mathbf{paths}(P)$ *there exists a path* $q \in \mathbf{paths}(Q)$ *such that* $p \leq q$.

Note that there is a stronger condition on the last node of a path. This is because the last node carries additional significance, since it is also a termination condition.

Proposition 12 $P \leq Q \Rightarrow P \sqsubseteq_o Q$

PROOF Since the transition traces are easily constructed from the paths, the proposition can be proved by showing that if $P \leq Q$ then the transition traces of P are contained in those of Q. Given the fact that the behaviours are extractable from the paths, a more direct (and arguably more useful) proof can be given by a compositional definition of the paths of a program. A compositional construction of paths is given in the appendix. $\square$

Consider, for example, the composed reduction system $\mathbf{r}_1 ; \mathbf{r}_3 ; (\mathbf{r}_2 \mathbin{+\!\!\!+} \mathbf{r}_4)$:

$$\mathbf{paths}(\mathbf{r}_1 ; \mathbf{r}_3 ; (\mathbf{r}_2 \mathbin{+\!\!\!+} \mathbf{r}_4)) = \{\langle \{\mathbf{r}_1\}, \{\mathbf{r}_1\} \{\mathbf{r}_3\} \{\mathbf{r}_3\} \{\mathbf{r}_2, \mathbf{r}_4\}\rangle\}.$$

Since we have $\langle \{\mathbf{r}_1, \mathbf{r}_3\}, \{\mathbf{r}_1\} \{\mathbf{r}_2, \mathbf{r}_3\} \{\mathbf{r}_3\} \{\mathbf{r}_2, \mathbf{r}_4\}\rangle \in \mathbf{paths}((\mathbf{r}_1 ; \mathbf{r}_2) \mathbin{+\!\!\!+} (\mathbf{r}_3 ; \mathbf{r}_4))$ then we can conclude that $\mathbf{r}_1 ; \mathbf{r}_3 ; (\mathbf{r}_2 \mathbin{+\!\!\!+} \mathbf{r}_4) \sqsubseteq_o (\mathbf{r}_1 ; \mathbf{r}_2) \mathbin{+\!\!\!+} (\mathbf{r}_3 ; \mathbf{r}_4)$.

Path Stuttering

The main limitation of the path-inclusion ordering is that we can only compare paths of equal length. So, for example, we cannot prove the inequaility:

$$(\mathbf{r}_1 \mathbin{+\!\!\!+} \mathbf{r}_3) ; (\mathbf{r}_2 \mathbin{+\!\!\!+} \mathbf{r}_4) \sqsubseteq_o (\mathbf{r}_1 ; \mathbf{r}_2) \mathbin{+\!\!\!+} (\mathbf{r}_3 ; \mathbf{r}_4)$$

since the path of $(\mathbf{r}_1 \mathbin{+\!\!\!+} \mathbf{r}_3) ; (\mathbf{r}_2 \mathbin{+\!\!\!+} \mathbf{r}_4)$ (there is only one) is shorter than all the paths of $(\mathbf{r}_1 ; \mathbf{r}_2) \mathbin{+\!\!\!+} (\mathbf{r}_3 ; \mathbf{r}_4)$.

The solution is to define an analogue of closure under stuttering, at the level of paths. We do not literally add stuttering paths, but rather, paths which give rise to stuttering. Consider a path of the form $p_1\langle n, a\rangle p_2$. The arc a represents an internal termination step. Operationally, after this step is performed, we could offer some reductions from a, say n', and none will be applicable—and hence we can converge for all reductions in n'. Hence the path $p_1\langle n, a, n', n'\rangle p_2$ describes no more (but no fewer) behaviours than $p_1\langle n, a\rangle p_2$. This leads us to a definition of *path stuttering equivalence*

Definition 13 *Let path stuttering equivalence, $=_{\mathrm{s}}$, be the least equivalence relation on paths such that*

for all paths p_1, p_2 (p_1 possibly empty), and for all sets of reductions n, a, n' such that $n' \subseteq a \subseteq n$,

$$p_1\langle n, a\rangle p_2 =_{\mathrm{s}} p_1\langle n, a, n', n'\rangle p_2$$

For example, $\langle \{\mathbf{r}_1, \mathbf{r}_2\} \{\mathbf{r}_1, \mathbf{r}_2\} \{\mathbf{r}_3\}\rangle =_{\mathrm{s}} \langle \{\mathbf{r}_1, \mathbf{r}_2\} \{\mathbf{r}_1, \mathbf{r}_2\} \{\mathbf{r}_1\} \{\mathbf{r}_1\} \{\mathbf{r}_3\}\rangle$. With the development that follows, we will be able to conclude that

$$(\mathbf{r}_1 \mathbin{+\!\!\!+} \mathbf{r}_2) ; \mathbf{r}_3 \equiv_o (\mathbf{r}_1 \mathbin{+\!\!\!+} \mathbf{r}_2) ; \mathbf{r}_1 ; \mathbf{r}_3$$

Now we use path stuttering equivalence to coarsen the path inclusion ordering. As before we define a preorder on paths, and extend this to programs in the obvious way:

Definition 14 (Stuttered Path Inclusion) *Two paths, p and p', are in the stuttered path-inclusion ordering, written $p \leq_s p'$ if there exists p_1, p_2 such that*

$$p =_{\mathrm{s}} p_1 \leq p_2 =_{\mathrm{s}} p'.$$

On composed reduction systems we define $P \leq_s Q$ if and only if for all $p \in \mathbf{paths}(P)$ there exists a path $q \in \mathbf{paths}(Q)$ such that $p \leq_s q$.

Proposition 15 $P \leq_s Q \Rightarrow P \sqsubseteq_o Q$

PROOF (Outline) It is sufficient to show that $P \leq_s Q \Rightarrow \ddagger\mathsf{T}[\![P]\!] \subseteq \ddagger\mathsf{T}[\![Q]\!]$. The main step is to show that the closed ($\ddagger$) traces corresponding to a path $p_1\langle n, a\rangle p_2$ are equal to the closed traces of $p_1\langle n, a, n', n'\rangle p_2$, whenever $n' \subseteq a \subseteq n$. We omit the details. □

References

[Abr79] K. Abrahamson. Modal logic of concurrent nondeterministic programs. In *Proceedings of the International Symposium on Semantics of Concurrent Computation*, volume 70, pages 21–33. Springer-Verlag, 1979.

[Bac89a] R. Back. A method for refining atomicity in parallel algorithms. In *PARLE '89, volume II*, number 365 in LNCS. Springer-Verlag, 1989.

[Bac89b] R. Back. Refinement calculus, part ii: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, number 430 in LNCS. Springer-Verlag, 1989.

[BM92] J.-P. Banâtre and D. Le Métayer, editors. *Research Directions in High-level Parallel Programming Languages.* Springer-Verlag, LNCS 574, 1992.

[BM93] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *CACM*, January 1993. (INRIA research report 1205, April 1990).

[Bro93] S. Brookes. Full abstraction for a shared variable parallel language. In *Logic In Computer Science (LICS).* IEEE, 1993.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[dBKPR91] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. The failure of failures in a paradigm for asynchronous communication. In *CONCUR '91*, number 527 in Lecture Notes in Computer Science, pages 111–126. Springer-Verlag, 1991.

[Dij76] E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DJ89] N. Dershowitz and J-P. Jouannaud. *Rewrite Systems*, volume B, chapter 15. North-Holland, 1989.

[Hin69] R. Hindley. An abstract form of the Church-Rosser theorem. *J. Symbolic Logic*, 34(1), 1969.

[HMS92] C. Hankin, D. Le Métayer, and D. Sands. A calculus of Gamma programs. Research Report DOC 92/22 (28 pages), Department of Computing, Imperial College, 1992. (short version to appear in the Proceedings of the Fifth Annual Workshop on Languages and Compilers for Parallelism, Aug 1992, Springer-Verlag).

[HP79] M. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming lanuage. In *Mathematical Foundations of Computer Science*, volume 74 of *LNCS*, pages 108–120. Springer-Verlag, 1979.

[Klo92] J. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. OUP, 1992.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *C. ACM*, 31(1):32–45, January 1989.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *TCS*, 94, 1992.

[MW91] J. Meseguer and T. Winkler. Parallel programming in maude. In *In [BM92]*, 1991.

[Par79] D. Park. On the semantics of fair parallelism. In *Abstract Software Specifications (1979 Copenhagen Winter School Proceedings)*, number 86 in Lecture Notes in Computer Science, pages 504–526. Springer-Verlag, 1979.

[Plo76] G. D. Plotkin. A powerdomain construction. *Siam J. Comput.*, 5(3):452–487, September 1976.

[Ros73] B. Rosen. Tree-manipulating systems and Church-Rosser theorems. *J. ACM*, 20(1):160–187, January 1973.

[San93a] D. Sands. A compositional semantics of combining forms for Gamma programs. In *International Conference on Formal Methods in Programming and Their Applications.* Springer-Verlag, 1993.

[San93b] D. Sands. Laws of parallel synchronised termination. In *Theory and Formal Methods 1993: Proceedings of the First Imperial College, Department of Computing, Workshop on Theory and Formal Methods*, Isle of Thorns, UK, 1993. Springer-Verlag Workshops in Computer Science.

[Sta74] J. Staples. Church-Rosser theorems for replacement systems. In *Algebra and Logic: papers from the Summer research institute of the Australian Mathematical Society*, number 450 in Lecture Notes in Mathematics. Springer-Verlag, 1974.

A Compositional Definition of Transition Traces

To give the compositional construction of transition traces we need to define the appropriate sequential and parallel composition operators over sets of traces.

Notation In what follows we will adopt the following notation. If S is a set, then S^* will denote the set of finite sequences of elements from S, S^+ will denote the finite non-empty sequences, and S^∞ will denote the infinite sequences. The power-set is denoted by $\wp(S)$.

Note in particular that for a reduction relation $\rightarrow_{\mathbf{r}}$, we will write $(\rightarrow_{\mathbf{r}})^*$ to denote the finite sequence of pairs contained in $\rightarrow_{\mathbf{r}}$, and *not* the transitive-reflexive closure of the relation.

Sequential Composition Sequential composition has an easy definition. We just take all concatenations of the atomic traces of the components. As is usual, if α and β are traces, then $\alpha\beta$ denotes their concatenation, which is just α when α is infinite. Define the following sequencing operation for trace sets:

$$T_1 \odot T_2 = \{\alpha\beta \mid \alpha \in T_1, \beta \in T_2\}$$

End-synchronised Merge Not surprisingly, parallel composition is described with the use of a merging combinator which interleaves traces. The peculiarities of parallel composition are prominent in the definition. To define the transition traces of $P_1 \mathbin{\#} P_2$ we must ensure that the traces of P_1 and P_2 are interleaved, but not arbitrarily; the termination step of a parallel composition requires an agreement, or synchronisation, at the point of their termination. To build up the picture, suppose α and β in $(\mathbf{U} \times \mathbf{U})^+$ are traces of some programs P_1 and P_2 respectively. The set of all interleavings of α and β which correspond to possible executions of $P_1 \mathbin{\#} P_2$ can be given inductively by:

$$(M, M) \sharp (N, N) = \begin{cases} \{(M, M)\} & if \quad M = N \\ \emptyset & otherwise \end{cases}$$

$$\begin{array}{rcl} (M, M')\alpha \sharp (N, N')\beta & = & \{(M, M')\gamma \mid \alpha \neq \epsilon, \gamma \in \alpha \sharp (N, N')\beta\} \\ & \cup & \{(N, N')\gamma \mid \beta \neq \epsilon, \gamma \in (M, M')\alpha \sharp \beta\} \\ & & \textit{if either } \alpha \neq \epsilon \textit{ or } \beta \neq \epsilon. \end{array}$$

To generalise the definition to incorporate the infinite traces as well as the finite, we need to define the interleavings via a *maximal* fixed-point rather than a *minimal* fixed point as implicit in the above definition. There are many possible ways of presenting this construction. We choose an implicit definition of the required maximal fixed-point:

Definition 16 *A function* $\mathbf{m}$*, from pairs of nonempty traces to sets of nonempty traces, is an end-synchronised merger (ESM), if the following conditions are satisfied:*

1. *if* $(M, N) \in \mathbf{m}(\beta, \gamma)$ *then* $\beta = \gamma = (M, N)$*;*

2. *if* $\alpha \in \mathbf{m}(\beta,\gamma)$ *then* $(M,N)\alpha \in \mathbf{m}((M,N)\beta,\gamma)$
$\quad \& \ (M,N)\alpha \in \mathbf{m}(\gamma,(M,N)\beta)$

Definition 17 *The generalised end-synchronised merge is given by the point-wise union of all end-synchronised mergers:*

$$\alpha \sharp \beta = \cup\{\mathbf{m}(\alpha,\beta) \mid \mathbf{m} \textit{ is an ESM}\}$$

Note that $\sharp$ is an ESM (this follows from the Knaster-Tarski fixed-point theorem), and therefore the largest ESM. The corresponding relation on sets of traces will provide the denotation of parallel composition:

$$T_1 \oplus T_2 = \{\gamma \mid \alpha \in T_1, \beta \in T_2, \gamma \in \alpha \sharp \beta\}$$

Definition 18 *The compositional atomic trace mapping* $\mathsf{T}_c[\![\_]\!] : \mathbf{P} \to \wp((\mathbf{U} \times \mathbf{U})^+ \cup (\mathbf{U} \times \mathbf{U})^\infty)$ *is given by induction on the syntax as:*

$$\begin{array}{rcl}
\mathsf{T}_c[\![\mathbf{r}]\!] & = & \ddagger((\rightarrow_{\mathbf{r}})^* \odot \{(M,M) \mid M{\downarrow}^{\mathbf{r}}\}) \quad \cup \quad \ddagger((\rightarrow_{\mathbf{r}})^\infty) \\
\mathsf{T}_c[\![P_1 \,;\, P_2]\!] & = & \ddagger(\mathsf{T}_c[\![P_2]\!] \odot \mathsf{T}_c[\![P_1]\!]) \\
\mathsf{T}_c[\![P_1 \mathbin{+\!\!\!+} P_2]\!] & = & \ddagger(\mathsf{T}_c[\![P_1]\!] \oplus \mathsf{T}_c[\![P_2]\!])
\end{array}$$

Soundness It is straightforward to show that $\mathsf{T}_c[\![\_]\!]$ is sound with respect to behaviours of a program. The following lemma gives the basic operational correspondence:

Lemma 19 1. $\langle P, M\rangle \to M \Rightarrow (M,M) \in \mathsf{T}_c[\![P]\!]$

2. $\langle P, M\rangle \to \langle P', N\rangle \ \&\ \alpha \in \mathsf{T}_c[\![P']\!] \Longrightarrow (M,N)\alpha \in \mathsf{T}_c[\![P]\!]$

Proposition 20 $\mathsf{T}_c[\![P]\!] \subseteq \mathsf{T}_c[\![Q]\!] \Longrightarrow P \sqsubseteq_o Q$

PROOF From Lemma 19 it is easy to see that $\mathsf{T}_c[\![P]\!] \subseteq \mathsf{T}_c[\![Q]\!] \Rightarrow \mathcal{B}(P) \subseteq \mathcal{B}(Q)$. The operations used to build the compositional definition are all monotone with respect to subset inclusion, and so a simple induction on contexts is sufficient to give

$$\mathsf{T}_c[\![P]\!] \subseteq \mathsf{T}_c[\![Q]\!] \Rightarrow \forall \mathbf{C}.\, \mathsf{T}_c[\![\mathbf{C}[P_1]]\!] \subseteq \mathsf{T}_c[\![\mathbf{C}[P_2]]\!].$$

Putting these together we have

$$\begin{array}{rcl}
\mathsf{T}_c[\![P]\!] \subseteq \mathsf{T}_c[\![Q]\!] & \Rightarrow & \forall \mathbf{C}.\, \mathcal{B}(\mathbf{C}[P_1]) \subseteq \mathcal{B}(\mathbf{C}[P_2]) \\
& \Longleftrightarrow & P_1 \sqsubseteq_o P_2.
\end{array}$$

□

B Path Constructions

Transition Traces from Paths The transition traces can be constructed from the paths as follows (overloading the mapping $\mathsf{T}[\![\_]\!]$):

$$\begin{array}{rcl}
\mathsf{T}[\![\langle R\rangle]\!] & = & (T_R)^\infty \cup ((T_R)^* \odot \{(M,M) \mid \mathbf{r} \in R, M{\downarrow}^{\mathbf{r}}\}) \\
\mathsf{T}[\![\langle R, R'\rangle\sigma]\!] & = & (T_R)^\infty \cup ((T_R)^* \odot \{(M,M) \mid \mathbf{r} \in R', M{\downarrow}^{\mathbf{r}}\} \odot \mathsf{T}[\![\sigma]\!]) \\
& \text{where} & T_R = \{(M,N) \mid \mathbf{r} \in R, M {\rightarrow_{\mathbf{r}}} N\}
\end{array}$$

Paths Constructed Compositionally

Proposition 21 *The following equations uniquely characterise the paths of a program:*

$$
\begin{array}{rcl}
\mathbf{paths}(\mathbf{r}) & = & \langle\{\mathbf{r}\}\rangle \\
\mathbf{paths}(P\,;Q) & = & \{\sigma_1\langle R\rangle\sigma_2 \mid \sigma_1 \in \mathbf{paths}(P), \sigma_2 \in \mathbf{paths}(Q), R = \mathit{last}(\sigma_1)\} \\
\mathbf{paths}(P \mathbin{+\!\!\!+} Q) & = & \{\sigma_1 \otimes \sigma_2 \mid \sigma_1 \in \mathbf{paths}(P), \sigma_2 \in \mathbf{paths}(Q)\}
\end{array}
$$

where

$$
\begin{array}{rcl}
\langle R\rangle \otimes \langle R'\rangle & = & \langle R \cup R'\rangle \\
\langle R_1, R_2\rangle\sigma \otimes \langle R'\rangle & = & \langle R'\rangle \otimes \langle R_1 R_2\rangle\sigma \\
 & = & \{\langle (R_1 \cup R'), R_2\rangle\sigma' \mid \sigma' \in \langle R'\rangle \otimes \sigma\} \\
 & & \\
\langle R_1, R_2\rangle\sigma_1 \otimes \langle R_1', R_2'\rangle\sigma_1' & = & \{\langle (R_1 \cup R_1'), R_2\rangle\sigma \mid \sigma \in \sigma_1 \otimes \langle R_1', R_2'\rangle\sigma_1'\} \\
 & \cup & \{\langle (R_1 \cup R_1'), R_2'\rangle\sigma' \mid \sigma' \in \langle R_1, R_2\rangle\sigma_1 \otimes \sigma_1'\}
\end{array}
$$

Towards Operational Semantics of Contexts in Functional Languages

David Sands
DIKU, University of Copenhagen*

Abstract

We consider operational semantics of contexts (terms with holes) in the setting of lazy functional languages, with the aim of providing a balance between operational and compositional reasoning, and a framework for semantics-based program analysis and manipulation.

Introduction

In this note we initiate a new direction in the semantics of functional programs. The approach is based on operational semantics; our aims are to provide a operational route to high-level semantic issues, such as program analysis and source-to-source transformation. We investigate the idea of giving a direct operational semantics to *program contexts*—that is, "incomplete" programs containing a number of *holes* in the place of some subexpressions.

The idea of providing an operational semantics for contexts has been studied by Larsen (et al) for process algebras [Lar86][LX91]. In that setting, a context is viewed as an *action transducer*, which consumes actions provided by its internal processes (the holes) and produces externally observable actions. The operational semantics of contexts contains transitions of the form

$$\mathcal{C} \xrightarrow[a]{b} \mathcal{C}'$$

which is interpreted as: *by consuming action a, context $\mathcal{C}$ can produce action b and change into $\mathcal{C}'$.*

We describe some initial steps towards providing an operational semantics for contexts in a functional setting.

Functional Action-Transducers

In the process setting a context is viewed as an *action transducer*. What is the corresponding notion for a context in the functional setting? In a functional language, the role of an "action" is played by the *observables* of the language: namely a lazy data constructor— `cons`, `true`, "λ".

*Universitetsparken 1, 2100 København Ø, DENMARK. e-mail: `dave@diku.dk`

We take a bold step, and demand that the "actions" should themselves be contexts—but not arbitrary contexts. They should be contexts built from the observables of the language. We will call these *observable contexts*. Observable contexts will be ranged-over by $\mathcal{O}$, $\mathcal{O}'$, etc. For some context $\mathcal{C}$ containing occurrences of a single hole, if we have a *transduction* of the form:

$$\mathcal{C} \xrightarrow[\mathcal{O}]{\mathcal{O}'} \mathcal{C}'$$

then we will require that:

$$\mathcal{C}[\mathcal{O}] \simeq \mathcal{O}'[\mathcal{C}'] \qquad (*)$$

where the notation $\mathcal{C}[e]$ denotes context $\mathcal{C}$ with e placed "in the hole", and $\simeq$ denotes the usual operational equivalence, extended to capture-free contexts in the obvious way. This equation does not capture everything we expect from a context semantics. What we expect is that the transduction should be as lazy as possible — ie. we could not have found a smaller "input" $\mathcal{O}$ that would have given the same observable "output" $\mathcal{O}'$. If the language is *sequential*[Ber78] then we expect the semantics to give us the *minimal* $\mathcal{O}$.

For example, a context of the form `if [] = 0 then` $\mathcal{C}_1$ `else (leaf` $\mathcal{C}_2$`)` (containing multiple occurrences of a single hole), where `leaf` is a constructor, would have a transduction:

$$\texttt{if [] = 0 then } \mathcal{C}_1 \texttt{ else (leaf } \mathcal{C}_2\texttt{)} \xrightarrow[\texttt{suc []}]{\texttt{leaf []}} \mathcal{C}_2 \circ \texttt{suc[]}$$

where $\circ$ is context composition, so $\mathcal{C}_2 \circ$ `suc[]` denotes the context $\mathcal{C}_2$`[suc[]]`.

Problems Giving a full operational semantics for contexts is difficult because:

- contexts can consume without producing an observable.

  For example `if [] = 0...` can consume a constructor without necessarily being able to produce anything. (Similar situation would arise in Larsen's work if one did not consider the silent action τ to be observable.)

- The number of occurrences of a given hole may increase under transduction (assuming we use some mechanism like β-reduction in our semantics).

- The number of distinct holes in a context can increase under transduction, in the presence of n-ary constructors.

  eg. If $\mathcal{C}_1$ has a single hole, and $\mathcal{C}_1 \xrightarrow[\texttt{cons[]}_1\texttt{[]}_2]{} \mathcal{C}_2$ then $\mathcal{C}_2$ has two distinct holes.

- How do we treat higher-order functions?

- Contexts can capture variables by means of binding operators

  eg.
  ```
  case e of           .
         nil    ⇒ e'
      cons h t  ⇒ C₂
  ```

A Simplified Case

As a first step we study only a very simple language. We avoid almost all of the above problems by considering a language with the following features:

- First-order functions
- No binding operators
- Unary constructors and constants (= nullary constructors) as the only values.

The language we consider consists of first-order recursion equations with possible pattern-matching on the first argument (non-nested), based on unary or nullary constructors. Here are some example definitions:

```
add 0 x         =  x
add (suc y) x   =  suc (add y x)

twice x         =  add x x
```

Notation Let u range over both unary constructors (eg. `suc`) and constants (eg. `0`,`true` etc.). The observable contexts are then given by

$$\mathcal{O} \quad ::= \quad [\,] \quad | \quad u \quad | \quad u\,\mathcal{O}$$

For simplicity of presentation, we will only consider contexts with a single hole (occurring zero or more times). (We will consider an expression to be a context with zero occurrences of the hole.) For this restricted language, the extension to handle *polyadic* contexts (contexts with several distinct holes) is straightforward (eg. borrowing the notations from [LX91]).

In the context transductions, if u is a unary constructor, then we write observable context $u\,[\,]$ as simply u, and occurrences of the trivial context $[\,]$ will simply be omitted from the transductions, so we will write $\mathcal{C} \xrightarrow{\texttt{suc}} \mathcal{C}'$ in place of $\mathcal{C} \xrightarrow[{[\,]}]{\texttt{suc}\,[\,]} \mathcal{C}'$. If u is a constant, then u can also be denoted $u()$. We add the unit expression $()$ to the language of contexts.

Language Rules

We define the following transductions involving terms of the language:

$$u\,\mathcal{C} \xrightarrow{u} \mathcal{C} \tag{1}$$

$$\texttt{f}\ \vec{\mathcal{C}} \longrightarrow \texttt{e}\{\vec{\texttt{x}} := \vec{\mathcal{C}}\} \qquad \texttt{if f}\ \vec{x} = e \tag{2}$$

$$\frac{\mathcal{C}_1 \xrightarrow[\mathcal{O}]{u} \mathcal{C}_2}{\mathtt{f}\ \mathcal{C}_1\ \vec{\mathcal{C}} \xrightarrow[\mathcal{O}]{} e\{y := \mathcal{C}_2\}\{\vec{x} := \vec{\mathcal{C}} \circ \mathcal{O}\}} \quad \text{if } \mathtt{f}\ (u\ y)\ \vec{x} = e \tag{3}$$

In the last rule, the composition $\vec{\mathcal{C}} \circ Con'$ denotes the vector of contexts obtained by composing each context in $\vec{\mathcal{C}}$ with Con'. To check that the rule satisfies the desired property $(*)$, assume that from the antecedent we have

$$\mathcal{C}_1 \circ \mathcal{O} \simeq u\ \mathcal{C}_2$$

then

$$\begin{aligned}(\mathtt{f}\ \mathcal{C}_1\ \vec{\mathcal{C}}) \circ \mathcal{O} &\simeq \mathtt{f}\ (\mathcal{C}_1 \circ \mathcal{O})\ (\vec{\mathcal{C}} \circ \mathcal{O}) \\ &\simeq \mathtt{f}\ (u\ \mathcal{C}_2)\ (\vec{\mathcal{C}} \circ \mathcal{O}) \\ &\simeq e\{y := \mathcal{C}_2\}\{\vec{x} := \vec{\mathcal{C}} \circ \mathcal{O}\}\end{aligned}$$

These rules are straightforward, since they follow the "small-step" semantics of the language (ie. they are non compositional.) We recover the ability to reason compositionally using the following context rules:

Context Rules

$$[\,] \xrightarrow[u]{u} [\,] \quad (u \text{ unary}) \tag{4}$$

$$[\,] \xrightarrow[u]{u} () \quad (u \text{ constant}) \tag{5}$$

$$\mathcal{C} \longrightarrow \mathcal{C} \tag{6}$$

$$\frac{\mathcal{C}_1 \xrightarrow[\mathcal{O}_1']{\mathcal{O}_1} \mathcal{C}_2 \quad \mathcal{C}_2 \xrightarrow[\mathcal{O}_2']{\mathcal{O}_2} \mathcal{C}_3}{\mathcal{C}_1 \xrightarrow[\mathcal{O}_1' \circ \mathcal{O}_2']{\mathcal{O}_1 \circ \mathcal{O}_2} \mathcal{C}_3} \tag{7}$$

$$\frac{\mathcal{C}_1 \xrightarrow[\mathcal{O}_2]{\mathcal{O}_1} \mathcal{C}_1' \quad \mathcal{C}_2 \xrightarrow[\mathcal{O}_3]{\mathcal{O}_2} \mathcal{C}_2'}{\mathcal{C}_1 \circ \mathcal{C}_2 \xrightarrow[\mathcal{O}_3]{\mathcal{O}_1} \mathcal{C}_1' \circ \mathcal{C}_2'} \tag{8}$$

The last rule is the *uniform rule* of [Lar89][LX91], and it characterises all the transductions of composed contexts.

Properties

Closed expressions can be viewed as contexts containing zero holes. In this way the rules above can be seen to subsume the usual large-step and small-step structural operational semantics. Suppose the large step semantics defines an evaluation relation $\Downarrow$ (we omit the routine definition), then we have the following:

$$e \Downarrow a \iff e \xrightarrow{u} e' \quad \wedge \quad (u\ e' \equiv a)$$

For example, if `I` is the identity function, then we have the following example proof:

$$\text{compose}\ \dfrac{2\,\dfrac{}{\texttt{I (suc 0)} \longrightarrow \texttt{(suc 0)}} \qquad 1\,\dfrac{}{\texttt{(suc 0)} \xrightarrow{\texttt{suc}} \texttt{0}}}{\texttt{I (suc 0)} \xrightarrow{\texttt{suc}} \texttt{0}}$$

But this is not the whole story for evaluation of closed expressions. Unlike the structural operational semantics for $\Downarrow$, the *proof* of $e \xrightarrow{u} e'$ is not unique.

An important point is that by use of the composition rule (8) we can vary the compositionality. This means that when we need to prove a property of a function application `f` e, we can split this into a context `f` [ ] and the sub-term e, and derive the transition of the composed system in terms of these components.

As an example, using the functions defined earlier, consider the term `twice (I (suc0))`. We can prove that

$$7\ \dfrac{2\,\dfrac{}{\texttt{twice [ ]} \rightarrow \texttt{add [ ] [ ]}} \qquad A}{\texttt{twice [ ]} \xrightarrow[\texttt{suc}]{\texttt{suc}} \texttt{add [ ] (suc [ ])}}$$

where A is the sub-proof:

$$7\ \dfrac{4,1\ \dfrac{\texttt{[ ]} \xrightarrow[\texttt{suc}]{\texttt{suc}} \texttt{[ ]}}{\texttt{add [ ] [ ]} \xrightarrow[\texttt{suc}]{} \texttt{suc add [ ] (suc [ ])} \xrightarrow{\texttt{suc}} \texttt{add [ ] (suc [ ])}}}{\texttt{add [ ] [ ]} \xrightarrow[\texttt{suc}]{\texttt{suc}} \texttt{add [ ] (suc [ ])}}$$

and so using the composition rule we obtain:

$$8\ \dfrac{\texttt{twice [ ]} \xrightarrow[\texttt{suc}]{\texttt{suc}} \texttt{add [ ] (suc [ ])} \qquad \texttt{I (suc 0)} \xrightarrow{\texttt{suc}} \texttt{0}}{\texttt{twice (I (suc 0))} \xrightarrow{\texttt{suc}} \texttt{add 0 (suc 0)}}$$

Note that in the example, because of the use of the compositional rule, there is only one sub-proof for the expression `I (suc 0)`, whereas under the standard call by name SOS we would have two sub-proofs.

We conjecture that proofs which always treat function calls compositionally in this way (we need to generalise to n-ary contexts to do this for n-ary functions) have size proportional to the number of evaluation steps required under standard call-by-need computation. This form of proof corresponds to the *demand function* used in Bjerner and Holmström's call-by-need time-analysis [BH89].

Further Work

In the remainder of this paper we consider the directions for further development, which mostly concern tackling the problems of richer languages.

Polyadic Contexts

The above semantics is easily extended to handle polyadic contexts, but if we go beyond just unary constructors then the extension quickly becomes notationally complex. The problem is that consuming an observable context may give rise to several new holes, and producing a observable context means that a transduction may result in several contexts. Our proposal for dealing with these problems is to adopt a different kind of transduction. Instead of requiring that a transduction

$$\mathcal{C} \xrightarrow[\mathcal{O}]{\mathcal{O}'} \mathcal{C}'$$

implies that $\mathcal{C} \circ \mathcal{O} \simeq \mathcal{O}' \circ \mathcal{C}'$, the requirement is that

$$\mathcal{C} \circ \mathcal{O} \simeq \mathcal{O}' \circ \mathcal{C}' \circ \mathcal{O}$$

In this way the "type" of the holes in the derived context $\mathcal{C}'$ is the same as in $\mathcal{C}$. The addition of n-ary constructors means that $\mathcal{C}'$ must be a vector of contexts.

With this interpretation of context transductions, the uniform composition rule now has the form:

$$\frac{\mathcal{C}_1 \xrightarrow[\mathcal{O}_2]{\mathcal{O}_1} \mathcal{C}_1' \quad \mathcal{C}_2 \xrightarrow[\mathcal{O}_3]{\mathcal{O}_2} \mathcal{C}_2'}{\mathcal{C}_1 \circ \mathcal{C}_2 \xrightarrow[\mathcal{O}_3]{\mathcal{O}_1} \mathcal{C}_1' \circ \mathcal{O}_2 \circ \mathcal{C}_2'}$$

Higher-Order Functions

If we focus on contexts which cannot capture variables, then higher-order functions can be thought of as introducing an extra hole. We anticipate that to deal with polyadic contexts in their full generality we will need a notation along the lines of Martin-Lö f's theory of arities, so $(X)\mathcal{C}$ will denote a context with a single hole named X. Then a lambda-context could have a transition

$$(\vec{Y})\lambda y.\mathcal{C} \xrightarrow{\lambda} (X)(\vec{Y})\mathcal{C} \qquad X \notin \vec{Y}.$$

Abstract Contexts and Relativised Equivalence

Two natural directions are to consider static analysis problems, and notions of relativised equivalence. We should consider:

- Bisimulation-like characterisations of context-equivalence along the lines of [Abr90][How89].
- Relativised equivalences $\simeq_{\mathcal{C}}$:

$$e \ \simeq_{\mathcal{C}} \ e' \iff \mathcal{C}[e] \simeq \mathcal{C}[e']$$

- Semantics for abstract observable contexts (whose meaning is a set of contexts);
- A notion of *environment* as a dual to (abstract) observable contexts, thus generalising the demand semantics of [San93].

These points should enable an operational formalisation of *context analysis* of the form of [Hug87][WH87].

Guarded Contexts

In a somewhat orthogonal study we have considered context semantics for a restricted class of contexts (a form of guarded contexts) for a higher-order language with binding operators and arbitrary lazy constructors. The principle technical problem in this context semantics is to handle holes which occur under bound variables. This operational semantics of contexts finds immediate application to the problem of correct folding in program transformation. It also provides a simple form of "applicative bisimulation up to context" proof technique, a la Sangiorgi [San94].

Acknowlegement An earlier investigation of the subject of this note was undertaken together with Sebastian Hunt a few years ago. Our attempt failed, because we were over-ambitious in trying to give *only* compositional rules. But a number of ideas crystalised from our attempt, and have influenced the current development. One idea in particular —that the "actions" should themselves be contexts—is due to Sebastian.

References

[Abr90] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison Wesley, 1990.

[Ber78] G. Berry. Stable models of typed lambda calculi. In *5th Coll. on Automata Languages and Programming*, LNCS 62. Springer-Verlag, 1978.

[BH89] B. Bjerner and S. Holmström. A compositional approach to time analysis of first order lazy functional programs. In *Functional Programming Languages and Computer Architecture, conference proceedings*, pages 157–165. ACM press, 1989.

[How89] D. J. Howe. Equality in lazy computation systems. In *Fourth annual symposium on Logic In Computer Science*, pages 198–203. IEEE, 1989.

[Hug87] R. J. M. Hughes. Backwards analysis of functional programs. Research Report CSC/87/R3, University of Glasgow, March 1987.

[Lar86] K. G. Larsen. *Context-Dependent Bisimulation Between Processes.* PhD thesis, Department of Computing, University of Edinburgh, 1986.

[Lar89] K. G. Larsen. Compositinal theories based on an operational semantics of contexts. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, number 430 in LNCS. Springer-Verlag, 1989.

[LX91] K. G. Larsen and L. Xinxin. Compositinality through an operational semantics of contexts. *J. Logic and Computation*, 1(6):761–795, 1991.

[San93] D. Sands. A naïve time analysis and its theory of cost equivalence. TOPPS report D-173, DIKU, 1993. To appear: Journal of Logic and Computation, 1995.

[San94] D. Sangiorgi. On the bisimulation proof method. Technical report, University of Edinburgh, 1994.

[WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *1987 Conference on Functional Programming and Computer Architecture*, pages 385–407, Portland, Oregon, September 1987.

An Approach to the Category of Net Computations

Vladimiro Sassone°

BRICS* – Computer Science Dept., University of Aarhus

ABSTRACT. We introduce the notion of *strong concatenable process* as a refinement of concatenable processes [3] which can be expressed axiomatically via a *functor* $\mathcal{Q}[\_]$ from the category of Petri nets to an appropriate category of symmetric strict monoidal categories, in the precise sense that, for each net N, the strong concatenable processes of N are isomorphic to the arrows of $\mathcal{Q}[N]$. In addition, we identify a *coreflection* right adjoint to $\mathcal{Q}[\_]$ and characterize its *replete image*, thus yielding an axiomatization of the category of net computations.

Introduction

Petri nets, introduced by C.A. Petri in [8] (see also [10]), are unanimously considered among the most representative *models for concurrency*, since they are a fairly simple and natural model of *concurrent* and *distributed* computations. However, Petri nets are, in our opinion, not yet completely understood.

Among the semantics proposed for Petri nets, a relevant role is played by the various notions of *process* [9, 4, 1], whose merit is to provide a faithful account of computations involving many different transitions and of the *causal connections* between the events occurring in a computation. However, process models, at least in their standard forms, fail to bring to the foreground the *algebraic structure* of nets and their computations. Since such a structure is relevant to the understanding of nets, they fail, in our view, to give a comprehensive account of net behaviours.

The idea of looking at nets as *algebraic structures* [10, 7, 13, 14, 2] has been given an original interpretation by considering monoidal categories as a suitable framework [6]. In fact, in [6, 3] the authors have shown that the semantics of Petri nets can be understood in terms of *symmetric monoidal categories*—where objects are states, arrows processes, and the tensor product and the arrow composition model, respectively, the operations of parallel and sequential composition of processes. In particular, [3] introduced *concatenable processes*—the slightest variation of Goltz-Reisig processes [4] on which sequential composition can be defined—and structured the concatenable processes of a Petri net N as the arrows of the symmetric strict monoidal category $\mathcal{P}[N]$. This yields an axiomatization of the causal behaviour of a net as an *essentially algebraic theory* and thus provides a *unification* of the process and the algebraic view of net computations.

However, also this construction is somehow unsatisfactory, since it is not functorial. More strongly, given a morphism between two nets, i.e., a *simulation* between them, it may not be possible to identify a corresponding monoidal functor between the respective categories of computations. This fact, besides

* **B**asic **R**esearch **in** **C**omputer **S**cience, Centre of the Danish National Research Foundation.
° Supported by EU **H**uman **C**apital and **M**obility grant ERBCHBGCT920005.

showing that our understanding of the algebraic structure of Petri nets is still incomplete, prevents us from identifying the *category* (of the categories) *of net computations*, i.e., from axiomatizing the behaviour of Petri nets 'in the large'.

This paper presents an analysis of this issue and a solution based on the new notion of *strongly concatenable processes*, introduced in Section 4. These are a slight refinement of concatenable processes which are still rather close to the standard notion of process: they are Goltz-Reisig processes whose minimal and maximal places are linearly ordered. In the paper we show that, similarly to concatenable processes, also this new notion can be axiomatized as an algebraic construction on N by providing an abstract symmetric strict monoidal category $\mathcal{Q}[N]$ whose arrows are in one-to-one correspondence with the strongly concatenable processes of N. The category $\mathcal{Q}[N]$ constitutes our proposed axiomatization of the behaviour of N in categorical terms.

Corresponding directly to the linear ordering of pre- and post-sets which characterizes strongly concatenable processes, the key feature of $\mathcal{Q}[\_]$ is that, differently from $\mathcal{P}[\_]$, it associates to the net N a monoidal category whose objects form a free *non-commutative* monoid. The reason for renouncing to commutativity when passing from $\mathcal{P}[\_]$ to $\mathcal{Q}[\_]$, a choice that at first may seem odd, is explained in Section 2, where the following negative result is proved: under very reasonable assumptions, *no* mapping from nets to symmetric strict monoidal categories whose monoids of objects are commutative can be lifted to a functor, since there exists a morphism of nets which *cannot* be extended to a *monoidal* functor between the appropriate categories. Thus, abandoning the commutativity of the monoids of objects and considering *strings* as representatives of multisets, i.e., considering strongly concatenable processes, seem to be a choice forced upon us by the aim of a functorial algebraic semantics of nets. As a consequence of this choice, any transition of N has many corresponding arrows in $\mathcal{Q}[N]$, actually one for each linearization of its pre-set and of its post-set. However, such arrows are 'related' to each other by a *naturality* condition, in the precise sense that, when collected together, they form a natural transformation between appropriate functors. This naturality axiom is the second relevant feature of $\mathcal{Q}[\_]$ and it is actually the key to keep the computational interpretation of the new category $\mathcal{Q}[N]$, i.e., the strongly concatenable processes, surprisingly close to that of $\mathcal{P}[N]$, i.e., the concatenable processes.

Concerning our main issue, *viz.* functoriality, in Section 3 we introduce a category $\underline{\mathsf{TSSMC}}^{\otimes}$ of symmetric strict monoidal categories with free non-commutative monoids of objects, called *symmetric Petri categories*, whose arrows are equivalence classes—accounting for our view of strings as representatives of multisets—of those symmetric strict monoidal functors which preserve some further structure related to nets, and we show that $\mathcal{Q}[\_]$ is a functor from $\underline{\mathsf{Petri}}$, a rich category of nets introduced in [6], to $\underline{\mathsf{TSSMC}}^{\otimes}$. In addition, we prove that $\mathcal{Q}[\_]$ has a *coreflection* right adjoint $\mathcal{N}[\_]\colon \underline{\mathsf{TSSMC}}^{\otimes} \to \underline{\mathsf{Petri}}$. This implies, by general reasons, that $\underline{\mathsf{Petri}}$ is *equivalent* to an easily identified coreflective subcategory of $\underline{\mathsf{TSSMC}}^{\otimes}$, namely the *replete image* of $\mathcal{Q}[\_]$. The category $\underline{\mathsf{TSSMC}}^{\otimes}$, together with the functors $\mathcal{Q}[\_]$ and $\mathcal{N}[\_]$, constitutes our proposed axiomatization ('in the large') of Petri net computations in categorical terms.

Although this contribution is a first attempt towards the aims of a functorial algebraic semantics for nets and of an axiomatization of net behaviours 'in the large', we think that the results given here help to deepen the understanding of the subject. We remark that the refinement of concatenable processes into strongly concatenable processes is similar and comparable to the one which brought from Goltz-Reisig processes to them, and that the result of Section 2 makes strongly concatenable processes 'unavoidable' if a functorial construction is desired. In addition, from the categorical viewpoint, our approach is quite natural, since it is the one which simply observes that multisets are equivalence classes of strings and then takes into account the categorical paradigm, following which one always prefers to add suitable isomorphisms between objects rather than considering explicitly equivalence classes of them. Finally, concerning the use of category theory in semantics, and in particular in this paper, it may be appropriate to observe here that the categorical framework made it possible to discover and amend an 'anomaly' of $\mathcal{P}[\_]$ significant and of general relevance which could have not been noticed in other frameworks.

Due to the extended abstract nature of this paper, most of the proofs are omitted. Some preliminary related results appear also in [11].

<u>Notation</u>. When dealing with a category $\underline{\mathsf{C}}$ in which arrows are meant to represent computations, in order to stress its computational interpretation, we write arrow composition from left to right, i.e., in the diagrammatic order, and we denote it by $\_;\_$. The reader is referred to [5] for the categorical concepts used.

<u>Acknowledgements</u>. I wish to thank José Meseguer and Ugo Montanari to whom I am indebted for several discussions on the subject. Thanks to Mogens Nielsen, Claudio Hermida and Jaap Van Oosten for their valuable comments on an early version of this paper.

1 Concatenable Processes

In this section we recall the notion of concatenable processes [3].

<u>Notation</u>. Given a set S, we denote by $S^{\oplus}$ the set of *finite multisets* of S, i.e., the set of all functions from S to the set ω of natural numbers which yield nonzero values only on finitely many $s \in S$. We recall that $S^{\oplus}$ is a *commutative monoid*, actually the *free* commutative monoid on S, under the operation of multiset union, in the following denoted by $\oplus$, with unit element the empty multiset 0.

Definition 1.1 *(Petri Nets)*
A Petri net is a structure $N = (\partial^0_N, \partial^1_N\colon T_N \to S^{\oplus}_N)$, where T_N is a set of transitions, S_N is a set of places, and ∂^0_N and ∂^1_N are functions.
A morphism of Petri nets from N_0 to N_1 is a pair $\langle f, g\rangle$, where $f\colon T_{N_0} \to T_{N_1}$ is a function and $g\colon S^{\oplus}_{N_0} \to S^{\oplus}_{N_1}$ is a monoid homomorphism such that $\langle f, g\rangle$ respects source and target, i.e., $\partial^i_{N_1} \circ f = g \circ \partial^i_{N_0}$, for $i = 0, 1$.
This defines the category $\underline{\mathsf{Petri}}$ of Petri nets.

This describes a Petri net precisely as a graph whose set of nodes is a free commutative monoid, i.e., the set of *finite multisets* on a given set of *places*. The source and target of an arc, here called a *transition*, are meant to represent, respectively, the *markings* consumed and produced by the firing of the transition.

DEFINITION 1.2 *(Process Nets and Processes)*
A process net is a finite, acyclic net Θ such that for all $t \in T_\Theta$, $\partial^0_\Theta(t)$ and $\partial^1_\Theta(t)$ are sets (as opposed to multisets), and for all $t_0 \neq t_1 \in T_\Theta$, $\partial^i_\Theta(t_0) \cap \partial^i_\Theta(t_1) = \varnothing$, for $i = 0, 1$.

Given $N \in$ Petri, a process of N is a morphism $\pi: \Theta \to N$, where Θ is a process net and π is a net morphism which maps places to places (as opposed to morphisms which map places to markings).

We consider as identical process nets which are isomorphic. Consequently, we shall make no distinction between two processes $\pi: \Theta \to N$ and $\pi': \Theta' \to N$ for which there exists an isomorphism $\varphi: \Theta \to \Theta'$ such that $\pi' \circ \varphi = \pi$.

The equivalence of the following definition of $\mathcal{P}[N]$ with the original one in [3] has been proved in [12]. The reader is referred to the cited works for a more explicit description of $\mathcal{P}[N]$, a wider discussion, and for related examples.

DEFINITION 1.3
The category $\mathcal{P}[N]$ is the monoidal quotient of $\mathcal{F}(N)$, the symmetric strict monoidal category whose monoid of objects is $S_N^\oplus$ and whose arrows are freely generated from the transitions of N, modulo the axioms

$$\begin{array}{rcll} \gamma_{a,b} & = & id_{a\oplus b} & \text{if } a, b \in S_N \text{ and } a \neq b, \\ t; (id \otimes \gamma_{a,a} \otimes id) & = & t & \text{if } t \in T_N \text{ and } a \in S_N, \\ (id \otimes \gamma_{a,a} \otimes id); t & = & t & \text{if } t \in T_N \text{ and } a \in S_N, \end{array}$$

where γ is the symmetry isomorphism of $\mathcal{F}(N)$.

The arrows of $\mathcal{P}[N]$ have a nice computational interpretation as *concatenable processes*, a slight refinement of the classical notion of process consisting of a suitable labelling of the minimal and the maximal places of process nets which distinguishes among the different instances of a place in a process of N. The role of the symmetries is to *regulate the flow of causality* between subprocesses by permuting instances of places appropriately, i.e., by exchanging causes. In this view, the first axiom says that permuting different places does not change the causal relationships, and the remaining two say that the same happens when permuting places in the pre- and in the post-set of a transition. Using the labels, it is easy to define an operation of concatenation of concatenable processes and, thus, a category $\mathcal{CP}[N]$ whose objects are the multisets $S_N^\oplus$ and whose arrows are the concatenable processes of N. It has been proved in [3] that $\mathcal{CP}[N]$ is a symmetric strict monoidal category and that the following result holds.

THEOREM 1.4
$\mathcal{CP}[N]$ and $\mathcal{P}[N]$ are isomorphic.

2 A Negative Result about Functoriality

Among the primary requirements usually imposed on constructions like $\mathcal{P}[\_]$ there is that of *functoriality*. One of the main reasons supporting the choice of a categorical treatment of semantics is the need of specifying further the structure of the systems under analysis by giving explicitly the morphisms or, in other words, by specifying how the given systems simulate each other. This, in turn, means to choose precisely what the relevant (behavioural) structure of the systems is. It is then clear that such morphisms should be preserved at the semantic level. In our case, the functoriality of $\mathcal{P}[\_]$ means that if N can be mapped to N' via a morphism $\langle f, g\rangle$, which by the very definition of net morphisms implies that N can be simulated by N', there must be a way, namely $\mathcal{P}[\langle f, g\rangle]$, to see the processes of N as processes of N'. However, this is not possible for $\mathcal{P}[\_]$. The problem, as illustrated by the following example, is due to the first axiom in Definition 1.3 which, on the other hand, is exactly what makes $\mathcal{P}[N]$ capture quite precisely the notion of processes of N.

Example 2.1
Consider the nets N and $\bar{N}$ in the picture below, where we use the standard graphical representation of nets in which circles are places, boxes are transitions, and sources and targets are directed arcs. We have $S_N = \{a_0, a_1, b_0, b_1\}$ and T_N consisting of the transitions $t_0\colon a_0 \to b_0$ and $t_1\colon a_1 \to b_1$, while $S_{\bar{N}} = \{\bar{a}, \bar{b}_0, \bar{b}_1\}$ and $T_{\bar{N}}$ contains $\bar{t}_0\colon \bar{a} \to \bar{b}_0$ and $\bar{t}_1\colon \bar{a} \to \bar{b}_1$.

Consider now the net morphism $\langle f, g\rangle$ where $f(t_i) = \bar{t}_i$, $g(a_i) = \bar{a}$ and $g(b_i) = \bar{b}_i$, for $i = 0, 1$. We claim that $\langle f, g\rangle$ cannot be extended to a monoidal functor $\mathcal{P}[\langle f, g\rangle]$ from $\mathcal{P}[N]$ to $\mathcal{P}[\bar{N}]$. Suppose in fact that F is such an extension. Then, it must be $\mathsf{F}(t_0 \otimes t_1) = \mathsf{F}(t_0) \otimes \mathsf{F}(t_1) = \bar{t}_0 \otimes \bar{t}_1$. Moreover, since $t_0 \otimes t_1 = t_1 \otimes t_0$, we would have

$$\bar{t}_0 \otimes \bar{t}_1 = \mathsf{F}(t_1 \otimes t_0) = \bar{t}_1 \otimes \bar{t}_0,$$

which is impossible since the leftmost and the rightmost terms above are different processes in $\mathcal{P}[\bar{N}]$.

Formally speaking, the problem is that the category of symmetries sitting inside $\mathcal{P}[N]$, say Sym_N, is *not free*. Moreover, it is easy to verify that as soon as one imposes axioms on $\mathcal{P}[N]$ which guarantee to get a functor, one annihilates all the symmetries and, therefore, destroys the ability of $\mathcal{P}[N]$ of dealing with causality. It is important to observe that it would be definitely *meaningless* to try to overcome the problem simply by dropping from <u>Petri</u> the morphisms which 'behave badly': the morphism $\langle f, g\rangle$ of Example 2.1, for instance, is clearly a *simulation* and, as such, it should definitely be allowed by any serious attempt to formulate a definition of net morphisms. The following result shows that the problem illustrated in Example 2.1 is serious, actually deep enough to prevent any naive modification of $\mathcal{P}[\_]$ from being functorial.

THEOREM 2.2
Let $\mathcal{X}[\_]$ be a function which assigns to each net N a symmetric strict monoidal category whose monoid of objects is commutative and contains the places of N. Suppose that the group of symmetries at any object of $\mathcal{X}[N]$ is finite and suppose that there exists a net N with a place $a \in N$ such that, for each $n > 1$, we have that the components at (na, na) of the symmetry isomorphism of $\mathcal{X}[N]$ is not an identity. Then, there exists a Petri net morphism $\langle f, g\rangle: N_0 \to N_1$ which cannot be extended to a symmetric strict monoidal functor from $\mathcal{X}[N_0]$ to $\mathcal{X}[N_1]$.

Proof. (Sketch.) Let N' be a net such that, for each n, we have $c'_{na,na} \neq id$, where c' is the symmetry natural isomorphism of $\mathcal{X}[N']$, and let N be a net with two distinct places a and b and with *no* transitions, and let c' be the symmetry natural isomorphism of $\mathcal{X}[N]$. Since the group of symmetries at ab is finite, there is a *cyclic* subgroup generated by $c_{a,b}$, i.e., there exists $k > 1$, the order of the subgroup, such that $(c_{a,b})^k = id$ and $(c_{a,b})^n \neq id$ for any $1 \leq n < k$. Let p be any prime number greater than k. Then, exploiting general properties of monoidal categories and reasoning as in Example 2.1, one sees that the Petri net morphism $\langle f, g\rangle: N \to N'$, where f is the function $\varnothing \to T_{N'}$ and g is the monoid homomorphism such that $g(b) = (p-1)a$ and g is the identity on the other places of N, cannot be extended to a symmetric strict monoidal functor $\mathsf{F}: \mathcal{X}[N] \to \mathcal{X}[N']$. ✓

The contents of the previous proposition may be restated in different terms by saying that in the *free* category of symmetries on a commutative monoid M there are *infinite* homsets. This means that dropping axiom $\gamma_{a,b} = id_{a\oplus b}$ in the definition of $\mathcal{P}[N]$ causes an 'explosion' of the structure of the symmetries. More precisely, if we omit that axiom we can find some object u such that the group of symmetries on u has infinite order. Of course, since symmetries represent causality, and as such they are integral parts of processes, this makes the category so obtained completely useless for the application we have in mind.

The hypothesis of Theorem 2.2 can be certainly weakened in several ways, at the expense of complicating the proof. However, we avoided such complications since the conditions stated above are *already* weak enough if one wants to regard $\mathcal{X}[N]$ as a category of processes of N. In fact, since places represent the atomic bricks of which states are built, one needs to consider them in $\mathcal{X}[N]$, since symmetries regulate the 'flow of causality', there will be $c_{na,na}$ different from the identity, and since in a computation we can have only finitely many 'causality streams', there will not be categories with infinite groups of symmetries. Therefore, the given result means that there is no chance to have a functorial construction of the processes of N along the lines of $\mathcal{P}[N]$ whose objects form a commutative monoid.

3 The Category $\mathcal{Q}[N]$

In this section we introduce the symmetric strict monoidal category $\mathcal{Q}[N]$ which is meant to represent the processes of the Petri net N and which supports a functorial construction. This will allow us to characterize the category of the categories of net behaviours, i.e., to axiomatize net behaviours 'in the large'.

Theorem 2.2 shows that, necessarily, there is a price to be payed. Here, the idea is to renounce to the commutativity of the monoids of objects. More precisely, we build the arrows of $\mathcal{Q}[N]$ starting from the Sym^*_N, the 'free' category of symmetries over the *set* S_N of places of N. This choice makes each transition of N have many corresponding arrows in $\mathcal{Q}[N]$; however, the arrows of $\mathcal{Q}[N]$ which differ only by being instances of the same transition are linked together by a 'naturality' condition which guarantees that $\mathcal{Q}[N]$ remains close to the category $\mathcal{P}[N]$ of concatenable processes. Namely, the arrows of $\mathcal{Q}[N]$ correspond to Goltz-Reisig processes in which the minimal and the maximal places are *linearly* ordered.

Similarly to Sym_N, Sym^*_N serves a double purpose: from the categorical point of view it provides the symmetry isomorphism of a symmetric monoidal category, while from a semantic perspective it regulates the flow of causal dependency. Generally speaking, a symmetry in $\mathcal{Q}[N]$ should be interpreted as a *'reorganization'* of the tokens in the global state of the net which, when reorganizing multiple instances of the same place, yields a exchange of causes exactly as Sym_N does for $\mathcal{P}[N]$.

NOTATION. In the following, we use $S^{\otimes}$ to indicate the set of (finite) strings on set S, more commonly denoted by S^*. In the same way, we use $\otimes$ to denote string concatenation, while 0 denotes the empty string. As usual, for $u \in S^{\otimes}$, we indicate by $|u|$ the lenght of u and by u_i its i-th element.

DEFINITION 3.1 *(The Category of Permutations)*
Let S be a set. The category Sym^*_S has for objects the strings $S^{\otimes}$ and an arrow $p\colon u \to v$ if and only if p is a permutation of $|u|$ elements, and v is the string obtained by applying the permutation p to u, i.e., $v_{p(i)} = u_i$.
Arrows composition in Sym^*_S is obviously given by the product of permutations, i.e., their composition as functions, here and in the following denoted by $\_\,;\_$.

Graphically, we represent an arrow $p\colon u \to v$ in Sym^*_S by drawing a line between u_i and $v_{p(i)}$, as for example in Figure 1. Of course, it is possible to define a tensor product on Sym^*_S together with interchange permutations which make it a symmetric monoidal category (see also Figure 1 where γ is the permutation $\{1 \to 2, 2 \to 1\}$).

DEFINITION 3.2 *(Operations on Permutations)*
Given the permutations $p\colon u \to v$ and $p'\colon u' \to v'$ in Sym^*_S their parallel composition $p \otimes p'\colon u \otimes u' \to v \otimes v'$ is the permutation such that

$$i \mapsto \begin{cases} p(i) & \text{if } 0 < i \leq |u| \\ p'(i - |u|) + |u| & \text{if } |u| < i \leq |u| + |u'| \end{cases}$$

Given a permutation π of m elements and the strings $u_i \in S^{\otimes}$, $i = 1, \ldots, m$, the interchange permutation $\pi(u_1, \ldots, u_m)$ is the permutation p such that

$$p(i) = i - \sum_{j=1}^{h-1} |u_j| + \sum_{\pi(j) < \pi(h)} |u_j| \quad \text{if } \sum_{j=1}^{h-1} |u_j| < i \leq \sum_{j=1}^{h} |u_j|.$$

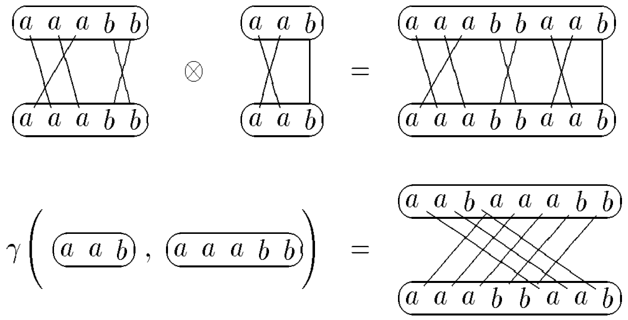


Figure 1: The monoidal structure of Sym^*_S

It is easy to see that $\otimes$ extends to a functor $\otimes\colon Sym^*_S \times Sym^*_S \rightarrow Sym^*_S$ making Sym^*_S a strict monoidal category. Moreover, the family of interchange permutations $\gamma = \{\gamma(u,v)\}_{u,v \in Sym^*_S}$ provides the symmetry isomorphism which makes Sym^*_S a symmetric strict monoidal category.

Theorem 3.3
Let S be a set, let $\underline{\mathsf{C}}$ be a symmetric strict monoidal category and let F be a function from S to the set of objects of $\underline{\mathsf{C}}$. Then, there exists a unique symmetric strict monoidal functor $\mathsf{F}\colon Sym^*_S \rightarrow \underline{\mathsf{C}}$ extending F.

The preceding result proves that the mapping $S \mapsto Sym^*_S$ extends to a *left adjoint* functor from $\underline{\mathsf{Set}}$, the category of sets, to $\underline{\mathsf{SSMC}}$, the category of symmetric strict monoidal categories. Equivalently, Sym^*_S is the free symmetric strict monoidal category on the set S, which is the key point about Sym^*_S.

In the following, given a string $u \in S^{\otimes}$, let $\mathcal{M}(u)$ denote the multiset corresponding to u, and, given a net N, let Sym^*_N the category $Sym^*_{S_N}$.

Definition 3.4 *(The category $\mathcal{Q}[N]$)*
Let N be a net in $\underline{\mathsf{Petri}}$. Then $\mathcal{Q}[N]$ is the category which includes Sym^*_N as subcategory and has as additional arrows those defined by the following inference rules:

$$\frac{t\colon \mathcal{M}(u) \rightarrow \mathcal{M}(v) \text{ in } T_N}{t_{u,v}\colon u \rightarrow v \text{ in } \mathcal{Q}[N]}$$

$$\frac{\alpha\colon u \rightarrow v \text{ and } \beta\colon u' \rightarrow v' \text{ in } \mathcal{Q}[N]}{\alpha \otimes \beta\colon u \otimes u' \rightarrow v \otimes v' \text{ in } \mathcal{Q}[N]} \qquad \frac{\alpha\colon u \rightarrow v \text{ and } \beta\colon v \rightarrow w \text{ in } \mathcal{Q}[N]}{\alpha\,;\,\beta\colon u \rightarrow w \text{ in } \mathcal{Q}[N]}$$

plus the axioms expressing the fact that $\mathcal{Q}[N]$ is a symmetric strict monoidal category with symmetry isomorphism γ, and the following axiom involving transitions and symmetries.

$$p\,;\,t_{u',v'} = t_{u,v}\,;\,q \quad \text{where } p\colon u \rightarrow u' \text{ in } Sym^*_N \text{ and } q\colon v \rightarrow v' \text{ in } Sym^*_N. \qquad (\Phi)$$

Exploiting the freeness of Sym^*_N, it is easy to prove the following completely axiomatic description of $\mathcal{Q}[N]$, which can be useful in many contexts.

PROPOSITION 3.5
$\mathcal{Q}[N]$ is (isomorphic to) the category $\underline{\mathsf{C}}$ whose objects are the elements of $S_N^{\otimes}$ and whose arrows are generated by the inference rules

$$\frac{u \in S_N^{\otimes}}{id_u\colon u \to u \text{ in } \underline{\mathsf{C}}} \qquad \frac{u, v \text{ in } S_N^{\otimes}}{c_{u,v}\colon u \otimes v \to v \otimes u \text{ in } \underline{\mathsf{C}}} \qquad \frac{t\colon \mathcal{M}(u) \to \mathcal{M}(v) \text{ in } T_N}{t_{u,v}\colon u \to v \text{ in } \underline{\mathsf{C}}}$$

$$\frac{\alpha\colon u \to v \text{ and } \beta\colon u' \to v' \text{ in } \underline{\mathsf{C}}}{\alpha \otimes \beta\colon u \otimes u' \to v \otimes v' \text{ in } \underline{\mathsf{C}}} \qquad \frac{\alpha\colon u \to v \text{ and } \beta\colon v \to w \text{ in } \underline{\mathsf{C}}}{\alpha;\beta\colon u \to w \text{ in } \underline{\mathsf{C}}}$$

modulo the axioms expressing that $\underline{\mathsf{C}}$ is a strict monoidal category, namely,

$$\begin{aligned} \alpha; id_v = \alpha = id_u;\alpha \quad &\text{and} \quad (\alpha;\beta);\delta = \alpha;(\beta;\delta), \\ (\alpha \otimes \beta) \otimes \delta = \alpha \otimes (\beta \otimes \delta) \quad &\text{and} \quad id_0 \otimes \alpha = \alpha = \alpha \otimes id_0, \\ id_u \otimes id_v = id_{u \otimes v} \quad &\text{and} \quad (\alpha \otimes \alpha');(\beta \otimes \beta') = (\alpha;\beta) \otimes (\alpha';\beta'), \end{aligned}$$

the latter whenever the righthand term is defined, the following axioms expressing that $\underline{\mathsf{C}}$ is symmetric with symmetry isomorphism c

$$\begin{aligned} c_{u,v\otimes w} &= (c_{u,v} \otimes id_w);(id_v \otimes c_{u,w}), \\ c_{u,u'};(\beta \otimes \alpha) &= (\alpha \otimes \beta);c_{v,v'} \quad \text{for } \alpha\colon u \to v,\ \beta\colon u' \to v', \\ c_{u,v};c_{v,u} &= id_{u \otimes v}, \end{aligned}$$

and the following axiom corresponding to axiom (Φ)

$$p\,;t_{u',v'}\,;q = t_{u,v} \quad \text{where } p\colon u \to u' \text{ and } q\colon v' \to v \text{ are symmetries.}$$

We show next that $\mathcal{Q}[\_]$ can be lifted to a functor from the category of Petri nets to an appropriate category of symmetric strict monoidal categories and equivalence classes of symmetric strict monoidal functors. The role of such an equivalence is to take into account that we look at the strings of $S_N^{\otimes}$ as concrete representatives of the multisets of $S_N^{\oplus}$ and, therefore, we want to consider perfectly *equal* those functors which differ only by picking different, yet compatible, linearizations of multisets.

DEFINITION 3.6 *(Symmetric Petri Categories)*
A symmetric Petri category is a symmetric strict monoidal category $\underline{\mathsf{C}}$ in $\underline{\mathsf{SSMC}}$ whose monoid of objects is the free monoid $S^{\otimes}$ for some set S.

For any pair $\underline{\mathsf{C}}$ and $\underline{\mathsf{D}}$ of symmetric Petri categories, consider the binary relation $\mathcal{R}_{\underline{\mathsf{C}},\underline{\mathsf{D}}}$ on the symmetric strict monoidal functors from $\underline{\mathsf{C}}$ to $\underline{\mathsf{D}}$ defined as $\mathsf{F}\,\mathcal{R}_{\underline{\mathsf{C}},\underline{\mathsf{D}}}\,\mathsf{G}$ if and only if there exists a *monoidal natural isomorphism* $\sigma\colon \mathsf{F} \cong \mathsf{G}$ whose components are all *symmetries*. Clearly, $\mathcal{R}_{\underline{\mathsf{C}},\underline{\mathsf{D}}}$ is an equivalence relation and the family $\mathcal{R} = \{\mathcal{R}_{\underline{\mathsf{C}},\underline{\mathsf{D}}}\}_{\underline{\mathsf{C}},\underline{\mathsf{D}} \in \underline{\mathsf{SSMC}}}$ is a congruence with respect to functor composition. Therefore, the following definition makes sense.

DEFINITION 3.7 *(The category $\underline{\mathsf{SSMC}}^{\otimes}$)*
Let $\underline{\mathsf{SSMC}}^{\otimes}$ be the quotient of the full subcategory of $\underline{\mathsf{SSMC}}$ consisting of the symmetric Petri categories modulo the congruence $\mathcal{R}$.

Theorem 3.8 *($\mathcal{Q}[\_]\colon \underline{\mathsf{Petri}} \to \underline{\mathsf{SSMC}}^{\otimes}$)*
$\mathcal{Q}[\_]$ extends to a functor from $\underline{\mathsf{Petri}}$ to $\underline{\mathsf{SSMC}}^{\otimes}$.

Proof. (Sketch.) Let $\langle f, g\rangle\colon N_0 \to N_1$ be a morphism of Petri nets. In order define $\mathcal{Q}[\langle f, g\rangle]$ we need to be able to embed N in $\mathcal{Q}[N]$. To this end, consider any function $in_{N_1}\colon S^{\oplus}_{N_1} \to S^{\otimes}_{N_1}$ such that $\mathcal{M}(in_{N_1}(\nu)) = \nu$. Since g is a monoid homomorphism from the free monoid $S^{\oplus}_{N_0}$ to $S^{\oplus}_{N_1}$, it corresponds to a unique function g' from S_{N_0} to $S^{\oplus}_{N_1}$, whence we obtain $\hat{g} = in_{N_1} \circ g'\colon S_{N_0} \to S^{\otimes}_{N_1}$, i.e., a function from S_{N_0} to the set of objects of $\mathcal{Q}[N_1]$. Then, from Theorem 3.3, we have the symmetric strict monoidal functor $\mathsf{F}'\colon Sym_{S_{N_0}} \to \mathcal{Q}[N_1]$. Finally, we extend F' to a functor $\mathcal{Q}[\langle f, g\rangle]$ from $\mathcal{Q}[N_0]$ to $\mathcal{Q}[N_1]$ by considering the symmetric strict monoidal functor F which coincides with F' on Sym_{N_0} and maps $t_{u,v}\colon u \to v$ to $f(t)_{\mathsf{F}(u),\mathsf{F}(v)}\colon \mathsf{F}(u) \to \mathsf{F}(v)$. Since monoidal functors map symmetries to symmetries, and since $f(t)$ is transition of N_1, it follows immediately that F preserves axiom (Φ), i.e., that F is well defined. Moreover, since a different choice of in_{N_1} would clearly give a functor G such that $\mathsf{F}\ \mathcal{R}\ \mathsf{G}$, we have that $\mathcal{Q}[\_]$ does not depend on in_{N_1}. It is easy to check that this definition makes $\mathcal{Q}[\_]$ into a functor. ✓

However, the category $\underline{\mathsf{SSMC}}^{\otimes}$ is still too general for our purpose. In particular, it is easily noticed that $\mathcal{Q}[\_]$ is not *full*. This signifies that $\underline{\mathsf{SSMC}}^{\otimes}$ has too little structure to represent net behaviours precisely enough; equivalently, since the structure of the objects of a category $\underline{\mathsf{C}}$ is 'encoded' in the morphisms of $\underline{\mathsf{C}}$, it signifies that the morphisms of $\underline{\mathsf{SSMC}}^{\otimes}$ do not capture the structure of symmetric Petri categories precisely enough. Specifically, the transitions, which are definitely primary components of nets, and as such are treated by the morphisms in $\underline{\mathsf{Petri}}$, have *no* corresponding notion in $\underline{\mathsf{SSMC}}^{\otimes}$: we need to identify such a notion and refine the choice of the category of net computations accordingly.

The key to accomplish our task is the following observation about axiom (Φ) in Definition 3.4: as already mentioned, it simply expresses that the collection of the arrows $t_{u,v}$ of $\mathcal{Q}[N]$, for $t \in T_N$ and $u, v \in S^{\otimes}_N$, is a natural transformation. Namely, for $\underline{\mathsf{C}}$ a symmetric Petri category with objects $S^{\otimes}$, and ν a multiset in $S^{\oplus}$, let $Sym_{\underline{\mathsf{C}},\nu}$ be the subcategory of $\underline{\mathsf{C}}$ consisting of those objects $u \in S^{\otimes}$ such that $\mathcal{M}(u) = \nu$ and the symmetries between them, and let $in_{\underline{\mathsf{C}},\nu}$ be the inclusion of $Sym_{\underline{\mathsf{C}},\nu}$ in $\underline{\mathsf{C}}$. Then, for $\nu, \nu' \in S^{\oplus}$, one obtains a pair of parallel functors $\pi_{\underline{\mathsf{C}},\nu}$ and $\pi_{\underline{\mathsf{C}},\nu'}$ by composing $in_{\underline{\mathsf{C}},\nu}$ and $in_{\underline{\mathsf{C}},\nu'}$ respectively with the first and with the second projection of $Sym_{\underline{\mathsf{C}},\nu} \times Sym_{\underline{\mathsf{C}},\nu'}$.

$$\begin{tikzcd}
& Sym_{\underline{\mathsf{C}},\nu} \arrow[dr, "in_{\underline{\mathsf{C}},\nu}"] & \\
Sym_{\underline{\mathsf{C}},\nu} \times Sym_{\underline{\mathsf{C}},\nu'} \arrow[ur, "\pi_0"] \arrow[rr, shift left, "\pi_{\underline{\mathsf{C}},\nu}"] \arrow[rr, shift right, "\pi_{\underline{\mathsf{C}},\nu'}"'] \arrow[dr, "\pi_1"'] & & \underline{\mathsf{C}} \\
& Sym_{\underline{\mathsf{C}},\nu'} \arrow[ur, "in_{\underline{\mathsf{C}},\nu'}"'] &
\end{tikzcd}$$

It follows directly from the definitions that, when $\underline{\mathsf{C}}$ is $\mathcal{Q}[N]$, axiom (Φ) states exactly that, for all $t\colon \nu \to \nu' \in T_N$, the set $\{t_{u,v} \mid \mathcal{M}(u) = \nu, \mathcal{M}(v) = \nu'\}$ is a natural transformation from $\pi_{\mathcal{Q}[N],\nu}$ to $\pi_{\mathcal{Q}[N],\nu'}$.

A further very relevant property of the transitions of N when considered as arrows of $\mathcal{Q}[N]$ is that of being decomposable as a tensor only trivially and as a composition only by means of symmetries. This is easily captured by the following notion of *primitive* arrow.

DEFINITION 3.9 *(Primitive Arrows)*
Let $\underline{\mathsf{C}}$ be a symmetric Petri category. An arrow τ in $\underline{\mathsf{C}}$ is primitive if

i) τ *is not a symmetry;*

ii) $\tau = \alpha; \beta$ *implies* α *is a symmetry and* β *is primitive, or viceversa;*

iii) $\tau = \alpha \otimes \beta$ *implies* $\alpha = id_0$ and β *is primitive, or viceversa.*

A simple inspection of Definition 3.4 shows that the only primitive arrows in $\mathcal{Q}[N]$ are the arrows $t_{u,v}$, for $t: \mathcal{M}(u) \to \mathcal{M}(v)$ a transition of N. As a consequence, the natural transformations $\tau: \pi_{\mathcal{Q}[N],\nu} \dot{\to} \pi_{\mathcal{Q}[N],\nu'}$ whose components are primitive are in one-to-one correspondence with the transitions of N. Following the usual categorical paradigm, we then use the properties that characterize the transitions of N in $\mathcal{Q}[N]$, expressed in abstract categorical terms, to define the notion of transition in any symmetric Petri category.

DEFINITION 3.10 *(Transitions of Symmetric Petri Categories)*
Let $\underline{\mathsf{C}}$ be a symmetric Petri category and let $S^\otimes$ be its monoid of objects. A transition of $\underline{\mathsf{C}}$ is a natural transformation $\tau: \pi_{\underline{\mathsf{C}},\nu} \dot{\to} \pi_{\underline{\mathsf{C}},\nu'}$, for ν, ν' in $S^\oplus$, whose components $\tau_{u,v}$ are primitive arrows of $\underline{\mathsf{C}}$.

It is clear now what the extra structure required in $\underline{\mathsf{SSMC}}^\otimes$ is: transitions must be preserved by morphisms of symmetric Petri categories. Formally, for $\underline{\mathsf{C}}$ and $\underline{\mathsf{D}}$ in $\underline{\mathsf{SSMC}}^\otimes$ and $\mathsf{F}: \underline{\mathsf{C}} \to \underline{\mathsf{D}}$ in $\underline{\mathsf{SSMC}}$, F *respects transitions* if, for each transition $\tau: \pi_{\underline{\mathsf{C}},\nu} \dot{\to} \pi_{\underline{\mathsf{C}},\nu'}$ of $\underline{\mathsf{C}}$, there exists a transition $\tau': \pi_{\underline{\mathsf{D}},\bar{\nu}} \dot{\to} \pi_{\underline{\mathsf{D}},\bar{\nu}'}$ of $\underline{\mathsf{D}}$ such that $\mathsf{F}(\tau_{u,v}) = \tau'_{\mathsf{F}(u),\mathsf{F}(v)}$ for all (u, v) in $Sym_{\underline{\mathsf{C}},\nu} \times Sym_{\underline{\mathsf{C}},\nu'}$; in this case, we say that τ' *corresponds to* τ *via* F.

The following lemma shows that a symmetric strict monoidal functor which preserves transitions defines a mapping between sets of transitions and that, moreover, this property extends to the arrows of $\underline{\mathsf{SSMC}}^\otimes$. It follows immediately that Definition 3.12 is well given.

LEMMA 3.11
If $\mathsf{F}: \underline{\mathsf{C}} \to \underline{\mathsf{D}}$ respects transitions, then for any transition τ of $\underline{\mathsf{C}}$, there exists a unique transition τ' of $\underline{\mathsf{D}}$ which corresponds to τ via F.

If $\mathsf{F} \mathrel{\Re} \mathsf{G}$, then F respects transitions if and and only if G does so, and then τ' corresponds to τ via F if and only if τ' corresponds to τ via G.

DEFINITION 3.12 *(Symmetric Petri Morphisms and the Category $\underline{\mathsf{TSSMC}}^\otimes$)*
A morphism of symmetric Petri category is an arrow in $\underline{\mathsf{SSMC}}^\otimes$ which respects transitions. We shall use $\underline{\mathsf{TSSMC}}^\otimes$ denote the (lluf) subcategory of $\underline{\mathsf{SSMC}}^\otimes$ whose arrows are the morphisms of symmetric Petri categories.

Finally, it is easy to prove that $\mathcal{Q}[\_]$ is actually a functor to $\underline{\mathsf{TSSMC}}^\otimes$.

Proposition 3.13 *($\mathcal{Q}[\_]$: $\underline{\mathsf{Petri}} \to \underline{\mathsf{TSSMC}}^{\otimes}$)*
The functor $\mathcal{Q}[\_]$ restricts to a functor from $\underline{\mathsf{Petri}}$ to $\underline{\mathsf{TSSMC}}^{\otimes}$.

Proof. It is enough to verify that, for any morphism $\langle f, g\rangle: N_0 \to N_1$ in $\underline{\mathsf{Petri}}$, a representative F of $\mathcal{Q}[\langle f, g\rangle]$ respects transitions. This follows at once, since f is a function $T_{N_0} \to T_{N_1}$, $\mathsf{F}(t_{u,v}) = f(t)_{\mathsf{F}(u),\mathsf{F}(v)}$, and the transitions of $\mathcal{Q}[N_i]$ are exactly the natural transformations $\{t_{u,v} \mid \mathcal{M}(u) = \nu, \mathcal{M}(v) = \nu'\}$ for $t: \nu \to \nu' \in T_{N_i}$. ✓

Interestingly enough, we can identify a functor from $\underline{\mathsf{TSSMC}}^{\otimes}$ to $\underline{\mathsf{Petri}}$ which is a *coreflection* right adjoint to $\mathcal{Q}[\_]$. It is worth remarking that this answers to a possible legitimate doubt about the category $\underline{\mathsf{TSSMC}}^{\otimes}$: in principle, in fact, the functoriality of $\mathcal{Q}[\_]$ could be due to a very tight choice of the target category, e.g., the congruence $\mathcal{R}$ could induce too many isomorphisms of categories and $\mathcal{Q}[\_]$ make undesirable identifications of nets. The existence of a coreflection right adjoint to $\mathcal{Q}[\_]$ is, of course, the best possible proof of the adequacy of $\underline{\mathsf{TSSMC}}^{\otimes}$: it implies that $\underline{\mathsf{Petri}}$ is embedded in it *fully* and *faithfully* as a coreflective subcategory. This result supports our claim that $\underline{\mathsf{TSSMC}}^{\otimes}$ is an axiomatization of the category of net computations.

Theorem 3.14 *($\mathcal{Q}[\_] \dashv \mathcal{N}[\_]$: $\underline{\mathsf{Petri}} \to \underline{\mathsf{TSSMC}}^{\otimes}$)*
Let $\underline{\mathsf{C}}$ be a symmetric Petri category, and let $S^{\otimes}$ be its monoid of objects. Define $\mathcal{N}[\underline{\mathsf{C}}]$ to be the Petri net $(\partial^0, \partial^1: T \to S^{\oplus})$, where

- *T is the set of transitions $\tau: \pi_{\underline{\mathsf{C}},\nu} \stackrel{\cdot}{\to} \pi_{\underline{\mathsf{C}},\nu'}$ of $\underline{\mathsf{C}}$;*
- *$\partial^0(\tau: \pi_{\underline{\mathsf{C}},\nu} \stackrel{\cdot}{\to} \pi_{\underline{\mathsf{C}},\nu'}) = \nu$ and $\partial^1(\tau: \pi_{\underline{\mathsf{C}},\nu} \stackrel{\cdot}{\to} \pi_{\underline{\mathsf{C}},\nu'}) = \nu'$.*

Then, $\mathcal{N}[\_]$ extends to a functor $\underline{\mathsf{TSSMC}}^{\otimes} \to \underline{\mathsf{Petri}}$ which is right adjoint to $\mathcal{Q}[\_]$. In addition, since the unit is an isomorphism, the adjunction is a coreflection.

Proof. For any symmetric Petri category $\underline{\mathsf{C}}$, there is a (unique) symmetric strict monoidal functor $\varepsilon_{\underline{\mathsf{C}}}: \mathcal{Q}\mathcal{N}[\underline{\mathsf{C}}] \to \underline{\mathsf{C}}$ which is the identity on the objects and which sends the component at (u, v) of the transition $\tau: \nu \to \nu'$ of $\mathcal{N}[\underline{\mathsf{C}}]$ to the component $\tau_{u,v}$ of the natural transformation $\tau: \pi_{\underline{\mathsf{C}},\nu} \stackrel{\cdot}{\to} \pi_{\underline{\mathsf{C}},\nu'}: Sym_{\underline{\mathsf{C}},\nu} \times Sym_{\underline{\mathsf{C}},\nu'} \to \underline{\mathsf{C}}$. Since it clearly preserves transitions, we have that $\varepsilon_{\underline{\mathsf{C}}}$ is a (representative of a) morphism of symmetric Petri categories. It is not difficult to prove that $\varepsilon_{\underline{\mathsf{C}}}$ enjoys the couniversal property making it the counit of the adjunction. The unit $\eta_N: N \to \mathcal{N}\mathcal{Q}[N]$ is the morphism $\langle f, id\rangle$, where f sends $t \in T_N$ to $\{t_{u,v}\} \in T_{\mathcal{N}\mathcal{Q}[N]}$, which is an iso. ✓

We end this section by identifying the replete image of $\mathcal{Q}[\_]$ in $\underline{\mathsf{TSSMC}}^{\otimes}$, i.e., the full subcategory of $\underline{\mathsf{TSSMC}}^{\otimes}$ consisting of those symmetric Petri categories isomorphic to $\mathcal{Q}[N]$, for some N in $\underline{\mathsf{Petri}}$.

Theorem 3.15 *($\underline{\mathsf{Petri}} \cong \underline{\mathsf{PSSMC}}$)*
Let $\underline{\mathsf{PSSMC}}$ be the full subcategory of $\underline{\mathsf{TSSMC}}^{\otimes}$ consisting of those symmetric Petri categories $\underline{\mathsf{C}}$ whose arrows can be generated by tensor and composition from symmetries, and components of transitions of $\underline{\mathsf{C}}$, uniquely up to the axioms of symmetric strict monoidal categories, i.e., the axioms in Proposition 3.5, and the naturality of transitions, i.e., axiom (Φ).

Then, $\underline{\mathsf{PSSMC}}$ and $\underline{\mathsf{Petri}}$ are equivalent.

Proof. By general results in category theory, it is enough to show that $\underline{\mathsf{C}}$ belongs to $\underline{\mathsf{PSSMC}}$ if and only if $\varepsilon_{\underline{\mathsf{C}}}: \mathcal{Q}\mathcal{N}[\underline{\mathsf{C}}] \to \underline{\mathsf{C}}$ is an isomorphism, which is easy. ✓

4 Strongly Concatenable Processes

In this section we introduce a slight refinement of concatenable processes and we show that they are abstractly represented by the arrows of the category $\mathcal{Q}[N]$. In other words, we find a process-like representation for the arrows of $\mathcal{Q}[N]$. This yields a functorial construction for the category of the processes of a net N.

Definition 4.1 *(Strongly Concatenable Processes)*
Given a petri net N in Petri, a strongly concatenable process of N is a tuple (π, ℓ, L) where $\pi: \Theta \to N$ is a process of N, and $\ell: \min(\Theta) \to \{1, \ldots, |\min(\Theta)|\}$ and $L: \max(\Theta) \to \{1, \ldots, |\max(\Theta)|\}$ are isomorphisms, i.e., total orderings of, respectively, the minimal and the maximal places of Θ.

An isomorphism of strongly concatenable processes is an isomorphism of the underlying processes which, in addition, preserves the orderings ℓ and L. As usual, we identify isomorphic strongly concatenable processes.

As in the case of concatenable processes, it is easy to define an operation of concatenation of strongly concatenable processes. We associate a source and a target in $S_N^{\otimes}$ to each strongly concatenable process by taking the *string* corresponding to the linear ordering of, respectively, $\min(\Theta)$ and $\max(\Theta)$. Then, the concatenation of $(\pi_0: \Theta_0 \to N, \ell_0, L_0): u \to v$ and $(\pi_1: \Theta_1 \to N, \ell_1, L_1): v \to w$ is the strongly concatenable process $u \to w$ obtained by merging the maximal places of Θ_0 and the minimal of Θ_1 according to L_0 and ℓ_1. (See Figure 2, where we enrich the usual representation of non-sequential processes by labelling the minimal and the maximal places with the values of, respectively, ℓ and L.)

Proposition 4.2
Under the above defined operation of sequential composition, the strongly concatenable processes of N form a category $\mathcal{CQ}[N]$ whose identities are those processes consisting only of places, which therefore are both minimal and maximal, and such that $\ell = L$.

Strongly concatenable processes admit a tensor product $\otimes$ such that, given $SCP = (\pi_0: \Theta_0 \to N, \ell_0, L_0): u \to v$ and $SCP' = (\pi_1: \Theta_1 \to N, \ell_1, L_1): u' \to v'$, $SCP \otimes SCP'$ is the strongly concatenable process $(\pi: \Theta \to N, \ell, L): u \otimes u' \to v \otimes v'$ given below (see also Figure 2), where $+$, besides the usual sum of natural numbers, denotes also the disjoint union of sets and functions, and in_0 and in_1 the corresponding injections.

- $\Theta = (\partial^0_{\Theta_0} + \partial^0_{\Theta_1}, \partial^1_{\Theta_0} + \partial^1_{\Theta_1}: T_{\Theta_0} + T_{\Theta_1} \to (S_{\Theta_0} + S_{\Theta_1})^{\oplus})$;
- $\pi = \pi_0 + \pi_1$;
- $\ell(in_0(a)) = \ell_0(a)$ and $\ell(in_1(a)) = |\min(\Theta_0)| + \ell_1(a)$;
- $L(in_0(a)) = L_0(a)$ and $L(in_1(a)) = |\max(\Theta_1)| + L_1(a)$.

Observe that $\otimes$ is a functor $\otimes: \mathcal{CQ}[N] \times \mathcal{CQ}[N] \to \mathcal{CQ}[N]$. The strongly concatenable processes consisting only of places are analogous in $\mathcal{CQ}[N]$ of the

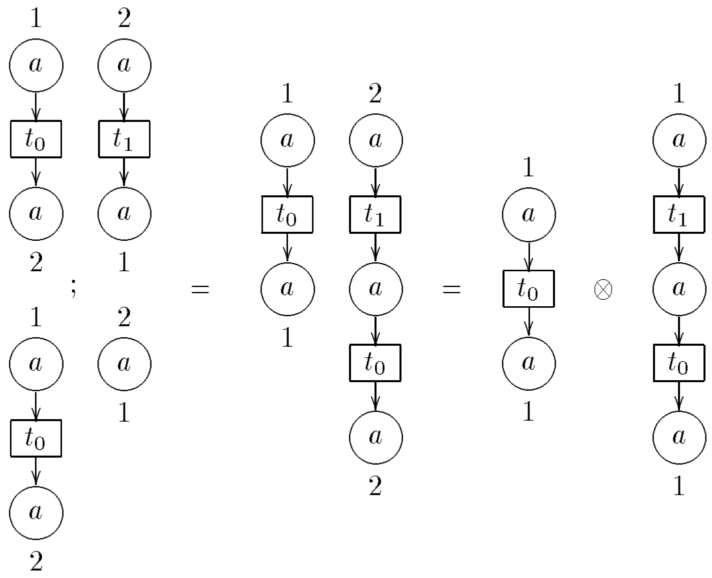


Figure 2: An example of the algebra of concatenable processes

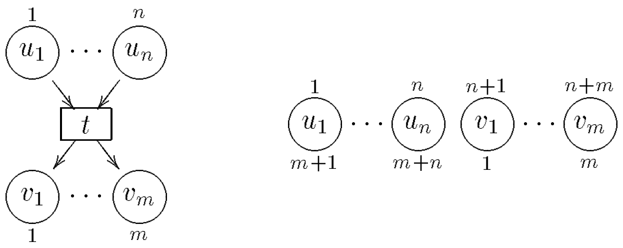


Figure 3: A transitions $t_{u,v}\colon u \to v$ and the symmetry $\gamma(u,v)$ in $\mathcal{C}\mathcal{Q}[N]$

permutations of $\mathcal{Q}[N]$. In particular, for any $u, v \in S^{\otimes}$, the strongly concatenable process $\bar{\gamma}(u,v)$ consisting of places in one-to-one correspondence with the elements of the string $u \otimes v$ mapped by π to the corresponding places of N, and such that $\ell(u_i) = i$, $\ell(v_i) = |u| + i$, $L(u_i) = |v| + i$ and $L(v_i) = i$, plays in $\mathcal{C}\mathcal{Q}[N]$ the role played by the permutation $\gamma(u,v)$ in $\mathcal{Q}[N]$ (see also Figure 3).

Proposition 4.3
Under the above defined tensor product $\mathcal{C}\mathcal{Q}[N]$ is a symmetric strict monoidal category whose symmetry isomorphism is the family $\{\bar{\gamma}(u,v)\}_{u,v \in S_N^{\otimes}}$.

The transitions t of N are faithfully represented in the obvious way by processes with a unique transition which is in the post-set of any minimal place and in the pre-set of any maximal place, minimal and maximal places being in one-to-one correspondence, respectively, with $\partial_N^0(t)$ and $\partial_N^1(t)$. Thus, varying ℓ and L on the process corresponding to a transition we obtain a representative in $\mathcal{C}\mathcal{Q}[N]$ of each instance $t_{u,v}$ of t in $\mathcal{Q}[N]$ (see also Figure 3).

Theorem 4.4
$\mathcal{C}\mathcal{Q}[N]$ and $\mathcal{Q}[N]$ are isomorphic.

Proof. (Sketch.) Consider the following mapping F from the arrows of $\mathcal{Q}[N]$ to strongly concatenable processes.

- An instance $t_{u,v}$ of a transition t of $\mathcal{Q}[N]$ is mapped to the strongly concatenable processes with a unique transition and two layers of places: the minimal, in one-to-one correspondence with $\partial_N^0(t)$ and ordered by ℓ to form the string u, and the maximal, in one-to-one correspondence with $\partial_N^1(t)$ and ordered to form v.
- The permutation $\gamma(u,v)$ is sent to the strongly concatenable process $\bar{\gamma}(u,v)$.
- F is extended inductively to a generic term α of $\mathcal{Q}[N]$, i.e., $\alpha_0 \otimes \alpha_1$ is mapped to $\mathsf{F}(\alpha_0) \otimes \mathsf{F}(\alpha_1)$ and $\alpha_0 \,;\, \alpha_1$ to $\mathsf{F}(\alpha_0); \mathsf{F}(\alpha_1)$.

Then, defining F to be the identity on the objects gives the required isomorphism $\mathsf{F}\colon \mathcal{Q}[N] \cong \mathcal{CQ}[N]$. ✓

References

[1] E. Best and R. Devillers. Sequential and Concurrent Behaviour in Petri Net Theory. *Theoretical Computer Science*, n. 55, pp. 87–136, 1987.

[2] C. Brown, D. Gurr, and V. de Paiva. *A Linear Specification Language for Petri Nets.* Technical Report DAIMI PB-363, Computer Science Dept., Aarhus University, 1991.

[3] P. Degano, J. Meseguer, and U. Montanari. Axiomatizing Net Computations and Processes. In *Proceedings of the 4th LICS Symposium*, pp. 175–185, IEEE, 1989.

[4] U. Goltz and W. Reisig. The Non-Sequential Behaviour of Petri Nets. *Information and Computation*, n. 57, pp. 125–147, 1983.

[5] S. MacLane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[6] J. Meseguer and U. Montanari. Petri Nets are Monoids. *Information and Computation*, n. 88, pp. 105–154, Academic Press, 1990.

[7] M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part 1. *Theoretical Computer Science*, n. 13, pp. 85–108, 1981.

[8] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

[9] C.A. Petri. *Non-Sequential Processes.* Interner Bericht ISF–77–5, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, Germany, 1977.

[10] W. Reisig. *Petri Nets.* Springer-Verlag, 1985.

[11] V. Sassone. *On the Semantics of Petri Nets: Processes, Unfoldings, and Infinite Computations.* PhD Thesis TD 6/94, Dipartimento di Informatica, Università di Pisa, 1994.

[12] V. Sassone. *Some Remarks on Concatenable Processes.* Technical Report TR 6/94, Dipartimento di Informatica, Università di Pisa, 1994.

[13] G. Winskel. A New Definition of Morphism on Petri Nets. In *Proceedings of STACS '84*, LNCS, n. 166, pp. 140–150, Springer-Verlag, 1984.

[14] G. Winskel. Petri Nets, Algebras, Morphisms and Compositionality. *Information and Computation*, n. 72, pp. 197–238, 1987.

Functional Logic Programming in GCLA

Olof Torgersson*
Department of Computing Science, Chalmers University of Technology
S-412 96 Göteborg,Sweden
oloft@cs.chalmers.se

Abstract

We describe a definitional approach to functional logic programming, based on the theory of Partial Inductive Definitions and the Programming Language GCLA. It is shown how functional and logic programming are easily integrated in GCLA using the features of the language, that is combining functions and predicates in programs becomes a matter of programming methodology. We also give a brief description of a way to automatically generate efficient procedural parts to the described definitions.

1 Introduction

Through the years there have been numerous attempts to combine the two main declarative programming paradigms functional and logic programming into one framework providing the benefits of both. The proposed methods varies from different kinds of translations, embedding one of the methods into the other, to more integrated approaches such as Horn Clause Logic with equality [5] and Constraint Logic Programming as in the language Life [2].

A notion shared between functional and logic programming is that of a definition, we say that we *define* functions and predicates. The programming language can then be seen as a formalism especially designed to provide the programmer with an as clean and elegant way as possible to define functions and predicates respectively. Of course these formalisms are not created out of thin air but are explained by an appropriate theory.

In GCLA [1, 6] we take a somewhat different approach, we do talk about definitions but these definitions are not given meaning by mapping them on some theory about something else, but are instead understood through a theory of *definitions* and their properties, the theory of *Partial Inductive Definitions* (PID) [4]. This theory is designed to express and study properties of definitions, so we look at the problem from a different angle and try to answer the questions; what are the specific properties of function and predicate definitions and how

*This work was carried out as part of the work in the ESPRIT working group GENTZEN and was funded by The Swedish National Board for Industrial and Technical Development (NUTEK).

can they be combined and interpreted to give an integrated functional logic computational framework based on PID.

A GCLA program consists of two communicating partial, inductive definitions which we call the (*object level*) *definition* and the *rule definition* respectively. The rule definition is used to give a meaning to the conditions in the definition and it is also through this the user queries the definition. We present a rule definition to a class of functional logic program definitions. This rule definiton implicitly determines the structure of function, predicate and integrated functional logic program definitions. We also try to give a brief description of how to write functional logic GCLA programs. Furthermore we show how the knowledge that a definition defines a functional logic program can be used to automatically generate better proof-search strategies, enhancing efficiency and enabling us to write more concise definitions. The approach taken to functional logic programming in GCLA in this paper is inspired by some earlier work found in [1, 6].

2 The GCLA Programming Language

The programming system GCLA [1, 6, 3](to be pronounced "gisela") is a logical programming language that is based on a generalization of Prolog. This generalization is unusual in that it takes quite a different view of the meaning of a logic program - a definitional view rather than the traditional logic view.

Compared to Prolog, what has been added to GCLA is the possibility of assuming conditions. For example, the clause

```
a <= (b -> c).
```

should be read as: "`a` holds if `c` can be proved while assuming `b`".

There is also a richer set of queries in GCLA than in Prolog. In GCLA, a query corresponding to an ordinary Prolog query is written

```
\- a.
```

and should be read as "Does `a` hold (in the definition D)?" We can also assume things in the query, for example

```
c \- a.
```

which should be read as "Assuming `c`, does `a` hold (in the definition D)?", or "Is `a` derivable from `c`?"

To execute a program, a query G is posed to the system asking wether there is a substitution σ such that $G\sigma$ holds according to the logic defined by the program. The goal G has the form $\Gamma \vdash c$, where Γ is a list of assumptions, and c is the conclusion from the assumptions Γ. The system tries to construct a deduction showing that $G\sigma$ holds in the given logic.

2.1 GCLA Programs

A GCLA program consists of two parts; One part is used to express the declarative content of the program, called the *definition* or the *object level*, and the other part is used to express rules and strategies acting on the declarative part, called the *rule definition* or the *meta level*.

2.1.1 The Definition.

The definition constitutes the formalization of a specific problem domain and in general contains a minimum of control information. The intention is that the definition by itself gives a purely declarative description of the problem domain while a procedural interpretation of the definition is obtained only by putting it in the context of the rule definition.

2.1.2 The Rule Definition.

The rule definition contains the procedural knowledge of the domain, that is the knowledge used for drawing conclusions based on the declarative knowledge in the definition. This procedural knowledge defines the possible inferences made from the declarative knowledge.

The rule definition contains *inference rule* definitions which define how different inference rules should act, and *search strategies* which control the search among the inference rules.

The general form of an inference rule is

$Rule(A_1, \ldots, A_m, PT_1, \ldots, PT_n)$ <=
    $Proviso$,(PT_1 -> Seq_1,...,(PT_n -> Seq_n)
    -> Seq.

and the general forms of a strategy are

$Strat(A_1, \ldots, A_m)$ <= $PT_1, \ldots, PT_n$.

$Strat(A_1, \ldots, A_m)$ <=
    ($Proviso_1$ -> Seq_1),...,($Proviso_k$-> Seq_k).
$Strat(A_1, \ldots, A_m)$ <= PT1, ..., PTn.

where

- A_i are arbitrary arguments.
- *Proviso* is a conjunction of provisos, that is calls to Horn clauses defined elsewhere. The *Proviso* could be empty.
- Seq and Seq_i are sequents which are on the form ($Antecedent \vdash Consequent$), where $Antecedent$ is a list of terms and $Consequent$ is an ordinary GCLA term.
- PT_i are proofterms, that is terms representing the proofs of the premises, Seq_i.

2.2 Example: Default Reasoning

Assume we know that an object can fly if it is a bird and if it is not a penguin. We also know that Tweety and Polly are birds as well as all penguins, and that Pengo is a penguin. This knowledge is expressed in the following definition:

```
flies(X) <= bird(X),(penguin(X) ->false).

bird(tweety).   bird(polly).   bird(X) <= penguin(X).

penguin(pengo).
```

One possible rule definition enabling us to use this definition the way we want, is:

```
fs <=
   right(fs),            % First try standard right rules.
   left_if_false(fs).    % else if consequent is false...

left_if_false(PT) <=     % Is the consequent false?
   (_ \- false).
left_if_false(PT) <=     %  If so perform left rules.
   no_false_assump(PT),false_left(_).

no_false_assump(PT) <= not(member(false,A)) -> (A \- _).
no_false_assump(PT) <= left(PT).

member(X,[X|_]).         % Proviso Definition
member(X,[_|Ys]) :- member(X,Ys).
```

If we want to know which birds can fly, we pose the query

```
fs \\- (\- flies(X)).
```

and the system will respond with `X` = `tweety` and `X` = `polly`. If we want to know which birds cannot fly, we can pose the query

```
fs \\- (flies(X) \- false).
```

and the system will respond with `X` = `pengo`.

3 Functional Logic Programming: A First Example

The key to using GCLA as a (kind of) first-order functional programming language is the inference rule *D-left*, *Definition Left* also called the rule of *Definitional Reflection* [4, 6], which gives us the opportunity to reason on the basis of assumptions with respect to a given definition. The rule of Definitional Reflection tells us that if we have an atom a as an assumption and C follows from everything that defines a then C follows from a:

$$\frac{\Gamma, A \vdash C}{\Gamma, a \vdash C}\ D\vdash \text{ for each } a \in D(a)$$

where $D(a) = \{A \mid (a \Leftarrow A) \in D\}$. With this rule at hand functional evaluation becomes a special case of hypothetical reasoning. Asking for the value of the function `f` in `x` is done with the query:

```
f(x) \- C.
```

3.1 Defining addition

Functional programming languages can be regarded as syntactical sugar for typed lambda-calculus, or as languages especially constructed to define functions. In ML we can write a definition of addition using successor arithmetic:

```
datatype nat = zero | s of nat

fun plus(zero,N) = N
  | plus(s(M),N) = s(plus(M,N))
```

Functions in GCLA are also defined by a number of equations so the naive way to define addition as a function in GCLA would be to mimic the ML program and write the definition:

```
plus(zero,N)<= N.
plus(s(M),N)<= s(plus(M,N)).
```

Unfortunately this will not give us a function that computes anything useful since we have not defined what the value of `s(plus(M,N))` should be. What we forgot was that in the (strict) functional language ML the type definition and the computation rule together give a way to compute a value from `s(plus(M,N))`. The type definition says that `s` is a constructor function and since the language is strict the argument of `s` is evaluated before a dataobject is constructed. One way to achieve the same thing in GCLA is to introduce an object constructing function `succ`:

```
succ(X) <= (X ->Y) -> s(Y).
```

We read this as "the value of `succ(X)` is defined to be `s(Y)` if the value of `X` is `Y`" . The function `succ` plays much the same role as the constructor function `s` in the functional program, it is used to build dataobjects. We will sometimes call such functions that are used to build dataobjects *objectfunctions*.

How the function `succ` works can be seen in the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\mathtt{Y} = \mathtt{0}\}}{\mathtt{0} \vdash \mathtt{Y}}\ \textit{Axiom}}{\vdash \mathtt{0} \rightarrow \mathtt{Y}}\ \textit{A-right} \quad \dfrac{\{\mathtt{s(Y)} = \mathtt{X}\}}{\mathtt{s(Y)} \vdash \mathtt{X}}\ \textit{Axiom}}{(\mathtt{0} \rightarrow \mathtt{Y}) \rightarrow \mathtt{s(Y)} \vdash \mathtt{X}}\ \textit{A-left}}{\mathtt{succ(0)} \vdash \mathtt{X}}\ \textit{D-left}$$

The functional GCLA definition we have given so far only allow nested function calls in its second argument, an expression like `plus(plus(zero,zero),zero)` is undefined. Since we have no external computation rule telling us to evaluate arguments to functions before they are called we must add the following extra clause to handle this case:

```
plus(Exp,N)#{Exp\=zero, Exp\=s(_)}<=(Exp -> M) -> plus(M,N).
```

The guard is needed to make the different clauses defining addition mutually exclusive.

So at last we have a useful definition of addition. If we use the standard rules we are unfortunately still able to construct some undesired derivations like

$$\cfrac{\cfrac{\cfrac{}{\mathtt{false} \vdash \mathtt{X}}\,\textit{False-left}}{\mathtt{s(zero)} \vdash \mathtt{X}}\,\textit{D-left}}{\mathtt{plus(zero,s(zero))} \vdash \mathtt{X}}\,\textit{D-left}$$

which succeeds without binding `X`. The general problem that needs to be solved is to choose correctly between the rules *D-left* and *Axiom*. To do this we need to distinguish canonical values from non-canonical values. We achieve this by giving the canonical values circular definitions:

```
zero <= zero.
s(X) <= s(X).
```

Now the canonical values are defined so they can not be absurd, but on the other hand the definition is empty in content so there is nothing to be gained from using *D-left*. We also restrict our inference rules so that the *axiom* rule only is applicable to circularly defined atoms, and symmetrically restrict the rule *D-left* to be applicable only to atoms not circularly defined. We call atoms with circular definitions *canonical objects*.

In the final definition below the first three clauses corresponds to the *type definition* in the ML program, while the last three really constitutes the *definition* of the addition function.

```
0 <= 0.
s(X) <= s(X).

succ(X) <= (X <- Y) -> s(Y).

plus(0,N)<= N.
plus(s(M),N) <= succ(plus(M,N)).
plus(Exp,N)#{Exp \= 0, Exp \= s(_)}<= (Exp -> M) -> plus(M,N).
```

4 A Calculus for Functional Logic Programming

In this section we describe a basic rule definition to functional logic program definitions. We present the inference rules as a sequent calculus to enhance readability.

We call the rule definition consisting of the rules below *FL* (for functional logic). This rule definition shows implicitly the choices we have made as for what counts as a valid functional logic program. We have chosen to work with the common condition constructors '`,`' ,' `;`' and ' `->`' , which all have more or less their usual meaning. However, the inference rules are restricted to specialize them to functional logic programs. We have also added a special condition constructor ,'`not`' ,for negation.

One of the goals of *FL* is to give a useful rule definition producing as few redundant answers as possible, to achieve this we have made the calculus deterministic in the sense that at most one inference-rule can be applied to each object-level sequent.

FL is very similar to the calculus *DOLD* [6] used to interpret the meta-level of a GCLA program (that is, *DOLD* is used to interpret the GCLA code of the rules here described) This should not be surprising since the meta-level of a GCLA program is nothing but an indeterministic functional logic program run backwards.

4.1 Rules of Inference

The inference rules of *FL* can be naturally divided into two groups, rules relating atoms to a definition and rules for constructed conditions.

4.1.1 Rules Relating Atoms to a Definition.

$$\frac{\vdash C\sigma}{\vdash c \quad \sigma}\ \textit{D-right} \quad (b \Leftarrow C) \in D, \sigma = mgu(b, c), C\sigma \neq c\sigma$$

$$\frac{D(a\sigma) \vdash C\sigma}{a \vdash C \quad \sigma}\ \textit{D-left}\ \ \sigma \text{ is an } a\text{-sufficient substitution }, a\sigma \neq D(a\sigma)$$

$$\frac{}{a \vdash c \quad \sigma\tau}\ \textit{D-Axiom}\ \ \sigma \text{ is an } a\text{-sufficient substitution }, a\sigma = D(a\sigma), \tau = mgu(a\sigma, c\sigma)$$

The restrictions we put on these definitional rules are such that they are mutually exclusive,a very important feature to minimize the number of possible answers.

4.1.2 Rules for Constructed Conditions.

The rules for constructed conditions are essentially Gentzen's sequent calculus rules restricted to allow at most one element in the antecedent. We only show a few of them as examples:

$$\frac{}{false \vdash false}\ \textit{False-left}$$

$$\frac{A \vdash B}{\vdash A \rightarrow B}\ \textit{A-right}$$

$$\frac{C_1 \vdash C \quad C_2 \vdash C}{(C_1; C_2) \vdash C}\ \textit{O-left}$$

4.2 Queries

Both the left and right hand side of sequents in queries may be arbitrarily complex as long as we remember that whenever there is something in the left hand side the right hand side *must* be a *variable* or a (partially instantiated) *canonical object.*

Assuming that `f`,`g` and `h` are functions, `p` and `q` predicates and `a`, `b` and `c` canonical objects some possible examples are

```
\- p(X).
```

"Does `p` hold for some `X`"

```
p(b) -> g(a,b) \- C
```

"What is the value of `g` in `(a,b)` provided that `p(b)` holds"

```
(p(X);q(X)) -> (f(X);h(X)) \- a.
```

"Find a value of `X` such that `p(X)` *or* `q(X)` holds and both `f`*and* `h` has the value `a` in X"

5 Writing Functional Logic Program Definitions

Now that we have given a calculus for functional logic program definitions, it is in order that we also describe the structure of the definitions it can be used to interpret and how to write programs. Note that all readings we give of definitions, conditions and queries are with respect to the given rule definition (calculus), *FL*, and that there is nothing intrinsical in the definitions *themselves* that forces this particular interpretation. For example, if we allow contraction, and several elements in the antecedent, function definitions as we describe them cease to make sense since it is then possible to prove things like

```
plus(s(0),s(0)) \- s(s(s(0))).
```

using the definition of addition given in Sect. 3.

Conditions are built up using the condition constructors ',' , ';' , '->' , '`true`' , '`false`' and '`not`'. Both functions and predicates are defined using the same condition constructors and there is no easily recognizable syntactic difference between functions and predicates. The difference in interpretation of functions and predicates instead comes from whether they are intended to be used to the left or to the right in queries.

When we read and write functional logic program definitions the condition constructors have different interpretations depending on wether they occur to the left or to the right,so when we write a function or a predicate we must always keep in mind wether the condition we are currently writing will be used to the left or to the right to understand its meaning. The basic principle is that predicates are used to the right (negation excepted), while functions are used to the left.

In order to show that it is not always so obvious to see what constitutes function and predicate definitions respectively, we look at a simple example:

```
q(X) <= p(X) -> r(X).
```

If we read this as defining a predicate, we read it as "`q(X)` holds if the value of `p(X)` is `r(X)`" , read as a (conditional) function it becomes "the value of `q(X)` is defined to be `r(X)` provided that `p(X)` holds" .

5.1 Defining Predicates

Since pure Prolog is a subset of GCLA, defining predicates is very much the same thing in this context as in Prolog even though the theoretical foundation is different. The predicate definitions allowed by *FL* also provides two extensions of pure Prolog in predicates ; use of functions in conditions defining predicates and constructive negation.

A predicate definition defining the predicate P consists of a number of definitional clauses

$P(t_1,\ldots,t_n) <= C_1.$
...
$P(t_1,\ldots,t_n) <= C_m\ .\ n \geq 0, m > 0$

where each C_i is a *Predicate Condition* as described below, and all variables occurring in C_i not occurring in the head of the clause are understood as existentially quantified.

We say that a condition C is a Predicate Condition if

- C is an atom
- C = $true$ or C = $false$
- C = (C_1,C_2) , where both C_1 and C_2 are predicate conditions. We read this as "C holds if C_1 and C_2 holds"
- C = (C_1;C_2), where both C_1 and C_2 are predicate conditions. We read this as "C holds if C_1 or C_2 holds"
- C = (C_1 -> C_2), where C_1 is a *Functional Condition* as described below and C_2 is a variable or a (partially instantiated) canonical object. We read this as "C holds if the value of C_1 is C_2"
- C = not(C_1) , where C_1 is a predicate condition. We read this as "C holds if C_1 can be proven false".

5.2 Defining Canonical Objects

Each atom that should be regarded as a canonical object in a definition, in the sense that it could possibly be the result of some function, is given a circular definition. From an operational point of view this is essential to prevent further application of the rule *D-left* and allow application of the rule *D-axiom*. These circular definitions also set canonical objects apart as belonging to a special class of terms since they can not be proven true or false in *FL*.

Generally the definition of the canonical objects S of arity n is

$$S(X_1, \ldots, X_n) <= S(X_1, \ldots, X_n).$$

where each X_i is a variable.

Note that we make a distinction between a *canonical object* and a *canonical value*. Any atom which has a circular definition is a canonical object, while a canonical value is a canonical object where each subpart is a canonical value (a canonical object of arity zero is also a canonical value).

5.2.1 Object-Functions and Implicit Type Definitions

In the definition of addition in Sect. 3 we used the objectfunction `succ` to ensure that all numbers we constructed were fully evaluated canonical values built up from `s` and `0` only. Generally to each canonical object S of arity n we create an objectfunction F to be used whenever we want to build an object of type S in a definition. F has the definition:

$$F(X_1, \ldots, X_n) <= (X_1 \text{ -> } Y_1),\ldots,(X_n \text{ -> } Y_n) \text{ -> } S(X_1, \ldots, X_n) \text{ .}$$

The canonical objects and objectfunctions together form an *Implicit Type Definition*. The implicit type definition for lists is:

```
[] <= [].
[X|Xs] <= [X|Xs] .

cons(X,Xs) <= (X -> Y),(Xs ->Ys) -> [Y|Ys].
```

5.2.2 Laziness and Strictness.

When we use *FL* to evaluate programs it is the definition and not the computation rule that determines if a function is strict or lazy. To define a strict function we always use an objectfunction when we construct a dataobject in a program, this will ensure that evaluation continues until we reach a fully evaluated value. The definition of `plus` given in Sect. 3 follows this convention, as does the following definition of `append` which uses the implicit type definition of lists:

```
append([],Ys) <= Ys.
append([X|Xs],Ys) <= cons(X,append(Xs,Ys)).
append(E,Ys)#{E\=[],E\=[_|_]}<= (E -> Xs) -> append(Xs,Ys).
```

In lazy functional languages expressions are evaluated to WHNF, in our context this means that evaluation stops whenever we reach a canonical object, whether or not its subparts are fully evaluated. When we write function definitions this simply means that we omit the object-functions whose purpose is to evaluate the subparts of canonical objects. The lazy version of `append` then becomes:

```
append([],Ys) <= Ys.
append([X|Xs],Ys) <= [X|append(Xs,Ys)].
append(E,Ys)#{E\=[],E\=[_|_]}<= (E -> Xs) -> append(Xs,Ys).
```

If we use lazy evaluation function definitions must be uniform and have only shallow patterns [7].

5.3 Function Definitions

A definition of a function F really consists of two parts, an implicit type definition T made up of definitions of canonical objects and objectfunctions used in F, and the definition of F itself.

A function definition defining F then consists of a number of definitional clauses

$F(t_1,\ldots,t_n) <= C_1.$
...
$F(t_1,\ldots,t_n) <= C_m \ . \ n \geq 0, m > 0$

where each C_i is a *Functional Condition* as described below, and all variables occurring in C_i not occurring in the head are understood as universally quantified.

We say that a condition C is a Functional Condition if

- C is an atom
- $C = (C_1,C_2)$, where both C_1 and C_2 are functional conditions. We read this as "the value of C is the value of C_1 or C_2"
- $C = (C_1$ ->$C_2)$, where C_1 is a predicate condition and C_2 is functional condition. We read this as "the value of C is C_2 provided that C_1 holds"
- $C = (C_1;C_2)$, where both C_1 and C_2 are functional conditions. We read this as "the value of C is B if B is the value of both C_1 and C_2".

If the heads of two or more clauses defining a function are overlapping all the corresponding bodies must evaluate to the same value, since the definiens operation used in *D-left* collects all clauses defining an atom. For example consider the function definition:

```
f(0) <= 0.
f(N) <= s(0)
```

Even though `f(0)` is defined to be `0` the derivation of the query

```
f(0) \- C.
```

will fail since `f(0)` is also defined to be `s(0)`, the definition of `f` is *ambiguous.*

5.4 Examples

If we use the lazy version of append and ask the query

```
append([0],[s(0)]) \- C.
```

we will get the single answer `C = [0|append([],[s(0)])]`, since this is a canonical value and cannot be evaluated further. To get a fully evaluated answer we have to force evaluation somehow, one way is to use the following function `show` which takes canonical values apart and evaluate the subparts:

```
show(0) <= 0.
show(s(X)) <= (show(X) -> Y) -> s(Y).
show([]) <= [].
show([X|Xs]) <= (show(X) ->Y),(show(Xs)->Ys) -> [Y|Ys].
show(E)#{E\=0,E\=s(_),E\=[],E\=[_|_]} <= (E -> V) -> show(V).
```

As another example we define a function `odd_double` defined only for odd numbers, this function uses the definition of plus from Sect.3. Note how the predicate `odd` is used to restrict the applicability of `odd_double` to odd numbers only.

```
odd_double(X) <= (X ->X1),
  odd(X1) -> plus(X1,X1).

odd(s(X)) <= even(X).

even(0).
even(s(X)) <= odd(X).
```

6 Generating Specialized Rule Definitions

In Sect. 3 we noticed that *FL* is deterministic, thus making search strategies more or less superfluous, the answers will be the same no matter in what order the inference rules are tried.

However, there are several reasons to use search strategies anyway. The most important concerns integration of functional logic programs into large systems using other programming paradigms as well; without a proper search strategy all the rules and strategies concerning the rest of the system will also be tested. Another reason is to enhance efficiency, the strategies we present in this section will almost always try the correct rule. Yet another reason is that the specialized rules we create allows us to write definitions without "evaluation clauses" for arguments thus making definitions shorter and more elegant.

In [3] we suggested the method *Local Strategies* as a way to write efficient rule definitions to a given definition. We also suggested that it should be possible to more or less automatically generate rule definitions according to this method to some programs, the rules we generate are applications of this method.

6.1 Specialized D-Rules

It is beyond the scope of this paper to describe the algorithms and heuristics used to generate rules, in short we try to decide which atoms are functions and which are predicates and then create specialized rules which as far as possible precompile the rule-sequences needed to use each particular function and predicate. Sometimes it is impossible to distinguish functions from predicates, in such cases an oracle (the user) is queried. It is also beyond the scope of this paper to describe the resulting rules and strategies in any greater detail, we simply give schematic descriptions of the D-Rules created for each function and predicate

6.1.1 Predicate Definitions and their D-right Rules

When we create specialized D-right rules to predicates we let them evaluate all arguments before we try to find a unifiable clause. This means that we may have functional expressions as arguments to predicates. It also means that the only allowed patterns in the heads of predicate definitions (and function definitions) are variables and canonical values. This corresponds closely to the approach taken in functional logic languages based on a constructor discipline [5]. Generally if P is a predicate of arity n its corresponding D-right rule becomes:

$$\frac{X_1 \vdash Y_1 \ldots X_n \vdash Y_n \quad \vdash B}{\vdash \ P(X_1, \ldots, X_n)} \quad B \in D(P(Y_1, \ldots, Y_n))$$

If we use strict functions in the arguments of predicates this approach works well enough, but if we combine lazy functions and predicates the patterns allowed in the heads of predicate definitions must be further restricted to shallow patterns with no repeated variables.

6.1.2 Function Definitions and their D-left Rules

The usual meaning of a strict function definition is that its arguments are evaluated before the function is called. It is therefore perfectly reasonable to associate with each strict function a rule which evaluates each of the arguments and then tests to see if the resulting atom is defined. If F is a function of arity n the rule becomes:

$$\frac{X_1 \vdash Y_1 \ldots X_n \vdash Y_n \quad Dp \vdash C}{F(X_1, \ldots, X_n) \ \vdash \ C} \quad Dp = D(F(Y_1, \ldots, Y_n))$$

In lazy functions we only evaluate arguments which have a non-variable pattern, if we do not want to risk evaluating to many arguments it is important that the function definitions are uniform, the patterns must also be linear, that is no variable may occur more than once in the head of a clause.

If we use lazy functions the rule-generator also generates a strategy which makes it possible to evaluate a functional condition to a fully evaluated canonical value which makes show-functions like the one given in Sect. 5 superfluous. It is up to the user to specify if a function is strict or lazy.

6.2 Examples

We now give some examples to show how functional logic programs may be written if we create specialized rules to each program.

6.2.1 Addition Revisited

The strict definition of addition given in Sect. 3 may now be reformulated as:

```
0 <= 0.
s(X) <= s(X).
succ(X) <= s(X).

plus(0,N) <= N.
plus(s(M),N) <= succ(plus(M,N)).
```

The definition of succ may look a bit strange but remember that succ has a corresponding specialized *D-left* rule which evaluates the argument.

6.2.2 Mixing Functions and Predicates

Let append and mem have the definitions:

```
[] <= [].
[X|Xs] <= [X|Xs].
cons(X,Xs) <= [X|Xs].

append([],Ys) <= Ys.
append([X|Xs],Ys) <= cons(X,append(Xs,Ys)).

mem(X,[X|_]).
mem(X,[_|Ys]) <= mem(X,Ys).
```

and let plus be defined as above, we may then ask queries like:

```
\- mem(plus(s(0),X),append(cons(0,[s(0)]),[s(s(0))])).
X = 0 ? ;
X = s(0) ? ;
```

6.2.3 Computing Prime Numbers

As our last example we show a definition that can be used to compute prime numbers using the well-known sieve of Erastothenes. To implement this algorithm we use a combination of lazy functions and predicates, and also the possibility to regard numbers as canonical objects and use the ordinary arithmetic operations. We also use a special condition constructor if (that has a corresponding inference rule) which chooses one of two functional conditions depending on whether a predicate is provable or not.

```
[] <= [].
[X|Xs] <= [X|Xs].

primes <= sieve(from(2).

sieve([P|Ps]) <= [P|sieve(filter(P,Ps))].
```

```
filter(_,[]) <= [].
filter(N,[X|Xs]) <= if(p(N,X),
                       filter(N,Xs),
                       [X|filter(N,Xs)]).

p(A,B) <= (B mod A) =:= 0.

from(M) <= (M -> N) -> [N|from(N+1)].
```

We also need something to force evaluation. We can not use a show strategy since the result is infinite. The solution is to define a predicate to print the elements of the infinite list of primes

```
print_list(E) <= (E -> [X|Xs]),put(X),print_list(Xs).
```

where `put` is defined elsewhere. Now the query

```
\- print_list(primes).
```

will print `2 3 5 7 11` and so on until we run out of memory.

References

[1] M. Aronsson, *GCLA, The Design Use and Implementation of a Program Development System*, Ph D thesis, Department of Computer Sciences, University of Stockholm, 1993.

[2] H. Ait-Kaci, A. Podelski, Towards a Meaning of Life, *Journal of Logic Programming*, vol.16 pp 195-234, 1993.

[3] G. Falkman, O. Torgersson, Programming Methodologies in GCLA, in, *Extensions of Logic Programming*, Springer LNCS 798, 1994.

[4] L. Hallnäs, Partial Inductive Definitions, *Theoretical Computer Science*, vol 87,pp 115-142,1991.

[5] M. Hanus, The Integration of Functions into Logic Programming; From Theory to Practice, *Journal of Logic Programming*, vol 19/20, pp 583-628,1994.

[6] P. Kreuger, GCLA II, A Definitional Approach to Control, in, *Extensions of Logic Programming*, Springer LNCS 596, 1992.

[7] H. Kuchen, J,J Moreno-Navarro, M. Rodriguez-Artalejo. Implementing a Lazy Functional Logic Language with Disequality Constraints, in *Proc. Proc. of the 1992 Joint International Conference and Symposium on Logic Programming.* MIT Press,1992.

Extensions of Structural Synthesis of Programs

Tarmo Uustalu
Dept of Teleinformatics, The Royal Institute of Technology
Electrum 204, S-164 40 Kista (Stockholm), Sweden
email tarmo@it.kth.se

Abstract

Structural synthesis of programs (SSP) is an approach to deductive synthesis of functional programs using types as specifications and based on the Curry-Howard correspondence and on an intensional treatment of the notion of type. The implemented programming environments employing SSP have been based on a fragment of intuitionistic propositional logic (simple type theory) and on a natural-deduction proof system. In the paper, we indicate that the proof search strategy used in these systems is applicable to a variety of natural-deduction proof systems (not necessarily of intuitionistic or propositional logics), and that the object-oriented user front-end specification language of the NUT programming environment can, in fact, be given a useful logical semantics in terms of intuitionistic first-order logic.

1 Introduction

Structural synthesis of programs (SSP) as proposed and described by Mints and Tyugu [MT82] is an approach to deductive synthesis of functional programs using types as specifications and based on the Curry-Howard correspondence and on an intensional treatment of types as descriptors of the conceptual roles of values in computations. The approach has a practical orientation; a feasible balance has been sought between expressiveness of the specification language and efficiency of proof search. The implemented systems of SSP have been based on a fragment of intuitionistic propositional logic (simple type theory). The work on SSP is related to automated software engineering, programming in type theories, and automated theorem proving.

In the paper, we indicate that the two cornerstones of SSP — the particular proof search strategy underlying the synthesis procedure, and idea of using types as specifications in program synthesis — are both special applications of something quite general.

The proof search technique employed in SSP has been described in different formulations by a number of authors: Maslov (the inverse method) [Mas64] (for an accessible exposition, see [Lif89]), Mints (general resolution) [Min90a, Min90b, Min93], Stålmarck (the assure-method) [Stå90]. We present the SSP proof search strategy as an instance of a generic strategy of forward-directed search in natural-deduction calculi where conclusion generation is limited to sequents that potentially may participate in a normal derivation of the goal sequent and where conclusion generation under an extra assumption is tried only if no new conclusions can be derived otherwise and where always the most general conclusion is drawn (unification). In the propositional case, decision of proof-theoretically easy goal formulae (this notion is formalized!) is polynomial-time and specifications one usually would write in practice are proof-theoretically easy.

As an example of taking advantages of the first-orderness in specifications and of the unification mechanism in proof search, we show how the object-oriented user front-end specification language of the NUT system (the most recent system in the Priz family of SSP-based programming environments), which used to be translated into propositional logic, can also be translated into predicate logic. The second translation yields faster proof search and shorter proofs.

The paper is organized as follows. Section 2 is a brief exposition of the kernel ideas of structural synthesis of programs. In Section 3, we define what kind of proof systems we consider to be natural-deduction systems, describe the proof search strategy, and present the main theoretic results about the strategy (concerning completeness and complexity). In Section 4, we present a semantics of the specification language of NUT in terms of intuitionistic first-order predicate logic. In Section 5, we formulate our conclusions.

2 Structural Synthesis of Programs

As we stated already, structural synthesis of programs is an approach to deductive synthesis of functional programs where types are used as specifications. It is based on the Curry-Howard correspondence [How80] and on an intensional treatment of types as descriptors of the conceptual roles of values in computations. The idea was proposed in the beginning of 1980s by Mints and Tyugu [MT82, MT90], and has since then been tried out, in combination with other ideas, in a family of programming environments (Priz, ExpertPriz, CPriz, NUT) [MT88, Tyu91, Tyu94]. Since the theoretical justification of the technique, together with descriptions of the concrete programming systems and examples of applications can be found in the cited sources, we will only mention some essentials.

Structural synthesis was intended as a software engineering technique, for application in programming-in-the-large. Thus, in structural synthesis, one usually does not synthesize programs from the scratch, but from knowing some realizations of some specifications. Such background knowledge about specifications and programs is called a computational model. Given the knowledge that specifications $s_1, \ldots, s_n$ are realized by programs $p_1, \ldots, p_n$, respectively, and given a specification s (a goal specification), the synthesis of a program p realizing s proceeds in two steps. At the first step, a proof is sought for s in the intuitionistic theory one obtains by regarding $s_1, \ldots s_n$ as extralogical axioms. Any such proof represents the structure of some program realizing s. At the second step, this program is extracted from the proof, using the realizability semantics of intuitionistic logic and the knowledge about realizations for the extralogical axioms.

Structural synthesis can be regarded as a kind of logic programming on functional values; one might perhaps call it programming with types. Similarly to conventional logic programming, which we will call relational programming for a moment in order to avoid confusion, computation in programming with types proceeds by proof search, and the result of a computation is something extracted from the proof found. The difference lies in what namely is extracted. In relational programming, the goal is an existential formula and the value extracted is a witness of its validity (semantically speaking, the value extracted is a witness of non-emptiness of a relation). In programming with types, the value extracted is a realization of the goal formula (semantically speaking, a witness of non-emptiness of a type).

Quite obviously, the work on structural synthesis is related to work on programming in type theories as advocated in e.g. [NPS90]. The difference lies in the emphases. In programming in type theories, the main interests are verification of programs against specifications, and transformational development of specifications into programs. The type languages used are of a very rich expressive power. In structural synthesis, the main objective has been efficient automatic synthesis, which obviously sets very restrictive constraints on the type language.

3 Proof Search in Natural-Deduction Proof Systems for Propositional Logics

In this section, we will show that the proof search strategy employed in SSP is applicable to a variety of natural-deduction proof systems of propositional logics; namely, to all those enjoying some form of a subformula property or some other property ensuring that the number of different formulae that potentially may be needed in a proof of a given formula is polynomial in the size of this formula.

We shall present the strategy and the theoretic results about it on the example of two proof systems: the "canonical" natural-deduction calculus for intuitionistic propositional logic, formulated by Gentzen and Prawitz [Pra65], and the Dilemma-system (a natural-deduction-inspired proof system for classical propositional logic), formulated by Stålmarck [Stå90].

SSP is an example of an application of the proof search strategy to the first proof system. Independently, Stålmarck formulated the strategy and the results for the second proof system and has applied it in verification of state-transition systems (where both implementations and specifications of such systems are formalized in the language of propositional logic). His method has been patented for commercial use.

Natural-deduction proof systems are a special case of sequent-based proof systems. Rules of such proof systems act upon sequents, which are figures of the form $\Gamma \vdash \Delta$, where Γ, Δ are finite sets of formulae. The left-hand side of a sequent is called its antecedent, whereas the right hand side of it is called its succedent. In every rule application, some formulae (more exactly, formula occurrences) in the premisses and in the conclusion appear as active, whereas the others serve as a passive context.

There exists no universally accepted definition of a natural-deduction proof system. For us and here, natural-deduction proof systems are sequent-based proof systems satisfying the following conditions:

- the succendents of sequents contain exactly one formula;
- no proof rule contains active formulae in the antecedent of the conclusion;
- just a few rules contain active formulae in the antecedents of the premisses.

The rules that have all their active formulae in the succedents of the premisses and of the conclusion are called simple rules. The rules that have some active formulae in the antecedents of the premisses are called assumption rules.

The kernel of a typical natural-deduction proof system is formed by introduction and elimination rules of the logical operators present in the language of the given logic. The active formulae of a rule for a certain operator are a formula with the given operator as the outermost operator (the major formula) and its immediate subformulae (the minor formulae). In an introduction rule, the major formula appears in the succedent of the conclusion, in an elimination rule, the major formula appears in the succedent of one of the premisses.

Intuitionistic propositional natural deduction

Iden	$\dfrac{}{\boxed{A} \vdash \boxed{A}}$	Weak	$\dfrac{\Gamma \vdash C}{\Gamma, D \vdash C}$
$\wedge_I$	$\dfrac{\Gamma \vdash \boxed{A} \quad \Gamma \vdash \boxed{B}}{\Gamma \vdash \boxed{A \wedge B}}$	$\wedge_E$	$\dfrac{\Gamma \vdash \boxed{A \wedge B}}{\Gamma \vdash \boxed{A}} \quad \dfrac{\Gamma \vdash \boxed{A \wedge B}}{\Gamma \vdash \boxed{B}}$
$\vee_I$	$\dfrac{\Gamma \vdash \boxed{A}}{\Gamma \vdash \boxed{A \vee B}} \quad \dfrac{\Gamma \vdash \boxed{B}}{\Gamma \vdash \boxed{A \vee B}}$	$\vee_E$	$\dfrac{\Gamma \vdash \boxed{A \vee B} \quad \Gamma, \boxed{A} \vdash C \quad \Gamma, \boxed{B} \vdash C}{\Gamma \vdash C}$
$\supset_I$	$\dfrac{\Gamma, \boxed{A} \vdash \boxed{B}}{\Gamma \vdash \boxed{A \supset B}} \quad \dfrac{\Gamma \vdash \boxed{B}}{\Gamma \vdash \boxed{A \supset B}}$	$\supset_E$	$\dfrac{\Gamma \vdash \boxed{A \supset B} \quad \Gamma \vdash \boxed{A}}{\Gamma \vdash \boxed{B}}$
$\neg_I$	$\dfrac{\Gamma, \boxed{A} \vdash \bot}{\Gamma \vdash \boxed{\neg A}}$	$\neg_E$	$\dfrac{\Gamma \vdash \boxed{\neg A} \quad \Gamma \vdash \boxed{A}}{\Gamma \vdash C}$

In this proof system, $\supset_I$, $\neg_I$ and $\vee_E$ are assumption rules; all other rules are simple. This proof system enjoys the following subformula property: only subformulae of the goal sequent are potentially needed in a proof; moreover, a formula A is potentially necessary in the position of a major formula of an application of an I-rule (resp. E-rule) only if the goal sequent contains a positive (resp. negative) occurrence of A. A formula A is potentially necessary as an antecedent formula only if the goal sequent contains a positive occurrence of some formula of the form $A \supset B$ or $\neg A$ or a negative occurrence of some formula of the form $A \vee B$ or $B \vee A$.

The Dilemma-system

Iden	$\dfrac{}{\boxed{A} \vdash \boxed{A}}$	Weak	$\dfrac{\Gamma \vdash C}{\Gamma, D \vdash C}$
Dilemma	$\dfrac{\Gamma, \boxed{A} \vdash C \quad \Gamma, \sim\boxed{A} \vdash C}{\Gamma \vdash C}$	Contrad	$\dfrac{\Gamma \vdash \boxed{A} \quad \Gamma \vdash \sim\boxed{A}}{\Gamma \vdash C}$
$\wedge_I$	$\dfrac{\Gamma \vdash \boxed{A} \quad \Gamma \vdash \boxed{B}}{\Gamma \vdash \boxed{A \wedge B}}$	$\wedge_E$	$\dfrac{\Gamma \vdash \boxed{A \wedge B}}{\Gamma \vdash \boxed{A}} \qquad \dfrac{\Gamma \vdash \boxed{A \wedge B}}{\Gamma \vdash \boxed{B}}$
$\sim\wedge_I$	$\dfrac{\Gamma \vdash \sim\boxed{A}}{\Gamma \vdash \sim\boxed{A \wedge B}} \qquad \dfrac{\Gamma \vdash \sim\boxed{B}}{\Gamma \vdash \sim\boxed{A \wedge B}}$	$\sim\wedge_E$	$\dfrac{\Gamma \vdash \sim\boxed{A \wedge B} \quad \Gamma \vdash \boxed{B}}{\Gamma \vdash \sim\boxed{A}}$
			$\dfrac{\Gamma \vdash \sim\boxed{A \wedge B} \quad \Gamma \vdash \boxed{A}}{\Gamma \vdash \sim\boxed{B}}$
$\vee_I$	$\dfrac{\Gamma \vdash \boxed{A}}{\Gamma \vdash \boxed{A \vee B}} \qquad \dfrac{\Gamma \vdash \boxed{B}}{\Gamma \vdash \boxed{A \vee B}}$	$\vee_E$	$\dfrac{\Gamma \vdash \boxed{A \vee B} \quad \Gamma \vdash \sim\boxed{B}}{\Gamma \vdash \boxed{A}}$
			$\dfrac{\Gamma \vdash \boxed{A \vee B} \quad \Gamma \vdash \sim\boxed{A}}{\Gamma \vdash \boxed{B}}$
$\sim\vee_I$	$\dfrac{\Gamma \vdash \sim\boxed{A} \quad \Gamma \vdash \sim\boxed{B}}{\Gamma \vdash \sim\boxed{A \vee B}}$	$\sim\vee_E$	$\dfrac{\Gamma \vdash \sim\boxed{A \vee B}}{\Gamma \vdash \sim\boxed{A}} \qquad \dfrac{\Gamma \vdash \sim\boxed{A \vee B}}{\Gamma \vdash \sim\boxed{B}}$
$\supset_I$	$\dfrac{\Gamma \vdash \sim\boxed{A}}{\Gamma \vdash \boxed{A \supset B}} \qquad \dfrac{\Gamma \vdash \boxed{B}}{\Gamma \vdash \boxed{A \supset B}}$	$\supset_E$	$\dfrac{\Gamma \vdash \boxed{A \supset B} \quad \Gamma \vdash \sim\boxed{B}}{\Gamma \vdash \sim\boxed{A}}$
			$\dfrac{\Gamma \vdash \boxed{A \supset B} \quad \Gamma \vdash \boxed{A}}{\Gamma \vdash \boxed{B}}$
$\sim\supset_I$	$\dfrac{\Gamma \vdash \boxed{A} \quad \Gamma \vdash \sim\boxed{B}}{\Gamma \vdash \sim\boxed{A \supset B}}$	$\sim\supset_E$	$\dfrac{\Gamma \vdash \sim\boxed{A \supset B}}{\Gamma \vdash \boxed{A}} \qquad \dfrac{\Gamma \vdash \sim\boxed{A \supset B}}{\Gamma \vdash \sim\boxed{B}}$
$\neg_I$	$\dfrac{\Gamma \vdash \sim\boxed{A}}{\Gamma \vdash \boxed{\neg A}}$	$\neg_E$	$\dfrac{\Gamma \vdash \boxed{\neg A}}{\Gamma \vdash \sim\boxed{A}}$
$\sim\neg_I$	$\dfrac{\Gamma \vdash \boxed{A}}{\Gamma \vdash \sim\boxed{\neg A}}$	$\sim\neg_E$	$\dfrac{\Gamma \vdash \sim\boxed{\neg A}}{\Gamma \vdash \boxed{A}}$

In this proof system, Dilemma is the single assumption rule; all other rules are simple. This proof system enjoys the following subformula property: only subformulae of the goal sequent are potentially needed in a proof; moreover, a formula A is potentially necessary in the position of a major formula of an application of an I- or $\sim E$- rule (resp. E- or $\sim I$-rule) only if the goal sequent contains a positive (resp. negative) occurrence of A. A formula A is potentially necessary as an antecedent formula only if it is a propositional variable occurring in the goal sequent.

We define the degree of assumption interaction of a proof as the maximum number of formulae in the antecedents of the sequents in the proof. We define an i-easy sequent as a formula sequent which has a proof with degree of assuption interaction $\leq i$. We define an i-hard sequent as a sequent which has no proof with degree of assumption interaction $< i$.

The proof search strategy exploits the subformula property, and its key ideas are the following:

- the search is forward-directed: one generates sequents that potentially may be needed in a proof of the goal sequent;
- a proof with minimum degree of assumption interaction is sought; i.e. at first, all potentially necessary 1-easy sequents are generated, then all potentially necessary 2-easy sequents are generated, etc., until the goal sequent is arrived at; this means that the simple rules are always applied in the first place, and the assumption rules are applied only if the simple rules cannot longer yield anything new and potentially necessary;
- for every potentially necessary antecent that is taken into consideration during the search, two datastructures are initialized and subsequently kept up-to-date: a bitvector which tells which potentially necessary sequents with the given antecedent have been generated already, and a queue of potentially necessary simple rule applications which have not yet been tried with the given antecedent, but whose premisses have been generated already;
- atomic names are invented for the subformulae of the goal formulae; instead of working with subformulae directly, one works with their names (to my knowledge, this idea was first put forward by Tseitin [Tse68]);

Two important results can be stated about the proof search strategy:

- it is complete, i.e. if a goal sequent is provable, then the search procedure finds a proof for it;
- the time-complexity of finding a proof of an i-easy goal sequent is

  $$O(NL^i)$$

  where

– N is the number of formulae potentially necessary in the succedents in a proof of the goal sequent, i.e. the number of subformulae of the goal formula (this is proportional to the length of the goal sequent),

– L is the number of formulae potentially necessary in the antecedents in a proof of the goal sequent; in the case of intuitionistic natural deduction, this is the number of implications and negations that occur positively in the goal sequent plus twice the number of disjunctions that occur negatively in the goal sequent; in the case of the Dilemma-system, this is the number of propositional variables occurring the goal sequent; in both cases, $N < L$.

The general decision problem is well-known to be NP-complete for classical propositional logic and PSPACE-complete for intuitionistic propositional logic. The last result tells us that, despite of this, proof search is polynomial-time for both logics, if we can be sure that we do not need high degrees of assumption interaction. Fortunately, it turns out that the goal sequents arising from application problems (tasks of program synthesis in the case of intuitionistic natural deduction, or tasks of state-transition system verification in the case of the Dilemma-system), although often large, are seldom very entangled (since they are usually based on a simple design idea), and, if provable, possess proofs of a quite low degree of assumption interaction. The experience shows that a bound of 2..4 leaves the proof search procedure "practically complete".

4 A Predicate Logic Semantics of an Object-Oriented Structural Specification Language

The NUT system, mentioned in Section 2, uses a specification language built upon the following idioms and principles of object-oriented programming:

- the primary entities of the language are objects; objects can be primitive or compound;
- every primitive object signifies a value with a certain conceptual role (being thus comparable with a "logical variable" in of a Prolog-like language);
- every compound object is associated to a finite set of its components, which are objects in their own right and are identified via component names (component a of object x is denoted as $x.a$ — the usual "dot-notation");
- every object is an instance of some class, classes being templates of objects with similar properties;

- all primitive objects are instances of the predefined class `prim` (in the real implementation, there are more primitive classes— different sorts of primitive values, such as numerics, strings, booleans etc.—, a detail that we shall ignore here);
- compound objects are instantiated from user-defined classes; the class of a compound object determines the names and classes of the components of the object, and may impose a number of realizabilities and definitional equalities on the deep components of the object[1];
- a user-defined class may have superclasses from where it inherits properties; inheritance in NUT is cumulative, not overriding.

From the point of view of specification and synthesis, primitive objects are types, and the object hierarchy is just a way of modular organization of the world of types. Every object is associated to a type variable.

Language

In the declarative part of the NUT language, the specification of the computational domain is presented as a collection of class definitions.

A class definition consists of a finite number of declarations of various kinds. NUT allows following kinds of declarations:

Superclass declarations	`super` cl'	where cl' is a class name
Component declarations	`comp` $a : cl'$	where a is a component name, and cl' is a class name
Equality declarations	`rel` $b_1 = b_2$	where b_1, b_2 are deep component names
Realizability declarations	`rel` f $\{t\}$	where f is a propositional formula built of deep component names, and t is a functional expression

Informal semantics

Given a definition of some class cl. Declarations in the definition have the following semantics:

[1] A deep component of an object is, by definition, a component or a deep component of a component of the object.

`super` cl'	means that every instance x of class cl has all properties of instance of class cl'
`comp` $a : cl'$	means that every instance x of class cl has a component a — object $x.a$ — which is an instance of class cl'
`rel` $b_1 = b_2$	means that, for every instance x of class cl, the primitive objects (types) $x.b_1$ and $x.b_2$ are definitionally equal
`rel` $f\ \{t\}$	means that, for every instance x of class cl, the formula (type) $x.f$ is realized by term t, where $x.f$ is a shorthand notation for the formula arising from f by prefixing every deep component name b in f with x

Translation into propositional logic

Formalization of the informal semantics in terms of propositional logic (simple theory of types) is straightforward.

Given a computational model, consisting of definitions of classes $cl_1, \ldots, cl_n$, the propositional logic semantics of these classes wrt the computational model are sets of realizability assertions defined as follows:

$$\begin{array}{lcl}
\mathbf{clsem}(\texttt{prim}) & = & \emptyset \\
\mathbf{clsem}(cl) & = & \bigcup\{decl \in \mathbf{cldef}(cl) \mid [\![decl]\!]_{cl}\} \\
[\![\texttt{super}\ cl']\!]_{cl} & = & \mathbf{clsem}(cl') \\
[\![\texttt{comp}\ a : cl']\!]_{cl} & = & a.\mathbf{clsem}(cl') \\
[\![\texttt{rel}\ b_1 = b_2]\!]_{cl} & = & \{b_1 \equiv b_2\ \{(\mathrm{id}, \mathrm{id})\}\} \\
[\![\texttt{rel}\ f\ \{t\}]\!]_{cl} & = & \{f\ \{t\}\}
\end{array}$$

where $a.\mathbf{clsem}(cl')$ is a shorthand notation for the set of realizability assertions obtained from $\mathbf{clsem}(cl')$ by prefixing all deep component names in formulae in $\mathbf{clsem}(cl')$ with a.

Finding a term that realizes $x.f$ for a generic instance x of class cl amounts to proving f from $\mathbf{clsem}(cl)$.

A similarly-formulated semantics of the Priz system (where differently from NUT, there are no classes; instead of instantiation from classes, cloning from objects is used) can be found in [MST91].

Translation into predicate logic

In the predicate logic semantics, we represent objects and deep component names by means of terms, and classes by means of binary predicates.

Given a computational model, consisting of definitions of classes $cl_1, \ldots, cl_n$, the predicate logic semantics of these classes are sets of realizability assertions defined as follows:

$$
\begin{array}{lcl}
\mathbf{clsem}(\texttt{prim}) & = & \emptyset \\
\mathbf{clsem}(cl) & = & \bigcup\{decl \in \mathbf{cldef}(cl) \mid [\![decl]\!]_{cl}\} \\
[\![\texttt{super}\ cl']\!]_{cl} & = & \{cl(x,z) \equiv cl'(x(cl),z)\ \{(\mathrm{id},\mathrm{id})\}\} \\
[\![\texttt{comp}\ \ a : cl']\!]_{cl} & = & \{cl(x,a.z) \equiv cl'(x(cl).a,z)\ \{(\mathrm{id},\mathrm{id})\}\} \\
[\![\texttt{rel}\ \ b_1 = b_2]\!]_{cl} & = & \{cl(x,b_1) \equiv cl(x,b_2)\ \{(\mathrm{id},\mathrm{id})\}\} \\
[\![\texttt{rel}\ \ f\ \{t\}]\!]_{cl} & = & \{cl(x,f)\ \{t\}\}
\end{array}
$$

where $cl(x, f)$ is a shorthand notation for the formula obtained from f by replacing every deep component name b in f with $cl(x, b)$.

All formulae above are implicitly universally closed wrt variables x, y.

Finding a term that realizes $x.f$ for a generic instance x of class cl amounts to proving $cl(x, f)$ from $\mathbf{clsem}(cl_1) \cup \ldots \cup \mathbf{clsem}(cl_n)$. Intuitively, provability of $cl(x_0, b)$ means that if object x_0 is of class c then object $x_0.b$ has a value.

Observe that the formula-translations of superclass and component declarations `super` cl' and `comp` $a : cl'$ and are skolemized versions of the formulae $cl(x, z) \equiv \exists y.cl'(y, z)$ and $cl(x, a.z) \equiv \exists y.cl'(y, z)$, respectively.

These two translations of the user front-end specification language into logical languages can be easily seen to be equivalent in the sense that the same synthesis tasks will be solvable and will lead to the same programs. The difference is that if one uses the first translation, synthesis will be done on the deep components of the generic instance x of class cl; if one uses the second translation, synthesis will be essentially done on their classes. The second translation yields faster proof search and shorter proofs, if x has many deep components of same classes.

5 Conclusions

We have shown that the proof search strategy employed in SSP is an instance of a generic strategy applicable to natural-deduction proof systems for various propositional logics (classical or intuitionistic, with modal, fixpoint operators etc.). For us, natural-deduction proof systems are sequent-based proof systems where succedents always consist of exactly one formula, where no proof rule contains active formulae in the antecedent of the conclusion, and where just a few rules may contain active formulae in the antecedents of the premisses. We have noted that the allowed degree of assumption interaction in proofs is a natural measure of hardness of tautologies from the point of view of at least two applications: synthesis of functional programs (by proof search in intuitionistic propositional logic) and verification of state-transition systems (by proof search in classical propositional logic). We have shown that the strategy ensures a polynomial time complexity of proof search if a bound is set on the allowed degree of assumption interaction. The applications of modal and fixpoint logics in structural synthesis are a direction for future study which we have started to look into.

We have also shown that the object-oriented structural specification language which has been employed in the most recent PRIZ-type system — NUT —, and used to be semantically explained terms of intuitionistic propositional logic, has a more concise semantic explanation in terms of intuitionistic predicate logic.

References

[How80] W. A. Howard. The formula-as-types of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, London, 1980.

[Lif89] Vladimir Lifshitz. What is the inverse method? *Journal of Automated Reasoning*, 5(1):1–23, 1989.

[Mas64] S. Yu. Maslov. The inverse method for establishing deducibility in classical predicate calculus. *Soviet Math. Doklady*, 5:1420–1423, 1964.

[Min90a] Grigory Mints. Gentzen-type systems and resolution rule, Part I: Propositional logic. In P. Martin-Löf and G. Mints, editors, *Proc. Int'l Conf. on Computer Logic, COLOG-88, Tallinn, Dec 1988*, pages 198–231, Berlin, 1990. Springer-Verlag. LNCS 417.

[Min90b] Grigory Mints. Gentzen-type systems and resolution rule, Part II: Predicate logic. In *Logic Colloquium '90*, 1990.

[Min93] Grigory Mints. Resolution calculus for the first order linear logic. *Journal of Logic, Language, and Information*, 2(1):59–83, 1993.

[MST91] Grigory Mints, Jan M. Smith, and Enn Tyugu. Type-theoretic semantics of some declarative languages. In J. Bārzdiņš and D. Bjørner, editors, *Baltic Computer Science: Selected Papers*, pages 18–32, Berlin, 1991. Springer-Verlag. LNCS 502.

[MT82] Grigory Mints and Enn Tyugu. Justification of the structural synthesis of programs. *Science of Computer Programming*, 2(3):215–240, 1982.

[MT88] Grigory Mints and Enn Tyugu. The programming system PRIZ. *Journal of Symbolic Computation*, 5(4):359–375, 1988.

[MT90] Grigory Mints and Enn Tyugu. Propositional logic programming and the PRIZ system. *Journal of Logic Programming*, 9(2–3):179–193, 1990.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Clarendon Press, Oxford, 1990.

[Pra65] Dag Prawitz. *Natural Deduction: A Proof-Theoretic Study*. Almqvist & Wiksell, Stockholm, 1965. Stockholm Studies in Philosophy 3.

[Stå90] Gunnar Stålmarck. The Assure method: A proof procedure for propositional logic. Technical Report 1, Logikkonsult NP AB, Stockholm, 1990.

[Tse68] G. S. Tseytin. On the complexity of propositional calculus. *Zapiski Nauchnykh Seminarov LOMI*, 8:235–259, 1968. In Russian.

[Tyu91] Enn Tyugu. Three new-generation software environments. *Communications of the ACM*, 34(6):46–59, 1991.

[Tyu94] Enn Tyugu. Using classes as specifications for automatic construction of programs in the NUT system. *J. Automated Software Engineering*, 1(3–4):315–334, 1994.

Algebra of Broadcasting Systems: Value Passing, Sequential Composition, and Fork

Martin Weichert[*†]

December 5, 1994

Abstract

This paper presents ACBS&, Algebra of Broadcasting Systems with Fork, a process calculus with broadcast communication, which combines value passing and sequential composition.

Putting value passing into a calculus with sequential composition turns out to be more complicated than with a prefixing calculus. A semantics for such a calculus can be defined using environments, as in imperative languages.

Parallel processes have parallel environments, which can lead to ambiguity *which* of the environments shall be passed on to the sequentially following process. ACBS& solves this problem by distinguishing a foreground and a background process. The resulting operator can be interpreted as a *fork* operator, which "forks off" a background process.

The calculus is presented both in a "pure" and a value passing version, together with a complete axiomatisation for the pure calculus.

1 Introduction

The basic ingredients of the calculus ACBS& are:

- CBS with its basic idea broadcast communication, as presented in [Pra91, Pra92],
- first the "pure" and then the value passing version,

into which I have incorporated two ideas from the Dutch work on process algebra, like ACP as presented in [BW90]:

- sequential composition, as opposed to prefixing; with the consequence of having two different actionless processes $\mathbf{o}$ and $\imath$, and
- parallel composition is split up into the three operators full merge, left merge, and communication merge; which facilitates axiomatisation.

Parallel composition, which is still present in the equational reasoning, has in the calculus itself been replaced by a *fork* operator, which "forks off" a background process.

[*] *Author's address:* Department of Computer Sciences, Chalmers University of Technology, S-412 96 Gothenburg, Sweden. *E-mail:* `martinw@cs.chalmers.se`

[†] This work was supported by the Esprit Basic Research Action CONCUR2

1.1 What is broadcasting?

In broadcasting communication as presented in [Pra91, Pra92], sending is autonomous and receiving is controlled. Any participant that is ready to speak can do so, while the listener(s) (of which there may be zero, one or many) have to wait until they hear something. The symmetric synchronisation of the handshake model is replaced by an asymmetric synchronisation: the listener is dependent on the speaker, but not the other way round.

The broadcasting model is based on local area networks as well as human speech, and gives rise to a number of interesting programming examples [Pra93].

1.2 Why prefer sequential composition over prefixing?

It is natural to think of events happening sequentially in time. You can express the notion of "repeating a process several times", or even "forever". Furthermore, it emphasizes the role of simple one-action-processes as the building blocks of bigger ones, thus enabling bottom-up process construction.

ACP and related calculi with sequential composition offer several models and axiom systems which are complete for those models. The left merge operator can helpturning an infinite axiom system into a finite one. But their communication primitive seems is predominantly handshake, due to the strong influence of CCS. Applying the ideas of ACP to broadcasting communication instead of handshaking has not been studied yet to my knowledge. This is the purpose of introducing ACBS.

The reader is assumed to be familiar with CCS-like calculi.

2 Discussion of the language constructs

The language presented here consists of a few, simple constructions. Several extensions with more operators and other concepts are possible - some of them face exactly the same problems as existing calculi, others pose quite unexpected new questions. A few of those will be discussed in section 5.

2.1 Simple Actions

The building blocks of processes are simple actions. There are two kinds of actions: send $a!$ and receive $a?$, where a is the actual signal being transmitted. Their meaning is the same as in CBS.

Unlike CBS, an action here is itself a process: The process $a!$ broadcasts the signal a into the environment, and then it terminates. It will ignore any incoming signals, and it does not depend on any other process(es) being ready to receive the signal. Sending is autonomous. The process $a?$ waits until it hears the signal a being broadcast, and then terminates. It cannot ignore a when it comes, but it will ignore anything else. Receiving is dependent on sending.

With value passing, the process $x?$, which can receive *any* value (represented by the variable x), is different from the process $a?$, receiving a particular signal a.

The special signal τ represents noise, a "signal" that carries no information. It is different from the other signals in that it cannot be received.

2.2 Choice

$P + Q$ can behave either as P, or as Q. *Eg*, the process $a! + b?$ can either send an a or receive a b sent from some other process.

2.3 Empty processes

In contrast to prefix-based calculi like CCS and CBS, the use of sequential composition gives rise to *two* empty processes:

2.3.1 Termination

$\mathbf{1}$ does no action, but terminates successfully. For any process P, $\mathbf{1} \cdot P$ is the same as P. It is called ε in [BW90].

2.3.2 Inaction

$\mathbf{0}$ does nothing at all, and prevents anything else from happening afterwards. For any process P, $\mathbf{0} \cdot P$ is the same as $\mathbf{0}$. Is called δ or *deadlock* in [BW90].

2.4 Sequential Composition

$P \cdot Q$ stands for "first P, then Q". If and when P has successfully terminated, Q can happen – like ";" in imperative programming languages.

Note that the condition "P has terminated" is not the same "P can do no more actions". The latter is also true if P just aborts or deadlocks, in which case Q will never start.

In connection with the concept of "forking" processes, the meaning of "termination" itself changes: Process P in $P \cdot Q$ may have started some background process which is still alive when Q is allowed to start executing. Q will then run in parallel with what remains of process P. Instead of "termination" one should better talk about "passing control" from P to Q.

2.5 Parallel Composition. Fork

$P \| Q$ represents the two processes P and Q running in parallel. They can communicate via broadcast. Any of several processes in parallel that is ready to speak can do so, autonomously, while those processes that are ready to listen hear the message, and all others ignore it. $P \| Q$ terminates if and when both P and Q have terminated.

This calculus uses a variant of parallel, the *fork* operator.

$P \& Q$ is the process which first *forks* off process P, and then starts Q, from where on they both run in parallel, possibly communicating with each other. $P \& Q$ behaves exactly as $P \| Q$, except for the passing of control to a process R sequentially composed after it: Whereas in $(P \| Q) \cdot R$ both P and Q have to terminate to let R start, in $(P\&Q)\cdot R$ the process R waits only for termination of Q. The latter expression is equivalent to $P\&(Q\cdot R)$.

The symbol & is chosen because of its use in UNIX shell scripts, where it has the same meaning. It can be defined either as a unary or a binary operator. Both of them are interdefinable in the presence of the other operators of this calculus: $P\&$ can be defined as $P\&\mathrm{1}$, and $P\&Q$ as $(P\&)\cdot Q$.

3 Pure ACBS&: The Syntax and Operational Semantics

3.1 Syntax

Process expressions in "pure" ACBS&, *ie* without value passing, are built using the following syntax:

$$Act ::= a? \Big| a! \Big| \tau!$$

$$P ::= \mathbf{o} \Big| \mathrm{1} \Big| Act \Big| P+P \Big| P\cdot P \Big| P\&P$$

The set of all process expressions is called $\mathcal{P}$.

Let $\mathcal{D}$ be a set of data values used as "signals", and $\mathcal{D}_\tau = \mathcal{D}\cup\{\tau\}$. a, b range over signals, including τ where appropriate.

An action consists of a signal together with a punctuation symbol: $a!$ "send a", $a?$ "receive a", or a: "ignore a". The latter is only part of the semantics, given below, it does not appear in the syntax. It is defined in such a way that "P ignores a" *iff* "P does not receive a".

In the following, $\sharp$ stands for !, ?, or :, $\natural$ stands for ! or ?, $\flat$ stands for ? or :.

3.2 Operational Semantics

The operational semantics of ACBS is given by a labelled transition relation on process expressions. There are two kinds of transitions:

$P \xrightarrow{a\natural} P'$ is to be read as "process P does action $a\natural$ and becomes P'".

$P \xrightarrow{\surd} P'$ is to be read as "process P passes on control to the next process in sequential composition, with a "residual" process P' remaining in the background." There is an important special case where P' is an inactive process, like $\mathbf{o}$. A transition like $P \xrightarrow{\surd} \mathbf{o}$ can be read simply as "P terminates".

The semantics is presented in Table 1.

4 Bisimulation and Axiomatisation

Our goal is to find a set of rules for equational reasoning about the calculus, which characterise the desired equivalence between processes, the bisimularity (bisimulation equivalence).

First we define bisimulation via the operational semantics, then we present a set of rules which are sound and complete w r t bisimulation.

<table>
<tr><th>Operator</th><th>Send</th><th>Receive</th><th>Ignore[1]</th><th>Pass Control</th></tr>
<tr><td>1, Termination</td><td></td><td></td><td></td><td>$1 \xrightarrow{\surd} \mathbf{o}$</td></tr>
<tr><td>$\mathbf{o}$, Inaction</td><td></td><td></td><td>$\mathbf{o} \xrightarrow{a:}$</td><td></td></tr>
<tr><td>?, Listen</td><td></td><td>$a? \xrightarrow{a?} 1$</td><td>$a? \xrightarrow{b:}$ (if $a \neq b$)</td><td></td></tr>
<tr><td>!, Send</td><td>$a! \xrightarrow{a!} 1$</td><td></td><td>$a! \xrightarrow{b:}$</td><td></td></tr>
<tr><td>+, Choice
($i \in \{1,2\}$)</td><td colspan="2">$\dfrac{P_i \xrightarrow{a\natural} P'}{P_1 + P_2 \xrightarrow{a\natural} P'}$</td><td>$\dfrac{P_1 \xrightarrow{a:} \quad P_2 \xrightarrow{a:}}{P_1 + P_2 \xrightarrow{a:}}$</td><td>$\dfrac{P_i \xrightarrow{\surd} P'}{P_1 + P_2 \xrightarrow{\surd} P'}$</td></tr>
<tr><td>$\cdot$, Sequential Composition</td><td colspan="3">$\dfrac{P \xrightarrow{a\sharp} P'}{P \cdot Q \xrightarrow{a\sharp} P' \cdot Q}$</td><td></td></tr>
<tr><td>”</td><td colspan="4">$\dfrac{P \xrightarrow{\surd} P' \quad P'\&Q \xrightarrow{\alpha} R}{P \cdot Q \xrightarrow{\alpha} R}$</td></tr>
<tr><td>&, fork[2]</td><td colspan="3">$\dfrac{P \xrightarrow{a\sharp_1} P' \quad Q \xrightarrow{a\sharp_2} Q'}{P\&Q \xrightarrow{a(\sharp_1 \bullet \sharp_2)} P'\&Q'}$ $(\sharp_1 \bullet \sharp_2 \neq \bot)$ $\begin{array}{c|ccc} \bullet & ! & ? & : \\ \hline ! & \bot & ! & ! \\ ? & ! & ? & ? \\ : & ! & ? & : \end{array}$</td><td></td></tr>
<tr><td>”</td><td colspan="2">$\dfrac{P \xrightarrow{\surd} P' \quad P'\&Q \xrightarrow{a\natural} R}{P\&Q \xrightarrow{a\natural} R}$</td><td>$\dfrac{P \xrightarrow{\surd} P' \quad P'\&Q \xrightarrow{a:}}{P\&Q \xrightarrow{a:}}$</td><td></td></tr>
<tr><td>”</td><td colspan="3"></td><td>$\dfrac{Q \xrightarrow{\surd} Q'}{P\&Q \xrightarrow{\surd} P\&Q'}$</td></tr>
</table>

[1] $P \xrightarrow{a:}$ is an abbreviation for $P \xrightarrow{a:} P$

[2] $\bot$ means “undefined” in the communication algebra (right) for $\bullet$

Table 1: Transition rules for the “pure” calculus

4.1 Strong Bisimulation

(Bi)simulation is defined as usual.[1]

Definition 1 *[Simulation, Bisimulation]*
$\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ *is a simulation if* $\mathcal{R}^\top \cdot \xrightarrow{a\sharp} \subseteq \xrightarrow{a\sharp} \cdot \mathcal{R}^\top$ *for all* $a\sharp$.
$\mathcal{R}$ *is a bisimulation if both* $\mathcal{R}$ *and* $\mathcal{R}^\top$ *are simulations.*

As in CCS/CBS, the largest bisimulation[2] is an equivalence, denoted by $\sim$.

Proposition 2 $\sim$ *is a congruence.*

4.2 Laws

The following laws make use of some auxiliary operators, which are not mentioned in the syntax given above. The purpose of these operators is not to build processes, they only appear in the reasoning about processes.

The process $P\&Q$ is identical to $P\|Q$, except for their termination behavior: $P\&Q$ terminates if and when Q does. Termination of the "background process" P has no effect at all. Therefore we have the equality:

$$P\&Q = (P \cdot \mathbf{o})\|Q$$

The three operators full merge $\|$, left merge $\lfloor\!\lfloor$, and communication merge $\#$ are taken from ACP [BW90]. We use the symbol $\#$ for communication merge; in ACP it is written $|$.

$P\|Q$ has in general three possibilities for its first action: Either P can do an action (which Q ignores), or Q does an action (which P ignores), or both of them engage in a communication action between them. These three possibilities are reflected by the important equality:

$$P\|Q = P\lfloor\!\lfloor Q + Q\lfloor\!\lfloor P + P\#Q$$

$P \# Q$ must start with a communication action between P and Q. $P\lfloor\!\lfloor Q$ must start with an action from P, which Q ignores. If Q queries for a signal a, then it *must* listen to it when it comes - it cannot ignore it. Thus, the process $P\lfloor\!\lfloor Q$ *cannot* do an a-action if Q queries for a.

The actions of $P\lfloor\!\lfloor Q$ are therefore restricted by the "query set" of process Q, *ie* the set of all labels a such that Q can start with an $a?$-transition, written as $Q^? = \{a \colon \exists Q'.\, Q \xrightarrow{a?} Q'\}$.

The restricted action $a\natural \setminus Q^?$ is the same as $a\natural$ *iff* a is not in the set $Q^?$; otherwise it is equal to $\mathbf{o}$.

The laws for strong bisimulation are given in Table 2.

---

[1] I prefer the concise formulation $\mathcal{R}^\top \cdot \xrightarrow{a\sharp} \subseteq \xrightarrow{a\sharp} \cdot \mathcal{R}^\top$, or, using the relational division operator, $\mathcal{R}^\top \subseteq \xrightarrow{a\sharp} \cdot \mathcal{R}^\top / \xrightarrow{a\sharp}$, over the usual one, which is equivalent:
"whenever $P\mathcal{R}Q$, if $P \xrightarrow{a\sharp} P'$ then $\exists Q'$ such that $Q \xrightarrow{a\sharp} Q'$ and $P'\mathcal{R}Q'$ ".
$\mathcal{R}^\top$ is the converse (or "transpose") of $\mathcal{R}$.

[2] "Bisimulation" in this paper always means "strong", unless explicitly stated that it is "weak".

$(+\textsc{com})$	$P+Q$	$=$	$Q+P$
$(+\textsc{ass})$	$(P+Q)+R$	$=$	$P+(Q+R)$
$(+\textsc{idp})$	$P+P$	$=$	P
$(+\mathbf{o})$	$P+\mathbf{o}$	$=$	P
$(\mathbf{o}\,\cdot)$	$\mathbf{o}\cdot P$	$=$	$\mathbf{o}$
$(\mathrm{1}\,\cdot)$	$\mathrm{1}\cdot P$	$=$	P
$(\cdot\,\textsc{ass})$	$(P\cdot Q)\cdot R$	$=$	$P\cdot(Q\cdot R)$
$(+\,\cdot)$	$(P_1+P_2)\cdot Q$	$=$	$P_1\cdot Q+P_2\cdot Q$
(*but:*	$P\cdot(Q_1+Q_2)$	$\neq$	$P\cdot Q_1+P\cdot Q_2$!)
$(\&\|)$	$P\&Q$	$=$	$(P\cdot\mathbf{o})\|Q$
$(\textsc{exp})$	$P\|Q$	$=$	$P\mathbin{\lfloor\!\lfloor}Q+Q\mathbin{\lfloor\!\lfloor}P+P\mathbin{\#}Q$
$(\mathbf{o}\mathbin{\lfloor\!\lfloor})$	$\mathbf{o}\mathbin{\lfloor\!\lfloor}Q$	$=$	$\mathbf{o}$
$(+\mathbin{\lfloor\!\lfloor})$	$(P_1+P_2)\mathbin{\lfloor\!\lfloor}Q$	$=$	$P_1\mathbin{\lfloor\!\lfloor}Q+P_2\mathbin{\lfloor\!\lfloor}Q$
$(\natural\mathbin{\lfloor\!\lfloor})$	$a\natural\mathbin{\lfloor\!\lfloor}Q$	$=$	$a\natural\setminus Q^?\cdot Q$
$(\cdot\mathbin{\lfloor\!\lfloor})$	$(a\natural\cdot P)\mathbin{\lfloor\!\lfloor}Q$	$=$	$a\natural\setminus Q^?\cdot(P\|Q)$
$(\#\textsc{com})$	$P\mathbin{\#}Q$	$=$	$Q\mathbin{\#}P$
$(\mathbf{o}\#)$	$\mathbf{o}\mathbin{\#}Q$	$=$	$\mathbf{o}$
$(+\#)$	$(P_1+P_2)\mathbin{\#}Q$	$=$	$P_1\mathbin{\#}Q+P_2\mathbin{\#}Q$
$(\natural\,\#)$	$a\natural\mathbin{\#}(b\natural'\cdot Q)$	$=$	$(a\natural\mathbin{\#}b\natural')\cdot Q$
$(\cdot\,\#)$	$(a\natural\cdot P)\mathbin{\#}(b\natural'\cdot Q)$	$=$	$(a\natural\mathbin{\#}b\natural')\cdot(P\|Q)$
$(!\#!)$	$a!\mathbin{\#}a!$	$=$	$\mathbf{o}$
$(!\#?)$	$a!\mathbin{\#}a?$	$=$	$a!$
$(?\#?)$	$a?\mathbin{\#}a?$	$=$	$a?$
$(\#\neq)$	$a\natural\mathbin{\#}b\natural'$	$=$	$\mathbf{o}$, if $a\neq b$
$(\setminus\in)$	$a\natural\setminus Q^?$	$=$	$\mathbf{o}$, if $a\in Q^?$
$(\setminus\notin)$	$a\natural\setminus Q^?$	$=$	$a\natural$, if $a\notin Q^?$
$(\mathbf{o}^?)$	$\mathbf{o}^?$	$=$	$\emptyset$
$(!\,^?)$	$(a!)^?$	$=$	$\emptyset$
$(?\,^?)$	$(a?)^?$	$=$	$\{a\}$
$(+\,^?)$	$(P+Q)^?$	$=$	$P^?\cup Q^?$
$(\cdot\,^?)$	$(P\cdot Q)^?$	$=$	$P^?$

Table 2: Laws for strong bisimulation

Let

$$\begin{aligned}\mathcal{L}aws \;=\; \{\; & (+\textsc{com}), (+\textsc{ass}), (+\textsc{idp}), (+\mathbf{o}), (\mathbf{o}\,\cdot\,), (1\,\cdot\,), (+\,\cdot\,), (\,\cdot\,\textsc{ass}), (\&\|), (\textsc{exp}),\\ & (\mathbf{o}\lfloor\!\lfloor), (+\lfloor\!\lfloor), (\natural\lfloor\!\lfloor), (\,\cdot\,\lfloor\!\lfloor), (\#\textsc{com}), (\mathbf{o}\#), (+\#), (\natural\,\#), (\,\cdot\,\#),\\ & (!\#!), (!\#?), (?\#?), (\#\neq), (\backslash\in), (\backslash\notin), (\mathbf{o}^?), (!\,^?), (?\,^?), (+^?), (\,\cdot\,^?)\;\}\end{aligned}$$

Proposition 3 *All of the laws are sound w r t $\sim$.*

To prove *eg* that the law $(\,\cdot\,\textsc{ass})$ is sound w r t $\sim$, I show that the relation

$$\mathcal{R} \stackrel{def}{=} \{\langle (P\cdot Q)\cdot R, P\cdot(Q\cdot R)\rangle \colon P, Q, R \in \mathcal{P}\} \cup Id_{\mathcal{P}}$$

is a bisimulation.

(The relation

$$\mathcal{R}' \stackrel{def}{=} \{\langle (P\cdot Q)\cdot R, P\cdot(Q\cdot R)\rangle \colon P, Q, R \in \mathcal{P}\}$$

is not enough: For some P, Q and R, both $(P\cdot Q)\cdot R$ and $P\cdot(Q\cdot R)$ may have a transition to the *same* process, rather than to two processes which are in the relation $\mathcal{R}'$.)

The soundness proofs can be machine checked. So far, I have translated the syntax and semantics of the calculus into the logical framework ALF [MN94], and have proved the soundness of laws like $1\cdot P \sim \mathbf{o}\&P$ and $\mathbf{o}\&P \sim P$.

4.3 Basic Terms

A process expression is called a *basic term* if it is in the following form:

$$B = \sum_i a_i\natural_i + \sum_j a_j\natural_j \cdot B_j$$

where the B_j themselves are basic terms. An empty sum stands for $\mathbf{o}$.

4.4 Completeness

The above laws can be turned into a Term Rewriting System, by reading all the equal signs as arrows from left to right. That TRS reduces every process to a basic term:

Proposition 4 *For every process $P \in \mathcal{P}$, there is a basic term $B \in \mathcal{P}$, such that*

$$\mathcal{L}aws \vdash P = B$$

The proof works by induction over the structure of basic terms.

Basic terms that are bisimilar must have the same (*ie*, bisimilar) summands (modulo commutativity, associativity, and idempotence of $+$ and occurrence of $\mathbf{o}$).

Let

$$\mathcal{L}aws^+ = \{(+\textsc{com}), (+\textsc{ass}), (+\textsc{idp}), (+\mathbf{o})\}$$

Proposition 5 *For all basic terms $B_1, B_2 \in \mathcal{P}$: if $B_1 \sim B_2$ then*

$$\mathcal{L}aws^{+} \vdash B_1 = B_2$$

Thus,

Proposition 6 *The set $\mathcal{L}aws$ is complete w r t bisimulation.*

4.5 More laws

Some other interesting equalities, which are not needed for the completeness result, are stated below.

$$\begin{array}{rrcl} (\|\mathbf{o}) & P\|\mathbf{o} & = & P \cdot \mathbf{o} \\ (\|\textsc{com}) & P\|Q & = & Q\|P \\ (\|\textsc{ass}) & (P\|Q)\|R & = & P\|(Q\|R) \end{array}$$

4.6 Expansion Theorem

CCS-like calculi usually have an axiomatisation including an "Expansion Theorem", which is really not an axiom but an axiom schema for infinitely many axioms. Note that the list above does not contain any explicit Expansion Theorem. Instead it has the equality ($\textbf{EXP}$), which together with the distributivity laws for $\lfloor\!\lfloor$ and $\#$ plays a similar role, thus making a *finite* axiomatisation possible. Faron Moller showed in his thesis [Mol89], that the $\lfloor\!\lfloor$ operator is necessary for finite axiomatisation.

5 Extensions

5.1 Recursion

So far, all processes have finite transition trees. There are various ways the calculus can be extended to include some form of recursion and thus processes that have infinite transition trees.

The most general form of recursion allows a process to contain itself within any operator, *eg*

$$X = a! + b! \cdot X \cdot c!$$

$$Y = a! \cdot (Y \| b!)$$

Both processes X and Y have infinitely many *different* states in their transition trees.

If however recursion is restricted such that the recursive appearance of the process is not allowed within a parallel, and only on the right side of a sequential, then the process can only reach a finite number of different states.

The first form of recursion therefore appears to be more expressive. This difference in expressivity reminds of the one between context-free (Chomsky 2) and regular (Chomsky 3) languages.

5.1.1 Loop

A simple form of recursion is made possible by the sequential composition, P^ω, "repeat P forever". It has the semantics rule:

$$\frac{P \xrightarrow{a\sharp} P'}{P^\omega \xrightarrow{a\sharp} P' \cdot P^\omega}$$

and the following laws:

$$\begin{array}{rrcl} (\textsc{unfold}) & P^\omega & = & P \cdot P^\omega \\ (\textsc{fold}) & \text{if } X & = & P \cdot X, \text{ then } X = P^\omega \end{array}$$

5.1.2 Loop and Fork

Suppose we want to define a generator process Gen which, every time it receives a signal gen, creates a new copy of some process P to run in parallel with Gen itself. Gen can be written recursively as: $Gen = gen? \cdot (P \| Gen)$. Using the (unary or binary) fork operator, this is equivalent to $Gen = gen? \cdot (P \& Gen)$ or $Gen = gen? \cdot P\& \cdot Gen$. Thus it can be rewritten with a loop as $Gen = (gen? \cdot P\&)^\omega$. Thus, the loop operator as a "restricted" form of recursion is not so restricted after all. It is not clear whether this rewriting offers any advantage or only shifts the complexity to a different operator.

6 Discussion: Value Passing and Environments

6.1 Why environments? Prefixing vs. Sequential Composition

We extend the calculus with value passing.

Process expressions can be built with the following actions:

$$Act ::= \tau! \;\Big|\; e! \;\Big|\; x?$$

where x stands for a variable, and e for any expression, possibly containing variables, yielding a result. The transition system defined in the operational semantics, however, is still labelled with single actions, not with variables or expressions.

In a prefixing calculus, processes can be defined as:

$$P ::= \mathbf{o} \;\Big|\; Act.\, P \;\Big|\; P + P \;\Big|\; P \| P$$

Action $x?$ *binds* the variable name x, which may occur free in e. In $x!\,.\,P$, x is free, in $x?\,.\,x!\,.\,P$ it is bound.

With sequential composition, processes look like this:

$$P ::= \mathbf{o} \;\Big|\; Act \;\Big|\; P \cdot P \;\Big|\; P + P \;\Big|\; P \| P$$

$x?$ cannot be seen as a variable binding here, if we want $\cdot$ to be associative: The scope of x in $(x? \cdot P) \cdot Q$ would be P, but in $x? \cdot (P \cdot Q)$ it would be $P \cdot Q$.

Now we want to give meaning to processes like $x? \cdot x!$, by separately interpreting its parts $x?$ and $x!$ and combining them. In a prefixing calculus like CBS one can express value passing with a rule

$$x?\,.\,P \xrightarrow{v?} P[v/x]$$

(note that the two terms on both sides of the arrow have no free variables, though P itself does!). We have no directly corresponding rule here. If we try to define

$$x? \xrightarrow{v?} 1$$

we find that the information about the actually transmitted value v is lost. The prefixing calculus does not need to give any semantics to expressions with free variables, but we have to. By introducing *environments*, assignments of variable names to values, the information can be conveyed from one process to its successor. Receiving a value changes the environment:

$$\langle x?\,, \rho\rangle \xrightarrow{v?} \langle 1, \rho[v/x]\rangle$$

Sending a value uses the information stored in the environment:

$$\langle e!\,, \rho\rangle \xrightarrow{(\llbracket e \rrbracket \rho)!} \langle 1, \rho\rangle$$

The operational semantics is no longer a transition system between processes, but between processes augmented with environments:

$$\rightarrow \;\subseteq\; (\bar{\mathcal{P}} \times \mathcal{E}nv) \times Act \times (\bar{\mathcal{P}} \times \mathcal{E}nv)$$

Defining an operational semantics is more difficult, and also more imperative-style. There is no value passing in ACP as presented in [BW90].

6.2 Environments of parallel processes

Each process running in parallel has its own environment, as given by the rule:

$$\frac{\langle P, \rho\rangle \xrightarrow{a\sharp_1} \langle P', \rho'\rangle \quad \langle Q, \sigma\rangle \xrightarrow{a\sharp_2} \langle Q', \sigma'\rangle}{\langle P, \rho\rangle \| \langle Q, \sigma\rangle \xrightarrow{a\sharp_1 \bullet \sharp_2} \langle P', \rho'\rangle \| \langle Q', \sigma'\rangle}$$

This approach makes the variables local to each parallel component. It works fine for static systems, where the number of parallel processes stays constant all the time.

If we want to have dynamic systems, where processes can split into two, or join into one, we need some more rules.

6.3 Splitting processes

A process splits into two in the expression $P \cdot (Q \| R)$. Both Q and R start with the same environment that P ended up with. This will be made formal later, using augmented processes and configurations.

6.4 Joining processes

In $(P \| Q) \cdot R$, two processes join into one. When P and Q terminate with different environments ρ and σ, which environment should R inherit? (P and Q could for example perform two parts of a computation in parallel, the final result of which shall then be used by R.)

There are several possibilities.

6.4.1 Ranking

Out of several parallel components, one is declared as "main", and its environment will be inherited.

6.4.2 Random

R inherits either ρ or σ by non-deterministic choice.

6.4.3 Combine

ρ and σ are somehow merged into one environment $\rho \oplus \sigma$. A restriction on P and Q could be introduced that shared variables must be read-only.

6.4.4 Agree

P and Q both give a result; and the results given by both must agree. (*cf* LOTOS).

6.4.5 Nothing

R gets no information out of P and Q. The scope of any variables occurring in P and Q is limited to these.

6.4.6 Disallow

Joining of processes if not allowed. The first operand of a sequential composition must not contain any $\|$.

6.5 Fork

The problem with the joining processes is partly due to the symmetry of the parallel operator. In $(P \| Q) \cdot R$, R has to wait for both P and Q to terminate, and we don't know from which of them R should inherit the environment. In contrast, in $(P \& Q) \cdot R$, R waits only for Q to terminate, so it seems natural to let R inherit the environment of Q.

7 ACBS& with Value Passing: The Syntax and Operational Semantics

After all these theoretic considerations, here is the calculus with value passing.

7.1 Syntax

ACBS& process expressions are built using the following syntax:

$$P ::= \mathbf{o} \;\Big|\; 1 \;\Big|\; x? \;\Big|\; e! \;\Big|\; \tau! \;\Big|\; P + P \;\Big|\; P \cdot P \;\Big|\; P \& P$$

7.2 Operational Semantics

For the operational semantics with value passing we need some auxiliary definitions.

Processes are *augmented* with *environments* in certain positions.

The syntax for augmented processes is given by:

$$\bar{P} ::= \mathbf{o} \;\Big|\; 1 \;\Big|\; x? \;\Big|\; e! \;\Big|\; \tau! \;\Big|\; \bar{P} + \bar{P} \;\Big|\; \bar{P} \cdot P \;\Big|\; \bar{P}, \rho \& \bar{P}$$

where P stands for a process, $\bar{P}$ for an augmented process, and ρ for an environment. Note that the second component of a sequential composition is *not* augmented with an environment - it will inherit the one from its predecessor when that passes over control.

We will call a pair consisting of an *augmented process* and an *environment*, such as $\bar{P}, \rho$ (dropping the pairing parentheses), a *configuration*.

Augmenting a process P with an *environment* ρ is defined as follows:

$$\begin{array}{lcl}
\mathbf{o}^\rho & = & \mathbf{o} \\
1^\rho & = & 1 \\
x?^\rho & = & x? \\
e!^\rho & = & e! \\
(P_1 + P_2)^\rho & = & P_1{}^\rho + P_2{}^\rho \\
(P \cdot Q)^\rho & = & P^\rho \cdot Q \\
(P \& Q)^\rho & = & P^\rho, \rho \& Q^\rho
\end{array}$$

The operational semantics of ACBS with value passing is given by a labelled transition relation between configurations; so the semantics associated to a process P starting in an environment ρ will be the transition system of the configuration P^ρ, ρ.

The semantics is presented in Table 3. $\tilde{P}$, $\tilde{P}'$, $\tilde{Q}$... range over configurations.

8 Denotational semantics. Implementation

The calculus can also be given a denotational semantics, where the process domain of transition trees is defined with the help of, *eg*, complete partial orders [Abr91] or metric spaces [vB93]. I have implemented the calculus (with value passing) in **LML**, a lazy functional language, by translating process terms into transition trees which I represent as lazy infinite data structures. In ALF [MN94] I have done another implementation, so far only the "pure" calculus, and done some machine-assisted correctness proofs.

Operator	Send	Receive	Ignore[1]	Pass Control
1, Termination				$1,\rho \xrightarrow{\surd} \mathbf{o},\rho$
$\mathbf{o}$, Inaction			$\mathbf{o},\rho \xrightarrow{v:}$	
?, Listen		$x?\,,\rho \xrightarrow{v?} 1,\rho[v/x]$	$x?\,,\rho \xrightarrow{\tau:}$	
!, Send	$e!\,,\rho \xrightarrow{[\![e]\!]\rho!} 1,\rho$		$e!\,,\rho \xrightarrow{v:}$	
$+$, Choice ($i \in \{1,2\}$)	$\dfrac{P_i,\rho \xrightarrow{a\natural} \tilde{P}'}{P_1+P_2,\rho \xrightarrow{a\natural} \tilde{P}'}$		$\dfrac{P_1,\rho \xrightarrow{v:} \quad P_2,\rho \xrightarrow{v:}}{P_1+P_2,\rho \xrightarrow{v:}}$	$\dfrac{P_i,\rho \xrightarrow{\surd} \tilde{P}'}{P_1+P_2,\rho \xrightarrow{\surd} \tilde{P}'}$
$\cdot$, Sequential Composition	$\dfrac{P,\rho \xrightarrow{a\sharp} P',\rho'}{P\cdot Q,\rho \xrightarrow{a\sharp} P'\cdot Q,\rho'}$			
„	$\dfrac{P,\rho \xrightarrow{\surd} \tilde{P}' \quad \tilde{P}'\&Q^{\rho},\rho \xrightarrow{\alpha} \tilde{R}}{P\cdot Q,\rho \xrightarrow{\alpha} \tilde{R}}$			
&, fork[2]	$\dfrac{\tilde{P} \xrightarrow{a\sharp_1} \tilde{P}' \quad \tilde{Q} \xrightarrow{a\sharp_2} \tilde{Q}'}{\tilde{P}\&\tilde{Q} \xrightarrow{a(\sharp_1\bullet\sharp_2)} \tilde{P}'\&\tilde{Q}'}\ (\sharp_1 \bullet \sharp_2 \neq \bot)$ $\quad \begin{array}{cccc} \bullet & ! & ? & : \\ \hline ! & \bot & ! & ! \\ ? & ! & ? & ? \\ : & ! & ? & : \end{array}$			
„	$\dfrac{\tilde{P} \xrightarrow{\surd} \tilde{P}' \quad \tilde{P}'\&\tilde{Q} \xrightarrow{a\natural} \tilde{R}}{\tilde{P}\&\tilde{Q} \xrightarrow{a\natural} \tilde{R}}$		$\dfrac{\tilde{P} \xrightarrow{\surd} \tilde{P}' \quad \tilde{P}'\&\tilde{Q} \xrightarrow{v:}}{\tilde{P}\&\tilde{Q} \xrightarrow{v:}}$	
„				$\dfrac{\tilde{Q} \xrightarrow{\surd} \tilde{Q}'}{\tilde{P}\&\tilde{Q} \xrightarrow{\surd} \tilde{P}\&\tilde{Q}'}$

[1] $P,\rho \xrightarrow{v:}$ is an abbreviation for $P,\rho \xrightarrow{v:} P,\rho$

[2] $\bot$ means "undefined" in the communication algebra (right) for $\bullet$

Table 3: Transition rules for the value passing calculus

9 Conclusions. Where to go on

9.1 Broadcast vs. Handshake

The work so far shows that broadcast communication can be realised in a sequential setting just as well as handshake communication. Nothing in ACP forces it to use handshaking. The difficulties encountered with value passing in a sequential setting are independent of the underlying communication model.

9.2 Sequential vs. Prefixing

The idea of sequential composition, a more "symmetric", homogeneous operator than prefixing, seems appealing. As long as only the pure calculus is considered, sequential composition works just as well as prefixing. With value passing, however, things start getting difficult, without any obvious advantage of sequential composition to be seen.

The problem of passing values in some context with sequential and parallel composition is however quite a common one and appears in many different formalisms, such as UNIX shell scripts, LOTOS, "Fork Calculus" [Hav92], Concurrent ML, OCCAM, μCRL [Gro91]. The concept of forking seems promising towards a solution.

9.3 Other

A question not yet treated here is how to define a suitable *weak bisimulation* and which properties it will have. We are also looking for more motivating examples. Space didn't permit me to put in here any of the examples I have, unfortunately.

10 Acknowledgements

Special thanks to my supervisor K.V.S. Prasad, without whom this work would not have been possible, to the concurrency group at Chalmers, and Thorsten Altenkirch for discussion and encouragement.

References

[Abr91] S. Abramsky. A domain equation for bisimulation. *Information and Computation*, 92(2):161–218, June 1991.

[BB93] J. C. M. Baeten and J. A. Bergstra. On sequential composition, action prefixes and process prefix. Technical Report P9305, University of Amsterdam, March 1993.

[BBK86] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. Technical Report CS-R8632, Centre for Mathematics and Computer Science, Amsterdam, 1986.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Tracts in Theoretical Computer Science*. Cambridge Univ. Press, 1990.

[CHM93] Søren Christensen, Yoram Hirshfeld, and Faron Moller. Bisimulation equivalence is decidable for all basic parallel processes. In *CONCUR-93*, Hildesheim, 1993.

[Gro91] Jan Friso Groote. *Process Algebra and Structured Operational Semantics*. PhD thesis, Universiteit van Amsterdam, November 1991.

[Hav92] Klaus Havelund. The fork calculus, 1992. presented at the Workshop in Bergen, fall 1992.

[HL92] Matthew Hennessy and Huimin Lin. Symbolic bisimulations. Computer Science Internal Report 1/92, University of Sussex, April 1992.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science. Springer-Verlag, 1980.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[Mol89] Faron Moller. *Axioms for Concurrency*. PhD thesis, University of Edinburgh, October 1989.

[Pra91] K. V. S. Prasad. A calculus of broadcasting systems. In *TAPSOFT'91 Volume 1: CAAP*, April 1991. Springer Verlag LNCS 493.

[Pra92] K. V. S. Prasad. A calculus of value broadcasts. Technical Report 69, Department of Computer Science, Chalmers University of Technology, October 1992.

[Pra93] K. V. S. Prasad. Programming with broadcasts. Technical Report 74, Department of Computer Science, Chalmers University of Technology, September 1993.

[vB93] Franck van Breugel. Three metric domains of processes for bisimulation. Technical Report CS-R9335, CWI, Amsterdam, 1993.

[Wei93] Martin Weichert. Algebra of communicating systems. In *Wintermötet Tanum Strand*, January 1993. Draft Proceedings.

[Wei94] Martin Weichert. Value passing and sequential composition. In *Proceedings of La Wintermöte '94 Lundsbrunn*, January 1994.

From Branching to Linear Metric Domains
(and back)*

Franck van Breugel†

McGill University
School of Computer Science
3480 University Street, Montreal H3A 2A7, Canada

Abstract

Besides partial orders, also *metric spaces* have turned out to be very useful to give semantics to programming languages (see, e.g., the collection of papers of the Amsterdam Concurrency Group [BR92]). In the literature, one encounters two main classes of metric domains: *linear domains* and *branching domains*. Linear domains were already studied by topologists in the early twenties. Branching domains have been introduced by, e.g., De Bakker and Zucker [BZ82, BZ83], Golson and Rounds [GR83, Gol84], and the author [Bre93]. The elements of these linear and branching domains are convenient to model—one might even say that they represent—*trace equivalence* classes and *bisimulation equivalence* classes, respectively. The former is a simple observation. The latter has been proved by Van Glabbeek and Rutten [GR89].

Linear domains are more abstract than branching domains. Our aim is to show that linear domains can be embedded in branching domains. We focus on the branching domain $\mathcal{B}$ introduced by De Bakker and Zucker in [BZ83] and the linear domain $\mathcal{L}$ the elements of which can be viewed as nonempty and compact sets of sequences.

By abstracting from the branching structure of the branching domain $\mathcal{B}$ we arrive at the linear domain $\mathcal{L}$. This abstraction operator—called *linearize operator* in the sequel—can be defined conveniently in terms of a *metric labelled transition system*. The theory of metric labelled transition systems has been outlined in the author's [Bre94a] and has been developed further in his thesis [Bre94b]. The branching domain $\mathcal{B}$ can be seen as a labelled transition system as De Bakker,

*This work was carried out during a visit of BRICS, Department of Computer Science, University of Aarhus, Denmark with financial support of BRICS.

†The author is supported by the Netherlands Organization for Scientific Research.

Bergstra, Klop, and Meyer noted in [BBKM84]. It can even be viewed as a metric labelled transition system. This metric labelled transition system is *compactly branching*—being a generalization of finitely branching. As a consequence, we can apply a theorem—being a generalization of a theorem reminiscent to König's lemma—obtaining the linearize operator *lin*. The additional metric structure of a metric labelled transition system (with respect to a labelled transition systems) is essential in the above. Similarly, we can linearize the more involved branching domains—used to model object-oriented and higher-order features—introduced by Rutten [Rut90] and De Bakker and the author [BB93].

Also the linear domain $\mathcal{L}$ can be viewed as a compactly branching metric labelled transition system. This can be done in two obvious ways. Both compactly branching metric labelled transition systems give rise to a *branch operator*: $branch_0$ and $branch_1$.

$$\begin{tikzcd} \mathcal{L} \arrow[r, shift left=2, "branch_0"] \arrow[r, shift right=2, "branch_1"'] & \mathcal{B} \arrow[l, "lin"'] \end{tikzcd}$$

The linearize and branch operators are related as follows:

$$lin \circ branch_0 = id_{\mathcal{L}} \text{ and } lin \circ branch_1 = id_{\mathcal{L}}.$$

We make the relationship between the linearize and branch operators even more precise. We follow the work of Nielsen and Winskel et al. [Win84, SNW93, WN94] using category theory—in particular functors—to classify the domains. The linear and branching domains are both turned into a *quasimetric space* which induces a preorder and hence a category. Lately, there is a growing interest in quasimetric spaces. See, e.g., Wagner's thesis [Wag94] and Flagg and Kopperman's [FK94]. The quasimetrics are obtained from the metrics the d omains are endowed with by dropping one half of the Hausdorff metric. The morphisms of the branching domain can be seen as simulations and the morphisms of the linear domains can be viewed simply as inclusion functions. The linearize operator and both the branch operators are functors. These functors form a reflection and a coreflection—our main result.

$$\begin{tikzcd} \mathcal{L} \arrow[r, hook, shift left=2] \arrow[r, hook, shift right=2] & \mathcal{B} \end{tikzcd}$$

By means of this reflection and coreflection we have expressed that the linear domain $\mathcal{L}$ can be embedded in the branching domain $\mathcal{B}$. These adjunctions can also be used for the transfer of (categorical) techniques from one domain to the other.

The details of the above will appear in [Bre95].

References

[BB93] J.W. de Bakker and F. van Breugel. Topological Models for Higher Order Control Flow. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of the 9th International Conference on Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 122–142, New Orleans, April 1993. Springer-Verlag.

[BBKM84] J.W. de Bakker, J.A. Bergstra, J.W. Klop, and J.-J.Ch. Meyer. Linear Time and Branching Time Semantics for Recursion with Merge. *Theoretical Computer Science*, 34(1/2):135–156, 1984.

[BR92] J.W. de Bakker and J.J.M.M. Rutten, editors. *Ten Years of Concurrency Semantics, selected papers of the Amsterdam Concurrency Group*. World Scientific, Singapore, 1992.

[Bre93] F. van Breugel. Three Metric Domains of Processes for Bisimulation. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of the 9th International Conference on Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 103–121, New Orleans, April 1993. Springer-Verlag.

[Bre94a] F. van Breugel. Generalizing Finiteness Conditions of Labelled Transition Systems. In S. Abiteboul and E. Shamir, editors, *Proceedings of the 21th International Colloquium on Automata, Languages, and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 376–387, Jerusalem, July 1994. Springer-Verlag.

[Bre94b] F. van Breugel. *Topological Models in Comparative Semantics*. PhD thesis, Vrije Universiteit, Amsterdam, September 1994.

[Bre95] F. van Breugel. From Branching to Linear Metric Domains (and back). BRICS report, University of Aarhus, Aarhus, 1995. To appear.

[BZ82] J.W. de Bakker and J.I. Zucker. Processes and the Denotational Semantics of Concurrency. *Information and Control*, 54(1/2):70–120, July/August 1982.

[BZ83] J.W. de Bakker and J.I. Zucker. Compactness in Semantics for Merge and Fair Merge. In E. Clarke and D. Kozen, editors, *Proceedings of 4th Workshop on Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 18–33, Pittsburgh, June 1983. Springer-Verlag.

[FK94] B. Flagg and R. Kopperman. Continuity Spaces: reconciling domains and metric spaces - part I. Draft, May 1994.

[Gol84] W.G. Golson. *Denotational Models based on Synchronous Communicating Processes*. PhD thesis, University of Michigan, Ann Arbor, 1984.

[GR83] W.G. Golson and W.C. Rounds. Connections between Two Theories of Concurrency: metric spaces and synchronization trees. *Information and Control*, 57(2/3):102–124, May/June 1983.

[GR89] R.J. van Glabbeek and J.J.M.M. Rutten. The Processes of De Bakker and Zucker represent Bisimulation Equivalence Classes. In *J.W. de Bakker, 25 jaar semantiek*, pages 243–246. CWI, Amsterdam, 1989.

[Rut90] J.J.M.M. Rutten. Semantic Correctness for a Parallel Object-Oriented Language. *SIAM Journal of Computation*, 19(2):341–383, April 1990.

[SNW93] V. Sassone, M. Nielsen, and G. Winskel. A Classification of Models for Concurrency. In E. Best, editor, *Proceedings of the 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 82–96, Hildesheim, August 1993. Springer-Verlag.

[Wag94] K.R. Wagner. *Solving Recursive Domain Equations with Enriched Categories*. PhD thesis, Carnegie Mellon University, Pittsburgh, June 1994. Draft.

[Win84] G. Winskel. Synchronization Trees. *Theoretical Computer Science*, 34(1/2):33–82, 1984.

[WN94] G. Winskel and M. Nielsen. Models for Concurrency. Number 12 in BRICS Notes Series. University of Aarhus, Aarhus, May 1994. To appear in S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, Oxford University Press, Oxford.

Automatic Synthesis of Real Time Systems *

Jørgen H. Andersen *Kåre J. Kristoffersen* *Kim G. Larsen*
Jesper Niedermann

BRICS †
Department of Math. & Comp. Sc., Aalborg University

Abstract

This paper presents a method for automatically constructing real time systems directly from their specifications. The model–construction problem is considered for implicit specifications of the form:

$$(A_1 \,|\, \ldots \,|\, A_n \,|\, X) \; \mathsf{sat} \; S$$

where S is a real time (logical) specification, $A_1 \ldots A_n$ are given (regular) timed agents and the problem is to decide whether there exists (and if possible exhibit) a real time agent X which when put in parallel with $A_1 \ldots A_n$ will yield a network satisfying S. The method presented proceeds in two steps: first, the implicit specification of X is transformed into an equivalent direct specification of X; second, a model for this direct specification is constructed (if possible) using a direct model construction algorithm. A prototype implementation of our method has been added to the real time verification tool EPSILON.

Introduction

During the last few years the area of real time systems has received a lot of attention from the research community. In particular, a variety of specification formalisms has emerged allowing real time properties to be expressed explicitly. These specification formalisms may roughly be divided into two groups, namely: real time logics (e.g. [RT89, HNSY92]) and real time process algebras (e.g. [Wan90, NRJV90]).

Central to the ongoing research has been the construction of *model–checking* algorithms; i.e. algorithms for deciding whether a given real time system satisfies a given specification. A number of model–checking algorithms exists for real timed logical specifications [ACD90] and more recently algorithms for model–checking [1] timed process algebraic specifications have been given [Cer92, LW93].

*This work has been partially supported by the European Communities under CONCUR2, BRA 7166.

†Basic Research in Computer Science, Centre of the Danish National Research Foundation

[1] In process algebra model–checking consists of checking a suitable behavioural relationship (bisimilarity, say) between the implementation and the specification.

In this work, we deal with the more ambitious goal of *model–construction*: i.e. given a real time specification (logical or process algebraic) we want to automatically synthesize a real time system satisfying the specification (if such a system exists). Moreover, we consider the model–construction problem in the setting of *implicit specifications*, i.e.:

$$(A_1 \mid \ldots \mid A_n \mid X) \ \mathsf{sat}\ S \tag{1}$$

The requirement of (1) represents a certain stage in a top–down development of a network satisfying a given overall specification S: namely, the stage where some components $A_1 \ldots A_n$ have already been constructed, but for the completion of the development one component X remains to be constructed. We call S an implicit specification of X as it specifies the behaviour of X in a certain context.

In this paper we present a method for automatically constructing the component X (if possible) such that (1) is met. Our method is applicable to logical as well as process algebraic specifications and proceeds in two steps: First, the implicit specification S is (effectively) transformed into a *direct* specification S' describing the sufficient and necessary requirement to X in order for (1) to hold; i.e.:

$$X\ \mathsf{sat}\ S' \quad \textit{if and only if} \quad (A_1 \mid \ldots \mid A_n \mid X)\ \mathsf{sat}\ S \tag{2}$$

Second, a real time system satisfying S' is generated (if possible) using a *direct* model–construction algorithm.

Our work can be seen as a real time extension of existing model–constructing algorithms for finite–state systems. For $n = 0$ the model–construction problem for (1) extends classical model–construction methods. For S a process algebraic specification the model–construction problem for (1) is a real time extension of the equation solving problem studied in [LX90b, Shi, Par89, LQ90]. For S a logical specification our work is related to and extends the work on contexts as property transformers studied in [LX91, LS92].

Our method assumes that the network components $A_1 \ldots A_n$ and X are all regular timed agents [Wan90] or equivalently one–clock timed automata [AD90]. For reasons of clarity we have chosen to present our solution method in a somewhat simplified setting:

—**T**he notion of parallel composition used in this paper is simply that of interleaving on actions; however our method extends smoothly to a wide range of existing notions of parallel compositions through parameterization on a so called synchronization function;

—**T**he specification language considered is a timed extension of the well–known Hennessy–Milner Logic [HM85]; however our method extends to a recursive (and very expressive) extension of this logic using already developed and well understood techniques [LX90a, LX91, JLJL93]. Also, our method is applicable to implicit model–construction based on process algebraic specifications by exhibition of suitable characteristic properties.

In the concluding remarks the above suggested extensions will be discussed in more details. Also, a prototype implementation of the implicit model–construction method for the full extensions has been given and is available as part of the EPSILON tool [CGL93].

DELAY TRANSITIONS: $$\langle A, v\rangle \xrightarrow{\epsilon(d)} \langle A, v+d\rangle$$
ACTION TRANSITIONS: $$\langle \sum_i [l_i, u_i].a_i.A_i, v\rangle \xrightarrow{a_i} \langle A_i, 0\rangle \text{ if } v \in [l_i, u_i]$$ $$\frac{\langle A, v\rangle \xrightarrow{a} \langle A', v'\rangle}{\langle N, v\rangle \xrightarrow{a} \langle A', v'\rangle} \text{ if } N \stackrel{def}{=} A$$

Table 1: Operational Semantics for Regular Timed Agents

1 Timed Processes and Timed Logic

Regular Timed Agents

Let $\mathcal{A}$ be a fixed set of actions range over by $a, b, c, \ldots$. We denote by $\mathbf{R}_{>0}$ the set of positive reals ranged over by $d, d_1, d_2, \ldots, d', d'', \ldots$. Similarly $\mathbf{R}$ denotes the set of non–negative reals, $\mathbf{N}$ denotes the set of natural numbers (including 0), and $\mathcal{D}$ denotes the set $\{\epsilon(d) \mid d \in \mathbf{R}_{>0}\}$. Regular timed agents are terms of the following grammar:

$$A \;::=\; \sum_{i=1}^{n} [l_i, u_i].a_i.A_i \;\mid\; N$$

where $l_i, u_i \in \mathbf{N}$, $a_i \in \mathcal{A}$ and N ranges over a finite set of agent identifiers For each agent identifier we assume a defining equation $N \stackrel{def}{=} A$. We shall use *nil* to denote the empty summation, and on occasions we will use the expanded notation $[l_1, u_1].a_1.A_1 + \cdots + [l_n, u_n].a_n.A_n$ for the general summation. Also, we shall omit trailing *nil*'s; hence $[4, 5].a$ denotes the agent $[4, 5].a.nil$. The maximum delay $M(A)$ of an agent A is defined recursively as $M(\sum_{i=1}^{n}[l_i, u_i].a_i.A_i) = \max\{u_i, M(A_i) \mid i = 1 \ldots n\}$, and $M(N) = M(A)$ where $N \stackrel{def}{=} A$.

Intuitively, the term $\sum_{i=1}^{n}[l_i, u_i].a_i.A_i$ describes an agent which is able to perform the action a_i between the time bounds l_i and u_i after which the agent will perform according to A_i. Formally, the semantics of regular timed agents are given in terms of a $\mathcal{A} \cup \mathcal{D}$ labelled transition system, where the *configurations* are pairs of the form $\langle A, v\rangle$, with $v \in \mathbf{R}$ denoting the amount by which the agent A has been delayed. The *transitions* between configurations are either delay– or action–transitions and are given by the rules of Table 1. Thus, we adopt the two–phase functioning priciple [NSY91] present in most real–time process algebras: i.e. the behaviour of a system is regarded as being split in two alternating phases, one where all components agree to let time progress, and one where the components compute.

Delay Transitions:
$$\langle \overline{A}, \overline{v} \rangle \xrightarrow{\epsilon(d)} \langle \overline{A}, \overline{v} + d \rangle$$
Action Transitions:
$$\frac{\langle A_i, v_i \rangle \xrightarrow{a} \langle A_i', 0 \rangle}{\langle A_1 \mid \cdots A_i \cdots \mid A_n, \overline{v} \rangle \xrightarrow{a} \langle A_1 \mid \cdots A_i' \cdots \mid A_n, \overline{v}[v_i := 0] \rangle}$$

Table 2: Operational Semantics for Networks

Timed Networks

Syntactically a timed network agent is a parallel composition of a number of regular timed agents; thus network agents are terms of the following grammar:

$$N \quad ::= \quad (A_1 \mid \ldots \mid A_n)$$

Behaviourally, we shall simply assume that a network agent interleaves components actions, whereas components are required to synchronize with respect to delay. Formally, a *network configuration* is a pair $\langle \overline{A}, \overline{v} \rangle$, where $\overline{A} = A_1 | \ldots | A_n$ is a network and $\overline{v} = (v_1, \ldots, v_n)$ is a delay vector, indicating how much each component of the network has been delayed. The transitions between network configurations are given by Table 2, where for $d \in \mathbf{R}_{>0}$, $\overline{v} + d$ denotes the delay vector $(v_1 + d, \ldots, v_n + d)$, and $\overline{v}[v_i := 0]$ denotes the delay vector obtained by replacing v_i with 0 in the vector $\overline{v}$.

Example 1.1 Consider the network agents $[0,2].a \mid [2,3].b$ and $nil \mid [2,3].b$. The possible configurations involving these two networks are indicated by the two coordinate systems in Figure 1; thus in the left coordinate system the x–axis indicates the delay of $[2,3].b$ and the y–axis gives the delay of $[0,2].a$. Using the inference rules of Tables 1 and 2 we can infer the following transitions from the initial network configuration (see also Figure 1):

$$\begin{aligned} &A = \langle [0,2].a \mid [2,3].b \,, (0,0) \rangle \xrightarrow{\epsilon(1.5)} B = \langle [0,2].a \mid [2,3].b \,, (1.5,1.5) \rangle \xrightarrow{a} \\ &C = \langle nil \mid [2,3].b \,, (0,1.5) \rangle \xrightarrow{\epsilon(1)} D = \langle nil \mid [2,3].b \,, (1,2.5) \rangle \xrightarrow{b} \end{aligned}$$

Timed Logic

The specification language used in this presentation is the Extended Timed Modal Logic introduced in [HLY92], here referred to as TL. The logic is an extension of the well known Hennessy–Milner Logic [HM85], and the formulae of the logic are given by the following abstract syntax:

$$\phi \;::=\; \mathrm{tt} \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \langle a \rangle \phi \mid \exists [l,u] \phi$$

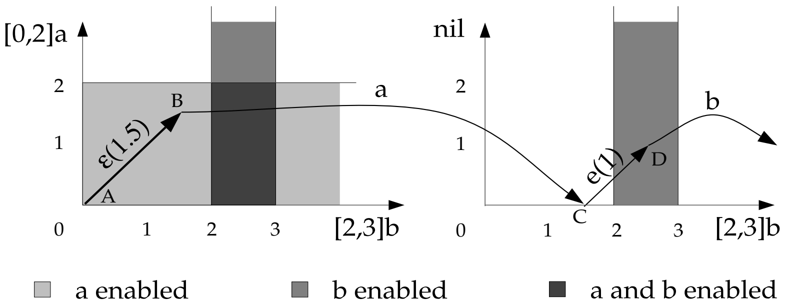


Figure 1: Transition Sequence

We shall freely use ff as abbreviation for $\neg$tt, $\phi_1 \vee \phi_2$ for $\neg(\neg\phi_1 \wedge \neg\phi_2)$, $[a]\phi$ for $\neg\langle a\rangle\neg\phi$ and $\forall[l,u]\phi$ for $\neg\exists[l,u]\neg\phi$.

For the interpretation of TL we define the satisfaction relation $\models$ between network configurations K and TL formulae ϕ inductively as follows:

$$\begin{array}{rlll}
i) & K \models \mathrm{tt} & \Leftrightarrow & \text{true} \\
ii) & K \models \phi_1 \wedge \phi_2 & \Leftrightarrow & K \models \phi_1 \text{ and } K \models \phi_2 \\
iii) & K \models \neg\phi & \Leftrightarrow & \text{not } K \models \phi \\
iv) & K \models \langle a\rangle\phi & \Leftrightarrow & \exists K'.\, K \xrightarrow{a} K' \text{ and } K' \models \phi \\
v) & K \models \exists[l,u]\phi & \Leftrightarrow & \exists K', d.\, d \in [l,u] \text{ and } K \xrightarrow{\epsilon(d)} K' \text{ and } K' \models \phi
\end{array}$$

We shall often write $\overline{A} \models \phi$ for $\langle \overline{A}, \overline{0}\rangle \models \phi$, where $\overline{0}$ is the (initial) delay vector with all components being 0. In this case we say that the network $\overline{A}$ satisfies the property ϕ.

Example 1.2 Consider the network $[0,2].a \,|\, [2,3].b$ from Example 1.1. Then it is easily seen that this network satisfies the formula $\forall[1,2]\langle a\rangle\forall[1,1]\langle b\rangle$tt. To see this simply observe that whenever $x \in [1,2]$ we can infer the following transition sequence:

$$\begin{array}{l}
\langle [0,2].a \,|\, [2,3].b \,,\, (x,x)\rangle \xrightarrow{a} \langle nil \,|\, [2,3].b \,,\, (0,x)\rangle \xrightarrow{\epsilon(1)} \\
\langle nil \,|\, [2,3].b \,,\, (1, x+1)\rangle \xrightarrow{b}
\end{array}$$

□

2 Symbolic Processes and Model Checking

Using the by now well–known region technique of Alur and Dill [ACD90] one may obtain an algorithm for model–checking: i.e. an algorithm for deciding whether a given network agent satisfies a TL formula. The region technique provides an abstract interpretation of network agents sufficiently complete that all information necessary for model–checking with respect to TL is maintained. At the same time the abstract interpretation yields a finite–state symbolic representation of networks thus enabling standard algorithmic model–checking techniques to be applied.

For $t \in \mathbf{R}$, let $\lfloor t \rfloor \stackrel{def}{=} \max\{n \in \mathbf{N} \mid n \leq t\}$ denote the integral part of t, and let $\{t\} \stackrel{def}{=} t - \lfloor t \rfloor$ denote its fractional part. We now recall from [ACD90]:

Definition 2.1 *Let $\overline{m} \in \mathbf{N}^n$ be a delay vector. Then $\overline{u}, \overline{v} \in \mathbf{R}^n$ are equivalent with respect to $\overline{m}$, denoted by $\overline{u} \doteq \overline{v}$ if*

- *For each $i = 1 \ldots n$, $u_i > m_i$ iff $v_i > m_i$,*
- *For each $i = 1 \ldots n$ such that $u_i \leq m_i$*
  1. $\lfloor u_i \rfloor = \lfloor v_i \rfloor$
  2. $\{u_i\} = 0$ *iff* $\{v_i\} = 0$
- *For each $i, j = 1 \ldots n$ such that $u_i \leq m_i$ and $u_j \leq m_j$, it is the case that $\{u_i\} \leq \{u_j\}$ iff $\{v_i\} \leq \{v_j\}$*

Observe that $\mathbf{R}^n / \doteq$ is finite. For $\overline{v} \in \mathbf{R}^n$, we denote by $[\overline{v}]$ the equivalence class of $\overline{v}$ under $\doteq$. The equivalence classes determined by $\doteq$ are called *regions* (see Figure 2).

For model–checking with respect to TL it is important to note that integer delays of equivalent delay vectors are again equivalent. Thus, whenever $\overline{u} \doteq \overline{v}$ then $\overline{u} + n \doteq \overline{v} + n$ whenever $n \in \mathbf{N}$. Hence, we may without ambiguity write $[\overline{u}] + n$ for the region $[\overline{u} + n]$. In general, it can be shown (see e.g. [LW93]) that two equivalent delay vectors $\overline{u}$ and $\overline{v}$ go through the same future regions; i.e. $\{[\overline{u} + d] \mid d \in R\} = \{[\overline{v} + d] \mid d \in R\}$. Moreover, $\overline{u}$ and $\overline{v}$ also agree on the order in which these regions are visited according to the following notion of successor region (see Figure 2):

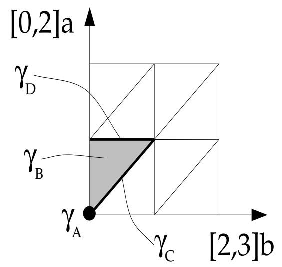
Definition 2.2 *Let $\gamma = [\overline{v}]$ be a region. Then the* successor *region* $\mathrm{succ}(\gamma)$ *is the region $[\overline{v'}]$, where:*

$$v'_i = \begin{cases} v_i + \min\{1 - \{v_j\} \mid j = 1 \ldots n\} & \text{if } \forall i.\{v_i\} > 0 \\ v_i + \min\{1 - \{v_j\} \mid j = 1 \ldots n\}/2 & \text{if } \exists i.\{v_i\} = 0 \end{cases}$$

We denote by $\mathrm{succ}^k(\gamma)$ *the region obtained by applying* succ *k times to γ.*

Now, it may be shown that the future regions from a delay vector $\overline{u}$ are precisely the regions $[\overline{u}], \mathrm{succ}^1([\overline{u}]), \mathrm{succ}^2([\overline{u}]), \mathrm{succ}^3([\overline{u}]), \ldots$ and that they are visited in this order. For γ a region and n a natural number we shall by n_γ denote the unique successor number such that $\gamma + n = \mathrm{succ}^{n_\gamma}(\gamma)$. Thus, when d ranges between two integer bounds l and u the delay vector $\overline{v} + d$ resides in regions between $\mathrm{succ}^{l_{[\overline{v}]}}([\overline{v}])$ and $\mathrm{succ}^{u_{[\overline{v}]}}([\overline{v}])$. Also, as agents enable actions within integer bounds, two network configurations with identical network agent and equivalent delay vectors agree on the action transitions they can perform in the following sense:

Lemma 2.3 *Let $\overline{A}$ be a network agent and $\overline{u} \doteq \overline{v}$. If $\langle \overline{A}, \overline{u} \rangle \stackrel{a}{\longrightarrow} \langle \overline{A}', \overline{u}' \rangle$, then also $\langle \overline{A}, \overline{v} \rangle \stackrel{a}{\longrightarrow} \langle \overline{A}', \overline{v}' \rangle$ for some $\overline{v}'$ such that $\overline{v}' \doteq \overline{u}'$.*



The figure illustrates four typical dyadic regions:

$$\begin{aligned}\gamma_A &= \{(0,0)\}\\ \gamma_B &= \{(x,y) \mid 0 < x, y < 1, x < y\}\\ \gamma_C &= \{(x,y) \mid 0 < x, y < 1, x = y\}\\ \gamma_D &= \{(x,y) \mid 0 < x < 1, y = 1\}\end{aligned}$$

Now using Definition 2.2 it follows that $\mathrm{succ}(\gamma_A) = \gamma_C$ and $\mathrm{succ}(\gamma_B) = \gamma_D$. In general successor regions are determined by following 45^o lines upwards to the right.

Figure 2: Regions and Their Successors

Based on the above observations it can be concluded that network configurations with equivalent delay vectors satisfy the same TL formulae:

Theorem 2.4 *Let $\overline{A}$ be a network agent, $\overline{u}, \overline{v}$ two delay vectors, and ϕ a TL formula. Then if $\overline{u} \doteq \overline{v}$ the following holds:*

$$\langle \overline{A}, \overline{u} \rangle \models \phi \quad \Leftrightarrow \langle \overline{A}, \overline{v} \rangle \models \phi$$

To obtain the model–checking algorithm we extract a finite–state symbolic semantics of network configurations, by identifying configurations with equivalent delay vector. Thus *symbolic* network configurations (or simply symbolic states) are pairs of the form $[\overline{A}, \gamma]$, where $\overline{A}$ is a network agent and γ is a $\doteq$–equivalence class with respect to the delay vector $M(\overline{A}) = (M(A_1), \ldots, M(A_n))$. As network agents have only finitely many sub-terms [2], and there are only finitely many regions with respect to $M(\overline{A})$ it follows that the set of symbolic states reachable from $[\overline{A}, \gamma]$ is finite. The transitions between symbolic states are either un–quantified delay transitions (labelled χ) or action transitions and are defined by the axiom and rule of Table 3. Moreover, symbolic transitions may be computed effectively. This rests on an effective representation of regions [3] allowing effective computation of a representative of a region as well as effective computation of the region from a delay vector. Also, due to Lemma 2.3, it suffices to consider a single representative $\overline{v}$ of γ when inferring symbolic action transitions.

We may now give an alternative interpretation of TL based on the above symbolic semantics of networks.

$$\begin{array}{rlcl} i) & [\overline{A}, \gamma] \models \mathrm{tt} & \Leftrightarrow & \text{true} \\ ii) & [\overline{A}, \gamma] \models \phi_1 \wedge \phi_2 & \Leftrightarrow & [\overline{A}, \gamma] \models \phi_1 \text{ and } [\overline{A}, \gamma] \models \phi_2 \\ iii) & [\overline{A}, \gamma] \models \neg\phi & \Leftrightarrow & \text{not } [\overline{A}, \gamma] \models \phi \\ iv) & [\overline{A}, \gamma] \models \langle a \rangle \phi & \Leftrightarrow & \exists [\overline{B}, \eta].\, [\overline{A}, \gamma] \xrightarrow{a} [\overline{B}, \eta] \text{ and } [\overline{B}, \eta] \models \phi \\ v) & [\overline{A}, \gamma] \models \exists [l, u]\phi & \Leftrightarrow & \exists l_\gamma \leq k \leq u_\gamma.\, [\overline{A}, \mathrm{succ}^k(\gamma)] \models \phi \end{array}$$

[2] with the usual application of unfolding in the case of recursive definition

[3] The obvious effective representation of a region is as a linear inequation system. An alternative effective representation where each region has a canonical representation is given in [GL94, God94].

Delay Transitions:	$$[\overline{A}, \gamma] \xrightarrow{\chi} [\overline{A}, \mathrm{succ}(\gamma)]$$
Action Transitions:	$$\frac{\langle \overline{A}, \overline{v} \rangle \xrightarrow{a} \langle \overline{B}, \overline{u} \rangle}{[\overline{A}, [\overline{v}]] \xrightarrow{a} [\overline{B}, [\overline{u}]]}$$

Table 3: Symbolic Semantics of Networks

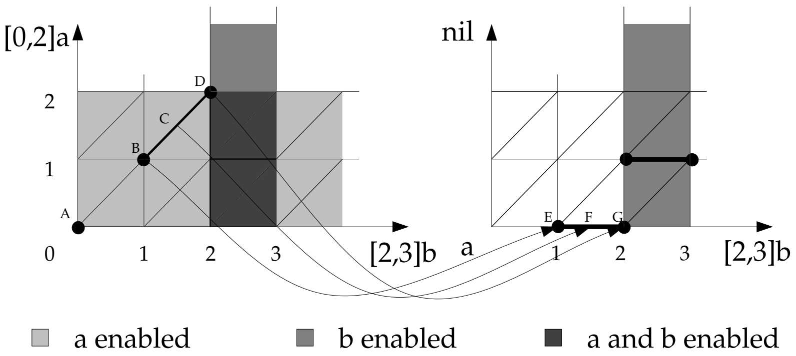


Figure 3: Symbolic Transitions

Clearly, due to the finite–state nature of the symbolic semantics of networks, the above symbolic interpretation is decidable using classical finite–state model–checking techniques. Moreover, the symbolic interpretation of TL is closely related to the standard interpretation as stated in the following theorem:

Theorem 2.5 *Let $\overline{A}$ be a network agent, $\overline{v}$ a delay vector, and ϕ a TL formula. Then the following equivalence holds:*

$$[\overline{A}, [\overline{v}]] \models \phi \quad \Leftrightarrow \quad \langle \overline{A}, \overline{v} \rangle \models \phi$$

It follows that model–checking of network configurations with respect to TL formulae is decidable.

Example 2.6 Figure 3 indicates some symbolic states and transitions associated with the network $[0,2].a \,|\, [2,3].b$. Note that $A \xrightarrow{\chi^2} B$, $A \xrightarrow{\chi^3} C$ and $A \xrightarrow{\chi^4} D$, and that $\{2,3,4\}$ are precisely the successor numbers associated with the delay interval $[1,2]$ when considering state A. Using the symbolic interpretation of TL it can now easily be checked that $[0,2].a|[2,3].b$ does indeed satisfy $\forall[1,2]\langle a \rangle \forall[1,1]\langle b \rangle \mathrm{tt}$. $\square$

3 Symbolic Contexts

We want to decompose logical properties required of a network agent into necessary and sufficient properties of one of its agents. More precisely, for any given regular agents $A_1, \ldots, A_n$ and any given TL formula ϕ, we want to find a formula ψ such that the following holds:

$$(A_1 \,|\, \ldots \,|\, A_n \,|\, A) \models \phi \quad \textit{if and only if} \quad A \models \psi \tag{3}$$

Clearly, the component property ψ will in general depend on the overall property ϕ as well as the agents $A_1, \ldots, A_n$. In the next section we shall define a *property transformer* $\mathcal{W}$, that — given the network agent $\overline{A} = (A_1|\ldots|A_n)$ and the property ϕ — will construct a property $\psi = \mathcal{W}(\overline{A}, \phi)$ satisfying the requirement of (3).

In (3), the property ϕ expresses constraints on transitions of the complete network $(A_1|\ldots|A_n|A)$, whereas ψ constrains transitions of the component agent A. Thus, in order to solve the above decomposition problem we must have a way of interrelating transitions of a network with transitions of one of its components. To achieve this we provide a symbolic operational semantics of the network contexts $C = (A_1|\ldots|A_n|[\,])$ in terms of *action transducers*. That is, a network context is semantically viewed as an object which consumes actions from its component agent and produces actions for the external environment, thus acting as an interface between the two. Obviously, we expect the new operational semantics of network contexts to be consistent with the existing operational semantics of network agents. That is, if the component agent A has an a transition, and the context $(A_1|\ldots|A_n|[\,])$ can consume this action while producing the action b, then we expect the combined network $(A_1|\ldots|A_n|A)$ to have a b–transition.

The idea of modelling contexts as action transducers has already been pursued for finite state systems [LX90a, LX91]. In our real–time setting we need in addition to take into account the delay of the context agents $A_1, \ldots, A_n$ as well as the delay of the component to be placed in the hole $[\,]$. However, as our transductional semantics is intended to provide the basis of an effective transformation of properties, we deal with delays in a symbolic manner using regions. Thus, formally, a symbolic $n+1$–ary network context is a pair of the form:

$$\Big[(A_1|\ldots|A_n|[\,])\,,\,[(v_1\ldots v_n, v)]\Big]$$

Here $[(v_1\ldots v_n, v)]$ is an $n+1$–ary region with $(v_1\ldots v_n)$ giving delay information of $A_1, \ldots, A_n$ and v providing the delay information of the $[\,]$–component. The transductions between symbolic network contexts is given by the axioms and rule of Table 4. For γ being an $n+1$–ary region $[(v_1\ldots v_n, v)]$, $\gamma^{\downarrow}$ denotes the unary region $[v]$ [4]. For $\overline{v} = (v_1\ldots v_n)$ an n–ary delay vector and u a non–negative real, $\overline{v}u$ denotes the $n+1$–ary delay vector $(v_1\ldots v_n, u)$.

Transductions may be inferred in two ways depending on whether the $[\,]$–component "participates" in the transduction or not. Thus for action trans-

[4] In the unary case regions are either integer points $[n, n]$, open intervals $]n, n+1[$ or open infinite intervals $]m, \infty[$, where m is the maximum delay bound.

DELAY TRANSDUCTIONS: $$\left[\overline{A}|[\,],\gamma\right] \xrightarrow[\chi]{\chi} \left[\overline{A}|[\,],\mathrm{succ}(\gamma)\right] \text{ if } \mathrm{succ}(\gamma)^{\downarrow} = \mathrm{succ}(\gamma^{\downarrow})$$ $$\left[\overline{A}|[\,],\gamma\right] \xrightarrow[0]{\chi} \left[\overline{A}|[\,],\mathrm{succ}(\gamma)\right] \text{ if } \mathrm{succ}(\gamma)^{\downarrow} = \gamma^{\downarrow}$$
ACTION TRANSDUCTIONS: $$\left[\overline{A}|[\,],[\overline{v}u]\right] \xrightarrow[a]{a} \left[\overline{A}|[\,],[\overline{v}0]\right]$$ $$\frac{\langle\overline{A},\overline{v}\rangle \xrightarrow{a} \langle\overline{B},\overline{w}\rangle}{\left[\overline{A}|[\,],[\overline{v}u]\right] \xrightarrow[0]{a} \left[\overline{B}|[\,],[\overline{w}u]\right]}$$

Table 4: Symbolic Transduction Semantics of Contexts

ductions the first axiom of Table 4 requires the [ ]–component to perform the a–action (after which the [ ]–delay is reset to 0). In the second action transduction rule, the a–action is performed entirely by the network context $\overline{A}$ without any involvement of the [ ]–component. This is modelled by a transduction using a unique 0–action (i.e. $0 \notin \mathcal{A}$). The symbolic semantics is extended in the obvious way to 0–actions by $[\overline{A},\gamma] \xrightarrow{0} [\overline{A}',\gamma']$ if and only if $\overline{A}' = \overline{A}$ and $\gamma' = \gamma$. Delay transductions model progression to a successor region. The two delay transduction axioms reflect that the projected [ ]–region may either remain unchanged or change to its (unary) successor region.

Example 3.1 In Figure 4 three types of transductions for the network $[0,2].a|[\,]$ are illustrated. As clearly $\gamma_B^{\downarrow} = \mathrm{succ}(\gamma_A^{\downarrow})$ and $\gamma_E^{\downarrow} = \gamma_D^{\downarrow}$, it follows from Table 4 that:

$$\left[([0,2].a|[\,]),\gamma_A\right] \xrightarrow[\chi]{\chi} \left[([0,2].a|[\,]),\gamma_B\right]$$
$$\left[(nil|[\,]),\gamma_D\right] \xrightarrow[0]{\chi} \left[(nil|[\,]),\gamma_E\right]$$
$$\left[([0,2].a|[\,]),\gamma_C\right] \xrightarrow[0]{a} \left[(nil|[\,]),\gamma_D\right]$$

□

The following Lemma demonstrates that the transductional semantics of contexts does indeed provide the key to relating symbolic transitions of a network and its component:

Lemma 3.2 *Let $\overline{A} = (A_1|\ldots|A_n)$ be an n–ary network agent, A a regular timed agent, γ an $n+1$–ary region, and let $\alpha \in \mathcal{A} \cup \{\chi\}$. Then the following equivalence holds:*

$$\left[\overline{A}|A,\gamma\right] \xrightarrow{\alpha} \left[\overline{A}'|A',\gamma'\right]$$

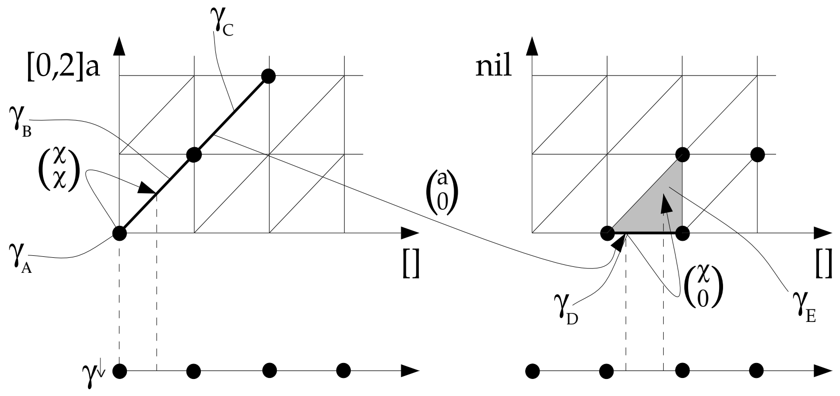


Figure 4: Context Transductions

if and only if

$$\left[\overline{A}|[\,],\gamma\right] \xrightarrow[\beta]{\alpha} \left[\overline{A}'|[\,],\gamma'\right] \text{ and } \left[A,\gamma^{\downarrow}\right] \xrightarrow{\beta} \left[A',\gamma'^{\downarrow}\right]$$

for $\beta = \alpha$ *or* $\beta = 0$*.*

4 Contexts as Property Transformers

As shown in the previous Lemma 3.2, contexts relate *symbolic* transitions of networks with *symbolic* transitions of their components. To facilitate the transformation of logical properties we extend our logic TL with a modality explicitly concerned with symbolic delay transitions ($\xrightarrow{\chi}$). Syntactically, we add the following production to the syntax for formulae:

$$\phi \;::=\; \odot\phi$$

We refer to the extended logic as $\mathrm{TL}^{\odot}$. Formulae with no occurrence of $\exists[l,u]$–modalities are called *pure* and the corresponding sublogic is refered to as $\mathrm{TL}_p^{\odot}$. Semantically, we interpret formulae $\odot\phi$ with respect to (standard) network configurations as well as symbolic network configurations thus extending the two existing interpretations of TL:

$$\langle \overline{A}, \overline{u}\rangle \models \odot\phi \;\;\Leftrightarrow\;\; \langle \overline{A}, \overline{v}\rangle \models \phi \text{ for some } \overline{v} \in \mathrm{succ}([\overline{u}])$$
$$[\overline{A}, \gamma] \models \odot\phi \;\;\Leftrightarrow\;\; [\overline{A}, \mathrm{succ}(\gamma)] \models \phi$$

It is straightforward to show that with these semantic definitions both Theorem 2.4 and Theorem 2.5 generalize to $\mathrm{TL}^{\odot}$. Furthermore, for any given network configuration, the original (interval) delay modalities of TL can be expressed using the new $\odot$–modality in the following way:

$$\langle \overline{A}, \overline{v}\rangle \models \exists[l,u]\phi \;\;\Leftrightarrow\;\; \langle \overline{A}, \overline{v}\rangle \models \bigvee_{k=l_{[\overline{v}]}}^{u_{[\overline{v}]}} \odot^k\phi$$

For $C = [\overline{A}|[\,],\gamma]$ an $n+1$–ary context and ϕ a TL$^{\odot}$–formula we now define the *transformed formula* $\mathcal{W}(C,\phi)$ as follows:

$$
\begin{aligned}
i)&\quad \mathcal{W}(C,\mathrm{tt}) = \mathrm{tt}\\
ii)&\quad \mathcal{W}(C,\phi_1\wedge\phi_2) = \mathcal{W}(C,\phi_1)\wedge\mathcal{W}(C,\phi_2)\\
iii)&\quad \mathcal{W}(C,\neg\phi) = \neg\mathcal{W}(C,\phi)\\
iv)&\quad \mathcal{W}(C,\langle a\rangle\phi) = \bigvee_{C\,\xrightarrow[a]{a}\,C'} \langle a\rangle\mathcal{W}(C',\phi)\ \vee \bigvee_{C\,\xrightarrow[0]{a}\,C'} \mathcal{W}(C',\phi)\\
v)&\quad \mathcal{W}(C,\exists[l,u]\phi) = \mathcal{W}(C,\bigvee_{k=l_\gamma}^{u_\gamma}\odot^k\phi)\\
vi)&\quad \mathcal{W}(C,\odot\phi) = \begin{cases}\mathcal{W}(C',\phi) & \text{if } C\xrightarrow[0]{\chi}C'\\ \odot\mathcal{W}(C',\phi) & \text{if } C\xrightarrow[\chi]{\chi}C'\end{cases}
\end{aligned}
$$

Note that $\mathcal{W}(C,\phi)$ is always a pure TL$^{\odot}$formula. The following Theorem and Corollary shows that the transformer $\mathcal{W}$ does indeed yield the sufficient and necessary requirement to a networks component in order that the network itself satisfy a given property:

Theorem 4.1 *Let $C = [\overline{A}|[\,],\gamma]$ be an $n+1$–ary context. Then the following equivalence holds for any regular timed agent A:*

$$[\overline{A}|A,\gamma]\models\phi \quad\Leftrightarrow\quad [A,\gamma^{\downarrow}]\models\mathcal{W}(C,\phi)$$

Corollary 4.2 *Let $C = [\overline{A}|[\,],\gamma]$ be an $n+1$–ary context. Then the following equivalence holds for any regular timed agent A, whenever $\overline{v}u\in\gamma$:*

$$\langle\overline{A}|A,\overline{v}u\rangle\models\phi \quad\Leftrightarrow\quad \langle A,u\rangle\models\mathcal{W}(C,\phi)$$

Corollary 4.3 *Let $\overline{A} = (A_1|\ldots|A_n)$ be an n–ary network, and let A be a regular timed agent. Then the following equivalence holds:*

$$(A_1|\ldots|A_n|A)\models\phi \quad\Leftrightarrow\quad A\models\mathcal{W}([\overline{A}|[\,],[\overline{0}]],\phi)$$

Example 4.4 Using $\mathcal{W}$ we may now compute the necessary and sufficient requirement to the component A in order that $[0,2].a\mid A$ satisfies $\phi = \forall[1,2]\langle a\rangle\forall[1,1]\langle b\rangle\mathrm{tt}$. After some calculations based on the transductional semantics of $[0,2].a|[\,]$ (part of which is illustrated in Example 3.1) we get the following:

$$
\begin{aligned}
&\mathcal{W}\Big(\Big[[0,2].a|[\,],\,[\overline{0}]\Big],\phi\Big) =\\
&\quad\odot^2\Big(\odot^2\langle b\rangle\mathrm{tt}\vee\langle a\rangle\odot^2\langle b\rangle\mathrm{tt}\Big)\ \wedge\ \odot^3\Big(\odot^2\langle b\rangle\mathrm{tt}\vee\langle a\rangle\odot^2\langle b\rangle\mathrm{tt}\Big)\ \wedge\\
&\quad\odot^4\Big(\odot^2\langle b\rangle\mathrm{tt}\vee\langle a\rangle\odot^2\langle b\rangle\mathrm{tt}\Big)
\end{aligned}
$$

□

Using the easily established fact that $(A_1|\ldots|A_n)$ and $(A_1|\ldots|A_n|nil)$ satisfy the same formulae the transformer $\mathcal{W}$ can also be used to obtain an alternative model–checking algorithm:

Corollary 4.5 *Let $\overline{A} = (A_1|\ldots|A_n)$ be an n–ary network, and let ϕ be a $TL^{\odot}$formula. Then the following equivalence holds:*

$$(A_1|\ldots|A_n) \models \phi \quad \Leftrightarrow \quad nil \models \mathcal{W}([\overline{A}|[\,],[\overline{0}]],\phi)$$

To decide whether the agent nil satisfies a property is particularly easy as nil satisfies no $\langle a\rangle\phi$ formula and all $[a]\phi$ formulae; also nil satisfies formulae $\exists[l,u]\phi$, $\forall[l,u]\phi$ and $\odot\phi$ precisely if nil satisfies ϕ.

Example 4.6 To decide $[0,2].a|[2,3]b$ satisfies the property $\phi = \forall[1,2]\langle a\rangle\forall[1,1]\langle b\rangle\mathrm{tt}$ we should simply certify that nil satisfies the following transformed property (obtained after some calculations):

$$\begin{aligned}
&\mathcal{W}\Big(\Big[[0,2].a|[2,3].b|[\,]\,,\,[\overline{0}]\Big],\phi\Big) = \\
&\qquad\qquad \odot^2\left(\odot^2(\mathrm{tt}\vee\langle b\rangle\mathrm{tt})\vee\langle a\rangle\odot^2(\mathrm{tt}\vee\langle b\rangle\mathrm{tt})\right)\ \wedge \\
&\qquad\qquad \odot^3\left(\odot^2(\mathrm{tt}\vee\langle b\rangle\mathrm{tt})\vee\langle a\rangle\odot^2(\mathrm{tt}\vee\langle b\rangle\mathrm{tt})\right)\ \wedge \\
&\qquad\qquad \odot^4\left(\odot^2(\mathrm{tt}\vee\langle b\rangle\mathrm{tt})\vee\langle a\rangle\odot^2(\mathrm{tt}\vee\langle b\rangle\mathrm{tt})\right)
\end{aligned}$$

Now, using the simplification rules pointed out above (i.e. simplify all modalities) it is obvious that nil satisfies the above property. □

5 Direct Model Construction

In this section we provide an algorithm that given a pure $\mathrm{TL}^{\odot}$–formula ϕ will decide whether ϕ is satisfiable by some regular agent. Moreover if ϕ is satisfiable the algorithm will construct a satisfying agent. The technique applied is based on classical tableau methods applied for modal logic (see e.g. [HC68]). To simplify this part of the presentation we use an alternative version of $\mathrm{TL}^{\odot}_p$ with no negation but with all dual operators included (i.e. ff, $\vee$ and $[a]$).

Let Γ be the set of all unary regions of the form $[n,n]$ and $]n,n+1[$, where $n\in\mathbf{N}$. Then a *problem* Π is a finite subset of $\Gamma\times\mathrm{TL}^{\odot}_p$. We say that a problem Π is *satisfiable* if there exists a regular timed agent A such that $[A,\gamma]\models\phi$ whenever $(\gamma,\phi)\in\Pi$. In this case we call A a *solution* to Π. It follows from the results of the previous sections that if A is a solution to an *initial* problem of the form $\{(\mathcal{O},\phi)\}$ where $\mathcal{O}=\{0\}$ then $A\models\phi$.

A problem Π is called *simple* if whenever $(\gamma,\phi)\in\Pi$ then ϕ is of the form $\langle a\rangle\psi$ or $[a]\psi$; i.e. all conjunctions, disjunctions and $\odot$–modalities have been resolved. As we shall see in the following it is particularly easy to decide satisfiability of simple problems. However, we first provide a reduction mechanism for transforming problems into simple ones. The reduction relation $\leadsto$ between problems is defined as the least relation satisfying the following axioms [5]:

$$\begin{array}{rl} i) & \Pi \uplus \{(\gamma, \mathrm{tt})\} \leadsto \Pi \\ ii) & \Pi \uplus \{(\gamma, \phi_1 \wedge \phi_2)\} \leadsto \Pi \cup \{(\gamma, \phi_1)\} \cup \{(\gamma, \phi_2)\} \\ iii) & \Pi \uplus \{(\gamma, \phi_1 \vee \phi_2)\} \leadsto \Pi \cup \{(\gamma, \phi_1)\} \\ iv) & \Pi \uplus \{(\gamma, \phi_1 \vee \phi_2)\} \leadsto \Pi \cup \{(\gamma, \phi_2)\} \\ v) & \Pi \uplus \{(\gamma, \odot\phi)\} \leadsto \Pi \cup \{(\mathrm{succ}(\gamma), \phi)\} \end{array}$$

As the use of $\leadsto$ always strictly decreases the total size of the formulae in Π it is clear that any reduction sequence from Π must be finite. In fact any problem determines a finite reduction tree with the leaves being the irreducible reductions of Π; i.e. Π' is an irreducible reduction of Π if $\Pi \leadsto^{\star} \Pi'$ and $\Pi' \not\leadsto$. Now it follows directly from the semantic definition of the various operators of $\mathrm{TL}_p^{\odot}$ that there is a close connection between the satisfiability of a problem and its irreducible reductions:

Lemma 5.1 *A problem is satisfiable if and only if one of its irreducible reductions is satisfiable.*

Moreover, it is clear from the definition of $\leadsto$ that any irreducible problem is either simple or contains a pair of the form (γ, ff) in which case it is obviously not satisfiable. Thus, we are left with the problem of deciding satisfiability of simple problems.

First we define for $a \in \mathcal{A}$, $\gamma \in \Gamma$ and Π a problem the *projected* problem Π_a^γ as follows:

$$\Pi_a^\gamma = \{(\mathcal{O}, \phi) \mid (\gamma, [a]\phi) \in \Pi\}$$

From the symbolic interpretation of $[a]\phi$ it follows directly that whenever A is a solution to Π and $[A, \gamma] \xrightarrow{a} [A', \mathcal{O}]$, then A' is a solution to Π_a^γ. Moreover, when $\gamma =]n, n+1[$, A' is also a solution to $\Pi_a^{[n,n]}$ and $\Pi_a^{[n+1,n+1]}$ as regular agents enable actions within *closed* integer–bound intervals.

Theorem 5.2 *Let Π be a simple problem. Then Π is satisfiable if and only if whenever $([n,n], \langle a\rangle\phi) \in \Pi$ then*

$$\Big\{(\mathcal{O}, \phi)\Big\} \cup \Pi_a^{[n,n]} \tag{4}$$

is satisfiable, and whenever $(]n, n+1[, \langle a\rangle\phi) \in \Pi$ then

$$\Big\{(\mathcal{O}, \phi)\Big\} \cup \Pi_a^{[n,n]} \cup \Pi_a^{]n,n+1[} \cup \Pi_a^{[n+1,n+1]} \tag{5}$$

is satisfiable.

Proof: *Then*–direction: follows directly from the symbolic interpretation of formulae and the comments above.
If–direction: For $(\gamma, \langle a\rangle\phi) \in \Pi$ let $A_{(\gamma,\langle a\rangle\phi)}$ be a regular timed agent satisfying

[5] $\uplus$ denotes disjoint union of sets.

(4) if $\gamma = [n,n]$ and (5) if $\gamma =]n, n+1[$. Let $l_{[n,n]} = u_{[n,n]} = n$, $l_{]n,n+1[} = n$ and $u_{]n,n+1[} = n+1$. Then the agent A defined as:

$$A = \sum_{(\gamma,\langle a\rangle\phi)\in\Pi} [l_\gamma, u_\gamma].a.A_{(\gamma,\langle a\rangle\phi)}$$

satisfies Π. $\square$

It now follows from the properties of $\rightsquigarrow$ and Theorem 5.2 that satisfiability of problems is decidable: to determine satisfiablity of a problem Π first (non–deterministically) reduce it to a simple problem Π' and then use the construction of Theorem 5.2. This leaves the satisfiability of the problems in (4) and (5) to be settled. However, as these problems all have strictly smaller maximum modal depth than Π' (and Π) we can apply the method recursively, with termination guaranteed.

Example 5.3 In order to synthesize the missing component A in Example 4.4 we should determine satisfiability of the problem:

$$\Pi = \Big\{(\mathcal{O}, \odot^2\psi \wedge \odot^3\psi \wedge \odot^4\psi)\Big\}$$

where $\psi = \odot^2\langle b\rangle\mathrm{tt} \vee \langle a\rangle \odot^2 \langle b\rangle\mathrm{tt}$. Now using the axioms for $\rightsquigarrow$ we obtain:

$$\begin{aligned}
&\Pi \rightsquigarrow^\star \\
&\Big\{([1,1],\psi), (]1,2[,\psi), ([2,2],\psi)\Big\} \rightsquigarrow^\star \\
&\Big\{([1,1],\odot^2\langle b\rangle\mathrm{tt}), (]1,2[,\odot^2\langle b\rangle\mathrm{tt}), ([2,2],\odot^2\langle b\rangle\mathrm{tt})\Big\} \rightsquigarrow^\star \\
&\Big\{([2,2],\langle b\rangle\mathrm{tt}), (]2,3[,\langle b\rangle\mathrm{tt}), ([3,3],\langle b\rangle\mathrm{tt})\Big\} = \Pi'
\end{aligned}$$

Now — as *nil* obviously satisfies $\{(\mathcal{O},\mathrm{tt})\}$ — we obtain from the proof of Theorem 5.2 that the following agent:

$$A = [2,2].b + [2,3].b + [3,3].b$$

satisfies Π' and hence Π. $\square$

Concluding Remarks

The presentation of this paper has been based on a somewhat simplified setting, and we want here to comment in slightly more detail on how our results extends.

The logic TL considered may be extended with constructs for defining properties recursively. The symbolic interpretation of TL extends easily to this recursive extension, thus providing the basis for decidability of model–checking. As for transforming recursive properties the techniques given in [LX90a, LX91] can be directly applied. Our direct model–construction method extends to maximal recursively defined properties using the techniques of [JLJL93].

The notion of parallel composition considered in this paper is simply that of interleaving of actions. However, our results extend to a variety of parallel compositions via parameterization on a *synchronization function* as studied in [HL89]. Thus, we may consider *parameterized* network of the form

$(A_1, \ldots, A_n)|_f$, where f is a synchronization (partial) function of type $(\mathcal{A} \cup \{0\})^n \hookrightarrow \mathcal{A}$. The use of the special no–action 0 enables the modelling of synchronizations where only some components participate. Also, the partiality of f enables synchronization of certain combinations of actions to be disallowed.

In this presentation we have not yet considered implicit process algebraic specifications; i.e. specifications of the form:

$$(A_1 \,|\, \ldots \,|\, A_n \,|\, X) \equiv B \tag{6}$$

where B is a regular timed agent and $\equiv$ is some abstracting equivalence (timed bisimilarity, say). However, (6) may easily be transformed into an equivalent logical implicit specification by using a *characteristic formula* ϕ_B for B; i.e. a formula such that $A \equiv B$ if and only if $A \models \phi_B$. Both for timed and time–abstracting bisimilarity such characteristic formulae can be effectively constructed.

Future work includes extension of our method to implicit specifications for arbitrary n–clock automata. However, it is already known that implicit specifications for general timed networks, i.e. specifications of the form:

$$(A_1 \,|\, \ldots \,|\, A_n \,|\, X_1 \,|\, \ldots \,|\, X_m) \ \mathsf{sat}\ S$$

where $X_1 \ldots X_m$ are regular timed agents (or 1–clock automata) are undecidable when $m > 1$ [Liu93].

References

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model–checking for Real–Timne Systems. *In Proceedings of Logic in Computer Science*, 1990.

[AD90] R. Alur and D. Dill. Automata for Modelling Real–Time Systems. *Lecture Notes in Computer Science*, 443, 1990. In Proceedings of ICALP.

[Cer92] K. Cerans. Decidability of Bisimulation Equivalences for Processes with Parallel Timers. *In Proceedings of CAV'92*, 1992.

[CGL93] K. Cerans, J.C. Godskesen, and K.G. Larsen. Timed modal specifications — theory and tools. *Lecture Notes in Computer Science*, 1993.

[GL94] J.C. Godskensen and K.G. Larsen. Synthesizing Distiniguishing Formulae for Real Time Systems. Technical report, Aalborg University, 1994.

[God94] J.C. Godskesen. *Timed Modal Specifications — A theory for verification of real–time concurrent systems.* PhD thesis, Aalborg University, 1994.

[HC68] G.E. Hughes and M.J. Cresswell. *An Introduction to Modal Logic.* Methuen and Co., 1968.

[HL89] Hans Hüttel and Kim G. Larsen. The Use of Static Constructs in a Modal Process Logic. *Lecture Notes in Computer Science*, (363), 1989. Logic at Botik'89.

[HLY92] U. Holmer, K.G. Larsen, and W. Yi. Decidability of bisimulation equivalence between regular timed processes. *Lecture Notes In Computer Science, Springer Verlag*, 575, 1992. In Proceedings of CAV'91.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, pages 137–161, 1985.

[HNSY92] T. A. Henzinger, Z. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Logic in Computer Science*, 1992.

[JLJL93] O.H. Jensen, J.T. Lang, C. Jeppesen, and K.G. Larsen. Model construction for implicit specifications in modal logic. *Lecture Notes in Computer Science*, 1993.

[Liu93] Xinxin Liu. On Small Model Property and Decidability of Process Equations. Unpublished note, Sussex University., 1993.

[LQ90] P. Lewis and H. Qin. Factorization of finite state machines under observational equivalence. *Lecture Notes In Computer Science, Springer Verlag*, 458, 1990.

[LS92] K.G. Larsen and A. Skou. Compositional verification of probabilistic processes. *In Proceedings of CONCUR'92. To appear in Lecture Notes in Computer Science.*, 1992.

[LW93] K.G. Larsen and Y. Wang. Time Abstracted Bisimulation: Implicit Specifications and Decidability. *In Proceedings of MFPS'93*, 1993.

[LX90a] K.G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. *Lecture Notes In Computer Science, Springer Verlag*, 443, 1990. In proceedings of International Colloquium on Algorithms, Languages and Programming 1990.

[LX90b] K.G. Larsen and L. Xinxinx. Equation Solving Using Modal Transition systems. *In Proceedings of Logic in Computer Science*, 1990.

[LX91] K.G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. *Journal of Logic and Computation*, 1(6):761–795, 1991.

[NRJV90] X. Nicollin, J.L. Richierand, J.Sifakis, and J. Voiron. ATP: an algebra for timed processes,. *In Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods*, 1990.

[NSY91] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to Timed Graphs and Hybrid Systems. *Lecture Notes in Computer Science*, 600, 1991. In Real–Time: Theory in Practice.

[Par89] J. Parrow. Submodule construction as equation solving in CCS. *Theoretical Computer Science*, 68, 1989.

[RT89] R.Alur and T.A.Henzinger. A Really Temporal Logic. *In Proceeding of IEEE Symp. on Foundations of Computer Science*, 1989.

[Shi] M.W. Shields. A note on the simple interface equation. Technical report, University of Kent at Canterbury.

[Wan90] Yi Wang. A Calculus of Real Time Systems. *Lecture Notes in Computer Science*, 458, 1990. In Proceedings of CONCUR.

Demonstration of ALF

Catarina Coquand   Lena Magnusson

Chalmers Technical University
Inst. for Computer Science
Göteborg, Sweden
E-mail: {catarina,lena}@cs.chalmers.se

We refer to our abstracts [Coq] and [Mag] in these proceedings for further information about the demonstrated system.

References

[Coq] Catarina Coquand. Introduction to ALF — a background. Chalmers Technical University, Inst. for Computer Science, Göteborg, Sweden. E-mail: `catarina@cs.chalmers.se`.

[Mag] Lena Magnusson. Introduction to ALF — an interactive proof editor. Chalmers Technical University, Inst. for Computer Science, Göteborg, Sweden. E-mail: `lena@cs.chalmers.se`.

Intelligent Software Synthesis in the DaCapo Environment

B. Steffen* B. Freitag A. Claßen T. Margaria U. Zukowski

Universität Passau
Germany

Abstract

The DaCapo software synthesis environment presented in this paper supports the *automatic construction of software* from existing components according to a (possibly incomplete) specification. Based on a specification language that uniformly combines taxonomic component specifications, interface conditions, and ordering constraints, our method adds a global view to conventional single component retrieval. Together with its semi-automatic archiving facility for the 'retrievable' inclusion of a satisfactory new software component into the software repository, our systems supports a full software lifecycle, as we illustrate along a user session.

1 Introduction

Systematic computer aided component reuse is essential to improve the productivity as well as the quality of software development. However, the complexity of the task requires to go beyond 'classical' library management tools, which are mainly based on single component retrieval, in order to sufficiently deal with the *global* dependencies between different software components. By their nature, such extensions are algorithmically very expensive. Thus only extremely carefully designed systems, which limit the exponential worst case behaviour to the 'unavoidable', are practically relevant. This requires the application specific structuring of the software repository together with a flexible specification language which allows users to formulate all the available knowledge without being forced to overspecify.

The DaCapo system supports a full software lifecycle satisfying these requirements for a 'transformational' setting by

- structuring the repository according to type constraints and semantic properties formulated in a taxonomic logic,
- providing a flexible specification language that captures *local* properties like taxonomic component descriptions and type conditions, and *global*

*Fakultät für Mathematik und Informatik, Universität Passau, D-94030 Passau (Germany), tel: +49 851 509-345, fax: +49 851 509-346, steffen@fmi.uni-passau.de

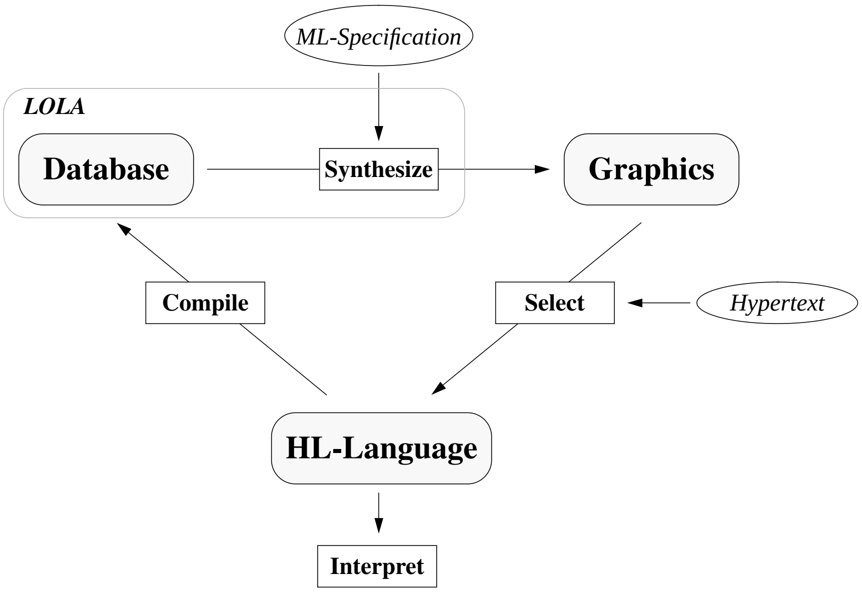


Figure 1: The DaCapo lifecycle

constraints fixing ordering, precedence, eventuality and conditional occurrence. This language implicitly encodes dynamic search strategies for sequential component compositions, which allows us to tailor the search effort according to the stage of the analysis.

- supporting the inclusion of a satisfactory new software component into the software repository by proposing an appropriately precomputed default taxonomic classification. These classifications are sufficient for future retrieval, but can also be modified by means of a hypertext system.

Figure 1 summarizes the architecture of our prototype implementation. Its database as well as its retrieval and synthesis algorithm are implemented on top of the deductive database system *LOLA* [FrKe94, FrSS91], and the graphical interface as well as the Hypertext browser are built on top of the TCL/TK graphics library [Oust94]. The target language is a high-level language, which is taylored for the combination of complex programs written in different programming languages. It is implemented in C++.

The development of the DaCapo system was influenced by industrial experience made with the hypertext-based software library system SEL [Frei94] that has been installed at BMW, Munich, Germany. The SEL system supports the classification and taxonomic grouping of components as well as the definition of typed links that allow a context dependent retrieval. However, neither type and signature information nor component composition have been considered, which are the essence of the DaCapo approach.

Current applications of our system range from the automatic construction of interfacing modules bridging the gap between the output type and the input type for a given sequence of software components, to the automatic synthesis of complete 'transformational' programs, which will be illustrated by configuring UNIX piping sequences for text and image processing (see Section 4). Other practically relevant applications that we are currently investigating concern

- semi-automatic intelligent network programming in a cooperation with Siemens Nixdorf (Munich),
- automatic composition of heterogeneous verification algorithms for hardware and software systems from basic transformation and analysis components (see [ClPS93, BuSt92, BuSt94]), and
- automatic generation of dataflow analysis algorithms from logic specifications for both finite and infinite state programs (see [Stef93, KnRS92, KnRS94a, KnRS94b]).

The paper is organized as follows: Section 2 introduces our UNIX application, Section 3 defines the metalevel specification language SLTL, Section 4 illustrates the full DaCapo lifecycle by means of a typical user session, Section 5 discusses the relations with other approaches, and Section 6 draws our conclusions.

2 Application: Synthesis of Complex UNIX Commands

We demonstrate the power of our approach by attacking a problem most UNIX users are confronted with: the synthesis of their own set of custom UNIX shell commands. The success of the UNIX operating system is in fact also due to its potential for composition-based software reuse: the UNIX shell can be viewed as a collection of reusable components that can be composed using the pipe mechanism. Let us consider text processing for illustration. Most UNIX environments offer the commands and applications summarized in Table 1, which we will use through the exemplary development in this paper.

The Models

UNIX commands and applications can be seen as transformations that take an input and produce the corresponding output. Thus they can abstractly be described as (*input-type*, `operation`, *output-type*). Accordingly, the `latex` application transforming a document *source.tex* into a device independent formatted document *source.dvi* is modelled as

$$(\mathit{latex\_source}, \texttt{latex}, \mathit{dvi})$$

Similarly, the X-Windows display program `xdvi` which takes a *dvi*-file as input and displays it on the *screen* is described as

$$(\mathit{dvi}, \texttt{xdvi}, \mathit{screen})$$

module name	input type	output type	description
`dvips`	*dvi*	*ps*	converts *dvi* to *ps*
	dvi/eps	*ps*	
`ghostview`	*ps*	*screen*	displays PostScript documents
`latex`	*ltx*	*dvi*	LaTeX-Compiler
	ltx/eps	*dvi/eps*	
`latex2html`	*ltx*	*html*	converts LaTeX source to *html*
`lpr`	*ps*	*printout*	prints documents
`more`	*ascii*	*screen*	displays plain text
`psnup-2`	*ps*	*ps*	formats multiple pages per sheet
`psselect`	*ps*	*ps*	selects pages from a document
`tex`	*tex*	*dvi*	TeX-Compiler
	tex/eps	*dvi/eps*	
`xdvi`	*dvi*	*screen*	TeX-Previewer
`xmosaic`	*html*	*screen*	World Wide Web browser

Table 1: The text processing module database

basic type	description
ascii	plain ASCII text
dvi	pure DVI file
dvi/eps	DVI file with embedded PostScript code
fig	vector graphic format
html	source file for www/xmosaic
ltx	pure LaTeX source
ltx/eps	LaTeX source with embedded PostScript code
pbm	portable bitmap graphic format
pgm	portable graymap graphic format
ppm	portable pixmap graphic format
printout	output type for print commands
ps	PostScript file
screen	output type for display commands
tex	pure TeX source
tex/eps	TeX source with embedded PostScript code

Table 2: The basic types

Table 1 and Table 2 summarize the commands and types we will use in this paper. Note that there can be more than one input/output combination for a command, thus allowing polymorphy.

Our goal is the specification and automatic synthesis of *legal program compositions*, i.e., of executable finite sequences of programs. Obviously, two programs can be sequentially combined iff they are *neighbours*, i.e. the output type of the first is compatible with the input type of the second. E.g. the `latex` and `xdvi` commands are neighbours, but `latex` and `ghostview` are not. Thus legal program compositions are simply sequential compositions of successively neighbouring programs, like e.g.

$$(latex\_source, \texttt{display\_latex}, screen) = (latex\_source, \texttt{latex}, dvi, \texttt{xdvi}, screen)$$

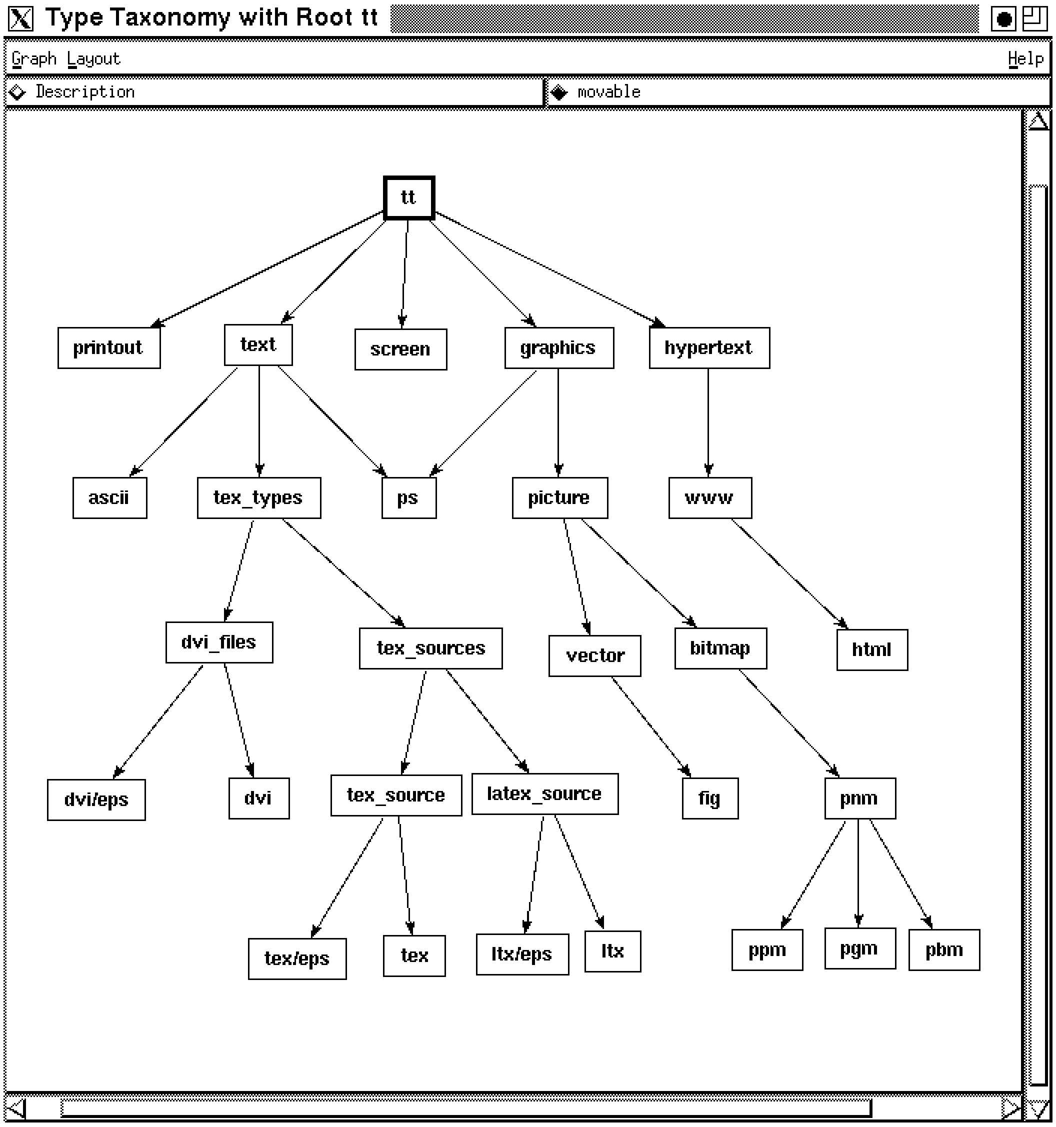


Figure 2: Hypertext browser - the type hierarchy

3 The Metalevel Specification Language

The DaCapo system is designed to synthesize the set of solutions to specifications given in terms of global and local constraints. These constraints can

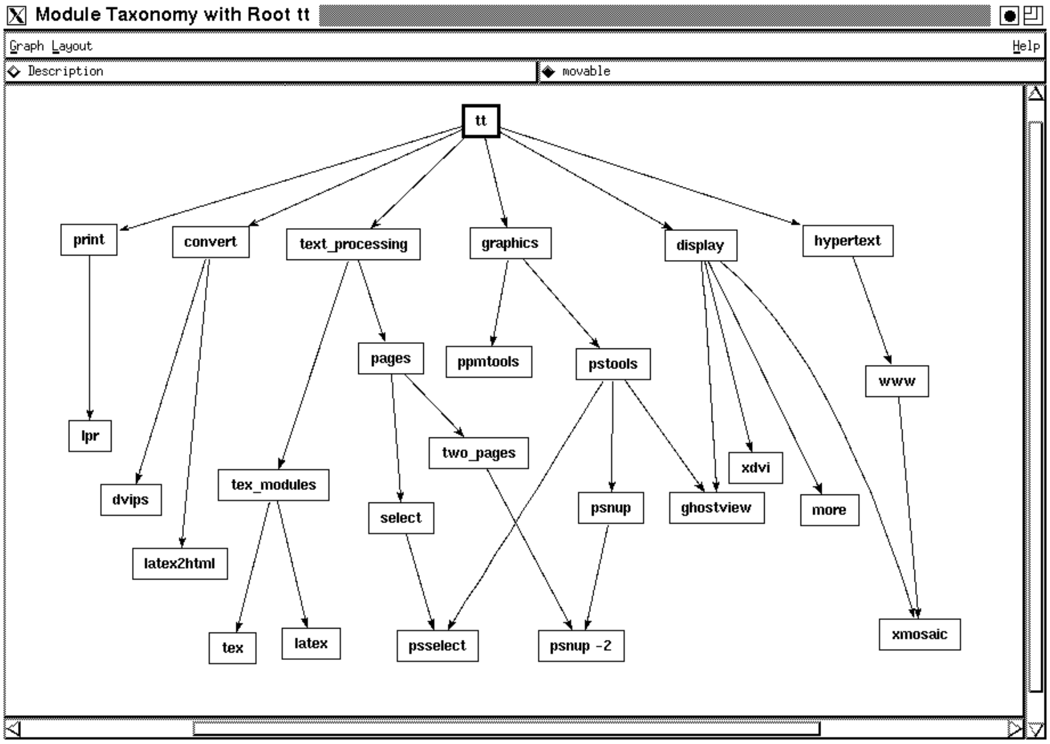


Figure 3: Hypertext browser - the module hierarchy

uniformly and elegantly be formulated within the temporal logic SLTL (Semantic Linear Time Logic), which combines *type*, *module*, and *ordering* constraints to a specification language with a coherent semantics. Thus more precisely, given an SLTL specification, the DaCapo system determines the set of all legal program compositions satisfying these constraints.

In practice, the set of all solutions is often very large. Thus it is convenient to define some default constraints in order to keep the set of delivered solutions manageable. In our system, *minimality* of the solutions is taken as such a default constraint. This choice reflects the desire for simple solutions and it does not affect the solvability of the specification.

The following two subsections present our languages for expressing *local* constraints capturing interfacing and module specifications, and *global* constraints restricting the ordering between modules.

3.1 Local Constraints

Local constraints are the selection criteria for single components. Currently they are based on a taxonomic classification of the repository components and the responsibility of an appropriate classification is left to the designer of the taxonomy. This choice does not impose any limitation, as our approach is parameterized in the local constraint mechanisms. In particular, the taxonomic classification of components described by Prieto-Diaz [Prie91, PrFr87], which can be found in most software library systems, could be easily integrated in our system.

Interface constraints describe the 'neighbourhood' of programs. In our UNIX application we consider type compatibility as the only criterion: Types are organized into a taxonomy according to their hierarchical classification (see Figure 2). A *type constraint* is formulated via simple logic formulas over the type taxonomy, and is interpreted in terms of sets of basic types. E.g. the formula $\mathsf{type}((text\ \mathsf{or}\ html)\ \mathsf{and\ not}\ ps)$ is satisfied by the basic types

$$ascii, dvi, dvi/eps, html, ltx, ltx/eps, tex, tex/eps$$

Module constraints restrict the set of legal elementary programs which may appear in the solution. Like types, also the modules are organized in a taxonomic hierarchy (see Figure 3), and module constraints are represented by simple logic formulas over the module taxonomy that specify sets of elementary components. E.g. the formula module(`display` and not `hypertext`) is satisfied by the elementary commands

`ghostview`, `more`, `xdvi`

In general the full power of the deductive database language can be used to express local constraints. It is therefore possible to incorporate any sophisticated classification scheme and taxonomic retrieval into our system.

3.2 Global Constraints: The Temporal Aspect

Ordering constraints allow users to specify a relative ordering between (some of) the components that form a solution. Using a modal logic as a framework, we are able to extend the two local kinds of specifications to an elegant and uniform specification language for global constraints, which is defined as follows:[1]

Definition 3.1 (SLTL)

The syntactic structure of Semantic Linear Time Logic (SLTL) is given by:

$$\begin{aligned}\Phi ::= \;& \mathtt{tt} \mid \mathsf{type}(C_{Type}) \mid \mathsf{not}\ \Phi \mid \Phi\ \mathsf{and}\ \Phi \mid \Phi\ \mathsf{or}\ \Phi \mid \\ & \langle C_{Mod} \rangle\ \Phi \mid \mathsf{generally}\ \Phi \mid \Phi\ \mathsf{until}\ \Psi\end{aligned}$$

where C_{Type} and C_{Mod} represent type and module constraints, respectively. □

SLTL formulas are interpreted over the set of legal program compositions. Intuitively,

- `tt` is satisfied by every program composition.

[1] The specification language can be seen as an 'interpreted version' of Propositional Linear Temporal Logic (PLTL) [Emer90, SiCl85] or a linear time variant of the temporal Logic CTL [ClES86].

- $\mathsf{type}(C_{Type})$ is satisfied by every program composition whose first component satisfies the type constraint C_{Type}[2].

- Conjunction, disjunction, and negation are interpreted in the usual fashion.

- $\mathsf{<}C_{Mod}\mathsf{>}\ \Phi$ is satisfied by program compositions whose first component satisfies C_{Mod} and whose *continuation*[3] satisfies Φ. In particular, $\mathsf{<.>}\ \Phi$ is satisfied by every program composition whose continuation satisfies Φ.

- generally Φ requires that Φ is satisfied for every postfix[4] of the program composition. As we are dealing with finite paths only, certain properties of this kind will never hold. Still, this operator is useful to express global type constraints.

- Φ until Ψ expresses that the property Φ holds for all programs of the program composition, until we reach a position where the corresponding continuation satisfies the property Ψ. Note that Φ until Ψ guarantees that the property Ψ holds eventually (strong until).

The on-line introduction of **derived operators** allows a modular and intuitively easy to understand formulation of complex properties. In fact, as can be seen in Figures 4 and 6, our built-in macro facility allows natural language-like specifications. Here we only introduce some convenient derived operators, which are illustrated in Section 4:

$$\begin{array}{lllll} \textit{Eventually}: & \mathsf{eventually}\ \Phi & =_{df} & \mathtt{tt}\ \ \mathsf{until}\ \ \Phi \\ \textit{Before}: & \Phi\ \ \mathsf{before}\ \ \Psi & =_{df} & (\neg\Psi\ \ \mathsf{until}\ \ \Phi)\ \ \mathsf{and}\ \ \mathsf{eventually}\ \ \Psi \end{array}$$

While the eventually operator requires that Φ will hold at some later point, Φ before Ψ guarantees that Ψ will become valid after Φ has been satisfied but not earlier.

4 The DaCapo Lifecycle

We illustrate the DaCapo lifecycle, sketched in Figure 1, by means of a typical user session.
Starting with the following loosely specified task

> *Display pages 3 and 4 of a latex source in two-page format on the screen*

we are going to synthesize a corresponding custom command from the library of basic commands and applications introduced in Section 2.

---

[2]Observe that taxonomies are always completed with a top node.
[3]The continuation is simply the program composition starting from the second module
[4]'Postfix' is a straightforward generalization of continuation.

The specification Already the following still natural-language like reformulation of the informal specification is understood by the system (cf. also Figure 4 top right).

```
type latex_source,
     use two_pages,
     use select
         and finally module display
```

Internally, it is transformed into the following SLTL formula

```
type(latex_source)
    and (tt U <two_pages>tt)
    and (tt U <select>tt)
    and (tt U <display>tt)
```

The solution space Given this formula, the system returns the set of four alternative program compositions in graph representation: every path from the start to the success node in the graph of Figure 4 (left) is a possible solution to our synthesis problem. Solutions can be investigated by means of the hypertext browser, which provides information about single modules and types as well as about the type and module taxonomies. The hypertext system offers a card for each node or edge of the graph. As an example, the actual high-level language implementation of the call to the module latex is reported in Figure 4 (right), together with a textual description, which may be complemented by typical examples.

The execution All the proposed solutions are directly executable. In fact, each path in the graph corresponds to a target program in our high level programming language, which is implemented on top of C++. In order to invoke a solution, its corresponding path must be selected by means of the mouse. The system offers the option for execution, as soon as the selection is unambiguous, cf. Figure 5 (front). The chosen solution

(*ltx*, `latex`, *dvi*, `dvips`, *ps*, `psnup -2`, *ps*, `psselect`, *ps*, `ghostview`, *screen*)

which requires an interactive selection of the desired pages as in Figure 5 (upper right), delivers the result shown in Figure 5 (back).

Modifying the query In the *prototyping* phase, the user approaches the desired solution by successively refining or relaxing an initial (usually loose) specification. In fact, the output in Figure 5 reveals that the selection was done on the *double-sided* document[5]. This is unsatisfactory, since it does not reflect the obvious intent, which refers to pages 3 to 4 of the *original* document.

[5]You may recognize that we use *this* paper as source file

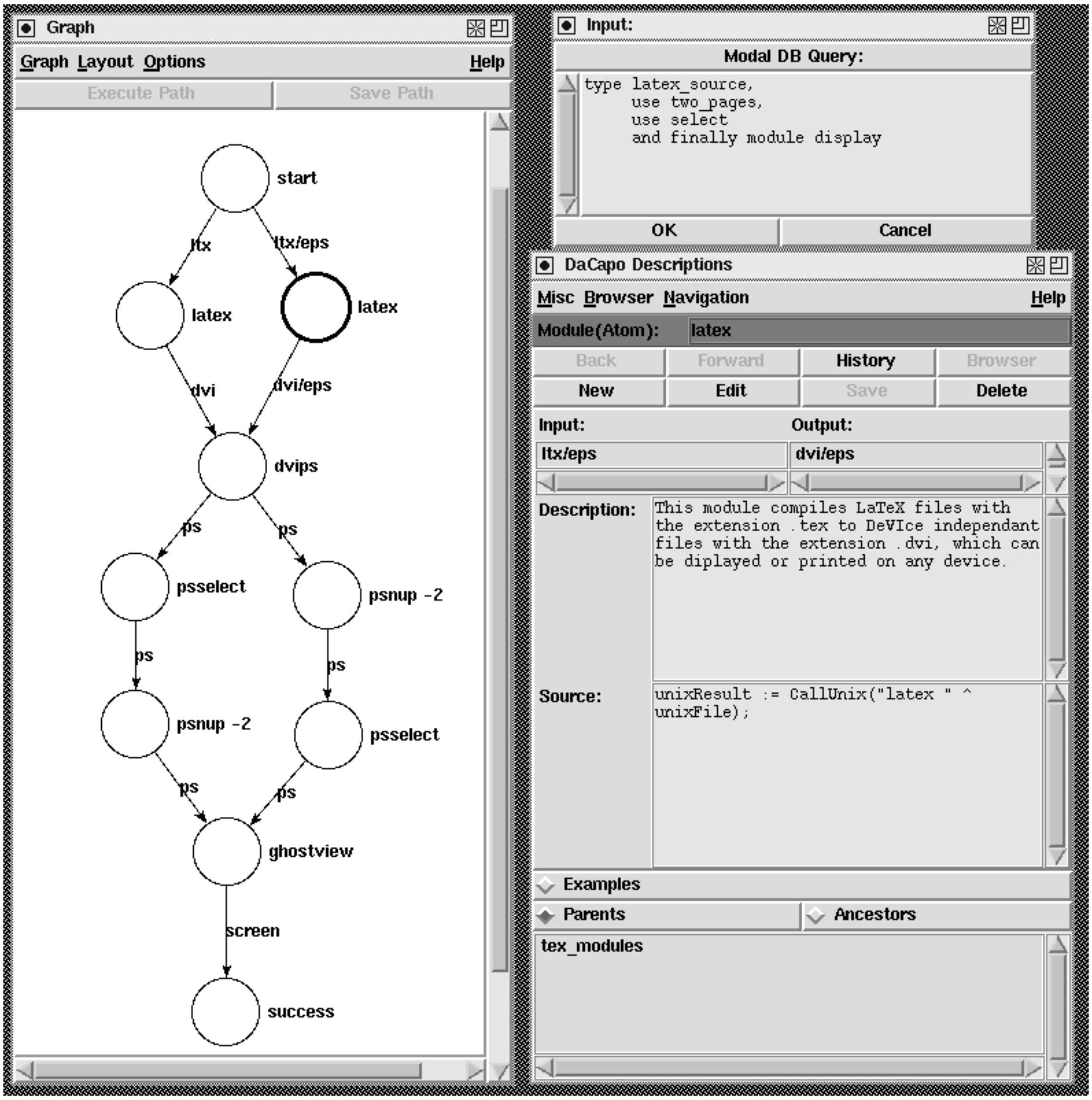


Figure 4: The solution graph

This intuition can be made precise in SLTL by means of ordering constraints requiring the page selection to happen before the two-page formatting:

```
type latex_source,
     module select before module two_pages
     and finally module display
```

as shown in Figure 6 (top right).

Figure 6 (left) shows the solution space corresponding to the modified query, which only contains the correct sequencing (the selection precedes the twopage formatting). Thus the execution of the highlighted path provides the desired result (see Figure 7).

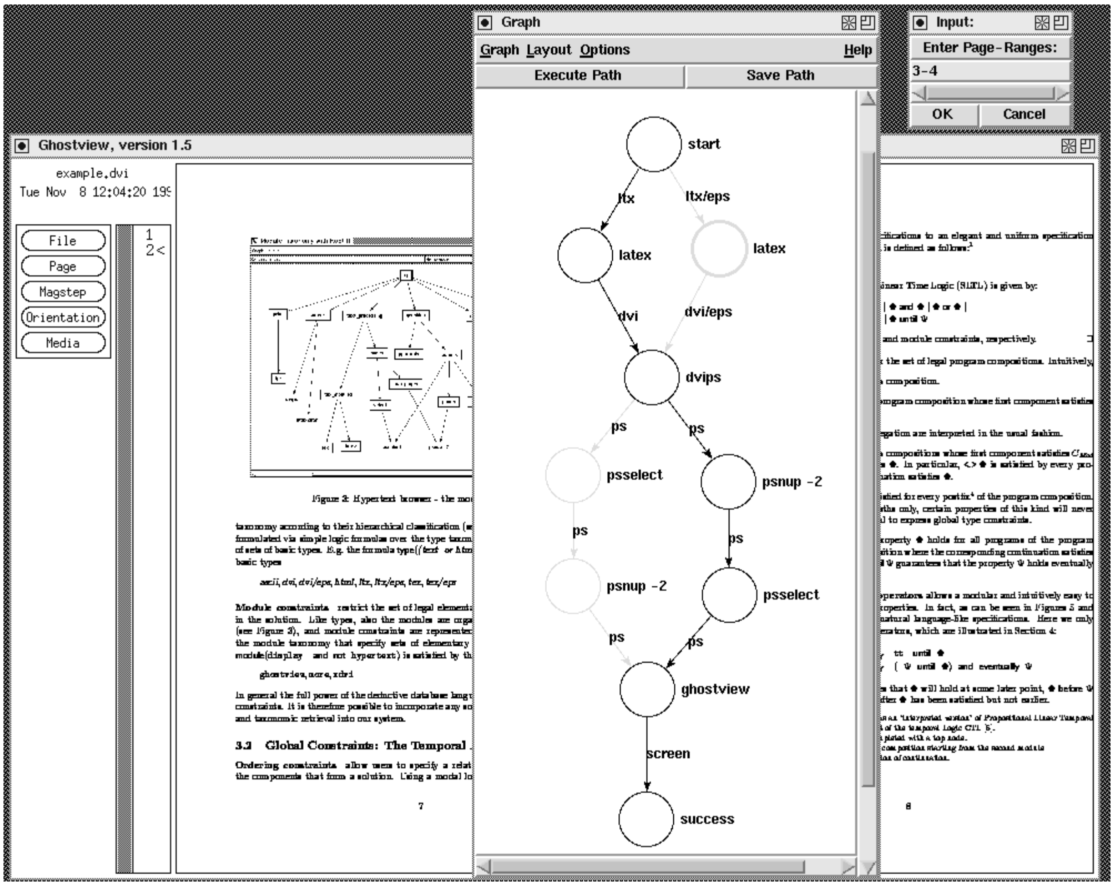


Figure 5: The executed program

The compilation Satisfactory solutions can be made persistent through compilation into a new module that can be saved in the component repository for later reuse. These repository updates are supported by the hypertext system by automatically associating with each new module a new card, which by default contains appropriately precomputed taxonomic criteria and a short textual description as shown in Figure 6 (right). These criteria are generated by the system from the corresponding criteria of its components. Thus the consistency of the taxonomies is automatically maintained. The user only needs to provide a new name, e.g. print_2_latex in this case, then the repository and taxonomy updates follow completely automatically. Of course, the proposed content of the new hypertext card can be also modified interactively.

Figure 7 shows the effect of resubmitting the specification to the updated repository. The execution of the new solution

$$(\texttt{ltx}, \texttt{print\_2\_latex}, \texttt{screen})$$

is identical to the one for its defining path.

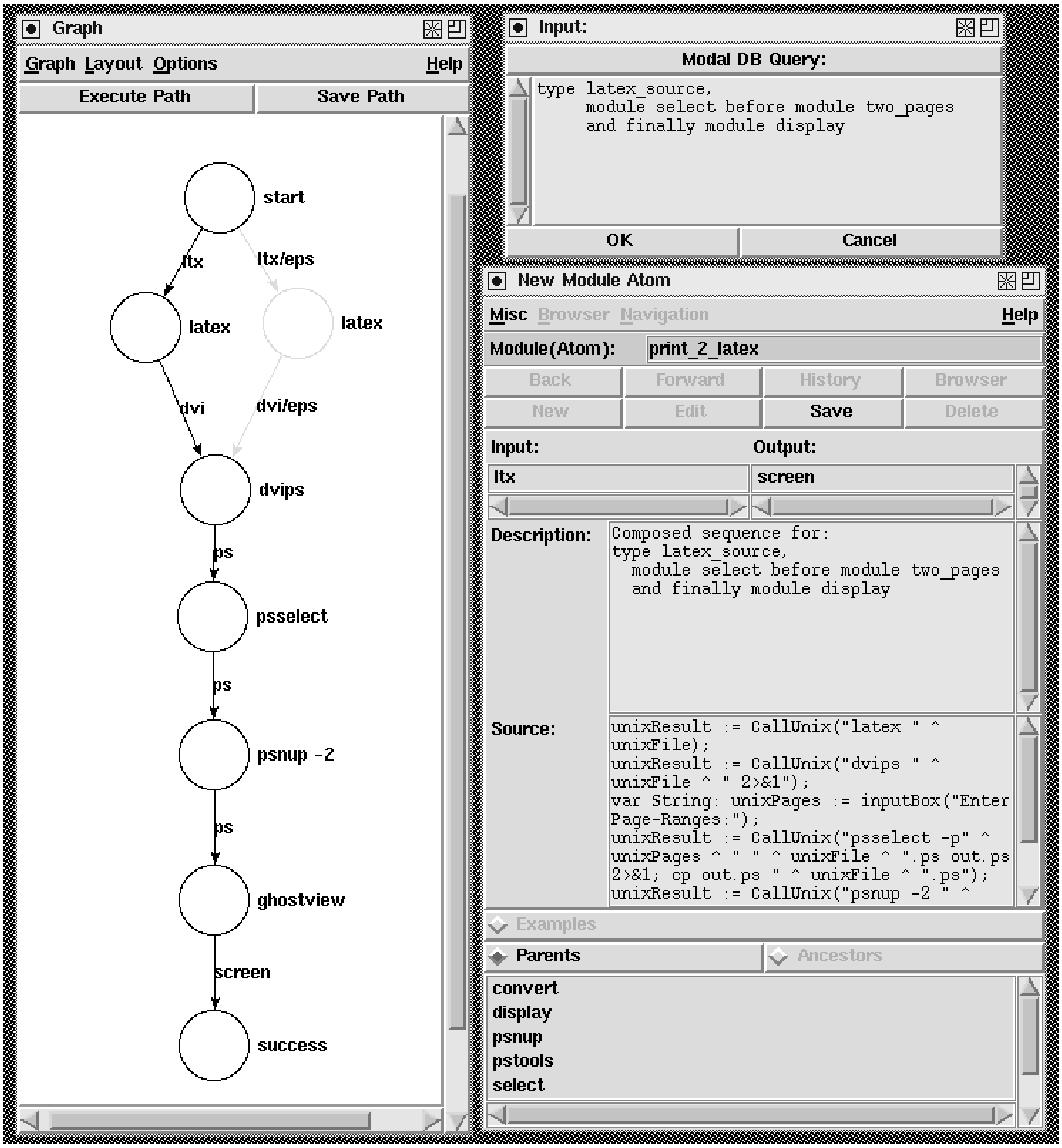


Figure 6: The modified solution graph

5 Comparison with Related Work

Most software library systems described in the literature are **local**, in the sense that they help retrieving single components. E.g.:

- LaSSIE [DBSB91] uses the knowledge representation language KANDOR to store information about a large software project in order to provide semantic retrieval. A taxonomy of classes is induced by a frame-based representation of objects.

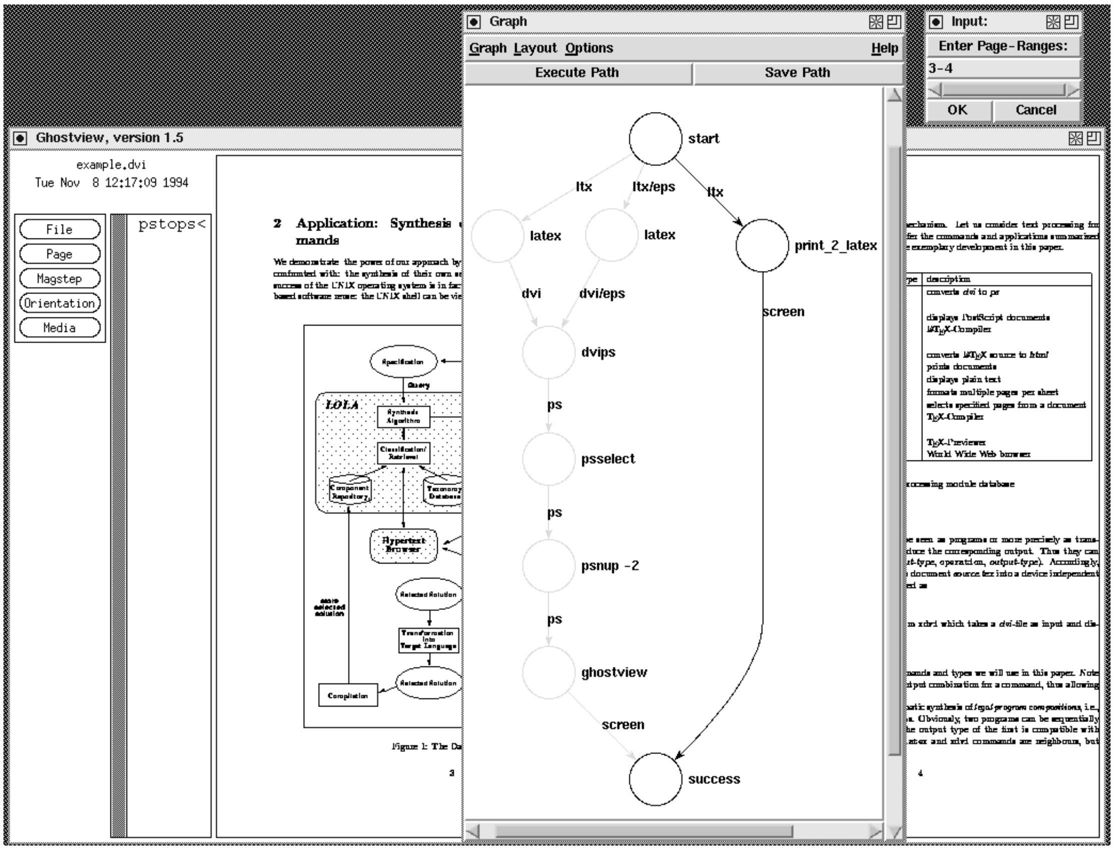


Figure 7: The modified execution

- The software engineering environment NORA [SFGK94] is based on logical inference and supports specification-driven single component retrieval.
- Other systems share a core constituted by the knowledge representation language Telos [MBJK90] and the deductive object-oriented database system ConceptBase [Jark91]:
  - In the ITHACA Environment [FuNP92], an application frame contains application classes with a functional component description in terms of a list of operations and information about the meta-level classes with design knowledge describing the use of components. Component selection is based on similarity in the spirit of Prieto-Diaz' work [Prie91, PrFr87].
  - A logic-based software configuration management system is presented in [JJMG92] using the Telos data model for the representation of configuration information. Given a system interface, the search for an implementation tries to reuse existing subsystems and components if possible. Otherwise, proposals are generated applying the deductive and abductive techniques of ConceptBase.

- An approach based on algebraic specifications and refinement is described in [ChHJ93]. Given an algebraic specification of the desired component in Telos, ConceptBase is used to retrieve software components with matching signatures.

The point of the DaCapo system, however, is its synthesis capability according to **global** constraints. We know only of two other systems with this feature:

- The system reported by Batory and O'Malley [BaMa92] constructs hierarchical software systems from reusable components using grammar rules for high level specification. Their approach strongly relies on structural restrictions of the underlying environment like open architecture software and a strict interface discipline.
- Möhring, Müller and Radermacher, [MöMR94], propose the idea to use scheduling methods to synthesise implementations satisfying precedence constraints. However, they do not provide an appropriate specification language or hints towards its intended expressive power.

The DaCapo system is based on an expressive, well-structured and more intuitive specification language than the first system, and it uses an, in some sense dynamic, scheduling algorithm for synthesis, which varies its search strategy according to the stage of the synthesis.

6 Conclusions

We have presented the DaCapo software library system that supports the *automatic construction of software* from existing components according to a (possibly incomplete) specification. Based on a specification language that uniformly combines taxonomic component specifications, interface conditions, and ordering constraints, our method adds a global view to conventional single component retrieval. Moreover, following a user session, we have illustrated the DaCapo software lifecycle management, which supports the inclusion of a satisfactory new software component into the repository by proposing an appropriately precomputed default taxonomic classification. This guarantees convenient retrieval for later use.

Acknowledgements

We are grateful to Achim Dannecker, Carsten Friedrich, Andreas Holzmann, Dirk Koschützki, Gerald Lüttgen, Falk Schreiber, Matthias Seul, and Barbara Zündorf for their constructive cooperation in the design and implementation of the DaCapo system.

References

[BaMa92] D. Batory, S. O'Malley. The design and implementation of hierachical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, Oct. 1992.

[BuSt92] O. Burkart, B. Steffen. Model checking for context–free processes. In *3rd Int. Conf. on Concurrency Theory (CONCUR '92)*, LNCS Vol. 630, Springer–Verlag, 1992.

[BuSt94] O. Burkart, B. Steffen. Pushdown Processes: Parallel Composition and Model Checking. In *5th Int. Conf. on Concurrency Theory (CONCUR '94)*, Uppsala (Sweden) LNCS Vol. , Springer–Verlag, 1994.

[ChHJ93] P. S. Chen, R. Hennicker,, M. Jarke. On the retrieval of reusable software components. *Advances in Software Reuse - Selected Papers from the 2nd Int. Workshop on Software Reusability, Lucca, Italy*, pp. 99–107. IEEE Computer Society, 1993.

[ClES86] E. Clarke, E. Emerson, A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[ClPS93] R. Cleaveland, J. G. Parrow, B. Steffen. The concurrency workbench: A semantics-based verification tool for finite state systems. *ACM Transactions on Programming Languages and Systems*, Jan. 1993.

[DBSB91] P. Devanbu, R. J. Brachman, P. G. Selfridge, B. W. Ballard. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34(5):35–49, May 1991.

[Emer90] E. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pp. 995–1072. Elsevier, 1990.

[Frei94] B. Freitag. A hypertext-based tool for large scale software reuse. *Proc. 6th International Conference on Advanced Information Systems Engineering, CAiSE'94, Utrecht, The Netherlands, June 1994, LNCS* Vol. 811, pp. 283–296, Berlin, 1994. Springer-Verlag.

[FrKe94] B. Freitag, J. Kempe. Multidatabase access with the *LOLA* system. Technical Report TUM-I9416, Technische Universität München, Munich, Germany, 1994.

[FrMS94] B. Freitag, T. Margaria, B. Steffen. A pragmatic approach to software synthesis. In *Proc. ACM SIGPLAN POPL'94 Post-Conference Workshop on Interface Definition Languages*, Portland, Oregon, Jan. 1994. (ACM SIGPLAN Notices, 1994).

[FrSS91] B. Freitag, H. Schütz, G. Specht. *LOLA* - a logic language for deductive databases and its implementation. *Proc. 2nd Intl. Symp. on Database Systems for Advanced Applications (DASFAA '91)*, pp. 216 – 225, Tokyo, 1991.

[FuNP92] M. Fugini, O. Nierstrasz, B. Pernici. Application development through reuse: the ITHACA tools environment. *ACM SIGOIS Bulletin*, 13(2):38–47, Aug. 1992.

[Jark91] M. Jarke. ConceptBase V3.0 User Manual. Technical Report MIP-9106, Universität Passau, 1991.

[JJMG92] M. Jarke, M. Jeusfeld, A. Miethsam, M. Gocek. Towards a logic-based reconstruction of software configuration management. In *Proc. of the seventh Knowledge-Based Software Engineering Conference, McLean, Virginia*, pp. 132–141. IEEE Computer Society Press, 1992.

[KaRT89] S. Katz, C. Richter, K. The. PARIS: A system for reusing partially interpreted schemas. In *Software Reusability: Volume I — Concepts and Models*, Frontier Series, chapt. 10, pp. 257–273. ACM Press, New York, 1989.

[KnRS92] J. Knoop, O. Rüthing, B. Steffen. Lazy code motion. *Proc. PLDI '92, ACM SIGPLAN Notices*, Vol.27, pp. 224 - 234, San Francisco, June 1992.

[KnRS94a] J. Knoop, O. Rüthing, B. Steffen. Partial Dead Code Elimination, *Proc. SIGPLAN PLDI Conf.'94, ACM SIGPLAN Notices*, Orlando (FL), June 1994.

[KnRS94b] J. Knoop, O. Rüthing, B. Steffen. *Optimal Code Motion: Theory and Practice. ACM Transactions on Programming Languages and Systems*, Vol. 16, N.4, July 1994, pp.1117-1155.

[MiMM94] A. Mili, R. Mili, R. Mittermeier. Storing and retrieving software components: A refinement based system. *Proc. 16th International Conference on Software Engineering, Sorrento, Italy.* IEEE Computer Society, 1994.

[MöMR94] R. H. Möhring, R. Müller, F. J. Radermacher. Advanced DSS for scheduling: Software engineering aspects and the role of eigenmodels. In *Proc. 27th Annual Hawaii Int. Conf. on System Sciences, Maui, Hawaii*, 1994.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis. Telos – representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, Oct. 1990.

[Oust94] John K. Ousterhout. TCL and the TK Toolkit. Addison–Wesley, April 1994.

[Prie91] R. Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5), May 1991.

[PrFr87] R. Prieto-Diaz, P. Freeman. Classifying software for reusibility. *IEEE Software*, 4(1):6–16, Jan. 1987.

[SiCl85] A. P. Sistla, E. M. Clarke. The complexity of the propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, July 1985.

[SFGK94] G. Snelting, B. Fischer, F. J. Grosch, M. Kievernagel, A. Zeller. Die inferenzbasierte Softwareentwicklungsumgebung NORA. *Informatik Forschung und Entwicklung*, 9(3), Aug. 1994. (In German).

[Stef93] B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, N. 21, pp. 115-139, 1993. (a shorter version appeared in the*Proc. TACS '91*, Sendai, Japan, Sept. 1991, *LNCS* N. 526, pp. 346–364, Springer–Verlag.)

Demonstration of the B-Toolkit

Ib Holm Sørensen

B-Core Limited
Magdalen Centre, Oxford
England
Email: Ib.Sorensen@comlab.ox.ac.uk

The B-Toolkit is a suite of integrated programs which implement the B-Method for Software Development. The B-Method is a collection of formal techniques which give a basis to those activities of Software Development that range from technical software specification, through design and integration, to code generation and into maintenance. The B-Method and the specification language AMN (Abstract Machine Notation) are in many respects similar to other "model oriented" formal methods. They employ a conventional "pseudo" programming style. The B-Tool is a language interpreter for the B Theory Language. This language is a special purpose language for writing interactive and automatic proof assistants and other systems where pattern matching, substitution and re-write mechanisms can be used. The B-Toolkit's component tools are implemented in the B Theory Language and is interpreted by the B-Tool.

Description

The B-Toolkit, which supports the B-Method, underwent eight years of research and development at the Programming Research Group in Oxford and British Petroleum Research, and its commercial development is now continuing inside B-Core (UK) Ltd in Oxford. The B-Toolkit is an integrated suite of computer programs, built partly on the B-Tool interpreter, and covers many aspects of software engineering, including:

a) Syntax and type-checking of specification documents as well as low level design documents, with comprehensive error reporting .

b) Verification condition generation (which generates the proof-obligations needed to guarantee specification consistency and correctness of refinement). The refinement rules, which originate from Oxford (Hoare, Sanders, He) and J-R Abrial's formulation in terms of Predicate Transformers, are used within the tool.

c) Automatic & interactive provers for discharging the verification conditions. (Other provers can be used in conjunction with B-Core (UK)'s B-Toolkit). Specification animation, enabling the specification to be "run", validated and tested; pre-conditions, guards for conditional statements, and the values of local and state variables may be inspected during animation.)

d) A translator for translating low level design documents into C.

e) Code generation from declarative descriptions, facilitated by a re-usable library of code modules.

f) A Library of reusable code modules, which are all accessed and used according to their given specification.

g) Rapid prototyping, facilitated by an interface generator, built on the reusable library.

h) Automatic Markup and Indexing of documentation of complete developments (requires LaTeX).

Availability

The B-Toolkit is commercially available in a Trial version or through full release licences. The B-Toolkit is delivered for IBM/RS6000 running AIX and Sun Sparc running SunOS 4.1.x. (5.x is also supported). The Toolkit requires 16Mbytes of RAM and approximately 20Mbytes of Disc. The B-Tookit is highly portable, and will run under most Unix systems implementations. The current B-Toolkit requires as a minimum a Unix Operating System, and for full functionality it further requires LaTeX and a C-Compiler.

Recent Publications in the BRICS Notes Series

NS-94-6 **Uffe H. Engberg, Kim G. Larsen, and Peter D. Mosses, editors.** ***Proceedings of the 6th Nordic Workshop on Programming Theory*** **(Aarhus, Denmark, 17–19 October, 1994), December 1994. v+483pp.**

NS-94-5 **Andrew M. Pitts.** ***Some Notes on Inductive and Co-Inductive Techniques in the Semantics of Functional Programs,*** **DRAFT VERSION. December 1994. vi+135 pp.**

NS-94-4 **Peter D. Mosses, editor.** ***Abstracts of the 6th Nordic Workshop on PROGRAMMING THEORY*** **(Aarhus, Denmark, 17–19 October, 1994), October 1994. v+52 pp.**

NS-94-3 **Sven Skyum, editor.** ***Complexity Theory: Present and Future*** **(Aarhus, Denmark, 15–18 August, 1994), September 1994. v+213 pp.**

NS-94-2 **David Basin.** ***Induction Based on Rippling and Proof Planning. Mini-Course.*** **August 1994. 62 pp.**

NS-94-1 **Peter D. Mosses, editor.** ***Proc. 1st International Workshop on Action Semantics*** **(Edinburgh, 14 April, 1994), May 1994. 145 pp.**