Abstracts of the 6th Nordic Workshop on PROGRAMMING THEORY

17–19 October 1994, Aarhus, Denmark

Peter D. Mosses (editor)

BRICS Notes Series NS-94-4

ISSN 0909-3206 October 1994

Abstracts of the 6th Nordic Workshop on

PROGRAMMING THEORY

17–19 October 1994 — Aarhus, Denmark

Peter D. Mosses (editor)

Foreword

The main objective of this 6th Nordic Workshop on Programming Theory is to bring together researchers from the Nordic and Baltic countries, in order to improve mutual contacts and cooperation. The 63 registered participants come from: Norway (3), Sweden (11), Finland (5), Latvia (1), Lithuania (2), Estonia (1), England (3), Germany (4), Denmark (33).

Presentations:
The following invited speakers are to give 60-minute presentations in plenary sessions: Matthew Hennessy (University of Sussex), Bernhard Steffen (Universität Passau), and Ib Holm Sørensen (B-Core (UK) Limited). The remainder of the workshop consists mainly of 30-minute presentations, selected on the basis of submitted abstracts, in parallel sessions. Moreover, four systems are to be demonstrated, all closely related to talks to be given during the workshop.

Proceedings:
Selected participants, based on the quality and topic of the presentations at the workshop, will be invited to submit a full paper after the workshop to the Nordic Journal of Computing.

Acknowledgements:
The 6th Nordic Workshop is financially supported by grants from BRICS[1] and the Danish Science Research Council. Technical and administrative support is 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

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

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

Contents

Higher-Order Processes and Their Models

by

Matthew Hennessy
University of Sussex

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, Germany
`steffen@fmi.uni-passau.de`

Automated verification often relies on some kind of temporal logic as specification language for systems and system properties. The modal mu-calculus is a particularly flexible representative: not only may a number of other temporal logics be translated into it, but it may also be used to encode various behavioral equivalences and preorders for finite state systems. Moreover, it is a well-structured and convenient specification language for other application areas where global constraints are of interest, like e.g. dataflow analysis or software configuration. In fact, the modal mu-calculus uniformly supports algebraic, operational, and logic-based approaches to verification. Model checking, the central automatic verification technique associated with the modal mu-calculus, is now well-established for finite-state system behaviors. More recent is the development of techniques for infinite-state behaviours: Bradfield and Stirling observed that tableaux-based model-checking covers general infinite-state systems, but their method is not effective. Muller and Schupp proved that the Monadic Second Order Logic is decidable for pushdown transition graphs, which are a strict generalization of context-free processes with respect to bisimulation semantics. This implies the decidability of the corresponding model checking problem for the full mu-calculus, but their decision procedure is non-elementary and thus not applicable to practical problems. More recently, practice-oriented model checking algorithms were developed together with Burkart and Hungar for the alternation-free fragment of the modal mu-calculus, which e.g. form the basis for the generation of efficient interprocedural dataflow analysis algorithms from modal logic specifications.

This presentation focusses on the structure of iterative model checking techniques depending on the generality of the model and the version of the modal mu-calculus considered, together with their potential for optimization. The first part incrementally develops an iterative algorithm for finite-state processes covering the full mu-calculus, by successively extending an algorithm for uniform fixpoints to hierarchical and to alternating fixpoints. The second part generalizes this development for context-free or pushdown processes. The key to this generalization is its second order approach: rather than determining *properties* for the *states* of a finite-state system, we compute *property transformers* for the specific incomplete portions (*fragments*) of the considered pushdown process, which describe the set of subformulas valid at the start state of a fragment relative to the set of subformulas valid at the end states of the fragment. We will see that this approach carries through all the way, although the correctness of the model checker for the full mu-calculus obtained in this fashion is still a conjecture. The development of the various algorithms is illustrated in the context of data flow analysis, while focussing on their differences, complexity, and optimization potential.

The B-Technologies : A system for computer aided programming

D.S. Neilson and I.H.Sorensen

Abstract

The paper introduces to the B-Technologies, a mathematically based formal method and a tool-set 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. *B-Method* and the *B-Toolkit* are described in the 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 and a set of software development tools: an AMN type checker, a specification animator and code generators. All tools are integrated with the proof assistants into a window-based development environment.

On decidability of simulation and bisimulation for Lossy Channel Systems

Parosh Abdullah **Mats Kindahl**

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 both weak and strong simulation preorder and prove their correctness. Furthermore, we show that weak bisimulation equivalence is undecidable.

Cpo-Models for GSOS Languages

Luca Aceto
BRICS, Aalborg University Centre
9220 Aalborg, Denmark

The meta-theory of process description languages like CCS, ACP and CSP has recently been the object of considerable research effort in the literature. So far, this line of research has produced a wealth of results which generalize and explain many of the congruence theorems and complete axiomatizations for behavioural equivalences that have been proposed in the literature. Most of this work is entirely based upon operational semantics, following a bias towards operational methods in process theory that dates back to Milner's original development of the theory of CCS. Notable exceptions to this trend are, among others, Abramsky's work on a domain equation for synchronization trees that leads to a fully abstract semantics for SCCS, and Ingolfsdottir's extension of Abramsky's model to a form of value-passing CCS.

In this talk, I plan to present a general way of giving denotational semantics to 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.

In the first part of the talk, I plan to show how to use GSOS rules to associate a labelled transition system with divergence information with each GSOS language. This will be done in such a way that the bisimulation preorder of Hennessy and Plotkin is a precongruence with respect to all the operators in the language. In order to obtain the aforementioned substitutivity result, 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. I believe that this is a natural choice and I shall argue for it by means of examples.

I shall then show how to automatically give a denotational semantics for a GSOS language in terms of Abramsky's domain of synchronization trees. To this end, it is sufficient to endow Abramsky's model with an appropriate continuous algebra structure. This I do by showing how the GSOS rules defining the operational semantics of an operation f of the calculus can be used to define a continuous function F over the domain of synchronization trees.

As a result of the general framework, I shall then show that the denotational semantics so obtained is guaranteed to be in complete agreement with the chosen behavioural semantics. More precisely, the denotational semantics produced by the general approach presented in the talk is always fully abstract with respect to the finitary part of the bisimulation preorder.

This is joint work with Anna Ingólfsdóttir.

Type and Behaviour Reconstruction

Torben Amtoft
Aarhus University
Dept. of Computer Science
Aarhus, Denmark
Email: tamtoft@daimi.aau.dk

The topic of this talk will be how to design an algorithm for gaining information about the type and the behaviour of CML programs, and how to prove this algorithm sound and complete wrt. the inference system presented by the Nielson's in POPL'94.

The algorithm is an extension of the standard algorithm W, collecting a set of constraints on behaviours. Due to the laws imposed on behaviours these do not constitute a free algebra, and a consequence of this is that there seems to be no notion of "principal solutions". In the presense of let-polymorphism, this complicates matters significantly.

The talk describes ongoing work.

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@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.

The talk will 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 will be on the actual specification of the instrument and the technical results that have been obtained for PMC will only be touched briefly.

Specification and Verification of Real–Time Systems using Timed Modal Logic

Jørgen Hedegaard Andersen
BRICS
Aalborg University

There has been done a lot of work in developing formalisms for describing real-time systems. A lot of the efforts have concentrated on extending already existing algebras such as CCS and CSP with realtime. Such algebras have their strength in detailed descriptions of the communication patterns. However, verification of these systems can be quite complicated using miscellaneous equivalence and/or refinement relations. In addition, it is impossible to express liveness properties. Also, process algebraic specifictions tend to become quite extensive and very explicit. What a logic offers in these situations is a degree of loseness so that one can express properties of only parts of a system.

My talk will consist of the following:

- An example introducing Timed CCS.
- Examples of formulas in Timed Modal Logic. Both safety and liveness properties of the previous example are exhibited.
- Implementation of Model Checker for Timed Modal Logic.

A General Framework for Type Inference

Hosein Askari, Ole I. Hougaard, Michael I. Schwartzbach

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 that there exists a uniform conceptual framework for developing type inference algorithms, even for seemingly ad-hoc languages. This framework provides a systematic cookbook methodology for clarifying the concepts and crystallizing the ultimate algorithmic problem that must be solved.

Specifically, we show how a number of important components of the constraint-based approach to type inference each lie on a spectrum that allows considerable generalizations. The components we deal with include types, type equivalences, type variables, type constraints, and polymorphism.

To demonstrate the viability of our framework, we develop a type inference algorithm for a full version of the Turbo Pascal language. For each of the components mentioned above we demonstrate how Turbo Pascal is at one end and the ML language is at the opposite. However, there are really more similarities than differences. The largest gap arises in the final algorithmic problem, which we must solve from scratch.

Exploring Summation and Product Operators in the Refinement Calculus

R.J.R. Back and M.J. Butler
Dept. of Computer Science, Åbo Akademi, Finland

Abstract

Dijkstra introduced weakest-precondition predicate transformers as a means of verifying total correctness properties of sequential programs [3]. In the *refinement calculus* of Back and others, specifications and programs are regarded uniformly as predicate transformers, and refinement laws are derived from properties of predicate transformers [1, 5, 6].

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 [7] and by Martin [4] using category theoretic considerations. This *product* operator combines predicate transformers by forming the cartesian product of their state spaces. We examine the product operator using the higher-order logic formalisation of the refinement calculus of Back & von Wright [2]. 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*) 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 [7] and [4]. 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 dynamic binding, and that when combined with the product operator, provides an elegant model of inheritance in an object-oriented programming langauge. Thus our exploration provides the basis for a calculus of objects and inheritance.

References

[1] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.* Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[2] R.J.R. Back and J. von Wright. Refinement concepts formalised in higher order logic. *Formal Aspects of Computing*, 5:247–272, 1990.

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

[4] C.E. Martin. *Preordered Categories and Predicate Transformers.* D.Phil. Thesis, Programming Research Group, Oxford University, 1991.

[5] C.C. Morgan. *Programming from Specifications.* Prentice–Hall, 1990.

[6] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comp. Prog.*, 9(3):298–306, 1987.

[7] D.A. Naumann. *Two-Categories and Program Structure: Data Types, Refinement Calculi, and Predicate Transformers.* Ph.D. Thesis, University of Texas at Austin, 1992.

From Branching to Linear Metric Domains

Franck van Breugel

Vrije Universiteit
Department of Mathematics and Computer Science
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

Besides partial orders, also metric spaces have turned out to be very useful to give semantics to programming languages (see, e.g., [BR92] for an overview). In the literature, one encounters two main categories of metric domains: linear domains, characterizing trace equivalence, and branching domains, characterizing bisimilarity. The metric linear domains are spaces of subsets of the Baire space. The study of this Baire space belongs to the topological folklore of the twenties. Metric branching domains have been introduced in, e.g., [BZ82], [BZ83], [GR83], and [Bre93].

In the presentation, I will discuss how one can abstract from the branching structure of branching domains arriving at linear domains. For that purpose I will present various linearize operators. These linearize operators will be defined by means of metric labelled transition systems. The theory of metric labelled transition systems has been outlined in [Bre94a] and is further developed in [Bre94b]. One of the key observations needed is that branching domains can be viewed as metric labelled transition systems satisfying some generalized finiteness conditions (the observation that branching domains can be viewed as labelled transition systems seems to originate with [Acz88]). I will also point out that the additional metric structure of metric labelled transition systems (with respect to labelled transition systems) is essential in the development. Various properties of the linearize operators will be discussed (strengthening some of the results of [BBKM84]). Furthermore, we will see that the theory is also applicable to linearize the more involved branching domains—used to model object-oriented and higher-order features—of [Rut90] and [BB93].

At the moment, I am investigating whether these linearize operators give rise to (co)reflections between suitable categories of branching and linear domains (along the lines of [WN94]).

References

[Acz88] P. Aczel. *Non-Well-Founded Sets.* Number 14 in CSLI Lecture Notes. Centre for the Study of Languages and Information, Stanford, 1988.

[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.

[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.

[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.

[Rut90] J.J.M.M. Rutten. Semantic Correctness for a Parallel Object-Oriented Language. *SIAM Journal of Computation*, 19(2):341–383, April 1990.

[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.

A Calculus of Timed Refinement

Kārlis Čerāns

Institute of Mathematics and Computer Science,
University of Latvia, Riga, Latvia,
E-mail: karlis@mii.lu.lv

A Calculus of Timed Refinement (CTR) is a process algebraic theory of loose specifications for real time systems. The basic motivation behind CTR, as well as behind other similar theories (Timed Modal Specifications by Čerāns, Godskesen and Larsen, or Timed Interval CCS by Daniels, and others), is to provide means for expressing formally decisions in design of real time systems, which do not constrain the external behaviour of the described system to a singleton set modulo some behavioural equivalence (e.g. Park's and Milner's bisimulation equivalence, or some other). In particular, it appears important to have a possibility of specifying loosely the quantitative timing constraints controlling the system behaviour.

For a theory of loose specifications to be useful, it requires a well defined notion of specification - implementation relation, or more generally, a refinement relation which desribes when one term of the calculus is a more general specification, than the other. It is natural to require that this refinement obeys certain general mathematical properties (e.g., some substitutivity properties, being a preorder, etc). However, at least equally important is to provide also a "proper" (preferably simple) semantical justification of the presented refinement relation, shedding a light on the pragmatical aspects of the use of the theory.

The CTR is designed with having all abovementioned principles in mind, furthermore postulating a priori adherence to branching time process algebraic semantics (in linear time semantics there is a well developped theory of (live) trace inclusion for timed automata due to Alur, Courcoubetis and Dill, et.al.). Having as a departure point Milner's CCS (and Wang Yi Timed CCS), the main novel construct of CTR is that of time interval prefix: $[a,b].P$, where a,b are natural 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.

We define a timed operational semantics for CTR terms (with the emphasis that it is up to the specification itself to decide internally, when any of its delay prefixes $[a,b]$ is going to expire). Based on this semantics the observational equivalence and refinement relations (named after "continuous bisimulation" and "continuous refinement") are defined and are shown to obey the usual algebraic properties (equivalence/preorder, substitutivity (congruence/pre-congruence)). Both the equivalence and refinement relations are decidable for finite control specifications (which include, e.g., networks of regular specifications).

Furthermore, we show, how to modify the semantics of the already existing formalism of Timed Modal Specifications to obtain a calculus with exactly the same modelling abilities as CTR. We believe, that the new semantics (obtained basically by not allowing modalities for delay transitions) is also "more natural" for TMS (this can be justified by at least one very natural example). Actually, the coincidence of the modelling abilities of the two obtained formalisms allows one to choose his/her preferable notation, still keeping work in the same model. The author believes that the obtained resuts are to some extent clarifying the behaviour of transition modalities in a timed framework.

Further work is to have a stronger focus on pragmatical aspects of the developped calculi, what may include switching to other communication disciplines amongst the components running in parallel in a system. We plan to look also at possibility of incorporating data aspects in our refinement/design calculi, aiming at providing in a long run a general process algebra based user friendly environment for practical design of certified real time systems.

Petri Nets, Traces, and Local Model Checking

Allan Cheng[1]
Computer Science Department, Aarhus University
Ny Munkegade, DK–8000 Aarhus C, Denmark
e-mail:acheng@daimi.aau.dk
Phone: +45 8942 3188 Fax: +45 8942 3255

Abstract

Work on verification of algorithms in the field of multi programming and distributed programming has been presented in e.g. [7, 3, 6, 10, 1, 4]. One important aspect that distinguishes the work in e.g [1] from the one in [10] is that in the former, reasoning about an algorithm is done "by hand", whereas in the latter, verification is performed automatically. Proofs "by hand" can be intellectually much more challenging, demanding, and time consuming than automatic verification. Both approaches have their advantage and disadvantage. E.g.: proofs "by hand" allows one to reason about systems where the number of processes is variable, while automatic verification can mostly only been performed on finite state systems.

Both approaches have no serious problems handling safety properties. However, when it comes to liveness properties, one often needs to take certain fairness assumptions into consideration. In the case of proofs "by hand" one simply takes these assumptions into account when constructing a proof. But in the case of automatic verification matters can become complicated. In [3], for each system being considered specific fairness assumptions are added to the model checking algorithm. In [2], examples of encodings of liveness properties in the propositional modal mu-calculus are given. However these encodings are often hard to understand.

Recently attention has focused on behavioural views of concurrent systems in which concurrency or parallelism is represented explicitly [8, 5, 11, 9, 12]. 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*.

We present a *CTL*-like logic which is interpreted over labeled 1-safe nets. The interpretation reflects the desire to reason about these only with respect their progress fair behaviours. It turns out that Mazurkiewicz trace theory provides a useful setting. Hence, the explicit notion of independence between transitions enables us to incorporate fair progress assumptions in a *uniform* way. No encodings as formulas are needed.

We provide a set of proof rules and prove soundness and completeness with respect to the given interpretation of our logic. As a result we obtain a local model checker, that is: automatic verification (of finite state systems) taking fair progress into account.

keywords: automatic verification, fair progress, labeled 1-safe nets, local model checking, maximal traces, partial orders, inevitability

References

[1] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice-Hall, 1990.

[2] Julian Bradfield. *Verifying Temporal Properties of Systems with Applications to Petri Nets.* PhD thesis, The University of Edinburgh, 1991. PhD in computer science, report CST-83-91.

[3] 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.

[4] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1992.

[5] Antoni Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 279–324. Springer-Verlag (*LNCS* 255), 1986.

[6] Robin Milner. *Communication and Concurrency.* Prentice Hall International Series In Computer Science, C. A. R. Hoare series editor, 1989.

[7] Susan S. Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):531–537, 1976.

[8] Wolfgang Reisig. *Petri Nets – An Introduction.* EATCS Monographs in Computer Science Vol.4, 1985.

[9] Eugene W. Stark. Concurrent transition systems. *Theoretical Computer Science*, 64():221–269, 1989.

[10] 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.

[11] Glynn Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 325–390. Springer-Verlag (*LNCS* 255), 1986.

[12] 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.

[1]This work has been supported by The Danish Research Councils.

Introduction to ALF - a Background

Catarina Coquand
Chalmers Technical University
Inst. for Computer Science
Göteborg, Sweden
Email: catarina@cs.chalmers.se

We will give an introduction to type theory, focusing 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.

The Comparison of two Approaches to Separate of an Algorithm's Data Dependency from its Computational Aspect

Vytautas Cyras* and Magne Haveraaen**
*Vilnius University, Lithuania, Email: Vytautas.Cyras@maf.vu.lt
**University of Bergen, Norway, Email: Magne.Haveraaen@ii.uib.no

The data dependency pattern of the computation is essential when expressing solvers for general recurrences. The comparison of two approaches - Structural Blanks and Constructive Recursion - is presented.

The approach of Structural Blanks was first presented by Greshnev, Lyubimskii and Cyras in 1985, and is investigated currently by Cyras. This approach was developed to express solutions to general recurrences as reusable program components. The approach distinguishes between structural components (S-modules) and functional components (F-modules). Examples of F- and S-modules on n-dimensional arrays are provided.

The approach of Constructive Recursion was presented in 1990-s. This approach concerns: (1) the functional language to express recursion, (2) avoiding exponential growth during the interpretation of a compiled program, and (3) parallelizing. A recursive function is defined over a regular data dependency graph. The separation of the data dependency pattern is very useful in porting programs to, and between, parallel machines.

Program Separation in GCLA[*]

Abstract

Göran Falkman

Department of Computing Science
Chalmers University of Technology
S-412 96 Göteborg, Sweden
email: falkman@cs.chalmers.se

In this paper I present a programming methodology which makes algorithms a more explicit part of *declarative* programming. The work presented here is based on some earlier work on program separation in GCLA [2].

The basic idea of the methodology is to separate programs into two parts:

(i) The part that describes the overall global structure of the algorithm.
(ii) The part that describes the possible connections between different points in the computation space generated by the algorithm.

It is natural to think of this separation as a separation of *form* and *content*. The first part then gives the form of the algorithm and the second part gives the specific content needed to compute a particular function.

In GCLA I use this separation as a basis for a *definitional* higher order programming methodology.

The programming system GCLA [1, 3] is based on the idea of viewing programs as *definitions*. A GCLA program consists of two definitions, R and D, where D simply is referred to as the definition and R is called the rule definition, since it defines the rules for how the definitions in D should be interpreted. The communication between R and D is based upon a set of primitive notions which are entirely justified by the interpretation of D as a definition. Among these notions, the most important ones are $K(a)$, which gives the definiens of a in K, i.e. $K(a) = \{A \mid a = A \in K\}$ and $Dom(K)$, which gives the domain of K, i.e. $Dom(K) = \{a \mid \exists A(a = A \in K)\}$.

The program separation scheme outlined above naturally translates into the two-level architecture of GCLA programs; the first part, describing the form of the algorithm, corresponds to the rule definition and the second part, describing the content of the algorithm, corresponds to the definition[1]. Thus, the basic idea of the methodology can be reformulated as to separating programs into two definitions:

(iii) The definition that abstractly defines the main rules, i.e. the form, of the algorithm using primitive operations on definitions, e.g. $K(a)$, $Dom(K)$ etc.
(iv) The definition that defines connections between the concrete representation of possible points in the computation space generated by the algorithm, i.e. the specific content of the algorithm.

Since the communication between (iii) and (iv) only uses primitive operations on definitions it is possible to define the rules in (iii) without referring to any *specific* definition in (iv). The methodology is therefore, in some sense, a *definitional* higher order programming methodology.

References

1. M. Aronsson, *GCLA, The Design, Use and Implementation of a Program Development System*, Ph D thesis, Department of Computer and Systems Sciences, University of Stockholm, 1993.
2. G. Falkman, L. Hallnäs, O. Torgersson, Program Separation in GCLA, in: A. Momigliano, M. Ornaghi (eds.) *Proceedings of the Post-Conference Workshop on Proof-Theoretical Extensions of Logic Programming, Santa Margherita Ligure, Italy, 18 June, 1994*, pp 31-37, 1994.
3. P. Kreuger, GCLA II, A Definitional Approach to Control, in: L-H. Eriksson, L. Hallnäs, P. Shroeder-Heister (eds.), *Extensions of Logic Programming, Proceedings of the 2nd International Workshop held at SICS, Sweden, 1991, Springer Lecture Notes in Artificial Intelligence*, vol. 596, pp 239-297, Springer-Verlag, 1992.

[*]. This work has been 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.

[1]. Actually the second part may correspond to several definitions, but since the GCLA system at present only can handle one definition at a time I have to "simulate" the use of multiple definitions within a single definition.

Algebras with Structure

Øyvind B. Fredriksen

University of Bergen
Department of Informatics

Abstract

This talk is based on the one we gave at the 5th Nordic Workshop last year ([1]). The starting point of the work reported in that talk was the host of different kinds of algebras that have been suggested in the literature as models of algebraic specifications, among them

- ordinary total algebras
- partial algebras
- ordered/monotonic algebras
- continuous algebras
- topological algebras.

The goal of the work was to unify as many as possible of these algebra concepts into one.

In the talk we presented a partial solution to this problem, viz. a general construction based on the following *parameters*:

- a Cartesian category C
- functors
  - $F : Set \longrightarrow C$ preserving finite products
  - $G : C \longrightarrow Set$

  such that
  - $F \dashv G$
  - $F \circ G = I_{Set}$. (The composition is written in diagrammatical order.)

From these parameters we constructed

- a category of algebras of a given signature
- the subcategory of those satisfying a given set of equations.

We showed that the two categories had initial objects strongly related to the ordinary term and quotient algebras, respectively. Finally, we showed how to apply this general construction to obtain total and ordered algebras.

However, we felt that the requirement that $F \circ G = I_{Set}$ was rather *ad hoc*. In the present talk we will show that this restriction may be lifted, on the (rather weak) condition that the category C be *locally small*.

References

[1] Øyvind 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.

Structural Synthesis of Programs Using Regular Data Structures

Mait Harf, Jaan Penjam
Dept. of Computer Software, Institute of Cybernetics, Tallinn.

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 solvability of the problem [1]. This technique can be also treated as satisfaction of functional constraint network using value propagation [2]. The method is powerful under certain conditions (see [2]) but it can be optimized for several particular cases.

In this paper involving regular data structures is discussed. Regular data structures like arrays, sequences, vectors, etc. appear in many applications. The simulation of data parallel computations on systolic arrays serves as an example of such systems. The formal theory for synthesis of programs on regular data can be obtained by adding inference rule which enables to construct cyclic programs for computing i-th element of the sequence if j-th element of the sequence and the regular relationship between neighbouring elements is known. The completeness and soundness of the theory is introduced and proven.

The high order specification language which allows to define regular structures is observed. Two special abstract elements of a sequence "current" and "next" are invoked. Elements of a regular data structure can have complicated internal structure including another regular structure. The technique how to combine several one-dimensional structures to specify two- and n-dimensional arrays is described.

The method for program construction described here is implemented as a specific feature or the NUT programming system [3]. The paper contains several examples of usage regular data in specifications in NUT language as well as programs which can be synthesized for solving problems on these structures.

Demonstration of the NUT system including program synthesis on regular data structures is available.

Literature:

1. Tyugu E. Knowledge-Based Programming. - Addison-Wesley, N.Y., 1988.

2. Tyugu E.,Uustalu T. Higher-Order Functional Constraint Networks. In Constraint Programming. NATO ASI Series F: Computer and Systems Sciences, (to appear).

3. 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.

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 [Még90, HKK94] conservatively and are close to a two level, hierarchical fragment of unified algebra [Mos89].

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 [HKK93]. In [HKK93], we found that term declarations are hard to handle and underly strong restrictions w.r.t. their form, if they can be interpreted freely. 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 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. They 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.

We hope to get a compromise between the expressiveness of unified algebras and the operational aspects of order-sorted algebras when using two levels of specification - the higher one being dedicated to specification, the lower one for implementation issues. This may provide a way to effectively compute with a fragment of unified algebras using term rewriting systems. Comparisons and bigger examples are currently under investigation.

References

[HKK93] C. Hintermeier, C. Kirchner, and H. Kirchner. Dynamically-typed computations for order-sorted equational presentations. research report, INRIA, Inria Lorraine & Crin, November 1993.

[HKK94] 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.

[Még90] Aristide Mégrelis. *Algèbre galactique — Un procédé de calcul formel, relatif aux semi-fonctions, à l'inclusion et à l'égalité.* Thèse de Doctorat d'Université, Université de Nancy 1, 1990.

[Mos89] Peter D. Mosses. Unified algebras and institutions. In *Proceedings 4th IEEE Symposium on Logic in Computer Science, Pacific Grove*, pages 304–312, 1989.

Boolean Automata: a Compact Representation of Synchronous Reactive Systems

Leszek Holenderski, Axel Poigne

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 langauges 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 lineary with the growth of a reactive synchronous system it represents.

Bisimulations for a Label-passing Process Calculus with Asynchronous Communication

Hans Hüttel
Department of Mathematics and Computer Science
Aalborg University

(This is joint work with Martin Hansen and Josva Kleist.)

In recent years a number of bisimulation equivalences have been proposed for value-passing process calculi, the inspiration coming from the study of the π-calculus. Milner, Parrow and Walker have distinguished between the *late* and *early* bisimulation equivalences corresponding to differences in binding time. It turns out that late bisimulation is a strictly finer notion than that of early bisimulation. A similar distinction between late and early binding can be made for the testing equivalences of Hennessy and De Nicola; here however, it turns out that the late and early testing equivalences coincide, as shown by Ingólfsdóttir.

Most recently, Sangiorgi has proposed an even finer notion of bisimulation, namely *open bisimulation.* A pleasant property of open bisimilarity is that it has a sound and complete equational theory for finite agents in the π-calculus

In this talk we consider Plain LAL, a label-passing process calculus which differs from the π-calculus in that value communication is *asynchronous* in the sense of e.g. the Linda programming paradigm of Gelernter et al. We define the notions of early, late and open bisimulations and prove the surprising result that in the Plain LAL calculus, the three bisimulation equivalences *coincide.*

We next consider an extension of Plain LAL, namely LAL, a higher-order calculus similar to CHOCSstudied by Thomsen and define the notions of early, late and open bisimulation on LAL terms. It again turns out that the three equivalences coincide. To the best of our knowledge, this is the first such result for a higher-order (CHOCS-like) process calculus and the first time that open bisimulation has been studied for such a calculus.

Finally, we use the fact that the three equivalences coincide to give a sound and complete equational theory for finite Plain LAL agents.

Denotational Semantics for Value-Passing Calculi–Late Approach

A. Ingólfsdóttir, Ålborg Universitet

1 Abstract

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 input process has to accept whatever value the output 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 input 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 output 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 with the new constructions *abstractions* and *concretions*. For further motivation for this approach see for instance [MPW89], [Mil91] and [Ing93].

The aim of this talk is to develop a denotational semantics for languages with simple values based on the late approach described above although I will use a slightly different notation.

The talk is divided into two main parts: a general theory to handle value-passing followed by an application to a concrete definition. In the first part I will 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 I will 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 I only allow simple values and allow Σ to be any set of operators. I also introduce the general class of applicative (Σ, C)-domains to model the semantics of the (Σ, C)-terms. Here the late semantic approach is made explicit; the outcome of an input action is modelled as a function which takes a value and returns an element of the model, i.e. a processes whereas the outcome of an output action is modelled by a pair consisting of the out-putted value and the resulting process.

After having defined our general class of models I will modify the definition of evaluation mappings, i.e. the unique mapping from the process algebra into the domain known from

the theory for pure processes. As I want to be able to reason about a subset of the process algebra, I will extend the definition slightly. For this purpose I will introduce the notion of *recursively closed subsets* of a process algebra. This extension of the definition allows me to reason about the compact elements of an algebraic *cpo* on the syntactic level. This enables me to take advantages of the notion of algebraicity when comparing the semantics defined by the model to other kinds of semantics such as behavioural or axiomatic semantics.

Next I will apply the general result for algebraic *cpos* ([Bar84]): 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 me 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 I can take their unique extension to be their definition on the whole domain.

Then 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 I will apply the described theory to define the operators, i.e. by defining them as monotonic endofunctions on the *po* of compact elements and then extend them to the whole *cpo*.

Finally I will 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 me 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.

References

[Abr92] S. Abramsky: A Domain Equation For Bisimulation, *Information and Computation*, Vol. 92, pp 161-218, 1992.

[Bar84] Barendregt, H.P.: *The Lambda Calculus, Its Syntax and Semantics*, Studies in Logic and the Foundation of Mathematics, Vol. 103, North-Holland, Amsterdam 1984.

[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.

[Mil80] Milner, R. *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92. Springer-Verlag, Berlin, 1980.

[Mil91] R. Milner, Polyadic π-Calculus. A tutorial. Technical Report, CS-LFCS-91-180, LFCS, Department of Computer Science, University of Edinburgh, 1991. To appear *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.

Interpreting Broadcast Communication in CCS with Priority Choice

Claus Torp Jensen
Computer Science Department
Chalmers University of Technology
Goteborg, Sweden
E-mail: ctjensen@cs.chalmers.se

CBS (Calculus of Broadcasting Systems) is a process calculus in which broadcast is the fundamental communication paradigm. In a CBS system every parallel sub-process participates in any action that the system may perform. In that sense CBS is a synchronous calculus of the same type as SCCS.

CCS on the other hand is an asynchronous calculus. At most two subprocesses participate in any action of a particular parallel system.

We show that by adding a priority choice operator to CCS, these two different views of communication and parallellism may be reconciled in the sense that 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 of CBS terms is correct in the sense that two CBS terms are strongly bisimilar if 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.

Polytypic Programming

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 $f : a \rightarrow b$, 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).

Synthesizing Real Time Systems

Kim Guldstrand Larsen
BRICS
Aalborg University

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 explicitely. These specification formalisms may roughly be divided into two groups, namely: real time logics (e.g. [RT89, CHR91]) and real time process algebras (e.g. [Y.W90, NJV]).

Central to the ongoing research has been the construction of *model–checking* algorithms; i.e. algorithms for deciding whether a given real time systems [1] satisfies a given formula. A number of model–checking algorithms exists for real timed logics [ACD90] and recently algorithms for model–checking [2] in the setting of real time process algebra has been given [Cer92, LW93].

In this paper, 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). We consider the model–construction problem in the setting of *implicit specifications*, i.e.:

$$(A_1 \mid \ldots \mid A_n \mid X) \ \mathsf{sat}\ S \qquad (1)$$

Here $A_1 \ldots A_n$ are given real time systems [3], S is the (given) overall specification and the problem is to construct a solution for X which when put in parallel with $A_1 \ldots A_n$ will satisfy S. We provide a generic model–construction algorithm for (1) which will be applicable both for S being a logical specification [4] as well as a process algebraic specification [5]. Also, our notion of parallel composition is

[1]real time system = timed automata [AD90] $\vee$ TCCS agent [Y.W90] $\vee \ldots$

[2]In process algebra model–checking consists in checking a suitable behavioural relationship (bisimilarity, say) between the implementation and the specification.

[3]To be precise $A_1 \ldots A_n$ are one–clock real automatas or equivalently regular TCCS agents.

[4]In fact we introduce a real–time ν–calculus.

[5]For S being a process algebraic specification our method allows sat to be either timed bisimilarity or time–abstracting bisimilarity.

parameterized on a synchronization function allowing a range of existing notions of parallel composition (CCS, CSP and ACP) to be obtained as instances.

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 [LX90]. For S a logical specification our work is related to and extends the work on contexts as property transformers studied in [LX91]. The work reported is based on and extends that of [AKNP94].

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.

[CHR91] Z. Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, 1991.

[LW93] K.G. Larsen and Y. Wang. Time Abstracted Bisimulation: Implicit Specifications and Decidability. *In Proceedings of MFPS'93*, 1993.

[LX90] 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. *J. Logic Computat.*, 1(6):761–795, 1991.

[NJV] X. Nicollin, J.L. Richierand J.Sifakis, and J. Voiron. ATP: an algebra for timed processes, year = 1990. *In Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods.*

[RT89] R.Alur and T.A.Henzinger. A Really Temporal Logic. *In Proceeding of IEEE Symp. on Foundations of Computer Science*, 1989.

[Y.W90] Y.Wang. A Calculus of Real Time Systems. *Lecture Notes in Computer Science*, 458, 1990. In Proceedings of CONCUR.

[AKNP94] J. Andersen, K. Kristoffersen, J. Niederman and J. Pedersen. Specification, Verification and Construction of Real Time Systems. *Master Thesis*, 1994. Aalborg University.

Reasoning with Actions

S. B. Lassen
BRICS*
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C, Denmark
email: `thales@brics.aau.dk`

This talk explores a new formulation of the semantics of *actions*, the specification entities of action semantics. We specify a *reduction semantics* for actions and investigate how this formalism enables us to reason directly about operational equivalence of actions.

The reduction semantics is a small-step operational semantics defined as a term rewriting system specified in terms of evaluation contexts. Intermediate information is represented syntactically during processing. The syntactic representation of configurations in the operational semantics forms a convenient starting point for equational reasoning about operational equivalence.

Similar reduction semantics for Scheme- and ML-like functional languages with imperative operations and control operators have formed the basis for powerful operational reasoning in work by Matthias Felleisen, Ian Mason, Carolyn Talcott, and others. This talk is essentially an attempt to fit actions into the framework of Mason and Talcott and investigate whether it will prove as powerful a vehicle for action theory as it is in their functional setting.

The motivation for carrying over the techniques for operational reasoning to the world of actions is that action semantics is a general semantic description framework. Therefore, a strong equational theory for actions can be applied more easily and more generally for reasoning about programs and programming languages than previous work in the field based on various simple functional languages.

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

Introduction to ALF - an Interactive Proof Editor

Lena Magnusson
Chalmers Technical University
Inst. for Computer Science
Göteborg, Sweden
Email: 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 datatypes, 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 refinment 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.

Preprocessing by Program Specialization *

Karoline Malmkjær and Olivier Danvy
Aarhus University †

September 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), 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 naive 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*: 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.

The naive 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 (the left picture), which makes it possible to match a string linearly.

$$\begin{tikzcd}
& \odot \arrow[r, "b"] & \odot \arrow[r, "a"] & \odot \\
{} \arrow[r, Rightarrow] & \odot \arrow[u, "a"] \arrow[r, "b"] & \odot \arrow[r, "a"] & \odot
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
& \odot \arrow[dr, "b"] & & \\
{} \arrow[r, Rightarrow] & \odot \arrow[u, "a"] \arrow[r, "b"] & \odot \arrow[r, "a"] & \odot
\end{tikzcd}$$

Because it has to keep track of the positions, this tree grows quadratically with the text. We first try to fold it by relaxing the requirement upon 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. But this is not enough: the tree could be folded into a direct acyclic word graph (the right picture). We conclude by showing how to obtain this folded tree (a.k.a. a Weiner tree) by partial evaluation.

This approach generalizes the now traditional derivation of the Knuth, Morris, & Pratt linear string-matching program by partial evaluation of a naive and quadradic string-matching program.

*Submitted to the 6th Nordic Workshop on Programming Theory.

†Computer Science Department, Ny Munkegade, 8000 Aarhus C, Denmark. E-mail: (karoline, danvy) @daimi.aau.dk — Our work is supported by the DART project funded by The Danish Research Councils.

Static and Dynamic Processor Allocation for Higher-Order Concurrent Languages

Hanne Riis Nielson, Flemming Nielson

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

e-mail:{hrnielson,fnielson}@daimi.aau.dk
phone:+45.89.42.32.76 . fax:+45.89.42.32.55

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 statically determining for each fork operation on which processor all spawned processes should reside, whereas the other provides information useful for dynamically deciding for each newly spawned process which processor it should reside on. We prove the soundness of all 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.

A Proposal for a Process Logic

Olaf Owe
Department of Informatics
University of Oslo

June 1994

Abstract

We develop a logic for reasoning about requirement specification of processes with (internal and external) non-determinism. We try to achieve conceptual simplicity by avoiding the use of infinite sequences, and by formulating a proof system within first order logic, with a non-standard interpretation. In particular, non-deterministic processes are specified by means of a simple event relation, called the ready relation, whereas the underlying semantics is a set of models, each with a set of such relations.

By weakening the usual requirement of (sequence prefix) monotonicity, we obtain an expressive specification language, enabling us to relate incompatible executions in the same specification, without the risk of making meaningless or inconsistent specifications. It is possible to specify internal non-determinism without underspecification. Our language is expressive enough to avoid he Brock-Ackerman anomaly and the merge anomaly.

We present a sound and (relative) complete basic proof system. The classical rules and axioms of first order logic with equality are sound. In addition, some rules and axioms are needed for the ready relation and other process operators considered. Refinement may be done in two ways: By enriching a specification over a set of processes, one may refine several processes (reducing the set of possible models). By the explicit refinement operator, one may refine a single process (reducing the number of executions in each model).

A Type System Equivalent to Flow Analysis

Jens Palsberg
Aarhus University
BRICS, Computer Science Dept.
Aarhus, Denmark

Flow-based safety analysis of higher-order languages has been studied by Shivers, Palsberg, and Schwartzbach. Open until now is the problem of finding a type system that accepts exactly the same programs as safety analysis.

We have proved 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.

Specifying and Verifying Parametric Processes

Wiesław Pawłowski[*] Paweł Pączkowski[†] Stefan Sokołowski[*]

Abstract

A typical approach to software development is that of decomposition of large tasks into smaller subtasks. In more formal terms, a task of providing a construction that meets a given specification can be decomposed as follows:

1. provide sub-constructions that meet some sub-specifications into which the original specification is decomposed, and

2. provide a parametric construction which yields an object that meets the original specification when applied to sub-constructions described in point 1.

When the sub-constructions are again decomposed into smaller bits the parameters become parametric themselves and we have to deal with higher order parameters.

We 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 by E.-R. Olderog. However, rather than considering mixed terms as Olderog does, we introduce parametric processes that contain process variables. We allow higher order parametricity, where process variables can represent parametric processes. Technically speaking, Olderog'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 higher order specifications play the role of types. A proof system for deriving judgements "a process meets a specification" is provided.

The developed formalism allows one to do a systematic book-keeping of process dependencies that is useful in higher order constructions. At this stage of our research no attempt was made to extend process algebra operators to higher order constructions.

[*]Institute of Computer Science, PAS, Gdańsk, Poland; supported by ICS PAN, KBN grant PB-1312/P3/92/02 and CRIT.

[†]Chalmers Technical University, Göteborg, Sweden, and University of Gdańsk, Gdańsk, Poland; supported by CRIT and CONCUR2.

Formal Derivation of A FEAL Processor

R.Rukšėnas [*] K.Sere[†] Y.Zhao[*]

We present a method for formal derivation of asynchronous VLSI circuits. The proposed method focuses on transformational style of design and it uses methods familiar from the design of parallel programs. Refinement calculus and action systems are used as a framework in the formal design process. The refinement calculus and the stepwise refinement of parallel programs as action systems were introduced in the papers of Back and Sere [1, 2].

A novel method exploiting parallel programming techniques for VLSI design was introduced by Martin [3]. The method of Martin is, however, semi formal. Our goal is to formalize this method within the refinement calculus thereby giving it a completely formal basis.

The second goal of our work is the mechanisation of the design process. The refinement calculus itself has been formalised within a mechanical theorem prover by von Wright [4]. Hence, each transformation step can be expressed as a refinement rule and the entire circuit can be mechanically proved correct via refinement calculus.

As a case study we look at the design of an asynchronous FEAL (Fast Encipherment Algorithm) processor [5]. The synchronisation of communication between parts of the circuit is very tricky in its design. Here, a recent add to the action systems, the procedures mechanism [6], was found extremely useful .

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, 1978

2. R.J.R. Back, K. Sere. Stepwise Refinement of Action Systems. *Structured Programming*, 12:17-30, 1991

3. A.J. Martin. *Synthesis of Asynchronous VLSI Circuits.* California Institute of Technology, Technical Report Caltech-CS-TR-93-28, 1993

4. J. von Wright. *Program Refinement by Theorem Prover.* Åbo Akademi Report on Computer Science & Mathematics, Ser.A.No 146, 1994

5. S. Miyaguchi. The FEAL Cipher Family. *EUROCRYPT'90*,627-638, 1990

6. R.J.R. Back, A.J. Martin, K. Sere. *Specification of a Microprocessor.* Åbo Akademi Report on Computer Science & Mathematics, Ser.A.No 148, 1994

[*]Åbo Akademi University, Department of Computer Science, FIN-20520 Turku, Finland
[†]University of Kuopio, Department of Computer Science and Applied Mathematics, P.O.Box 1627, FIN-70211 Kuopio, Finland

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

Temporal logic is an appropriate formalism to reason about concurrent systems. We consider here the branching propositional temporal logic BPTL. An underlying model of the logic is a tree like construction, i.e. any instant of time may split into different possible futures. BPTL is a subsystem of branching time logic introduced in [BPM]. The language of BPTL contains the usual propositional connectives and the following temporal modalities (here u ranges over formulas): $\bigcirc u$ ("u is true in each next state"), $\odot u$("u is true in some next state), $\Box u$ ("u is always true from now on ") $\diamond u$("u is eventually true").

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. We prove soundness and completeness of the presented system. The proof of completeness uses tableau construction for BPTL. The idea to use tableau construction in order to prove completeness of nonclausal resolution proof system is adopted from [AM].

References

[AM] (1985) Abadi M. and Z.Manna, Nonclausal temporal deduction, Logics of programs (ed. R.Parikh), Springer-Verlag, LNCS 193.
[BPM] (1983) Ben-Ari M., A.Pnueli and Z.Manna, The temporal logic of branching time, Acta Informatica 20, 207–226.

A Graph-Form for Gamma Programs

David Sands
DIKU, University of Copenhagen

The Gamma model is a minimal programming language based on local multiset rewriting (with an elegant chemical reaction metaphor). A calculus of Gamma programs has been studied for programs built from basic "reactions", plus parallel and sequential composition operators. The salient (and unusual) feature of the composition operators for Gamma programs is that for termination, the parallel composition operator demands that its operands must terminate synchronously.

In this talk we discuss a new static graph representation for Gamma programs, and argue that it forms a better basis for the study of compositional semantics, refinement, true concurrency and program logics. Operationally, Gamma graphs are like flow-charts, where each node corresponds to a simple form of loop. A node represents a set of reactions which are concurrently active; an edge represents an internal termination step, where the child node may inherit some reactions from the parent but adds some new active reactions.

We compare this form with previous compositional semantics based on "reactive traces" (a la Brookes / de Boer et al) derived from SOS rules, and show that the refinement laws can be more easily obtained, with the help of a compositional construction of Gamma graphs. We also show that reasoning about relational properties using Gamma-graphs recovers the simplicity of reasoning seen in Banatre and Le Metayer's original study.

Towards Operational Semantics of Contexts in Functional Languages

David Sands
DIKU, University of Copenhagen

The idea of providing an operational semantics for contexts has been studied by Larson (et al) for process algebras. In that setting, a context is an action transducer which consumes actions provided by its internal processes (the holes) and produces eternally observable actions.

We describe some initial steps towards providing an operational semantics for contexts in a functional setting. In a functional language, the role of an "action" is played by a lazy data constructor, or a lambda term. Giving a full operational semantics for contexts is difficult because:

* contexts can consume without producing an observable;
* contexts can duplicate holes;
* contexts which consume an n-ary constructor must increase the number of distinct holes by n;
* contexts can capture variables by means of binding operators eg. lambda-expressions.

As a first step toward an operational semantics for contexts in a functional language, we study two restrictions:

a) a full context semantics for a restricted first-order functional language with recursive definitions and nullary and unary (lazy) constructors.
b) a 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.

In case (a), the context semantics subsumes the usual call-by-name semantics for expressions. We conjecture that for each derivation in the context semantics, a derivation exists in which any judgement about a function call can compositionally described in terms of the function context and the argument part. The corresponding proof abstractly represents the amount of computation performed in call-by-need computation.

The characterisation of contexts in case (b) finds immediate application to the problem of correct folding in program transformation. It also provides an applicative "bisimulation up to context" proof technique, a la Sangiorgi.

Some Results about the Category of Net Computations

Vladimiro Sassone

BRICS – Computer Science Dept., Aarhus University

ABSTRACT

KEYWORDS: Semantics, Semantics of Concurrency, Petri Nets, Categories.

In [1] the authors show that the non-interleaving behaviour 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—yielding in this way a unification of the process-oriented and the algebraic view of net computations. A natural complement to the ideas of [1] is provided by [3], which gives a purely categorical axiomatization of the category of the computations of a net, thus yielding a description of the causal behaviour of nets as an *essentially algebraic theory* (whose models are monoidal categories). However, this construction is somehow unsatisfactory, since it is *not* functorial. More strongly, given a morphism between two nets, it may not be possible to identify a corresponding monoidal functor between the respective categories of computations. This fact, besides showing that our understanding of the structure of Petri nets is still incomplete, has also other drawbacks, the most relevant of which is probably that it prevents us to identify the *category* (of the categories) *of net behaviours*, i.e., to axiomatize the behaviour of Petri nets *"in the large."*

The talk presents an analysis of the problem and a possible solution based on the newly introduced notion of *strong concatenable processes*. These are a slight refinement of the standard notion of process: namely, they are non-sequential processes whose minimal and maximal places are *linearly* ordered. We shall prove that strong concatenable processes are, in a very precise sense, the *least* refinement of non-sequential processes which can be expressed axiomatically in the style of [1, 3] via a *functorial* construction. As a first consequence of this result, we can formulate a possible definition of the *category of net computations*.

Although we are aware that this contribution is just 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 illustrated in the talk help to deepen the understanding of the subject. In addition, from the categorical viewpoint, our approach is quite natural and elegant.

These results appear also in [2, 4].

REFERENCES

[1] P. Degano, J. Meseguer, and U. Montanari. Axiomatizing Net Computations and Processes. In *Proceedings of the 4th LICS Symposium*, pp. 175–185, IEEE, 1989.

[2] V. Sassone. *On the Semantics of Petri Nets: Processes, Unfoldings, and Infinite Computations.* PhD Thesis TD 6/94, Dipartimento di Informatica, Università di Pisa, March 1994.

[3] V. Sassone. *Some Remarks on Concatenable Processes.* Technical Report TR 6/94, Dipartimento di Informatica, Università di Pisa, April 1994.

[4] V. Sassone. *Strong Concatenable Processes of Petri Nets.* To appear as technical reports BRICS, Computer Science Dept., Aarhus University and Dipartimento di Informatica, Università di Pisa, 1994.

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

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

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 $\perp$,
- $!^{\mathbf{n}}$t: the value has type t and is definitely *not* $\perp$, 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 $(!^{\mathbf{n}}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int}) \wedge (!^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{b}}\texttt{Int})$ 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 $(!^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int}) \wedge (!^{\mathbf{n}}\texttt{Int} \rightarrow !^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int}) \wedge (!^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{b}}\texttt{Int})$ 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

Functional Logic Programming in GCLA*

Olof Torgersson

Department of Computing Science, Göteborg University
S-412 96 Göteborg, Sweden
oloft@cs.chalmers.se

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 [4] and Constraint Logic Programming [1].

A notion shared between functional and logic programming is that of a *definition*, we say that we *define* functions and predicates. The programming language could 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 [5] 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) [3]. 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 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 the class of functional logic program definitions. This rule definition implicitly determines the structure of function, predicate and integrated functional logic program definitions. We will also 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. These rule definitions use and develop ideas from [2].

We illustrate with an example. The definition below combines lazy evaluation of functions with indeterminism and backtracking into a generate and test program producing all subsets of `Set` having a sum = `K`.

```
sum_of_subsets(Set,K) <= sum_eq(subset(Set),0,K).

sum_eq([],Acc,K) <= (Acc = K)-> [].   % read A->B as if A then B
sum_eq([X|Xs],Acc,K) <= (X+Acc=<K) -> [X|sum_eq(Xs,X+Acc,K)].

subset([]) <= [].
subset([X|Xs]) <= [X|subset(Xs)],subset(Xs).   % read this ',' as or
```

References

1. H. Aït-Kaci, A. Podelski, Towards a Meaning of Life, *Journal of Logic Programming*, vol 16, pp 195-234, 1993.
2. G. Falkman, O. Torgersson, Programming Methodologies in GCLA, in, *Extensions of Logic Programming, Springer Lecture Notes in Artificial Intelligence*, vol 798,Springer Verlag 1994.
3. L. Hallnäs, Partial Inductive Definitions, *Theoretical Computer Science*, vol 87,pp 115-142,1991.
4. M. Hanus, The Integration of Functions into Logic Programming; From Theory to Practice, *Journal of Logic Programming*, vol 19/20, pp 583-628, 1994.
5. P. Kreuger, GCLA II, A Definitional Approach to Control, in, *Extensions of Logic Programming, Springer Lecture Notes in Artificial Intelligence*, vol. 596, pp 239-297, Springer Verlag, 1992.

* This work was carried out as part of the work in ESPRIT working group GENTZEN and was funded by the Swedish National Board for Industrial and Technical Development (NUTEK).

Extensions of Structural Synthesis of Programs

Tarmo Uustalu
Dept of Teleinformatics, The Royal Institute of Technology
Electrum 204, S-164 40 Kista, Sweden
email tarmo@it.kth.se
fax +46 8 751 1793

Structural synthesis of programs (SSP) as proposed and described by Mints and Tyugu [MT90] is an approach to deductive synthesis of functional programs using types as specifications and based on the Curry-Howard correspondence and on a very intensional treatment of the notion of type. The approach has a practical orientation and a feasible balance has been sought between expressiveness of the specification language and efficiency of proof search. The work is related to automated software engineering, programming in type theories, and automated theorem proving.

The proof search technique employed in SSP has been described in different formulations by a number of authors: Maslov (the inverse method) [Ma64], Mints (generalized resolution) [Mi88], Stålmarck (the assure-method) [S90]. We like to think of the technique as a strategy of forward search in natural-deduction calculi where conclusion generation is limited to sequents that potentially might participate in a normal derivation of the goal formula and where conclusion generation under an extra hypothesis or variable substitution is tried only if no new conclusions can be derived otherwise. Importantly, derivation of proof-theoretically easy goal formulae (this notion is formalized!) is efficient and specifications one usually would write in practice are proof-theoretically easy.

In the implemented systems of SSP, the underlying logic has been propositional. The proof search technique, at the same time, is known to be applicable in first-order and modal logics. In the paper, we show that such extensions of the underlying logic allow for a style of specification where the hierarchical structure of the type world of the programming knowledge being specified is explicit and for proof search that takes real advantage of this explicitness. We discuss the specification methodology and the completeness and complexity of the appropriate proof search algorithms. The part concerning usage of modal logics as medium for specifying and reasoning about concept hierarchies is related to the author's earlier work [U92].

References

[Ma64] S. Yu. Maslov. The inverse method for establishing deducibility in classical predicate calculus. *Soviet Math. Doklady*, 5:1420–1423, 1964.

[Mi90] G. Mints. Gentzen-type systems and resolution rule, Part I: Propositional logic. In *Proc. Int'l Conf. on Computer Logic, COLOG-88, Tallinn, Dec 1988*, pp 198–231, Berlin, Springer-Verlag, 1990. (LNCS, Vol 417.)

[MT90] G. Mints and E. Tyugu Propositional Logic Programming and the PRIZ System. *J. Logic Programming*, 1990, 9(2–3):179–193.

[S90] G. Stålmarck. The Assure method: A proof procedure for propositional logic. Technical Report 1, Logikkonsult NP AB, Stockholm, 1990.

[U92] T. Uustalu. Combining object-oriented and logic paradigms: A modal logic programming approach. In *Proc. European Conf. on Object-Oriented Programming, ECOOP'92, Utrecht, June/July 1992*, pp 98–113. Berlin, Springer-Verlag, 1992. (LNCS, Vol 615.)

Algebra of Broadcasting Systems: Value Passing and Sequential Composition

Martin Weichert
Chalmers
Institute for Computer Science
Göteborg, Sweden
Email: martinw@cs.chalmers.se

This paper presents ACBS, Algebra of Broadcasting Systems, a process calculus characterised by value passing + sequential composition + broadcast communication.

ACBS is based on CBS, Calculus of Broadcasting Systems, a CCS-like calculus with broadcast communication instead of handshake, and on ACP, Algebra of Communicating Processes, a family of calculi built with sequential composition.

The paper presents a complete axiomatisation for non-recursive "pure" ACBS processes with respect to strong bisimulation. The use of three different "merge" operators as in ACP makes finite axiomatisation possible. It then proposes several extensions to the calculus, such as translation and recursion, and discusses their effects on axiomatisation. An example illustrates the equational style of reasoning provided by the axiomatisation.

The differences between prefixing and sequential composition, which are not so drastic in the "pure" calculi, become quite apparent when value passing is added. A section about value passing discusses how different programming styles emerge from the different connectives in the calculi. Whereas sequential composition leads to an imperative-style semantics involving program environments, prefixing more closely resembles evaluation in a functional-style language.

Finally, the discussions from the preceding section lead to the presentation of the value passing version of ACBS.

A Case Study in Timed Modal Specification

Carsten Weise
Lehrstuhl fuer Informatik I
University of Technology
52074 Aachen - Germany
Email: carsten@i1.informatik.rwth-aachen.de

We will specify processes by labelled modal transition system. The transitions in modal transition systems come in two flavors: may- and must-transitions. A refinement relation identifies the suitable implementations of a specification. An implementation is a transition system with only one type of transition (alternatively a modal transition system with only must-transitions). Intuitively an implementation must
consist of all must-transitions and may (or may not) have the may-transitions of the specification.

Instead of using a graphical representation (i.e. transition systems) one can also express the specification in logic formulae in the modal mu-calculus.

Time Modal Specifications (TMS) use a transition system with a time domain. The chosen domain here will consist of the real numbers. Despite the denseness of this domain, and therefore the typically infinitely large transition systems, using methods from the theory of timed automaton, namely timer regions, an automatic treatment of many problems (e.g. is P a refinement of Q?) is possible.

The talk will demonstrate the usefulness of this approach by applying Timed Modal Specifications to an example. The example is that of a lossy remote procedure call, implemented on top of a reliable memory. In the talk a specification of the remote procedure call and the memory component will be given, and several properties of the specifications will be proved.

The results presented are joint work with Kim Guldstrand Larsen and Bernhard Steffen.

Termination of order-sorted rewriting. Abstract.

Peter Ølveczky
Department of Informatics,
University of Oslo
e-mail:peterol@ifi.uio.no

Order-sorted *rewriting* is often used in connection with order-sorted *specifications* (i.e. many-sorted specifications with subsort relations), e.g. in automated theorem proving. Termination and confluence are important properties of a rewrite system, but in general it is undecidable whether a rewrite system satisfies these properties.

Numerous techniques have been developed that with varying degrees of success may be used to prove termination of *unsorted* rewrite systems. These techniques may also be used to prove termination of order-sorted rewrite systems by simply ignoring the sort information, because an order-sorted system always terminates whenever the corresponding unsorted system terminates. These methods are not satisfactory, since there are many terminating order-sorted rewrite systems that do not terminate if the sorts are ignored.

We propose a stronger method for proving termination of an order-sorted rewrite system by transforming it to an unsorted system, such that termination of the latter implies termination of the order-sorted system. We can thus use the well-known techniques for proving unsorted termination to prove order-sorted termination.

The main idea of our method is to label a function symbol with the sorts of its arguments. If a function symbol f takes arguments with sorts s and s', then f_s and $f_{s'}$ are treated as distinct function symbols in the unsorted system.

Our method, which can be used on non-sort-decreasing systems as well, includes as special cases previously published methods, including the method that just ignores sort information.

Demonstration of ALF

Catarina Coquand and Lena Magnusson
Chalmers Technical University
Inst. for Computer Science
Göteborg, Sweden

We refer to our abstracts on pages 15 and 31 for further information about the system to be demonstrated.

Demo of EPSILON by Jesper Niedermann

EPSILON is an automatic tool for analysing concurrent and non–deterministic real–time systems. The specification formalisms underlying EPSILON are Timed Modal Specifications (TMS) [1] and Timed Modal Logics. The current version of EPSILON is a prototype tool subject to constant development.

EPSILON contains algorithms for deciding various equivalences and refinements between specifications in TMS in particular diagnostic information is offered in cases where a derived equivalence or refinement does not hold, thus providing useful information in a subsequent debugging phase. Additionally EPSILON contains an algorithm for model checking with respect to logical formulae (with explicit time requirements). Ongoing work includes a procedure for constructing timed systems directly from logical formulae, an algorithm for transforming overall system properties into a property of a single component, and viewing the transition system of a process graphically.

I will use a running example throughout the demo namely a model of a real–time train crossing scenario [1], and through this show some of the features of EPSILON.

EPSILON is available by anonymous ftp from ftp.iesd.auc.dk and can be found in `'/pub/projects/EPSILON/'` in here is currently a version which should run on most Sun machines, with any operating system, a window system is not a requirement.

References

[1] Čerāns, Kārlis and Godskesen, Jens Christian and Larsen, Kim G., *1993 Timed Modal Specification – Theory and Tools*

High-Level Synthesis of Heterogeneous Analysis Systems

Bernhard Steffen and Tiziana Margaria*
Universität Passau, Germany
steffen/tiziana@fmi.uni-passau.de

Heterogeneity becomes an increasingly central characteristic of environments, to incorporate a variety of analysis and verification methods, like e.g. abstract interpretations, bisimulation checking, theorem proving and model checking. However, the combination of various methods to application specific heterogeneous tools is usually not (explicitly) supported. We present a practice-oriented environment for high-level synthesis of system-level analysis tools, which is unique in 1) constituting a framework for the development of application specific heterogeneous tools and 2) providing facilities for the automation of the synthesis process. Thus it constitutes a system-level prototyping environment that supports the rapid and reliable realization of efficient application-specific complex tools, without a sophisticated user interaction.

Specification language is a *modal logic*, SLTL, that uniformly and elegantly captures type descriptions, specifications of (elementary) algorithms and ordering constraints. Whereas the first two 'dimensions' are treated similarly by means of a simple logic over a taxonomy of types and algorithms respectively, the third is expressed by means of modalities. This allows an elegant and transparent specification of combinations of already existing algorithms.

As SLTL specifications can automatically be transformed into an executable high level functional program by means of a *minimal model generator* for the logic, this transformation can be seen as a specific form of software synthesis on top of a repository of reusable software components. We illustrate the generality of this approach by synthesizing 1) special purpose tools in our analysis and verification environment, and 2) complex parameter-correct UNIX commands from simple SLTL specifications.

*The work is joint with Burkhard Freitag, Olaf Burkart, Andreas Classen and Ulrich Zukowski

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 imploy 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.

<u>Description</u>
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 re-usable library.
h) Automatic Markup and Indexing of documentation of complete developments (requires LaTeX).

<u>Avaliability</u>
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.