Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems

19–20 May 1995, Aarhus, Denmark

Uffe H. Engberg
Kim G. Larsen
Arne Skou (editors)

BRICS Notes Series **NS-95-2**

ISSN 0909-3206 **May 1995**

Proceedings of the Workshop on

Tools and Algorithms for the Construction and Analysis of Systems

19–20 May 1995 — Aarhus, Denmark

Uffe H. Engberg
Kim G. Larsen
Arne Skou
(editors)

Foreword

Welcome to Aarhus, Denmark, and welcome to this workshop on *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, May 19–20, 1995. The aim of the workshop is to bring together researchers and practitioners interested in the development and application of tools and algorithms for specification, verification, analysis and construction of distributed systems. The overall goal of the workshop is to compare the various methods and the degree to which they are supported by interacting or fully automatic tools.

The workshop will present 23 papers covering the following topics: refinement-based verification and construction techniques; compositional verification methodologies; analysis and verification via theorem-proving; decision procedures for verification and analysis; specification formalisms, including process algebras and temporal and modal logics; analysis techniques for real-time and/or probabilistic systems; approaches for value-passing systems, tool sets for verification and analysis case studies. Special sessions for demonstration of verification tools are planned.

The workshop is organized as a satellite activity of the TAPSOFT '95 conference, May 22–26, University of Aarhus. The TAPSOFT '95 conference and its satellite workshops are hosted by BRICS, a centre of the Danish National Research Foundation at the Computer Science Departments of Aarhus and Aalborg Universities. We want to thank the TAPSOFT '95 organizers Peter D. Mosses and Karen K. Møller as well as Birger Nielsen for handling all practical matters concerning TACAS.

The following Program Committee has been responsible for the reviewing and selection of papers: Ed Brinksma (Twente University) Kim G. Larsen (BRICS, Aalborg University), Bernhard Steffen (University of Passau), Rance Cleaveland (North Carolina State University).

We want to thank the members of the Program Committee for their work in evaluating the submitted papers. We would also like to thank all referees who assisted the members of the Program Committee: L. Aceto, F. Andersen, V. Braun, E.H. Eertink, A. Geser, L. Heerink, A. Ingólfsdóttir, W. T. M. Kars, J-P. Katoen, K. J. Kristoffersen, R. Langerak, G. Luettgen, T. Margaria, A. Nymeyer, J. Tretmans, C. Weise, W. Yi.

BRICS, Aarhus and Aalborg, May 1995

Uffe H. Engberg
Kim G. Larsen
Arne Skou

Addresses

For a hardcopy of the proceedings, please contact:

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

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

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

`http://www.brics.aau.dk/BRICS/NS/95/2/BRICS-NS-95-2/`

`ftp ftp.brics.aau.dk (cd pub/BRICS/NS/95/2/TACAS)`

Contents

Combining Model Checking and Deduction for I/O-Automata

Olaf Müller* and Tobias Nipkow†
TU München‡

Abstract

We propose a combination of model checking and interactive theorem proving where the theorem prover is used to represent finite and infinite state systems, reason about them compositionally and reduce them to small finite systems by verified abstractions. As an example we verify a version of the Alternating Bit Protocol with unbounded lossy and duplicating channels: the channels are abstracted by interactive proof and the resulting finite state system is model checked.

1 Introduction

The purpose of this paper is to combine the two major paradigms for the verification of distributed systems: *model checking* and *theorem proving*. The advantages of each approach are well known: model checking is automatic but limited to finite state processes, theorem proving requires user interaction but can deal with arbitrary processes. Recently attempts have been made to combine the strength of both methods by using the deductive machinery of theorem provers to reduce "large" correctness problems to ones that are small enough for model checking. The key idea is *abstraction* whereby the state space is partitioned to obtain a smaller automaton which is amenable to model checking. Of course the abstraction has to be sound w.r.t. the property we want to check: if the abstracted automaton satisfies the property so should the original automaton.

In our approach the theorem prover provides a common representation language and tools for

- both finite and infinite state systems,
- checking the soundness of abstractions,
- reasoning about systems in a compositional manner.

Our work is based on Lynch and Tuttle's *Input/Output-Automata* (*IOA*) [14] as model of distributed processes which have been embedded in the theorem prover Isabelle/HOL [15]. We are interested in verifying safety properties of IOA. These safety properties are not expressed by temporal logic formulae but again by IOA. Hence we need to check that the traces of one IOA C (the implementation) are included in the traces of another IOA A (the specification). Assuming that C is infinite or at least too large to check $traces(C) \subseteq traces(A)$ automatically, we define an intermediate automaton B which is an abstraction of C and should satisfy $traces(C) \subseteq traces(B) \subseteq traces(A)$. Thus we achieve the following division of labor: $traces(C) \subseteq traces(B)$,

*Research supported by DFG, *Leibniz Programm*.

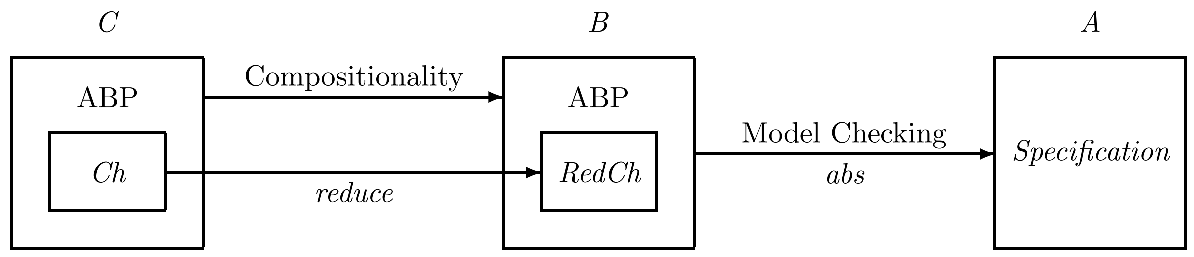
†Research supported by ESPRIT BRA 6453, *Types*.

‡Address: Institut für Informatik, Technische Universität München, 80290 München, Germany. Email: `{Mueller,Nipkow}@Informatik.TU-Muenchen.De`

i.e. the soundness of the abstraction, is proved interactively in Isabelle; $traces(B) \subseteq traces(A)$ is verified automatically by a model checker; finally, transitivity of $\subseteq$ yields the desired $traces(C) \subseteq traces(A)$.

The distinguishing feature of our approach is the ability to reason about the soundness of arbitrary abstractions because we have the meta-theory of IOA at our disposal. Assuming that the theorem prover and the formalization of IOA in it are correct, the only remaining source of errors is the model checker which is treated like an oracle by the theorem prover. Note that this includes the interface between model checker and theorem prover, which is particularly critical because we need to ensure that the theorem prover formalizes exactly the logic the model checker is based on.

The rest of the paper illustrates this approach using a particular example, namely an implementation of the Alternating Bit Protocol using unbounded channels. This is in contrast to pure model checking approaches where the channels are always of a fixed capacity (usually 1). The key to the success of our approach is the fact that channels may lose and duplicate, but not reorder messages. Thus is is possible to "compactify" channels without altering their behaviour by collapsing all adjacent identical messages. This is what our abstraction from C to B does. The full picture looks like this:



The implementation C contains unbounded channels *Ch* which are abstracted/compactified by a function *reduce*. It is shown interactively that *reduce* is indeed an abstraction function, i.e. $traces(Ch) \subseteq traces(RedCh)$. B is the same as C except that collapsing channels are used. Compositionality proves that C must be an implementation of B, i.e. $traces(C) \subseteq traces(B)$. Although *RedCh* is not a finite state system, it behaves like one if used in the context of the ABP because at any one time there are at most two different messages on each channel. Thus B is a finite state system. Note however, that we never need to prove this explicitly: It is merely an intuition which is later confirmed by the model checker when it is given a description of B and A together with an abstraction function *abs* between them. The model checker explores the full state space of B verifying transition by transition that *abs* is indeed an abstraction. It is only the successful termination of the model checker which tells us that B is finite.

1.1 Related work

Our paper is closely related to the work by Hungar [11] who embeds a subset of OCCAM in the theorem prover LAMBDA and combines it with an external model checker. The key difference is that Hungar relies much more on unformalized meta-theory than we do: he axiomatizes OCCAM's proof rules instead of deriving them from a semantics, and does not verify the soundness of his data abstractions.

The literature on abstraction for model checking is already quite extensive (see for example, [4, 8, 5]). The general idea is to compute an abstract program given a concrete one together with an abstraction function/relation. The approach of Clarke et al. is in principle also applicable to infinite concrete systems. However, since they compute an approximation to the real abstract program, the result is not necessarily finite state. Nevertheless it would be interesting to rephrase their ideas in terms of IOA and apply them to our example. In this case we would not give B explicitly but would compute (via the rewriting machinery of the theorem prover) a (hopefully finite state) approximation of it.

Our work differs from most approaches to model checking because we do not check if an automaton satisfies a temporal logic formula but if its traces are included in those of another automaton. Although theoretically equivalent, automata can be compared by providing an explicit abstraction function (or simulation relation), *abs* above. The same approach is followed in [12] where abstraction functions are also used for reduction, and in [9] where liveness is taken into account. If the documentation aspect of an explicit abstraction function is not considered important, one could also use a model checker which searches for an abstraction function using, for example, the techniques of [6], although this is bound to be less efficient.

Finally there is the result by Abdulla and Jonsson [1] that certain properties of finite state systems communicating via unbounded lossy channels are decidable, which they apply to the Alternating Bit Protocol. However, in our work the channels can both lose and duplicate messages, hence their result does not apply directly.

2 I/O-Automata in Isabelle/HOL

Isabelle notation. Set comprehension has the shape $\{e.\ P\}$, where e is an expression and P a predicate. Tuples are written between angle brackets, e.g. $<s, a, t>$, and are nested pairs with projection functions *fst* and *snd*. If f is a function of type $\tau_1 \to \tau_2 \to \tau_3$, application is written $f(x, y)$ rather than $f\ x\ y$. Conditional expressions are written $\mathit{if}(A, B, C)$. The empty list is written $[\,]$, and "cons" is written infix: $h :: tl$. Function composition is another infix, e.g. $f \circ g$.

2.1 I/O Automata

An IOA is a finite or infinite state automaton with labelled transitions. I/O automata, initially introduced by Lynch and Tuttle [14], are still under development, and the formalization we used represents only a fragment of the theory one can find in recent papers [7]. For example, we do not deal with fairness or time constraints. The details of the formalization can be found in a previous paper [15], so that we give only a brief sketch of the essential definitions inside Isabelle/HOL.

An action signature is described by the type

$$(\alpha)\mathit{signature} \equiv (\alpha)\mathit{set} \times (\alpha)\mathit{set} \times (\alpha)\mathit{set}.$$

The first, second and third components of an action signature S may be extracted with *inputs*, *outputs*, and *internals*. Furthermore, $\mathit{actions}(S) = \mathit{inputs}(S) \cup \mathit{outputs}(S) \cup \mathit{internals}(S)$, and $\mathit{externals}(S) = \mathit{inputs}(S) \cup \mathit{outputs}(S)$. Action signatures have to satisfy the following condition:

$$\begin{aligned} &\mathit{is\_asig}(\mathit{triple}) \equiv \\ &\quad (\mathit{inputs}(\mathit{triple}) \cap \mathit{outputs}(\mathit{triple}) = \{\}) \wedge \\ &\quad (\mathit{outputs}(\mathit{triple}) \cap \mathit{internals}(\mathit{triple}) = \{\}) \wedge \\ &\quad (\mathit{inputs}(\mathit{triple}) \cap \mathit{internals}(\mathit{triple}) = \{\}) \end{aligned}$$

An IOA is a triple with type defined by

$$(\alpha, \sigma)\mathit{ioa} \equiv (\alpha)\mathit{signature} \times (\sigma)\mathit{set} \times (\sigma \times \alpha \times \sigma)\mathit{set}$$

and it is further required that the first member of the triple be an action signature, the second be a non-empty set of start states and the third be an input-enabled state transition relation:

$$\begin{aligned} &\mathit{IOA}(<\mathit{asig}, \mathit{starts}, \mathit{trans}>) \equiv \\ &\quad \mathit{is\_asig}(\mathit{asig}) \wedge \mathit{starts} \neq \{\} \wedge \mathit{is\_state\_trans}(\mathit{asig}, \mathit{trans}). \end{aligned}$$

The property of being an input-enabled state transition relation is defined as follows:

$$\begin{array}{l} \mathit{is\_state\_trans}(\mathit{asig}, R) \equiv \\ \quad (\forall {<}s, a, t{>} \in R.\ a \in \mathit{actions}(\mathit{asig})) \wedge \\ \quad (\forall a \in \mathit{inputs}(\mathit{asig}). \forall s. \exists t.\ {<}s, a, t{>} \in R) \end{array}$$

The projections from an IOA are *asig_of*, *starts_of*, and *trans_of*. The actions of an IOA are defined $\mathit{acts} \equiv \mathit{actions} \circ \mathit{asig\_of}$.

An *execution-fragment* of an IOA A is a finite or infinite sequence that consists of alternating states and actions. In Isabelle it is represented as a pair of sequences: an infinite *state sequence* of type $\mathit{nat} \rightarrow \mathit{state}$ and an *action sequence* of type $\mathit{nat} \rightarrow (\mathit{action})\mathit{option}$. Here the *option* datatype is defined as $(\alpha)\mathit{option} = \mathit{None} \mid \mathit{Some}(\alpha)$ using an ML-like notation. A finite sequence in this representation ends with an infinite number of consecutive *None*s. Using this representation, a step of an execution-fragment ${<}\mathit{as}, \mathit{ss}{>}$ is ${<}\mathit{ss}(i), a, \mathit{ss}(i+1){>}$ if $\mathit{as}(i) = \mathit{Some}(a)$. Formally:

$$\begin{array}{l} \mathit{is\_execution\_fragment}(A, {<}\mathit{as}, \mathit{ss}{>}) \equiv \\ \quad \forall n\ a. \quad (\mathit{as}(n) = \mathit{None} \supset \mathit{ss}(\mathit{Suc}(n)) = \mathit{ss}(n)) \wedge \\ \quad \qquad\quad\ (\mathit{as}(n) = \mathit{Some}(a) \supset {<}\mathit{ss}(n), a, \mathit{ss}(\mathit{Suc}(n)){>} \in \mathit{trans\_of}(A)) \end{array}$$

An *execution* of A is an execution-fragment of A that begins in a start state of A. If we filter the action sequence of an execution of A so that it has only external actions, we obtain a *trace* of A. The traces of A are defined by

$$\mathit{traces}(A) \equiv \{\mathit{filter}(\lambda a. a \in \mathit{externals}(\mathit{asig\_of}(A)), \mathit{as})\ .\ \exists \mathit{ss}.\ {<}\mathit{as}, \mathit{ss}{>} \in \mathit{executions}(A)\}$$

where *filter* replaces $\mathit{Some}(a)$ by *None* if a is not an external action.

2.2 Composition and Refinement

I/O automata provide a notion of parallel composition. In Isabelle this mechanism is realized by a binary operator $\|$. The definition simply reflects the fact that each component performs its locally defined transition if the relevant action is part of its action signature, otherwise it performs no transition.

$$\begin{array}{l} A \parallel B \equiv \\ \quad {<}\mathit{asig\_comp}(\mathit{asig\_of}(A), \mathit{asig\_of}(B)), \\ \quad\ \{{<}u, v{>}\ .\ u \in \mathit{starts\_of}(A) \wedge v \in \mathit{starts\_of}(B)\}, \\ \quad\ \{{<}s, \mathit{act}, t{>}\ .\ (\mathit{act} \in \mathit{acts}(A) \vee \mathit{act} \in \mathit{acts}(B)) \wedge \\ \qquad \mathit{if}(\mathit{act} \in \mathit{acts}(A), {<}\mathit{fst}(s), \mathit{act}, \mathit{fst}(t){>} \in \mathit{trans\_of}(A), \mathit{fst}(s) = \mathit{fst}(t)) \wedge \\ \qquad \mathit{if}(\mathit{act} \in \mathit{acts}(B), {<}\mathit{snd}(s), \mathit{act}, \mathit{snd}(t){>} \in \mathit{trans\_of}(B), \mathit{snd}(s) = \mathit{snd}(t))\}{>} \end{array}$$

where an action signature composition is needed:

$$\begin{array}{l} \mathit{asig\_comp}(S_1, S_2) \equiv \\ \quad {<}(\mathit{inputs}(S_1) \cup \mathit{inputs}(S_2)) - (\mathit{outputs}(S_1) \cup \mathit{outputs}(S_2)), \\ \quad\ \mathit{outputs}(S_1) \cup \mathit{outputs}(S_2), \mathit{internals}(S_1) \cup \mathit{internals}(S_2){>} \end{array}$$

Action signature composition presumes compatibility of actions, which is defined by

$$\begin{array}{l} \mathit{compatible}(S_1, S_2) \equiv \\ \quad (\mathit{outputs}(S_1) \cap \mathit{outputs}(S_2) = \{\}) \wedge \\ \quad (\mathit{outputs}(S_1) \cap \mathit{internals}(S_2) = \{\}) \wedge \\ \quad (\mathit{outputs}(S_2) \cap \mathit{internals}(S_1) = \{\}) \end{array}$$

and is trivially extended to compatibility of automata.

For the aim of refinement, we make use of abstraction functions which Lynch and Tuttle call "weak possibility mappings". The set of these maps is described by the following predicate, which takes a function f (from concrete states to abstract states), a concrete automaton C, and an abstract automaton A.

$$\begin{array}{l}
\mathit{is\_weak\_pmap}(f, C, A) \equiv \\
\quad (\forall s_0 \in \mathit{starts\_of}(C).\, f(s_0) \in \mathit{starts\_of}(A)) \wedge \\
\quad (\forall s\ t\ a.\ \mathit{reachable}(C, s) \wedge \langle s, a, t\rangle \in \mathit{trans\_of}(C) \\
\quad \supset \mathit{if}(a \in \mathit{externals}(\mathit{asig\_of}(C)), \langle f(s), a, f(t)\rangle \in \mathit{trans\_of}(A), f(s) = f(t)))
\end{array}$$

The following theorem proved in Isabelle states that the existence of an abstraction function from C to A implies that the traces of C are contained in those of A.

$$\begin{array}{l}
\mathit{IOA}(C) \wedge \mathit{IOA}(A) \wedge \\
\mathit{externals}(\mathit{asig\_of}(C)) = \mathit{externals}(\mathit{asig\_of}(A)) \wedge \\
\mathit{is\_weak\_pmap}(f, C, A) \\
\supset \mathit{traces}(C) \subseteq \mathit{traces}(A)
\end{array}$$

2.3 Renaming

As in [13] we define an operation for renaming actions. The motivation for this is modularity: name clashes can be avoided and generic components can be plugged into different environments.

$$\mathit{rename} : (\alpha, \sigma)\mathit{ioa} \to (\beta \to (\alpha)\mathit{option}) \to (\beta, \sigma)\mathit{ioa}$$

In contrast to [13] we define the action renaming function with type $\beta \to (\alpha)\mathit{option}$ instead of $\alpha \to \beta$. Therefore it does not have to be injective, which facilitates reasoning about such functions.

$$\begin{array}{l}
\mathit{rename}(A, f) \equiv \\
\qquad \langle\langle \{\mathit{act}\ .\ \exists \mathit{act}'.\ f(\mathit{act}) = \mathit{Some}(\mathit{act}') \wedge \mathit{act}' \in \mathit{inputs}(\mathit{asig\_of}(A))\}, \\
\qquad\quad \{\mathit{act}\ .\ \exists \mathit{act}'.\ f(\mathit{act}) = \mathit{Some}(\mathit{act}') \wedge \mathit{act}' \in \mathit{outputs}(\mathit{asig\_of}(A))\}, \\
\qquad\quad \{\mathit{act}\ .\ \exists \mathit{act}'.\ f(\mathit{act}) = \mathit{Some}(\mathit{act}') \wedge \mathit{act}' \in \mathit{internals}(\mathit{asig\_of}(A))\}\rangle, \\
\qquad \mathit{starts\_of}(A), \\
\qquad \{\langle s, \mathit{act}, t\rangle\ .\ \exists \mathit{act}'.\ f(\mathit{act}) = \mathit{Some}(\mathit{act}') \wedge \langle s, \mathit{act}', t\rangle \in \mathit{trans\_of}(A)\}\rangle
\end{array}$$

3 Specification

The Alternating Bit Protocol [3] is designed to ensure that messages are delivered in order, from a sender to a receiver, in the presence of channels that can lose and duplicate messages. This FIFO-communication can be specified by a simple queue and therefore a single automaton *Spec*. As we are aiming for a finite state system, we have to consider an additional point: The sender buffer of the implementation will not be able to store an unbounded number of incoming messages. Restricting the number of input actions to yield a finite sender buffer is not allowed because of the input-enabledness of IOA.

What we really need is an assumption about the behaviour of the environment, namely that it will only send the next message if requested to do so by an explicit action *Next* issued by the system. In the IOA-model this can be expressed by including an environment IOA which embodies this assumption. Therefore the specification is a parallel composition of two processes:

$$\mathit{Specification} \equiv \mathit{Env} \parallel \mathit{Spec}$$

and the interaction between them is shown in Fig. 1. The two components *Env* and *Spec* are described in the following subsections.

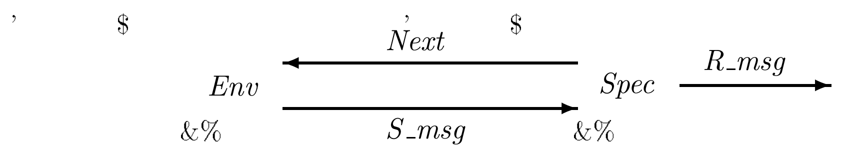


Figure 1: The Specification

3.1 The Environment

Env models the assumption that the environment only outputs *S_msg* when allowed to do so by *Next*. The state of *Env* is a single boolean variable *send_next*, initially true, which is set to true by every incoming *Next*. *S_msg* is enabled only if *send_next* is true and sets *send_next* to false as a result:

Next	input	$S\_msg(m)$	output
post:	$\mathit{send\_next}'$	pre:	$\mathit{send\_next}$
		post:	$\neg \mathit{send\_next}'$

where we use the following format to describe transition relations:

action	(input | output | internal)
pre:	P
post:	Q

Predicate P is the constraint on the state s that must hold for the transition to apply. If it is true, it is omitted. Predicate Q relates the state components before and after the transition; we refer to the state components after the transition by decorating their names with a $'$. If no state component changes, post is omitted.

3.2 The Specification

The state of the IOA *Spec* is a message queue q, initially empty, modelled with the type $(\mu)\mathit{list}$, where the parameter μ represents the message type. The only actions performed in the abstract system are: $S\_msg(m)$, putting message m at the end of q, $R\_msg(m)$, taking message m from the head of q, and *Next*, signaling the world outside to send the next message. Formally:

Next	output	$S\_msg(m)$	input	$R\_msg(m)$	output
pre:	true	post:	$q' = q@[m]$	pre:	$q = m :: \mathit{rst}$
				post:	$q' = \mathit{rst}$

4 Implementation

The system being proved correct also contains the component *Env* described in the previous section.

$$\mathit{Implementation} \equiv \mathit{Env} \parallel \mathit{Impl}$$

Impl represents the Alternating Bit Protocol and is itself a parallel composition of 4 processes:

$$\mathit{Impl} \equiv \mathit{Sender} \parallel S\_Ch \parallel \mathit{Receiver} \parallel R\_Ch$$

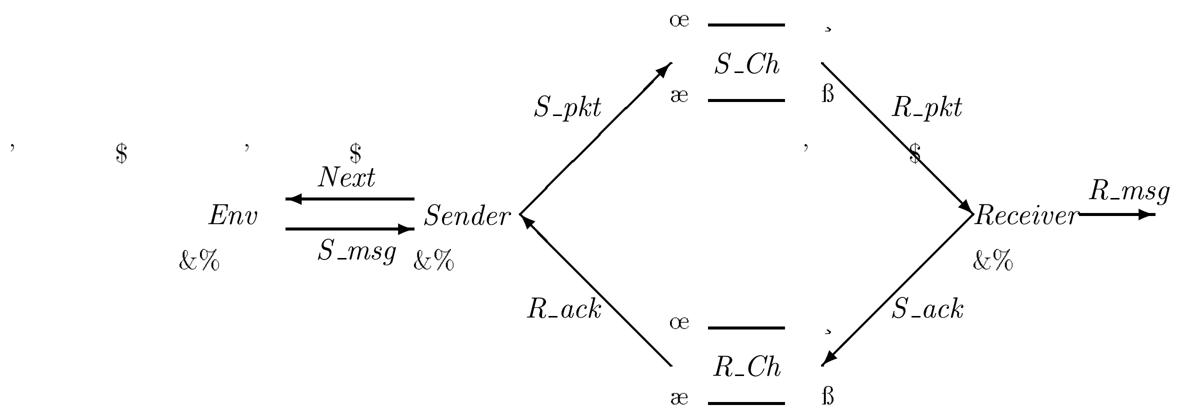


Figure 2: The Implementation

a sender, a receiver, and proprietary channels for both. The "dataflow" in the system is depicted in Fig. 2

Messages are transmitted from the sender to the receiver with a single header bit as packets of type $bool \times \mu$. The type of system actions, $(\mu)action$, is described in Isabelle by the following ML-style datatype:

$$\begin{aligned}(\mu)action \equiv\ & Next \mid S\_msg(\mu) \mid R\_msg(\mu) \mid S\_pkt(bool, \mu) \mid \\ & R\_pkt(bool, \mu) \mid S\_ack(bool) \mid R\_ack(bool)\end{aligned}$$

4.1 The Sender

The state of the process *Sender* is a pair:

Field	Type	Initial Value
message:	$(\mu)option$	*None*
header:	*bool*	*true*

The Sender makes the following transitions:

Next	output
pre:	$message = None$
$S\_msg(m)$	input
post:	$message' = Some(m)\ \wedge\ header' = header$
$S\_pkt(b, m)$	output
pre:	$message = Some(m)\ \wedge\ b = header$
$R\_ack(b)$	input
post:	**if** $b = header$
	then $message' = None\ \wedge\ header' = \neg header$
	else $message' = message\ \wedge\ header' = header$

Note that the presence of *Env*, i.e. the fact that the sender can control the flow of incoming messages via *Next*, enables us to get by with a buffer of length 1 (modelled by $(\mu)option$) in the sender; *Next* is only sent if the buffer is empty, i.e. $message = None$.

4.2 The Receiver

The state of the process *Receiver* is also a pair, differing from the Sender only in the initial value of the header variable:

Field	Type	Initial Value
$message$:	$(\mu)option$	$None$
$header$:	$bool$	$false$

The Receiver makes the following transitions:

$R\_msg(m)$	output
pre:	$message = Some(m)$
post:	$message' = None \ \wedge \ header' = header$
$R\_pkt(b, m)$	input
post:	**if** $b \neq header \wedge message = None$
	then $message' = Some(m) \ \wedge \ header' = \neg header$
	else $message' = message \ \wedge \ header' = header$
$S\_ack(b)$	output
pre:	$b = header$

Note that R_pkt does not change the state unless $message = None$. This ensures that the receiver has passed the last message on via R_msg before accepting a new one. Alternatively, one could add the precondition $message = None$ to S_ack which would preclude the sender getting an acknowledgment and sending a new message before the receiver has actually passed the old one on.

4.3 The Channels

The channels, R_Ch and S_Ch, have very similar functionality. Roughly speaking, messages are added to a queue by an input action and removed from it by the corresponding output action. In addition, there can be no change at all in order to model the possibility of losing messages, in case of the adding action, and of duplicating messages, in case of the removing action. The only differences between the channels are the type of the messages delivered, packets for S_Ch and booleans for R_Ch, and the specific names for input and output actions, S_pkt and R_pkt or S_ack and R_ack, respectively. Therefore both channels can be designed as instances of a generic channel using the renaming function described in section 2.

This is done by introducing a new datatype $(\alpha)act \equiv S(\alpha) \mid R(\alpha)$ of abstract actions and defining an IOA Ch with a single state component $q : (\alpha)list$ by the following transition relation:

$S(a)$	input	$R(a)$	output
post:	$q' = q \vee q' = q@[a]$	pre:	$q \neq [\,] \wedge a = hd(q)$
		post:	$q' = q \vee q' = tl(q)$

In Isabelle we use a set comprehension format to describe transition relations. In the case of Ch it looks like this:

$$
\begin{array}{llll}
Ch\_trans \ \equiv \ \{<s, act, s'> \,. & \texttt{case}\ act\ \texttt{of} & & \\
 & S(a) & \Rightarrow & s' = s \vee s' = s@[a] \\
 & R(a) & \Rightarrow & s \neq [\,] \wedge a = hd(s) \wedge \\
 & & & (s' = s \vee s' = tl(s))\,\}
\end{array}
$$

An automatic translation of the pre/post style into the set comprehension format is possible and desirable but not the focus of our research.

The concrete channels are obtained from the abstract channel by $rename(Ch, S\_acts)$ and $rename(Ch, R\_acts)$, where

$$
\begin{array}{lcl}
S\_acts & : & (\mu)action \to (bool \times \mu)\ act\ option \\
R\_acts & : & (bool)action \to (bool)\ act\ option
\end{array}
$$

map the concrete actions to the corresponding abstract actions. For example S_acts is defined by $S\_acts(S\_pkt(b,m)) = Some(S(<b,m>))$, $S\_acts(R\_pkt(b,m)) = Some(R(<b,m>))$ and $S\_acts(act) = None$ for all other actions act.

5 Abstraction

What we are aiming for is a finite-state description of the Alternating Bit Protocol that is refined by the given implementation described in the previous section. To achieve this, we have to remove two obstacles:

1. The channel queues have to be finite.
2. The message alphabet has to be finite.

5.1 Finite Channels

Our attention is focused on this requirement. We define an abstract version *RedCh* of *Ch* and an abstraction function *reduce* from *Ch* to *RedCh* and prove $is\_weak\_pmap(reduce, Ch, RedCh)$. The idea is based on the observation that at most two different messages are held in each channel. This is easily explained: each message is repeatedly sent to *S_Ch*, until the corresponding acknowledgment arrives. Once we switch to the next message, *S_Ch* can only contain copies of the previous message. Hence, *S_Ch*'s queue is always of the form old^*new^*. The same is true for *R_Ch*. Thus, if all adjacent identical messages are merged, the channels have size at most 2. Fortunately, this reasoning never needs to be formalized but is implicitly performed by the model checker.

5.1.1 Refinement of Channels

A compacting channel *RedCh* is obtained from *Ch* if new messages are only added provided they differ from the last one added. Thus *RedCh* is identical to *Ch* except for action S:

$$\begin{array}{lll} S(\alpha) & \text{input} & \\ \text{post:} & q' = q \vee & \textbf{if } a \neq hd(reverse(q)) \vee q = [\,] \\ & & \textbf{then } q' = q@[a] \\ & & \textbf{else } q' = q \end{array}$$

By renaming *RedCh* we obtain the collapsed versions of *R_Ch* and *S_Ch*, called *R_RedCh* and *S_RedCh*. Notice that the description is a priori not finite, as q is an unbounded list. Finiteness is only implied by the context, i.e. the behaviour of the protocol.

With the definition of an abstraction function *reduce*

$$\begin{array}{lcl} reduce([\,]) & \equiv & [\,] \\ reduce(x :: xs) & \equiv & \texttt{case}\ xs\ \texttt{of} \\ & & \quad [\,] \ \Rightarrow\ [x] \\ & & y :: ys \ \Rightarrow\ if(x = y, reduce(xs), x :: reduce(xs)) \end{array}$$

we get the following refinement goal:

$$is\_weak\_pmap(reduce, Ch, RedCh)$$

The proof of this obligation is rather straightforward, proceeding by case analysis on the type of actions. Using some lemmata on how *reduce* behaves when combined with operators like @ or

tl, most cases are automatically solved by the conditional and contextual rewriting of Isabelle. Finally, using the meta-theorem

$$\begin{array}{l} \mathit{is\_weak\_pmap}(\mathit{abs}, C, A) \\ \supset \mathit{is\_weak\_pmap}(\mathit{abs}, \mathit{rename}(C, f), \mathit{rename}(A, f)) \end{array}$$

we get the appropriate refinement results for the concrete channels S_Ch, R_Ch and their collapsed versions S_RedCh and R_RedCh.

5.1.2 Compositionality

In order to extend this refinement result from the channels to the whole system, we have to prove some compositionality theorems for refinements. Lynch and Tuttle [13] established the required lemma on the level of trace inclusions. We decided, however, to prove it on the level of abstraction functions for reasons of simplicity.

$$\begin{array}{l} \mathit{IOA}(C_1) \wedge \mathit{IOA}(C_2) \wedge \mathit{IOA}(A_1) \wedge \mathit{IOA}(A_2) \wedge \\ \mathit{externals}(\mathit{asig\_of}(C_1)) = \mathit{externals}(\mathit{asig\_of}(A_1)) \wedge \\ \mathit{externals}(\mathit{asig\_of}(C_2)) = \mathit{externals}(\mathit{asig\_of}(A_2)) \wedge \\ \mathit{compatible}(C_1, C_2) \wedge \mathit{compatible}(A_1, A_2) \wedge \\ \mathit{is\_weak\_pmap}(f, C_1, A_1) \wedge \mathit{is\_weak\_pmap}(g, C_2, A_2) \\ \supset \mathit{is\_weak\_pmap}(\lambda{<}c_1, c_2{>}.{<}f(c_1), g(c_2){>}, C_1 \parallel C_2, A_1 \parallel A_2) \end{array}$$

Unfortunately, trace inclusion does not imply the existence of an abstraction function. Hence the above theorem is not as general as the corresponding one about traces, in particular since $\mathit{is\_weak\_pmap}(id, A, A)$ only holds if A has no internal actions. We intend to formalize and prove compositionality on the trace level in the near future.

Performing the proofs of abstraction and compositionality in Isabelle, we encountered a mismatch between the time required for the refinement proof and that required for the compatibility checks. Nearly half the time (1.5 min on a SPARC station 10) was needed to establish that no component causes a name clash of input/output actions. These checks, although automated, are expensive if performed by a theorem prover. Partly this is caused by our decision to have *rename* translate action names in the opposite direction one would expect (see section 2.3), something we may need to rethink.

5.2 Finite Message Alphabet

The second requirement, the problem of abstracting out data from a data-independent program has already been addressed by Wolper [17]. In his paper he shows how to reduce an infinite data domain to a small finite one if data independence is guaranteed and the properties to be checked are expressible in propositional temporal logic. In [2] and [16] this method is applied to the Alternating Bit Protocol. There, only three different message values are needed to verify the protocol's functional correctness.

Basically, a program is data-independent if its behaviour does not depend on the specific data it operates upon. A sufficient condition for a program described by an IOA to be data independent is that everywhere in the automaton the transitions are independent of the value of messages being transmitted. An inspection of our description of the protocol shows that it satisfies the condition.

In contrast to [2] our specification is not given as a collection of temporal formulae, but in terms of I/O automata. Thus, the methods above are not directly applicable to our formalization and until now, we have not investigated how to transfer them formally into our setting. However, it is intuitively plausible that Wolper's theory of data-independence holds generally, independently of the respective formalization. That is why we analogously restricted our model checking algorithm to deal with only three different message values.

A formal treatment of data-abstraction in Isabelle/HOL needs a modification of the way we model data. Currently the diversity of data is modelled by polymorphic types[1]. But since types are a meta-level notion and cannot be talked about (e.g. quantified) in HOL, even formalizing data independence seems to be impossible. Using object-level sets instead of polymorphism would cure this problem but is likely to complicate the theory.

6 Model Checking

The task of the model checker is to verify that B, the implementation with collapsing channels refines A, the specification. It is done by a generic ML-function *check*

$$check(actions, internal, startsB, nextsB, startA, transA, abs)$$

where $actions : (\alpha)list$ is the list of all actions, $internal : \alpha \to bool$ recognizes internal actions of B, $startsB : (\sigma)list$ is the list of start states of B, $nextsB : \sigma \to \alpha \to (\sigma)list$ produces the list of successor states in B, $startsA : \tau \to bool$ recognizes start states of A, $transA : \tau \to \alpha \to \tau \to bool$ recognizes transitions of A, and $abs : \sigma \to \tau$ is the abstraction function.

It is easy to translate Isabelle's predicative description of A's transitions automatically into an ML-function *transA*. For *nextsB* this is only possible if the predicates have a certain recognizable form, for example disjunctions of assignments of values to the state components. Otherwise how are we to compute the set of next states satisfying an arbitrary predicate? If σ, the state space of B (as opposed to the set of reachable states!) is infinite, this is impossible. That is the main reason why we need to specify B, i.e. *RedCh* explicitly; otherwise we could have described *RedCh* implicitly in terms of *Ch* and *reduce*.

The abstraction function *abs* is given by

$$abs(s) \equiv l(R.message)@if(R.header = S.header, l(S.message), tl(l(S.message)))$$

where $l : (\alpha)option \to (\alpha)list$ is defined by the equations $l(Some(x)) = [x]$ and $l(None) = [\,]$. To distinguish between components of the receiver state and the sender state that have the same field names, we use a 'dotted identifier' notation, e.g. $S.header$ and $R.header$.

It is also possible to generate *abs* automatically as a set of corresponding state pairs as done in [10]. This would not allow the explicit documentation of *abs*, but it would mean a step forward towards fully automatic support — the major advantage of model checking.

check itself realizes the predicate $is\_weak\_pmap(abs, B, A)$ by simply performing full state space exploration. Beginning with *startsB* the algorithm examines all reachable states, checking for every transition $<s_1, a, s_2> \in trans\_of(B)$ that either $<abs(s_1), a, abs(s_2)> \in trans\_of(A)$ (if a is external) or $abs(s_1) = abs(s_2)$ (if a is internal).

At the moment the ML-code for the different arguments of *check* is still generated manually. However, we intend to automate this, subject to the restrictions on B described above. It should also be noted that *check* is just a prototype which should be replaced by some optimized model checker, for example the one described in [9].

References

[1] P. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 160–170. IEEE Press, 1993.

[1]It is not true that a polymorphic IOA is automatically data independent: HOL-formulae may contain the polymorphic equality "=" which destroys data independence.

[2] S. Aggarwal, C. Courcoubetis, and P. Wolper. Adding liveness properties to coupled finite-state machines. *ACM Transactions on Programming Languages and Systems*, 12(2):303–339, 1990.

[3] K. Bartlett, R. Scantlebury, and P. Wilkinson. A note on reliable full-duplex transmission over half-duplex lines. *Communications of the ACM*, 12(5):260–261, 1969.

[4] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proc. 19th ACM Symp. Principles of Programming Languages*, pages 343–354. ACM Press, 1992.

[5] D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET)*, pages 573–593. North-Holland, 1994.

[6] J.-C. Fernandez and L. Mounier. "On the Fly" verification of behavioural equivalences and preorders. In K. G. Larsen, editor, *Proc. 3rd Workshop Computer Aided Verification*, volume 575 of *Lect. Notes in Comp. Sci.*, pages 181–191. Springer-Verlag, 1992.

[7] R. Gawlick, R. Segala, J. Sogaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, Laboratory for Computer Science, MIT, Cambridge, MA., December 1993. Extended abstract in Proceedings ICALP'94.

[8] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In C. Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 71–84. Springer-Verlag, 1993.

[9] P. Herrmann, T. Kraatz, H. Krumm, and M. Stange. Automated verification of refinements of concurrent and distributed systems. Technical Report 541, Fachbereich Informatik, Universität Dortmund, 1994.

[10] P. Herrmann and H. Krumm. Report on analysis and verification techniques. Technical Report 485, Fachbereich Informatik, Universität Dortmund, 1993.

[11] H. Hungar. Combining model checking and theorem proving to verify parallel processes. In C. Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 154–165. Springer-Verlag, 1993.

[12] R. Kurshan. Reducibility in analysis of coordination. In K. Varaiya, editor, *Discrete Event Systems: Models and Applications*, volume 103 of *Lecture Notes in Control and Information Science*, pages 19–39. Springer-Verlag, 1987.

[13] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, MIT, Cambridge, MA., 1987.

[14] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[15] T. Nipkow and K. Slind. I/O automata in Isabelle/HOL. In *Proc. TYPES Workshop 1994*, Lect. Notes in Comp. Sci. Springer-Verlag. To appear.

[16] K. Sabnani. An algorithmic technique for protocol verification. *IEEE Transactions on Communications*, 36(8):924–930, 1988.

[17] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. Principles of Programming Languages*, pages 184–193. ACM Press, 1986.

A Constraint Oriented Proof Methodology based on Modal Transition Systems

Kim G. Larsen[1]
BRICS[2]
Aalborg Univ., Denmark,
kgl@iesd.auc.dk

Bernhard Steffen
FB Math. und Informatik,
Univ. of Passau, Germany,
steffen@fmi.uni-passau.de

Carsten Weise
LS für Informatik I,
Aachen Univ., Germany
carsten@informatik.rwth-aachen.de

Abstract

We present a constraint-oriented state-based proof methodology for concurrent software systems which exploits compositionality and abstraction for the reduction of the verification problem under investigation. Formal basis for this methodology are Modal Transition Systems allowing loose state-based specifications, which can be refined by successively adding constraints. Key concepts of our method are *projective views*, *separation of proof obligations*, *Skolemization* and *abstraction*. Central to the method is the use of *Parametrized* Modal Transition Systems. The method easily transfers to real-time systems, where the main problem are parameters in timing constraints.

1 Introduction

The use of formal methods and in particular formal verification of concurrent systems, interactive or fully automatic, is still limited to very specific problem classes. For state-based methods this is mainly due to the state explosion problem: the state graph of a concurrent systems grows exponentially with the number of its parallel components – and with the number of clocks in the real-time case –, leading to an unmanageable size for most practically relevant systems. Consequently, several techniques have been developed to tackle this problem. Here we focus on the four main streams and do not discuss the flood of very specific heuristics. Most elegant and ambitious are *compositional* methods (e.g. [ASW94, CLM89, GS90][3]), which due to the nature of parallel compositions are unfortunately rarely applicable. *Partial order* methods try to avoid the state explosion problem by suppressing unnecessary interleavings of actions [GW91, Val93, GP93]. Although extremely successful in special cases, these methods do not work in general. In practice, *Binary Decision Diagram*-based codings of the state graph are successfully applied to an interesting class of systems, see e.g. [Br86, BCMDH90, EFT91]. These codings of the state graph do not explode directly, but they may explode during verification, and it is not yet fully clear when this happens. All these techniques can be accompanied by *abstraction*: depending on the particular property under investigation, systems may be

[1]This author has been partially supported by the European Communities under CONCUR2, BRA 7166.

[2]Basic Research in Computer Science, Centre of the Danish National Research Foundation.

[3]In contrast to the first reference, the subsequent two papers address compositional reduction of systems rather than compositional verification.

dramatically reduced by suppressing details that are irrelevant for verification, see e.g. [CC77, CGL92, GL93]. Summarizing, all these methods cover very specific cases, and there is no hope for a uniform approach. Thus more application specific approaches are required, extending the practicality of formal methods.

We present a constraint-oriented state-based proof methodology for concurrent software systems which exploits compositionality and abstraction for the reduction of the verification problem under investigation. Formal basis for this methodology are Modal Transition Systems (MTS) [LT88] allowing loose state-based specifications, which can be refined by successively adding constraints. In particular, this allows extremely fine-granular specifications, which are characteristic for our approach: each aspect of a system component is specified by a number of independent constraints, one for each parameter configuration. This leads to a usually infinite number of extremely simple constraints which must all be satisfied by a corresponding component implementation. Beside exploiting compositionality in the standard (vertical) fashion, this extreme component decomposition also supports a horizontally compositional approach, which does not only separate proof obligations for subcomponents or subproperties but also for the various parameter instantiations. This is the key for the success of the following three step reduction, which may reduce even a verification problem for infinite state systems to a small number of automatically verifiable problems about finite state systems:

- *Separating the Proof Obligations.* Sections 4 and 5 present a proof principle justifying the separation and specialization of the various proof obligations, which prepare the ground for the subsequent reduction steps.

- *Skolemization.* The separation of the first step leaves us with problems smaller in size but larger in number. Due to the nature of their origin, these problems often fall into a small number of equivalence classes requiring only one prototypical proof each.

- *Abstraction.* After the first two reduction steps there may still be problems with infinite state graphs. However, the extreme specialization of the problem supports the power of abstract interpretation, which finally may reduce all the proof obligations to finite ones.

Our proof methodology is not complete, i.e., there is neither a guarantee for the possibility of a finite state reduction nor a straightforward method for finding the right amount of separation for the success of the succeeding steps or the adequate abstraction for the final verification. Still, as should be clear from the examples in the paper, there is a large class of problems and systems, where the method can be applied quite straightforwardly. Of course, the more complex the system structure the more involved will be the required search of appropriate granularity and abstraction.

Whereas complex data dependencies may exclude any possibility of 'horizontal' decomposition, our approach elegantly extends to real time systems, even over a dense time domain. In fact, this extension does not affect the possibility of a finite state reduction. For the real-time case, the basis are Timed Modal Transition Systems (TMS) [CGL93], where (weak) refinement is decidable. The TMS tool `Epsilon` (see again [CGL93]) can be used to find the refinements in demand.

However, in this paper *parametrized* timed modal transition systems are used. Parameters may appear either in actions (so-called *parametrized actions*) or in timing constraints.

While parametrized actions do not interfere with decidability questions, the decision procedures used in the Epsilon tool cannot be directly applied to transition systems with parameters in timing constraints.

We demonstrate our methodology by two examples: an extremely simple problem of pipelined buffers, and a specification problem of a Remote Procedure Call (RPC) posed by Broy and Lamport ([BL93]). The method is explained step by step by applying it to the simple example. For every step, we also give details on how to solve the RPC problem, thus demonstrating that the method is applicable to larger problems as well. Both problems have untimed and timed versions, where in the timed cases parameters are present in the timing constraints. For these special problems we are able to give a (simple) solution to the problem arising from parameters in timing constraints.

The next section recalls the basic theory of Modal Transition Systems, which we use for system specification. Thereafter we describe the RPC problem. The following sections explain our method in detail. Section 4 presents our notion of projective views and discusses the first reduction step. The subsequent two sections are devoted to the second and third reduction step, while Section 7 shows how to extend our method to real time systems over a dense time domain. Finally, Section 8 summarizes our conclusion and directions to future work.

2 Modal Transition Systems

In this section we give a brief introduction to the existing theory of modal transition systems. We assume familiarity with CCS. For more elaborate introductions and proofs we refer the reader to [LT88, HL89, Lar90].

When specifying reactive systems by traditional Process Algebras like e.g. CCS [Mil89], one defines the set of action transitions that can be performed (or observed) in a given system state. In this approach, any valid implementation *must* be able to perform the specified actions, which often constrains the set of possible implementations unnecessarily. One way of improving this situation within the framework of operational specification is to allow specifications where one can explicitly distinguish between transitions that are *admissible* (or allowed) and those that are *required*. This distinction allows a much more flexible specification and a much more generous notion of implementation, and therefore improves the practicality of the operational approach. Technically, this is made precise through the following notion of *modal transition systems*:

Definition 2.1 *A modal transition system is a structure* $\mathcal{S} = (\Sigma, A, \longrightarrow_\Box, \longrightarrow_\Diamond)$*, where* Σ *is a set of states,* A *is a set of actions and* $\longrightarrow_\Box, \longrightarrow_\Diamond \subseteq \Sigma \times A \times \Sigma$ *are transition relations, satisfying the consistency condition* $\longrightarrow_\Box \subseteq \longrightarrow_\Diamond$. □

Intuitively, the requirement $\longrightarrow_\Box \subseteq \longrightarrow_\Diamond$ expresses that anything which is required should also be allowed hence ensuring the consistency of modal specifications. When the relations $\longrightarrow_\Box$ and $\longrightarrow_\Diamond$ coincide, the above definition reduces to the traditional notion of labelled transition systems.

Syntactically, we represent modal transition systems by means of a slightly extended version of CCS. The only change in the syntax is the introduction of two prefix constructs $a_\Box.P$ and $a_\Diamond.P$ with the following semantics: $a_\Diamond.P \xrightarrow{a}_\Diamond P$, $a_\Box.P \xrightarrow{a}_\Box P$ and $a_\Box.P \xrightarrow{a}_\Diamond P$. The semantics for the other constructs follow the lines of CCS in the sense that each rule has a version for $\longrightarrow_\Box$ and $\longrightarrow_\Diamond$ respectively. We will call this version of CCS *modal* CCS.

As usual, we consider a design process as a sequence of *refinement steps* reducing the number of possible implementations. Intuitively, our notion of when a specification S refines another (weaker) specification T is based on the following simple observation. Any behavioural aspect *allowed* by a S should also be allowed by T; and dually, any behavioural aspect which is already guaranteed by the weaker specification T must also be guaranteed by S. Using the derivation relations $\longrightarrow_\Box$ and $\longrightarrow_\Diamond$ this may be formalized by the following notion of *refinement*:

Definition 2.2 *A refinement $\mathcal{R}$ is a binary relation on Σ such that whenever $S\,\mathcal{R}\,T$ and $a \in A$ then the following holds:*

1. *Whenever $S \xrightarrow{a}_\Diamond S'$, then $T \xrightarrow{a}_\Diamond T'$ for some T' with $S'\,\mathcal{R}\,T'$,*
2. *Whenever $T \xrightarrow{a}_\Box T'$, then $S \xrightarrow{a}_\Box S'$ for some S' with $S'\,\mathcal{R}\,T'$.*

S is said to be a refinement of T in case (S,T) is contained in some refinement $\mathcal{R}$. We write $S \lhd T$ in this case. $\Box$

Note that when we apply the above definition to traditional labelled transition systems (where $\longrightarrow = \longrightarrow_\Box = \longrightarrow_\Diamond$), we obtain the well–known notion of bisimulation [Par81, Mil89]. Using standard techniques, one straightforwardly establishes that $\lhd$ is a preorder preserved by all modal CCS operators.

$\lhd$ allows *loose* specifications. This important property, which can be best explained by looking at the 'weakest' specification $\mathcal{U}$ constantly allowing any action, but never requiring anything to happen. Operationally, $\mathcal{U}$ is completely defined by $\mathcal{U} \xrightarrow{a}_\Diamond \mathcal{U}$ for all actions a. It is easily verified that $S \lhd \mathcal{U}$ for any modal specification S.

Intuitively, S and T are *independent* if they are not contradictory, i.e. any action required by one is not constraint by the other. The following formal definition is due to the fact that for S and T to be *independent* all 'simultaneously'̀reachable processes S' and T' must be indenpendent too:

Definition 2.3 *An independence relation $\mathcal{R}$ is a binary relation on Σ such that whenever $S\,\mathcal{R}\,T$ and $a \in A$ then the following holds:*

1. *Whenever $S \xrightarrow{a}_\Box S'$, there is a unique T' such that $T \xrightarrow{a}_\Diamond T'$ and $S'\,\mathcal{R}\,T'$,*
2. *Whenever $T \xrightarrow{a}_\Box T'$, there is a unique S' such that $S \xrightarrow{a}_\Diamond S'$ and $S'\,\mathcal{R}\,T'$,*
3. *Whenever $S \xrightarrow{a}_\Diamond S'$ and $T \xrightarrow{a}_\Diamond T'$ then $S'\,\mathcal{R}\,T'$.*

S and T are said to be independent *in case (S,T) is contained in some independence relation $\mathcal{R}$.* $\Box$

Note in particular that two specifications are independent if none of them requires any actions. Independence is important, as it allows to define conjunction on modal transition systems by:

$$\frac{S \xrightarrow{a}_\Box S' \quad T \xrightarrow{a}_\Diamond T'}{S \wedge T \xrightarrow{a}_\Box S' \wedge T'} \qquad\qquad \frac{S \xrightarrow{a}_\Diamond S' \quad T \xrightarrow{a}_\Box T'}{S \wedge T \xrightarrow{a}_\Box S' \wedge T'}$$

$$\frac{S \xrightarrow{a}_{\Diamond} S' \qquad T \xrightarrow{a}_{\Diamond} T'}{S \wedge T \xrightarrow{a}_{\Diamond} S' \wedge T'}$$

Of course, $S \wedge T$ is always a well-defined modal specifications (i.e. any required transition is also allowed), and in fact, for independent arguments S and T it defines their *logical* conjunction:

Theorem 2.4 Let S and T be independent modal specifications. Then $S \wedge T \lhd S$ and $S \wedge T \lhd T$. Moreover, if $R \lhd S$ and $R \lhd T$ then $R \lhd S \wedge T$.

In order to compare specifications at different levels of abstraction, it is important to abstract from transitions resulting from internal communication. The way this is done for modal transition systems follows the lines of traditional labelled transition systems. That is, for a given modal transition system $\mathcal{S} = (\Sigma, A \cup \{\tau\}, \longrightarrow_{\Box}, \longrightarrow_{\Diamond})$ we derive the modal transition system $\mathcal{S}_{\varepsilon} = (\Sigma, A \cup \{\varepsilon\}, \Longrightarrow_{\Box}, \Longrightarrow_{\Diamond})$, where $\overset{\varepsilon}{\Longrightarrow}_{\Box}$ is the reflexive and transitive closure of $\xrightarrow{\tau}_{\Box}$, and where $T \overset{a}{\Longrightarrow}_{\Box} T', a \neq \varepsilon$, means that there exist T'', T''' such that

$$T \overset{\varepsilon}{\Longrightarrow}_{\Box} T'' \xrightarrow{a}_{\Box} T''' \overset{\varepsilon}{\Longrightarrow}_{\Box} T'$$

The relation $\Longrightarrow_{\Diamond}$ is defined in a similar manner.

The notion of *weak refinement* can now be introduced as follows: S weakly refines T in $\mathcal{S}$, $S \trianglelefteq T$, iff there exists a refinement relation on $\mathcal{S}_{\varepsilon}$ containing S and T.

Weak refinement $\trianglelefteq$ essentially enjoys the same pleasant properties as $\lhd$: it is a preorder preserved by all modal CCS operators except $+$ [HL89] (including restriction, relabelling and hiding). Moreover, for ordinary labelled transition systems weak refinement reduces to the usual notion of weak bisimulation ($\approx$).

3 The Remote Procedure Call Problem

We demonstrate our method by applying it to a specification problem given by Broy and Lamport. Due to space limitations we can only present part of the problem.

The original problem consists of a *memory component* and an *RPC mechanism.* The memory component accepts read and writes from several processes, and returns the requested values (none in case of write) or raises an exception. The only exception here is *memory failure*, i.e. the memory could not read from/write to the hardware. A component in which execptions do never occur is called a *reliable memory.*

The processes are connected to the memory component via an RPC (Remote Procedure Call) mechanism. The RPC mechanism simply forwards calls from the processes to the memory, and returns from the memory to the processes. The RPC should be transparent to the user, i.e. the composition of the memory component and the RPC should be an implementation of the memory. This is what we will call the *untimed RPC problem.*

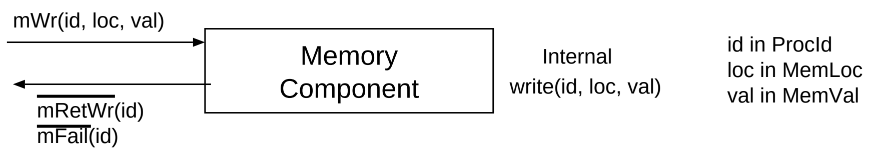
In the real-time case, the time to forward calls and returns by the RPC should be no more than δ. Further an exception should be raised if a call to the RPC does not return within $2\delta + \epsilon$ seconds. We will prove that if all calls to a reliable memory return within ϵ seconds, then the composition of the RPC and the reliable memory is an implementation of the reliable memory. This is the *timed RPC problem.*

The following is an informal specification of the memory component M, concentrating on write calls only. We assume sets `procId` of process identifiers, `memLocs` of memory locations and `memVals` of memory values, with typical elements `id`, `loc` and `val` resp. We will often use Z as an abbreviation for the product of the three sets, i.e. $Z :=$ `procId` $\times$ `memLocs` $\times$ `memVals`, with typical element $z \in Z$.

The events occurring in the memory component are described by *parameterized actions*, taking arguments from `procId`, `memLocs` and `memVals`. The actions of M are:

`mWr`(`id`, `loc`, `val`) :	write-call from process `id` of value `val` to location `loc`
`write`(`id`, `loc`, `val`) :	atomic write of value `val` to location `loc` initiated by process `id`
$\overline{\texttt{mRetWr}}$(`id`) :	send return from a write-request to process `id`
$\overline{\texttt{mFail}}$(`id`) :	signal memory failure to process `id`

The I/O-behaviour of the memory component M then looks like this:



The specification of the (reliable) memory component is a conjunction of the following properties:

$P0$ The memory component engages in actions only when it is called

$P1$ Each write operation (successful or not) performs a sequence of zero or more atomic writes of the correct value to the correct location at some time between the call and return. For a successful write operation, there must be at least one atomic write.

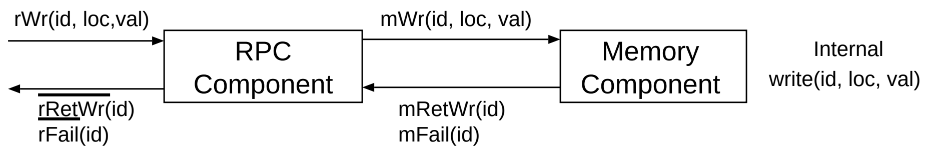
$P2$ A memory failure is never raised.

Clearly, the memory component M is specified by the conjunction of $P0$ and $P1$, while the reliable memory M_R is the conjunction of the M and $P2$.

The RPC R simply hands calls and returns (including the memory failure exception) through. These are the actions of the RPC:

`rWr`(`id`, `loc`, `val`)	:	remote write of value `val` to location `loc` issued by process `id`
$\overline{\texttt{rRetWr}}$(`id`)	:	return from remote write issued by process `id`
$\overline{\texttt{rFail}}$(`id`)	:	RPC returns an exception from a call issued by process `id`
$\overline{\texttt{mWr}}$(`id`, `loc`, `val`)	:	send a write of value `val` to location `loc` initiated by process `id`
`mRetWr`(`id`)	:	return from a write initiated by process `id`
`mFail`(`id`)	:	memory component raised a memory failure

The I/O-behaviour of the combined components can be depicted as:



In the next sections, we will explain our method directly using a much simpler example. At the end of each section we show how our method transfers to the RPC problem. We start with the untimed case.

4 Projective Views

In the following, we present, motivate and clarify our proof methodology by means of a minimal example, which is just sufficient to explain the various phenomena.

Consider the parallel system in Figure 1. Here two parameterized, disposable component media (supposed to transmit natural numbers) A and B are composed in parallel yielding a pipeline. Informally, the component A is supposed to input a natural number on port a, then output this number on port b after which it will terminate. The behaviour of B is similar. Using modal transition systems, the parallel system may be expressed as

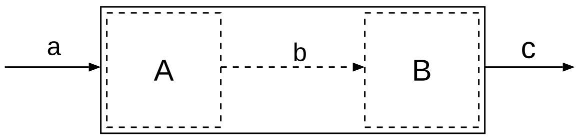


Figure 1: A Pipe Line of Two Disposable Media

follows:

$$\Big(\underbrace{a_{\Box}x.\overline{b_{\Box}}x}_{A} \mid \underbrace{b_{\Box}x.\overline{c_{\Box}}x}_{B}\Big)\backslash\{b\}$$

The behaviour of A and B are given by the two infinite–width transition systems of Figure 2. However, rather than using these direct specifications of A and B we specify the

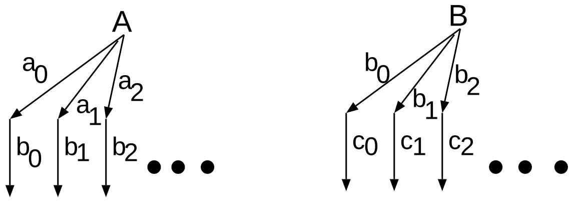
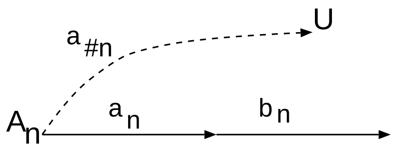


Figure 2: Behaviour of A and B.

two components behaviour using projective views A_n and B_n; one view for each possible natural number n. The projective view A_n specifies the *constraints* on the behaviour of the component A when focusing on transmission of the value n; this constraint can be expressed as the following modal transition system A_n (where we use solid lines for must- and dotted lines for may-transition):



Here $a_{\neq n}$ denotes all labels of the form a_m where $m \neq n$; also $\mathcal{U}$ denotes the universal modal transition system constantly allowing all actions. Note that this 'n-th view' imposes no constraint on the behaviour of A when transporting values different from n. The complete specification of the component A is the conjunction of all projective views[4] A_n. In fact it is easy to establish the following facts:

$$A \trianglelefteq \bigwedge_n A_n \quad \text{and} \quad \bigwedge_n A_n \trianglelefteq A \tag{1}$$

where A refers to the (infinite) transition system of Figure 2. Obviously, we may obtain similar projective views B_n for component B.

Let us now consider the problem of verifying that the overall system $\Big(A \,|\, B\Big)\backslash\{b\}$ is observationally equivalent to the system $C = a_{\square}x.\overline{c_{\square}}x$ (i.e. a slightly different disposable media). As A, B and C are standard transitions systems, i.e., everything allowed is also required, this problem is equivalent to showing

$$\Big(A \,|\, B\Big)\backslash\{b\} \ \trianglelefteq \ C$$

Thus (1), together with the observation that also C may be expressed as a conjunction of an infinite number of constraints C_n, leave us with the following refinement problem:

$$\Big(\bigwedge_n A_n \,|\, \bigwedge_n B_n\Big)\backslash\{b\} \ \trianglelefteq \ \bigwedge_n C_n \tag{2}$$

4.1 Application to the RPC problem

We give modal transition systems for the specification of properties $P0, P1$ and $P2$ of the memory component. Therefore we split $P1$ into two properties $P1a, P1b$ meaning

$P1a$ A write-call from process `id` cannot return unless an atomic write is performed.

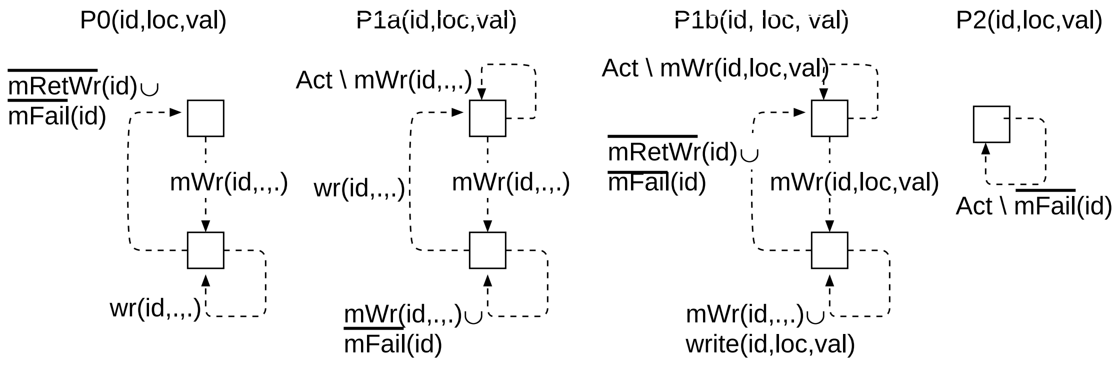
$P1b$ As long as a write-call from process `id` has not returned, no atomic write with a wrong location or value occurs

The labels in the following transition systems are sets of actions (called *abstracted actions*). A single action is a shorthand for the set containing this and only this action. For the other sets, we use the usual set-theoretic connectives, and a dot-notation, where a parametrized action with dots as parameters means "the set of all actions where the dotted position is replaced by all legal values for the parameter", e.g. for a fixed $\texttt{id} \in \texttt{procId}$, $\texttt{mWr}(\texttt{id}, ., .)$ is the set $\{\texttt{mWr}(\texttt{id}, \texttt{loc}, \texttt{val}) \mid \texttt{loc} \in \texttt{memLocs}, \texttt{val} \in \texttt{memVals}\}$.

Our specification assumes that calls from different processes are handled concurrently. As calls from different processes do not interfere, no actions parametrized with an identifier other than `id` is constrained in the specifications of calls from process `id`. This is modelled by allowing all actions with an identifier different from the fixed `id` in any state. Instead of adding to each state a loop where all these actions are allowed, we draw boxes meaning "a state with a loop for all non-`id` actions". By this the conjunction of the specifications for all processes is the same as their parallel composition.

The following modal transition systems specify the properties for a fixed value `id`:

[4]Note that all the projective views of A are pairwise independent.

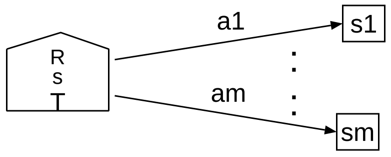


Note that only $P1b$ really depends on `loc` and `val`, and that the properties $P0, P1a, P1b$ and $P2$ are the conjunctions of the above modal specifications over all $z \in Z$.

Let $M(z)$ be the conjunction $P0(z) \wedge P1a(z) \wedge P1b(z)$, and $M_R(z) = M(z) \wedge P2(z)$. The memory component M is the conjunction of $M(z)$ over all $z \in Z$.

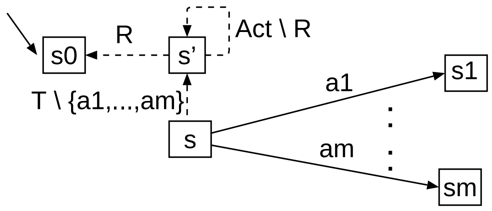
The transition systems for $P1a$ and $P1b$ are direct translation of their logical specifications using CTL with modalities. Transition systems resulting from this translation have may-transitions only, therefore conjunction of constraints is defined. CTL with modalities and its usefulness for our approach cannot be demonstrated here due to space limitations.

Let `Act` be the set of all actions. For two sets $R \subseteq$ `Act` (*return set*) and $T \subseteq$ `Act` (*tolerance set*), a state s and actions $a_1, \ldots, a_m \in$ `Act`. Then we use the following *macro state* for the specification of the RPC:



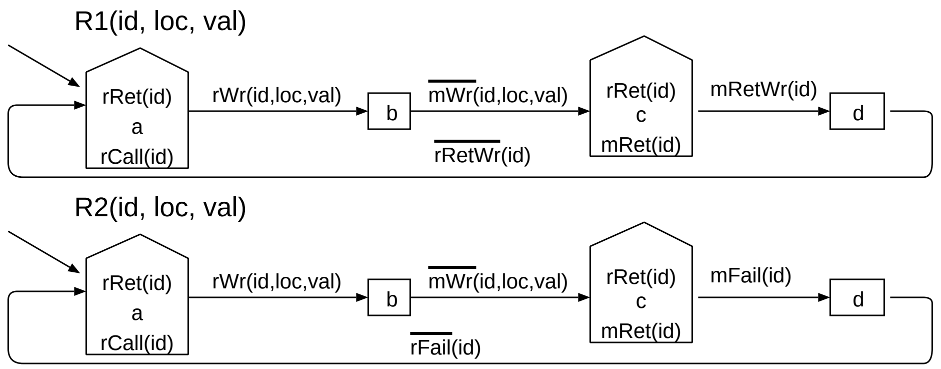
Here the edges leaving the "macro state" can be either may- or must-transition.

For a given transition system with start state s_0 and an auxiliary state s' not already in the transition system, this is meant to expand to



i.e. state s tolerates any action from T. If the behaviour of a tolerated action is already specified by an outgoing edge, nothing new happens. In the other case, the system goes to the auxiliary state s', where it accepts any action until a return action (from R) occurs. Return actions take the system back to the start state. Note that this corresponds to the idea of specifying procedures, where we specify each branch separately, and take care of possible other branches using the set T.

There are two main projective views of the RPC. In the first view, a write is handed through and a return received from the memory. In the second view, instead of a return a memory failure is received. These two views $R_1(\mathtt{id}, \mathtt{loc}, \mathtt{val})$ and $R_2(\mathtt{id}, \mathtt{loc}, \mathtt{val})$ are given in the following picture:



While it is natural to use must-transitions in the specification here, due to the fact that the memory component has no must-transitions at all it follows that must-transitions of the above specification can be replaced by may-transitions without affecting the correctness of the proof. This however makes all our specifications independent, so conjunction is defined. The sets in the macro states are defined as follows:

$$\begin{aligned} \mathtt{rCall}(\mathtt{id}) &:= \mathtt{rWr}(\mathtt{id},.,.) \\ \mathtt{rRet}(\mathtt{id}) &:= \overline{\mathtt{rRetWr}}(\mathtt{id}) \cup \overline{\mathtt{rFail}}(\mathtt{id}) \\ \mathtt{mRet}(\mathtt{id}) &:= \mathtt{mRetWr}(\mathtt{id}) \cup \mathtt{mFail}(\mathtt{id}) \end{aligned}$$

Let $R(z) := R_1(z) \wedge R_2(z)$. The untimed specification of the RPC R is the conjunction of $R(z)$ over all z.

Let f be a relabelling mapping all actions of the RPC to the appropriate actions of the memory component, and $A := \mathtt{Act}_M$ and $H := \mathtt{write}(.,.,.)$. Then the untimed verification problem is

$$\Big(R \,|\, M/H\Big)\backslash A[f] \;\trianglelefteq\; M/H \tag{3}$$

where the internal actions of the memory (i.e. the atomic writes) are hidden.

5 Sufficient Proof Condition

Due to the properties of conjunction (cf. Lemma 2.4) the proof of (2) can obviously be reduced to the verification of

$$\Big(\bigwedge_n A_n \,|\, \bigwedge_n B_n\Big)\backslash\{b\} \;\trianglelefteq\; C_n$$

for each natural n. Thus due to Lemma 2.4 and the fact that $\trianglelefteq$ is preserved by parallel composition and restriction, it suffices to prove

$$\forall n \in N.\; \Big(A_n \,|\, B_n\Big)\backslash\{b\} \;\trianglelefteq\; C_n \tag{4}$$

There is a general proof principle behind this reduction: in order to conclude:

$$\Big(\bigwedge_{i\in I_1} A_i^1 \,|\, \ldots \,|\, \bigwedge_{i\in I_k} A_i^k\Big)\backslash L \;\trianglelefteq\; \bigwedge_{j\in I} C_j$$

it suffices for each $j \in I$ to establish:

$$\Big(\bigwedge_{i \in I_{1,j}} A_i^1 \,|\, \ldots \,|\, \bigwedge_{i \in I_{k,j}} A_i^k \Big) \backslash L \;\trianglelefteq\; C_j$$

where $I_{\ell,j} \subseteq I_\ell$ for each $\ell = 1 \ldots k$.

Of course, in general the power of this proof principle strongly depends on a good choice of the $I_{\ell,j}$, which was trivial in our example.

5.1 Application to the RPC Problem

With the same argumentation, to prove (3) it is sufficient to show

$$\forall z \in Z.\ \Big(R(z) \,|\, M(z)/H \Big) \backslash A[f] \;\trianglelefteq\; M(z)/H \tag{5}$$

6 Skolemization and Abstraction

So far we have reduced the overall verification problem of (2) to that of (4). At first sight this doesn't seem much of a reduction as (4) requires a refinement proof to be established for each natural number. Fortunately, these proofs are not really sensitive to the actual value of the natural number n. Letting k be an arbitrary natural number (or a Skolem constant) it suffices to prove:

$$\Big(A_k \,|\, B_k \Big) \backslash \{b\} \;\trianglelefteq\; C_k \tag{6}$$

in order to infer (4). Thus we are now left with the problem of establishing a *single* refinement. But still, though finite *state* the specifications A_k and B_k both have infinitely many *transitions* (as $a_{\neq k}$ is an inifinite label set). This problem can be overcome using *abstraction* (or *factorization*) with respect to an appropriate equivalence relation.

Definition 6.1 *Let $\mathcal{S} = (\Sigma, S, \longrightarrow_\Box, \longrightarrow_\Diamond)$ be a modal transition system, let $\equiv_\Sigma$ and $\equiv_S$ be equivalence relations on Σ and S, and let $\Sigma^\equiv$ and $S^\equiv$ be the sets of equivalence classes. Then* the factorization of $\mathcal{S}$ *is the modal transition system $\mathcal{S}^\equiv = (\Sigma^\equiv, S^\equiv, \longrightarrow'_\Box, \longrightarrow'_\Diamond)$, where $\longrightarrow'_\Box, \longrightarrow'_\Diamond$ are defined as follows:*

$$\frac{s \xrightarrow{a}_\Box s'}{[s]^\equiv \xrightarrow{[a]^\equiv}{}'_\Box [s']^\equiv} \qquad \frac{s \xrightarrow{a}_\Diamond s'}{[s]^\equiv \xrightarrow{[a]^\equiv}{}'_\Diamond [s']^\equiv}$$

Equivalence relations $\equiv_\Sigma$ and $\equiv_S$ are called compatible *with the modal transition system $\mathcal{S}$ iff for all $a \equiv_\Sigma b, s \equiv_S t, s' \equiv_S t'$:*

$$s \xrightarrow{a}_\Box s' \quad \textit{iff} \quad t \xrightarrow{b}_\Box t' \qquad \textit{and} \qquad s \xrightarrow{a}_\Diamond s' \quad \textit{iff} \quad t \xrightarrow{b}_\Diamond t'$$

For compatible equivalence relations, the following reduction lemma is straightforward:

Lemma 6.2 *Let $S^\equiv$ and $T^\equiv$ be processes in the factorization $\mathcal{S}^\equiv$ of $\mathcal{S}$ with respect to compatible equivalence relations $\equiv_\Sigma$ and $\equiv_S$. Then we have for arbitrary representatives S of $S^\equiv$ and T of $T^\equiv$:*

$$S^\equiv \trianglelefteq T^\equiv \quad \textit{implies} \quad S \trianglelefteq T$$

This Lemma allows us to reduce verification problems for infinite systems to problems for finite systems, as soon as an appropriate factorization can be found.

For our example, let us consider the equivalence relation $\equiv$ defined by $x_k \equiv x_k$ and $x_i \equiv x_j$ whenever $i, j \neq k$, where x ranges over $\{a, b, c\}$. Obviously, $\equiv$ is compatible with the underlying transition system. Thus the verification of (2) can further be reduced to the refinement proof between the finite $\equiv$–abstracted versions of A_k, B_k and C_k

$$\Big({A_k}^{\equiv} \,|\, {B_k}^{\equiv} \Big) \backslash \{b\} \ \trianglelefteq\ {C_k}^{\equiv} \tag{7}$$

which can easily be done by means of the automatic verification tool `Epsilon`.

6.1 Application to the RPC Problem

The same is true for the RPC problem: instead of proving (5) for all z, a proof for a prototypical z is sufficient. Most of the abstraction is already carried out by using abstracted actions. Note however that the abstracted actions are in general *not* the required equivalence classes. For the RPC problem e.g. $\mathtt{write}(z)$ is an equivalence class of its own, and the set $\mathtt{write}(\mathtt{id}, ., .) \setminus \mathtt{write}(z)$ is another equivalence classes. This specific partitioning of the atomic write actions reflects the fact that we must distinguish between a write of the correct value to the correct location and all other writes from the same process.

From the diagrams in Sect. 4.1 it is easy to see that the resulting transition systems are of a manageable size, and the proof can be carried out using the `Epsilon` tool.

7 Specifications with Time

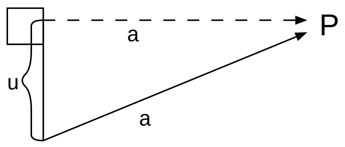
The above example can be extended to deal with real time. For the specification we use Wang Yi's Timed CCS (see [Yi91]) together with modal specifications. For details on these so called *Timed Modal Specifications* see [CGL93]. This method can be used with any totally ordered time domain, while in the following we will assume the positive real numbers.

The passing of time is modelled by a delay action $\varepsilon(d)$, where d is a positive real number. The intuitive meaning of such a delay is that d time units pass until the end of this action. Normal actions are enabled immediately, and can be taken at any time. As an example, the process $a_\Box x.\varepsilon(2).\overline{b_\Box} x$ can execute $a_\Box x$ at any time. Thereafter it must delay for at least two time units before it can engage in $b_\Box x$.

Further we assume *maximal progress*, i.e. a communication must be performed as soon as possible. Putting $a_\Box x.\varepsilon(2).\overline{b_\Box} x$ in parallel with $\overline{a_\Box} x.\varepsilon(3).b_\Box x$ would force the communication via channel a to take place immediately, and the communication via channel b to happen after exactly three time units.

For our specification, the macro $a[l, u]$ is convenient, where a is an action and l, u are real numbers with $l < u$. The intuition is that a process $a[l, u].P$ *may* enable a after l time units and *must* enable a after u time units. In other words, communication via a may be possible after at least l time units, and will be possible at any time after u time units. This macro is defined as $a[l, u].P = (\varepsilon(l).a_\Diamond + \varepsilon(u).a_\Box).P$.

In our examples, the lower bound is always zero. The graphical presentation we use for $a[0, u].P$ is:



Let d be a fixed real number. Then we specify a timed process $A(d)$, which reads port a and subsequently outputs its input onto port b within d time units, by $a_{\Box}x.\bar{b}x[0,d]$.Note that this is a timed version of process A. The same construction gives timed versions $B(d)$ and $C(d)$ of B and C.

We are now going to establish that a 'pipeline' with two components with delay d should not be slower than one component with delay $2d$, i.e.

$$\Big(A(d) \mid B(d)\Big) \setminus \{b\} \;\trianglelefteq\; C(2d)$$

The same method as in the untimed case reduces the situation to

$$\Big(A_k{}^{\equiv}(d) \mid B_k{}^{\equiv}(d)\Big) \setminus \{b\} \;\trianglelefteq\; C_k{}^{\equiv}(2d)$$

for a Skolem constant k and the equivalence relation of the previous section. Now, given a specific value for d this proof can be carried out using the Epsilon tool, which treats real valued timer domains by means of the clock region automaton technique (see [AD94] for details). This technique relies on integer values for all explicit timer constants in the specification, which can be achieved by multiplication with an appropriate constant in most applications. As all timer constants are multiplied by the same constant, this does not affect the principle behaviour of the system. In our example, the obvious choice for this constant is $1/d$, leaving us with the following refinement problem

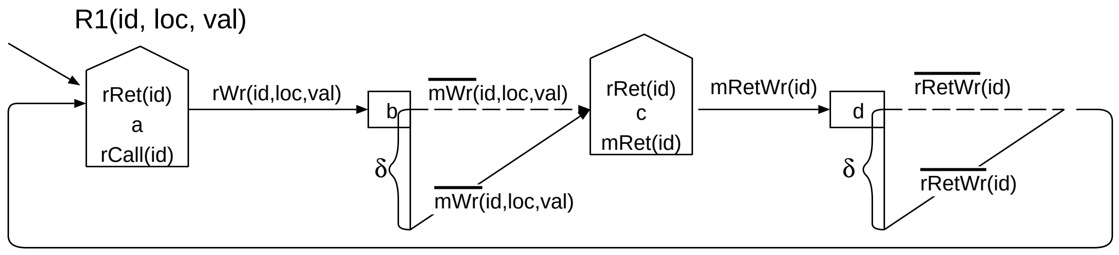
$$\Big(A_k{}^{\equiv}(1) \mid B_k{}^{\equiv}(1)\Big) \setminus \{b\} \;\trianglelefteq\; C_k{}^{\equiv}(2)$$

which can be solved using Epsilon.

Note that this proof indeed covers the statement for any d. Thus even in the presence of real time, the original verification problem is reduced to a very simple, automatically solvable problem.

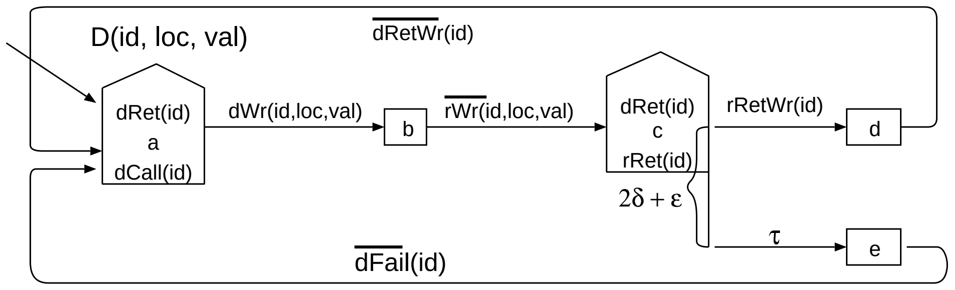
7.1 Application to the RPC Problem

The following is a timed version of R_1, where passing through the calls and returns takes not more than δ seconds:

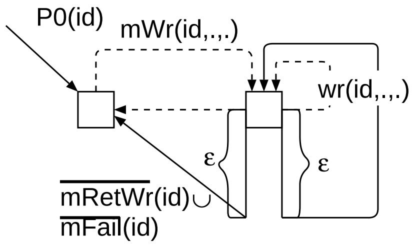


Note that actions without a timing constraint are enabled at any time. The timed version of R_2 is defined analogously (although unnecessary for the reliable memory). Call the timed RPC R^{δ}.

In the same way as the RPC we specify a demon which signals a failure if a call to the RPC does not return within $2\delta + \epsilon$ seconds. The actions of the demon are the same as those of the RPC, only the prefix r is replaced by a d. Timeout is modelled by a τ-transition. The specification of the demon $D_1(z)$ is



To define a timed reliable memory, we only need to alter property $P0$ by requiring the return to occur within ϵ seconds. This is done by the following:



We call the resulting timed specification of the reliable memory M_R^ϵ. The timed verification problem then is

$$\left(D^{2\delta+\epsilon} \,|\, R^\delta \,|\, M_R^\epsilon/H\right)\backslash A[f] \;\trianglelefteq\; M_R/H$$

Note that the memory on the right hand side is the "untimed" M_R, where we interprete all actions to be enabled all the time. Further the set A and the relabelling f have to be adjusted.

This problem can once again be reduced by our method to a problem where we only need to look at a prototypical z. The involved transition systems are of small size. The problem however are the parameters occuring in the timing constraints. With two parameters δ and ϵ present the multiplication trick does not work as before.

There is a simple workaround. Computing $R^\delta \,|\, M_R^\epsilon/H$ by hand one finds a transiton system where the expression $2\delta + \epsilon$ occurs in all timing constraints. Thus $2\delta + \epsilon$ can be replaced by d, leaving a system with only d as parameter in timing constraints. Now the previously used multiplication trick is applicable, so that the problem can be verified using Epsilon.

8 Conclusion and Future Work

We have introduced a new constraint-oriented method for the (automated) verification of concurrent systems. Key concepts of our 'divide and conquer' method are *projective views*, *separation of proof obligations*, *Skolemization* and *abstraction*, which together support a drastic reduction of the complexity of the relevant subproblems. Of course, our proof methodology does neither guarantee the possibility of a finite state reduction nor a straightforward method for finding the right amount of separation or the adequate abstraction. Still, there is a large class of problems and systems, where the method can be applied quite straightforwardly. Typical examples are systems with limited data dependence. Whereas involved data dependencies may exclude any possibility of 'horizontal' decomposition, our approach elegantly extends to real time systems, even over a dense time domain. In fact, the resulting finite state problems can be automatically verified using the `Epsilon` verification system. All this has been illustrated using a simple example of pipelined buffers. Our experience indicates that our method scales up to practically relevant problems, as demonstrated by the problem of the transparent RPC. Due to space limitations, we did not show how to do a logical specification in CTL with modalities. Such specifications are very natural problem descriptions, and they are in general easy to translate into Modal Transition Systems.

Beside further case studies and the search for good heuristics for proof obligation separation and abstraction, we are investigating the limits of tool support during the construction of constraint based specifications and the application of the three reduction steps. Whereas support by graphical interfaces and interactive editors is obvious and partly implemented in the META-Framework, a management system for synthesis, analysis and verification currently developed at the university of Passau, the limits of consistency checking and tool supported search for adequate separation and abstraction are still an interesting open research topic.

As pointed out, one major problem are parameters in the timing constraints. We are currently investigating methods – similar to the approach presented for parametrized timed automaton in [AHV93] – for checking bisimulation and (weak) refinement for *parametrized modal transition systems.*

References

[ASW94] H. Andersen, C. Stirling, G. Winskel. A Compositional Proof System for the Modal Mu-Calculus. in: Proceedings LICS, 1994.

[AD94] R. Alur, D.L. Dill. A Theory of Timed Automata. in: Theoretical Computer Science Vol. 126, No. 2, April 1994, pp. 183-236.

[AHV93] R. Alur, T.A. Henzinger, M.Y. Vardi. *Parametric real-time reasoning.* Proc. 25th STOC, ACM Press 1993, pp. 592–601.

[BL93] M. Broy, L. Lamport. Specification Problem. Case study for the Dagstuhl Seminar 9439, 1994.

[Br86] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. in: IEEE Transactions on Computation, 35 (8). 1986.

[BCMDH90] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. in: Proceedings LICS'90.

[CGL93] K. Čerāns, J.C. Godesken, K.G. Larsen. Timed Modal Specification - Theory and Tools. in: C. Courcoubetis (Ed.), Proc. 5th Int. Conf. on Computer Aided Verification (CAV '93), Elounda, Greece, June/July 1993. LNCS 697, Springer Berlin 1993, pp. 253–267.

[CGL92] E. Clarke, O. Grumber, D. Long. Model Checking and Abstraction. in: Proceedings XIX POPL'92.

[CLM89] E. Clarke, D. Long, K. McMillan. Compositional Model Checking. in: Proceedings LICS'89.

[CC77] R. Bryant. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction and Approximation of Fixpoints. in: Proceedings POPL'77.

[EFT91] R. Enders, T. Filkorn, D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. in: Proceedings CAV'91, LNCS 575, 1991, pp. 203–213

[GW91] P. Godefroid, P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. in: Proceedings CAV'91, LNCS 575, 1991, pp. 332–342.

[GP93] P. Godefroid, D. Pirottin. Refining Dependencies Improves Partial-Order Verification Methods. in: Proceedings CAV'93, LNCS 697, 1991, pp. 438–449.

[GL93] S. Graf, C. Loiseaux. Program Verification using Compositional Abstraction. in: Proceedings FASE/TAPSOFT'93.

[GS90] S. Graf, B. Steffen. Using Interface Specifications for Compositional Minimization of Finite State Systems. in: Proceedings CAV'90.

[HL89] H. Hüttel and K. Larsen. The use of static constructs in a modal process logic. Proceedings of Logic at Botik'89. LNCS 363, 1989.

[Lar90] K.G. Larsen. Modal specifications. In: Proceedings of Workshop on Automatic Verification Methods for Finite State Systems LNCS 407, 1990.

[LT88] K. Larsen and B. Thomsen. A modal process logic. In: Proceedings LICS'88, 1988.

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

[Par81] D. Park. Concurrency and automata on infinite sequences. In P. Deussen (ed.), 5th GI Conference, LNCS 104, pp. 167–183, 1981.

[Val93] A. Valmari. On-The-Fly Verification with Stubborn Sets. in: C. Courcoubetis (Ed.), Proc. 5th Int. Conf. on Computer Aided Verification (CAV '93), Elounda, Greece, June/July 1993. LNCS 697, Springer Berlin 1993, pp. 397–408.

[Yi91] W. Yi. CCS + Time = an Interleaving Model for Real-Time Systems, Proc.18th Int. Coll. on Automata, Languages and Programming (ICALP), Madrid, July 1991. LNCS 510, Springer New York 1991, pp. 217-228.

HyTech : The Cornell HYbrid TECHnology Tool*

Thomas A. Henzinger Pei-Hsin Ho

Computer Science Department
Cornell University
(tah|ho)@cs.cornell.edu

Abstract. HyTech, the Cornell Hybrid Technology Tool, is an automatic tool for analyzing hybrid systems. We review some of the formal technologies that have been incorporated into HyTech, and we illustrate the use of HyTech with two nontrivial case studies.

1 Introduction

Hybrid systems are digital real-time systems that interact with the physical world through sensors and actuators. Due to the rapid development of digital processor technology, hybrid systems directly control much of what we depend on in our daily lives. Many hybrid systems, ranging from automobiles to aircraft, operate in safety-critical situations, and therefore call for rigorous analysis techniques.

HyTech[1] is a symbolic model checker for linear hybrid systems. The underlying system model is *hybrid automata*, an extension of finite automata with continuous variables that are governed by differential equations [ACHH93]. The requirement specification language is the *integrator computation tree logic* Ictl, a branching-time logic with clocks and stop-watches for specifying timing constraints. Safety, liveness, real-time, and duration requirements of hybrid systems can be specified in Ictl [AHH93]. Given a hybrid automaton describing a system and an Ictl formula describing a requirement, HyTech computes the state predicate that characterizes the set of system states that satisfy the requirement.

In this report we review the formal technologies that have been incorporated into HyTech for solving reachability problems of linear hybrid automata; more advanced applications of HyTech, for Ictl model checking and the analysis of nonlinear hybrid systems [HH95], are detailed in the full version of this paper. In Section 2, we define the syntax and semantics of linear hybrid automata, which were introduced in [ACHH93, NOSY93]. In Section 3, we give an introduction to the reachability analysis of linear hybrid automata, which was presented in [AHH93, ACH$^+$95]. We concentrate on the reachability analysis of systems with unknown delay parameters, and use HyTech to derive sufficient and necessary conditions on the parameters such that the system satisfies a given safety requirement. We also demonstrate the use of abstract-interpretation operators, which are discussed in greater detail in [HH94]. Throughout, we use a temperature controller for a toy nuclear reactor as a running example to illustrate the use of HyTech. For the practitioning verifier, we present the actual input language for describing linear hybrid automata and verification commands.

In Section 4, we apply HyTech to two nontrivial benchmark problems. Both examples are taken from the literature, rather than devised by us. The first case study is a distributed control system introduced by Corbett [Cor94]. The system consists of a controller and two sensors, and is required to issue control commands to a robot within certain time limits. The two sensor processes are executed on a single processor, as scheduled by a priority scheduler. This scenario is modeled by linear hybrid automata with clocks and

*This research was supported in part by the National Science Foundation under grant CCR-9200794, by the United States Air Force Office of Scientific Research under contract F49620-93-1-0056, and by the Defense Advanced Research Projects Agency under grant NAG2-892.

[1]HyTech is available by anonymous ftp from ftp.cs.cornell.edu, cd ~pub/tah/HyTech. See also http://www.cs.cornell.edu/Info/People/tah/hytech.html.

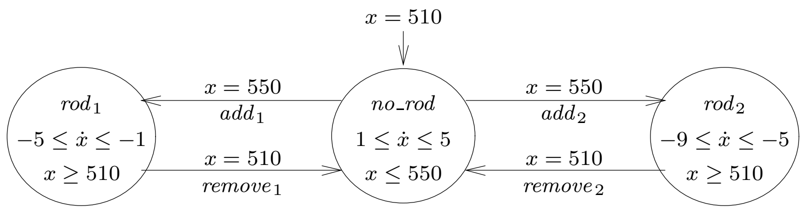


Figure 1: The reactor core automaton

stop-watches. HyTech automatically computes the maximum time difference between two consecutive control commands generated by the controller. It follows, for example, that a scheduler that gives higher priority to one sensor may meet the specification requirement, while a scheduler that gives priority to the other sensor may fail the requirement.

The second case study is a two-robot manufacturing system introduced by Puri and Varaiya [PV95]. The system consists of a conveyor belt with two boxes, a service station, and two robots. The boxes will not fall to the floor iff initially the boxes are not positioned closely together on the conveyor belt. HyTech automatically computes the minimum allowable initial distance between the two boxes.

2 Specification of Linear Hybrid Automata in HyTech

The system modeling language of HyTech is linear hybrid automata [AHH93]. Intuitively, a linear hybrid automaton is a labeled multigraph (V, E) with a finite set X of real-valued variables. The edges in E represent discrete system actions and are labeled with guarded assignments to X. The vertices in V represent continuous environment activities and are labeled with constraints on the variables in X and their first derivatives. The state of a hybrid automaton changes either through instantaneous system actions or, while time elapses, through continuous environment activities.

Example: reactor temperature control

We use a variant of the reactor temperature control system from [NOSY93] as a running example. The system consists of a reactor core and two control rods that control the temperature of the reactor core. The reactor core is modeled by the linear hybrid automaton in Figure 1. The temperature of the reactor core is represented by the variable x. Initially the core temperature is 510 degrees and both control rods are not in the reactor core. In this case, the core temperature rises at a rate that varies between 1 and 5 degrees per second. Notice that $\dot{x}$ is the first derivative of the variable x. The reactor is shut down once the core temperature increases beyond 550 degrees. In order to prevent a shutdown, one of two control rods can be put into the reactor core to dampen the reaction. If control rod 1 is put in, the core temperature falls at a rate that may vary between -5 and -1 degrees per second. Control rod 2 has a stronger effect; if it is put in, the core temperature falls at a rate that varies between -9 and -5 degrees per second. Either control rod is removed once the core temperature falls back to 510 degrees.

2.1 Syntax

A *linear term* over a set X of real-valued variables is a linear combination of variables with integer coefficients. A *linear inequality* over X is a nonstrict inequality between linear terms over X. A *linear hybrid automaton* A consists of the following components.

Data variables A finite ordered set $X = \{x_1, x_2, \ldots, x_n\}$ of real-valued *data variables*. For example, the reactor core automaton from Figure 1 has the single data variable x.

A *data state* is a point $(a_1, a_2, \ldots, a_n)$ in the n-dimensional real space $\mathbb{R}^n$ or, equivalently, a function that maps each variable x_i to a real value a_i. A *convex data region* is a convex polyhedron in $\mathbb{R}^n$, and a *data region* is a finite union of convex data regions. A *convex data predicate* is a conjunction of linear inequalities, and a *data predicate* is a disjunction of convex data predicates. Every (convex) data predicate ϕ defines a (convex) data region $[\![\phi]\!]$ of data states that satisfy ϕ.

Control locations A finite set V of vertices called *control locations*. For example, the reactor core automaton has the three control locations no_rod, rod_1, and rod_2.

A *state* (v, s) of the hybrid automaton A consists of a control location $v \in V$ and a data state $s \in \mathbb{R}^n$. A *region* $\bigcup_{v \in V}\{(v, S_v)\}$ is a collection of data regions $S_v \subseteq \mathbb{R}^n$, one for each control location $v \in V$. A *state predicate* is a collection $\bigcup_{v \in V}\{(v, \phi_v)\}$ of data predicates ϕ_v, one for each control location $v \in V$. When writing state predicates, we use the *location counter* l, which ranges over the set V of control locations. The location constraint $l = v$ denotes the state predicate $\{(v, \mathit{true})\} \cup \bigcup_{v' \neq v}\{(v', \mathit{false})\}$. Each state predicate $\bigcup_{v \in V}\{(v, \phi_v)\}$ defines the region $\bigcup_{v \in V}\{(v, [\![\phi_v]\!])\}$.

Location invariants A labeling function *inv* that assigns to each control location $v \in V$ a convex data predicate $inv(v)$, the *invariant* of v. The automaton control may reside in location v only as long as the invariant $inv(v)$ is true; so the invariants enforce progress in a hybrid automaton. The state (v, s) is *admissible* if the data state s satisfies the invariant $inv(v)$. We write Σ_A for the region $\bigcup_{v \in V}\{(v, [\![inv(v)]\!])\}$ of all admissible states of A.

In the reactor core automaton, we have $inv(no\_rod) = (x \leq 550)$, $inv(rod_1) = (x \geq 510)$, and $inv(rod_2) = (510 \leq x)$. In HyTech, we specify these invariants as follows:

```
inv[l[core] == norod] = x<=550
inv[l[core] == rodone] = 510<=x
inv[l[core] == rodtwo] = 510<=x
```

In the graphical representation of a hybrid automaton, we suppress invariants of the form *true*.

Continuous activities A labeling function *dif* assigns to each control location $v \in V$ and each data variable $x_i \in X$ a *rate interval* $dif(v, x_i) = [a_i, b_i]$, where a_i and b_i are integer constants. The rate interval $dif(v, x_i) = [a_i, b_i]$ specifies that the first derivative of the data variable x_i may vary within the interval $[a_i, b_i] \subset \mathbb{R}$ while the automaton control resides in location v. If $a_i = b_i$, then $dif(v, x_i)$ is called the *slope* of x_i in location v. A data variable is a *discrete variable* if it has the slope 0 in all locations; a *clock*, if it has the slope 1 in all locations; and a *stop-watch*, if in each location it has either the slope 1 or the slope 0.

In the reactor core automaton, $dif(no\_rod, x) = [1, 5]$, $dif(rod_1, x) = [-5, -1]$, and $dif(rod_2, x) = [-9, -5]$. In HyTech, we specify these rate intervals as follows:

```
dif[core,norod,x] = {1,5}
dif[core,rodone,x] = {-5,-1}
dif[core,rodtwo,x] = {-9,-5}
```

In the graphical representation of a hybrid automaton, we write $\dot{x} = a$ short for $\dot{x} \in [a, a]$, and we suppress rate intervals of the form $\dot{x} = 0$.

Transitions A finite multiset E of edges called *transitions*. Each transition (v, v') identifies a source location $v \in V$ and a target location $v' \in V$. For example, the core automaton has four transitions.

Synchronization letters A finite set L of letters called *synchronization alphabet*, and a labeling function *syn* that assigns to each transition $e \in E$ a letter from L. The synchronization letters are used to define the parallel composition of hybrid automata. The reactor core automaton has the four synchronization letters add_1, add_2, $remove_1$, and $remove_2$. In the graphical representation of a hybrid automaton, we suppress synchronization letters that do not occur in the alphabet of any other automata.

Discrete actions A labeling function *act* that assigns to each transition $e \in E$ a guarded command $act(e) = (\phi \rightarrow \alpha)$. The *guard* ϕ is a convex data predicate. The *command* α is a set of assignments $x_i := t_i$, at most one for each data variable $x_i \in X$, such that each t_i is a linear term over X. We write $dom(\alpha)$ for the set of variables that make up the left-hand sides of the assignments in α, and $t_i(s)$ for the value of the linear term t_i if interpreted in the data state s. The command α defines a function on data states that leaves the variables outside $dom(\alpha)$ unchanged: for all data states s, $\alpha_i(s) = t_i(s)$ if $x_i \in dom(\alpha)$, and $\alpha_i(s) = s_i$ if $x_i \notin dom(\alpha)$, where $\alpha_i(s)$ denotes the i-th component of the data state $\alpha(s)$. A *parameter* is a discrete variable that does not occur in the domain $dom(\alpha)$ of any command α.

In the reactor core automaton, $act(no\_rod, rod_1) = (x = 550 \rightarrow \emptyset)$, etc. In HyTech, we specify the transitions, synchronization letters, and guarded commands of the reactor core automaton as follows:

```
act[core,1] = { l[core]==norod && 550==x, add1, {l[core] -> rodone}}
act[core,2] = { l[core]==norod && 550==x, add2, {l[core] -> rodtwo}}
act[core,3] = { l[core]==rodone && 510==x, remove1, {l[core] -> norod}}
act[core,4] = { l[core]==rodtwo && 510==x, remove2, {l[core] -> norod}}
```

Notice that we encode the source and target locations of a transition within a guarded command. In the graphical representation of a hybrid automaton, we write α for the guarded command $true \rightarrow \alpha$, and ϕ for the guarded command $\phi \rightarrow \emptyset$, and we suppress the guarded command $true \rightarrow \emptyset$.

2.2 Semantics

At any time instant, the state of a hybrid automaton specifies a control location and the values of all data variables. The state can change in two ways: (1) by an instantaneous discrete transition that changes both the control location and the values of data variables, or (2) by a time delay that changes only the values of data variables in a continuous manner according to the rate intervals of the corresponding control location. Accordingly, we define the following two binary relations on the admissible states of the given automaton A.

Transition step For all admissible states (v, s) and (v', s') of A, and all synchronization letters σ, let $(v, s)\xrightarrow{\sigma}(v', s')$ iff there exists a transition e from v to v' such that (1) $syn(e) = \sigma$ and (2) $act(e) = (\phi \rightarrow \alpha)$ with $s \in [\![\phi]\!]$ and $s' = \alpha(s)$.

Time step For all admissible states (v, s) and (v, s') of A, and all nonnegative reals $\delta \geq 0$, let $(v, s)\xrightarrow{\delta}(v, s')$ iff there is a differentiable function $\rho\colon [0, \delta] \rightarrow \mathbb{R}^n$ such that (1) $f(0) = s$, (2) $f(\delta) = s'$, (3) for all reals $t \in [0, \delta]$, $\rho(t) \in [\![inv(v)]\!]$, and (4) for all reals $t \in (0, \delta)$ and each data variable x_i, $d\rho_i(t)/dt \in dif(v, x_i)$, where $\rho_i(t)$ denotes the i-th component of the data state $\rho(t)$.

The linear hybrid automaton A defines the labeled transition system $[\![A]\!] = \langle \Sigma_A, \mathcal{L}, \rightarrow_A \rangle$ that consists of the infinite state space Σ_A, the infinite label set $\mathcal{L} = L \cup \mathbb{R}_{\geq 0}$, and the binary transition relation $\rightarrow_A = \bigcup\{\xrightarrow{\sigma} \mid \sigma \in L\} \cup \bigcup\{\xrightarrow{\delta} \mid \delta \geq 0\}$ on Σ_A

For a region S, we define $pre(S)$ to be the set of all states σ such that $\sigma \rightarrow_A \sigma'$ for some state $\sigma' \in S$. Similarly, we define $post(S)$ to be the set of all states σ such that $\sigma' \rightarrow_A \sigma$ for some state $\sigma' \in S$. Both $pre(S)$ and $post(S)$ are again regions [AHH93]. We write $pre^*(S)$ for the infinite union $\bigcup_{i \geq 0} pre^i(S)$, and $post^*(S)$ for the infinite union $\bigcup_{i \geq 0} post^i(S)$. In other words, $pre^*(S)$ is the set of all states that can reach a state in S by a finite sequence of transitions of the labeled transition system $[\![A]\!]$; and $post^*(S)$ is the set of all states that can be reached from a state in S by a finite sequence of transitions of $[\![A]\!]$.

2.3 Parallel Composition

A hybrid system typically consists of several components that operate concurrently and communicate with each other. We describe each component as a linear hybrid automaton. The component automata may coordinate either through shared variables or via synchronization letters. The linear hybrid automaton that models the entire system is then constructed from the component automata using a product operation.

Let $A_1 = (X_1, V_1, inv_1, dif_1, E_1, L_1, syn_1, act_1)$ and $A_2 = (X_2, V_2, inv_2, dif_2, E_2, L_2, syn_2, act_2)$ be two linear hybrid automata. The product automaton $A_1 \times A_2$ generally interleaves the transitions of the component automata A_1 and A_2. If, however, a transition e_1 of A_1 is labeled with a synchronization letter σ that is contained also in the alphabet of A_2, then e_1 can be executed only simultaneously with a σ-labeled transition of A_2. Formally, the *product* $A_1 \times A_2$ is the linear hybrid automaton $A = (X_1 \cup X_2, V_1 \times V_2, inv, dif, E, L_1 \cup L_2, syn, act)$:

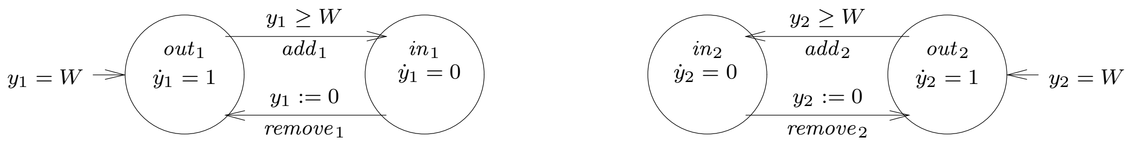


Figure 2: The control rod automata

- Each location (v, v') in $V_1 \times V_2$ has the invariant $inv(v, v') = (inv_1(v) \wedge inv_2(v'))$. For each variable $x \in X_1 \backslash X_2$, $dif((v, v'), x) = dif_1(v, x)$; for each variable $x \in X_2 \backslash X_1$, $dif((v, v'), x) = dif_2(v', x)$; and for each shared variable $x \in X_1 \cap X_2$, $dif((v, v'), x) = dif_1(v, x) \cap dif_2(v', x)$.
- E contains the transition $e = ((v_1, v_2), (v_1', v_2'))$ iff

  (1) $e_1 = (v_1, v_1') \in E_1$, $v_2 = v_2'$, and $syn_1(e_1) \notin L_2$; or
  (2) $e_2 = (v_2, v_2') \in E_2$, $v_1 = v_1'$, and $syn_2(e_2) \notin L_1$; or
  (3) $e_1 = (v_1, v_1') \in E_1$, $e_2 = (v_2, v_2') \in E_2$, and $syn_1(e_1) = syn_2(e_2)$.

  Suppose that $act_1(e_1) = (\phi_1 \to \alpha_1)$, and $act_2(e_2) = (\phi_2 \to \alpha_2)$. In case (1), $syn(e) = syn_1(e_1)$ and $act(e) = act_1(e_1)$. In case (2), $syn(e) = syn_2(e_2)$ and $act(e) = act_2(e_2)$. In case (3), $syn(e) = syn_1(e_1) = syn_2(e_2)$; moreover, $act(e) = (\phi_1 \wedge \phi_2 \to \alpha_1 \cup \alpha_2)$ if $dom(\alpha_1) \cap dom(\alpha_2) = \emptyset$, and $act(e) = false \to \emptyset$ if $dom(\alpha_1) \cap dom(\alpha_2) \neq \emptyset$,

HyTech automatically constructs the product automaton from a set of input automata.

For the reactor example, we use the two linear hybrid automata of Figure 2 to model the two control rods. Due to the mechanics of moving control rods, after a control rod is removed from the reactor core, it cannot be put back into the core for W seconds, where W is an unknown parameter. This requirement is enforced by the stop-watch y_1 that measures the time that has elapsed since control rod 1 was removed from the reactor core, and the stop-watch y_2 that measures the time that has elapsed since control rod 2 was removed. The rod automata synchronize with the core automaton through synchronization letters such as $remove_1$, which indicates the removal of control rod 1. The entire reactor system, then, is obtained by constructing the product of the core automaton of Figures 1 and the two rod automata of Figure 2.

We now show how the complete reactor temperature control system is specified in HyTech. First we declare the data variables:

```
AnaVariables = {x, y1, y2}
DisVariables = {w}
```

The data variables x, $y1$, and $y2$ are analog variables, and the data variable W is a discrete variable. We have already defined the reactor core automaton. Now we define the two control rod automata:

```
inv[l[rod1] == out] = 0<=y1
inv[l[rod1] == in] = 0<=y1
inv[l[rod2] == out] = 0<=y2
inv[l[rod2] == in] = 0<=y2

dif[rod1,in,y1] = {0,0}
dif[rod1,out,y1] = {1,1}
dif[rod2,in,y2] = {0,0}
dif[rod2,out,y2] = {1,1}

act[rod1,1] = { l[rod1]==out && w<=y1, add1, {l[rod1] -> in}}
act[rod1,2] = { l[rod1]==in, remove1, {l[rod1] -> out, y1 -> 0}}
act[rod2,1] = { l[rod2]==out && w<=y2, add2, {l[rod2] -> in}}
act[rod2,2] = { l[rod2]==in, remove2, {l[rod2] -> out, y2 -> 0}}
```

The synchronization alphabet of each automaton is defined by declaring a scope for each synchronization letter. The *scope* of the letter σ is the set of automata that contain σ in their synchronization alphabet. For the reactor temperature control system, we specify

```
    syn[remove1] = {rod1,core}
    syn[remove2] = {rod2,core}
    syn[add1] = {rod1,core}
    syn[add2] = {rod2,core}
```

For example, the letter $remove_1$ is used by the reactor core automaton and by the first control rod automaton. This means that the core automaton and the rod 1 automaton must synchronize on transitions labeled with $remove_1$.

While we have given symbolic names like *core* and *no_rod* to automata and locations, the analysis procedures of HyTech require that all automaton names and location names are integers starting from 1. To replace the symbolic names with integers, HyTech calls a macro language preprocessor when it reads an input file. Therefore, we need to define the integer values of the symbolic names at the beginning of the input file. The symbolic names that we use for the reactor temperature control system may be defined as follows:

```
    define(rod1,1)
    define(rod2,2)
    define(core,3)
    define(rodone,1)
    define(rodtwo,2)
    define(norod,3)
    define(out,1)
    define(in,2)
```

We also must declare the number of input automata, and the number of locations and transitions of each automaton:

```
    AutomataNo = 3
    locationo = {2,2,3}
    transitiono = {2,2,4}
```

The expression `locationo = {2,2,3}` means that the first (control rod 1), second (control rod 2), and third (reactor core) automaton has 2, 2, and 3 locations, respectively. The expression `transitiono = {2,2,4}` specifies the number of transitions in each input automaton.

Global invariants for modeling urgent transitions

Although the product automaton is constructed automatically by HyTech, it is sometimes useful to specify global conjuncts of all invariants of the product automaton. Such global invariants permit, in particular, the modeling of *urgent transitions*, which are transitions that must be taken as soon as possible. In the graphical representation of hybrid automata, we use boldface synchronization letters to mark urgent transitions. HyTech allows the user to specify location invariants for locations of the product automaton using the command `GlobalInvar`. We will show how urgent transitions can be modeled with global invariants as we analyze the examples of Section 4. The reactor temperature control system does not have any urgent transitions, so we write:

```
    GlobalInvar = {}
```

This completes the specification of the reactor temperature control system. Except for initial `define` statements, all HyTech input commands can be written in any order.

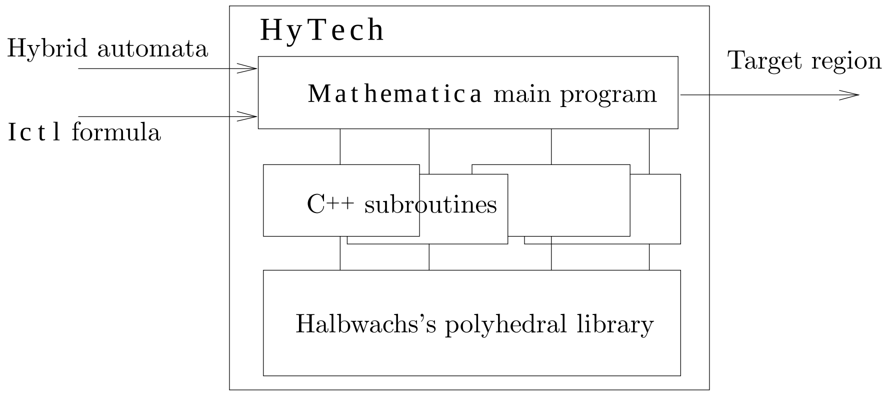


Figure 3: The architecture of HyTech

3 Symbolic Analysis of Linear Hybrid Automata in HyTech

The core of HyTech is a symbolic model-checking procedure, whose primitives are boolean, *pre*, and *post* operations on regions. The original implementation of HyTech represented regions as state predicates and manipulated regions by syntactic operations on formulas. We have improved the performance of HyTech by representing and manipulating regions geometrically: each data region is represented as a union of convex polyhedra. The current implementation of HyTech consists of a Mathematica main program and a collection of C++ subroutines that make use of a polyhedron-manipulation library by Halbwachs [Hal93, HRP94]. The architecture of HyTech is shown in Figure 3.

In this paper we do not discuss the full model-checking capabilities of HyTech, but restrict ourselves to reachability analysis, which amounts to checking safety requirements.

3.1 Reachability Analysis

The *reachability problem* $(A, \varphi_I, \varphi_F)$ for a linear hybrid automaton A, an initial state predicate φ_I, and a final state predicate φ_F, asks if the region $post^*([\![\varphi_I]\!]) \cap [\![\varphi_F]\!]$ is empty or, equivalently, if the region $[\![\varphi_I]\!] \cap pre^*([\![\varphi_F]\!])$ is empty. In other words, the reachability problem $(A, \varphi_I, \varphi_F)$ asks if there is a finite path in the underlying transition system $[\![A]\!]$ from some state in $[\![\varphi_I]\!]$ to some state in $[\![\varphi_F]\!]$. If $[\![\varphi_I]\!]$ represents the set of "initial" states of the automaton A, and $[\![\varphi_F]\!]$ represents the set of "unsafe" states specified by a safety requirement, then the safety requirement can be verified by reachability analysis: the automaton satisfies the safety requirement iff the reachability problem has the answer *yes* (i.e., $post^*([\![\varphi_I]\!]) \cap [\![\varphi_F]\!] = \emptyset$).

Unfortunately, the computation of $post^*([\![\varphi_I]\!])$ or $pre^*([\![\varphi_F]\!])$ may not terminate within a finite number of *post* or *pre* operations, because the reachability problem for linear hybrid automata is undecidable [ACHH93]. HyTech, in other words, offers a semidecision procedure for the reachability analysis. It is our experience, however, that for practical examples, including the examples in this paper, the computation does terminate and HyTech solves the corresponding reachability problems. Indeed, as for the practitioner there is little difference between a nonterminating computation and one that runs out of time or space resources, we submit that decidability questions are mostly of theoretical interest.

For the reactor temperature control system, we wish to check the safety requirement that *the reactor never needs to be shut down*; more precisely, whenever the core temperature reaches 550 degrees, then either y_1 or y_2 shows more than W seconds, thus allowing the corresponding control rod to be put into the reactor core. Let A denote the product of the reactor core automaton and the two control rod automata. We define the reachability problem $(A, \varphi_I, \varphi_F)$ as follows. The initial states are characterized by the state predicate

$$\varphi_I \ = \ (l[rod_1] = out \ \wedge \ l[rod_2] = out \ \wedge \ l[core] = no\_rod \ \wedge \ x = 510 \ \wedge \ y_1 = W \ \wedge \ y_2 = W);$$

that is, initially no rod is in the reactor core, the initial temperature is 510 degrees, and $y_1 = y_2 = W$ (we write $l[c]$ for the component of the location counter l that is associated with the component automaton c;

so $l[core]$ ranges over the locations of the reactor core automaton, etc.). The unsafe states are characterized by the state predicate

$$\varphi_F \;=\; (l[rod_1] = out \,\wedge\, l[rod_2] = out \,\wedge\, l[core] = no\_rod \,\wedge\, x = 550 \,\wedge\, y_1 \le W \,\wedge\, y_2 \le W);$$

that is, the unsafe situation is that the core temperature reaches 550 degrees and neither y_1 nor y_2 shows more than W seconds (and, thus, none of the control rods is available). The answer to the reachability problem $(A, \varphi_I, \varphi_F)$ is *yes* iff the reactor temperature control system satisfies the safety requirement.

In HyTech, the reachability problem is specified as follows:

```
InitialState = l[rod1]==out && l[rod2]==out && l[core]==norod && 510==x && w==y1
    && w==y2
Bad = l[rod1]==out && l[rod2]==out && l[core]==norod && 550==x && y1<=w && y2<=w
```

Forward versus backward analysis

HyTech can attack a reachability problem by forward analysis or by backward analysis. Given the reachability problem $(A, \varphi_I, \varphi_F)$, the *forward analysis* computes the state predicate that defines the region $post^*([\![\varphi_I]\!])$, and then takes the conjunction with the final state predicate φ_F; the *backward analysis* computes the state predicate that defines the region $pre^*([\![\varphi_F]\!])$, and then takes the conjunction with the initial state predicate φ_I. For a given reachability problem, one direction may perform better than the other direction. In fact, it may be that one direction terminates and the other does not. For example, only the backward analysis terminates for the reactor temperature control system.

We ask HyTech to perform a forward or backward analysis, respectively, by writing

```
Go := PrintTime[ Forward ]
```

or

```
Go := PrintTime[ Backward ]
```

These commands also print the CPU time consumed by the reachability analysis.

Parametric analysis

The automatic derivation of delay parameters was introduced for real-time systems in [AHV93] and applied to hybrid systems in [AHH93]. We can use HyTech to synthesize necessary and sufficient conditions on system parameters such that a hybrid automaton satisfies a requirement.

Recall that the reactor temperature control system contains the parameter w, which specifies the necessary rest time for a control rod. Clearly, the safety requirement will not be satisfied for large values of w. Indeed, the *target region* $[\![\varphi_I]\!] \cap pre^*([\![\varphi_F]\!])$ gives a sufficient and necessary condition on w such that the safety requirement is not satisfied. Typically the state predicate that defines the target region is too complex to see the conditions on the parameters clearly, but these can be isolated in HyTech using *projection operators*. By writing

```
EliminateLocList = {rod1,rod2,core}
EliminateVarList = {x,y1,y2}
```

we eliminate all location information from the state predicate that defines the target region, and we project out all information about the data variables x, y_1, and y_2. Then the resulting projection of the target region, as computed by HyTech using backward analysis, is

```
9w => 184
```

In other words, the target region is empty if and only if $9W < 184$. It follows that $9W < 184$ is a necessary and sufficient condition on the parameter W that prevents the reactor from shutdown. The verification requires 17.27 seconds of CPU time.[2]

[2]All performance figures are given for a SPARC 670MP station.

3.2 Abstract Interpretation

To expedite the reachability analysis and to force the termination of the analysis, HyTech provides several abstract-interpretation operators [CC77, HH94], including the convex-hull operator and the extrapolation operator. Our extrapolation operator is similar to the widening operator of [CH78, Hal93].

An abstract-interpretation operator approximates a set of convex data regions with a single convex data region. The *convex-hull operator* overapproximates a union of convex data regions by its convex hull. The *extrapolation operator* overapprovimates a directed chain $S \subset f(S) \subset f^2(S) \subset \cdots$ of convex data regions by a "guess" of the limit region $\bigcup_{i \geq 0} f^i(S)$. Either operator, or the combination of both operators, may cause the termination of a forward or backward reachability analysis that does not terminate otherwise. However, since the use of either operator results in an overapproximation of the target region, the abstract analysis is sound but not complete: if HyTech returns the answer *yes* to a reachability problem, then the approximate target region is empty, and therefore also the exact target region must be empty; but if the answer is *no*, then the exact target region may still be empty, and the correct answer to the reachability problem may be *yes*. In the latter case, we have to refine our approximation, by applying fewer abstract-interpretation operators, or by using two-way iterative approximation (see below).

In HyTech, we write

```
TakeConvex = True
```

or

```
TakeConvex = False
```

to turn the convex-hull operator on or off, respectively. The extrapolation operator can be turned on or off selectively for individual control locations. For example, if we want to apply the extrapolation operator only to data regions that correspond to the two locations $l[rod_1] = out \wedge l[rod_2] = in \wedge l[core] = no\_rod$ and $l[rod_1] = in \wedge l[rod_2] = out \wedge l[core] = no\_rod$ of the reactor temperature control system, then we write:

```
ExtraSet[lc_] = (lc === l[rod1]==out && l[rod2]==in && l[core]==norod) ||
(lc === l[rod1]==in && l[rod2]==out && l[core]==norod)
```

The commands

```
ExtraSet[lc_] = True
```

and

```
ExtraSet[lc_] = False
```

ask HyTech to apply the extrapolation operator to all or none of the control locations, respectively. (In our analysis of the reactor temperature control system, it was not necessary to use any abstract-interpretation operators.)

Two-way iterative approximation

If inconlusice, the approximate reachability analysis can be refined by alternating approximate forward and backward analysis [CC92, DW95]. If any abstract-interpretation operators are used, HyTech automatically performs a two-way iterative analysis, beginnig with the specified forward or backward pass. The readers should refer to [HH94] for the details about the two-way iterative analysis of hybrid systems.

4 Two Case Studies

We report on the application of HyTech to two nontrivial benchmark problems.

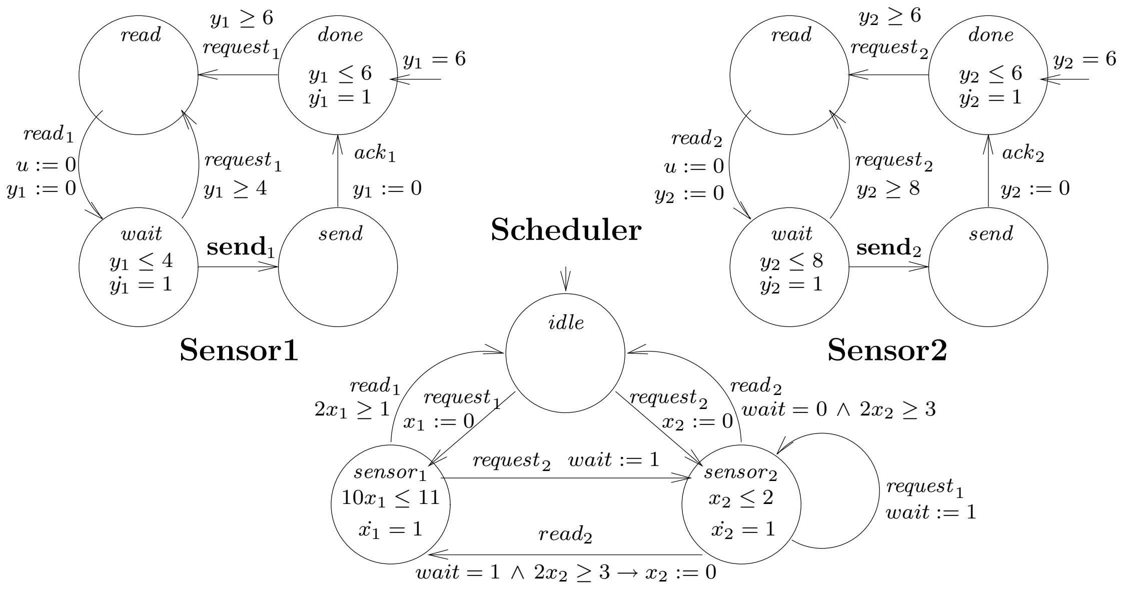


Figure 4: The two sensors and the scheduler

4.1 A Distributed Control System with Time-outs

The distributed control system of [Cor94] consists of two sensors and a controller that generates control commands to a robot according to the sensor readings. The programs for the two sensors and the controller are written in Ada. The two sensors share a single processor, and the priority of sensor 2 for using the processor is higher than the priority of sensor 1. In other words, if both sensor 1 and sensor 2 want to use the processor to construct a reading, only sensor 2 obtains the processor, and sensor 1 has to wait. The two sensors are modeled by the top two linear hybrid automata in Figure 4 and the priorities for using the shared processor are modeled by the scheduler automaton in the same figure.

Each sensor constructs a reading and sends the reading to the controller. The shared processor for constructing sensor readings is requested via *request* transitions, the completion of a reading is signaled via *read* transitions, and the reading is delivered to the controller via **send** transitions. Sensor 1 takes 0.5 to 1.1 milliseconds and sensor 2 takes 1.5 to 2 milliseconds of CPU time to construct a reading. These times are measured by the stop-watches x_1 and x_2 of the scheduler automaton. Notice that at most one of the two stop-watches x_1 and x_2 runs in a location, which reflects the fact that only one sensor can use the shared processor at a time. If sensor 1 loses the processor because of preemption by sensor 2, it can continue the construction of its reading after the processor is released by sensor 2. The value of the discrete variable *wait* is 1 if sensor 1 is waiting for the processor, and 0 otherwise.

Once constructed, the reading of sensor 1 expires if it is not delivered within 4 milliseconds, and the reading of sensor 2 expires if it is not delivered within 8 milliseconds. These times are measured by the clocks y_1 and y_2 of the sensor automata. If a reading expires, then a new reading must be constructed. After successfully delivering a reading, a sensor sleeps for 6 milliseconds (measured again by the clocks y_1 and y_2), and then constructs the next reading.

The controller is modeled by the automaton in Figure 5. The controller is executed on a dedicated processor, so it does not compete with the sensors for CPU time. We use the clock z to measure the delays and time-outs of the controller. The controller accepts and acknowledges a reading from each sensor, in either order, and then computes and sends a command to the robot. The sensor readings are acknowledged via *ack* transitions, and the robot command is delivered via a *signal* transition. It takes 0.9 to 1 milliseconds to receive and acknowledge a sensor reading. The two sensor readings that are used to construct a robot command must be received within 10 milliseconds. If the controller receives a reading from one sensor but does not receive the reading from the other sensor within 10 milliseconds, then the first sensor reading expires

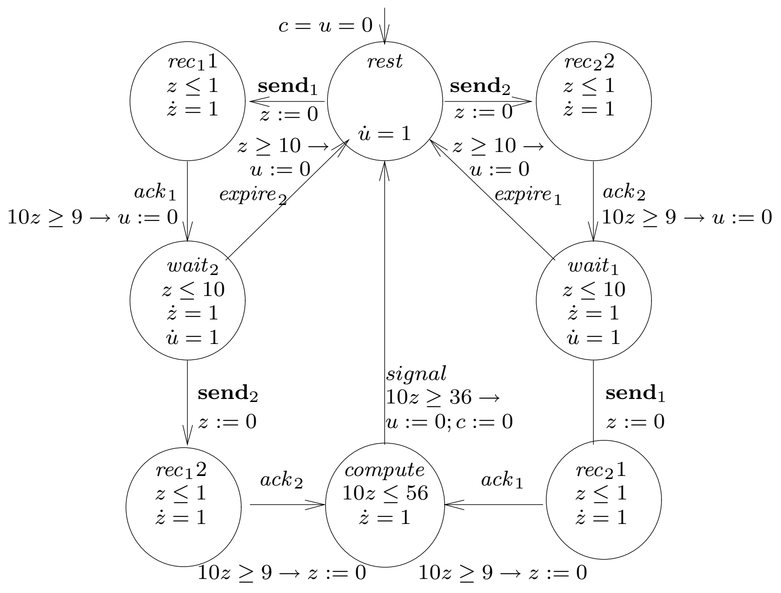


Figure 5: The controller

(via an *expire* transition). Once both reading are received, the controller takes 3.6 to 5.6 milliseconds to synthesize a robot command.

We want to know how often a robot command can be generated by the controller. For this purpose, we add a clock c to the controller automaton such that c measures the elapsed time since the last robot command was sent. The slope of the clock c is 1 in all locations of the controller automaton (this is omitted from Figure 5), and c is reset to 0 whenever a robot command is sent. We want to compute the maximum value of the clock c in all states that are reachable in the product of all four automata.

However, the product of the four automata does not model the system exactly according to Corbett's specification. This is because the **send** transitions should be urgent, that is, they should be taken as soon as they are enabled. We model the urgency of the **send** transitions by adding an additional clock, u, and global invariants. The clock u is reset whenever a sensor is ready to send a reading to the controller, and whenever the controller is ready to receive a sensor reading. Then we use the global invariant that $u = 0$ if both a sensor and the controller are ready for a transmission; that is,

$$\begin{aligned}
&(l[sensor_1] = wait \wedge l[controller] = rest \ \rightarrow \ u = 0) \ \wedge \\
&(l[sensor_2] = wait \wedge l[controller] = rest \ \rightarrow \ u = 0) \ \wedge \\
&(l[sensor_1] = wait \wedge l[controller] = wait_1 \ \rightarrow \ u = 0) \ \wedge \\
&(l[sensor_2] = wait \wedge l[controller] = wait_2 \ \rightarrow \ u = 0).
\end{aligned}$$

This invariant enforces whenever a transmission of a sensor reading is enabled, the transmission happens immediately.

In HyTech, the global invariant is defined as follows:

```
GlobalInvar = {{l[sensor1]==wait && l[controller]==rest, 0==u},
    {l[sensor2]==wait && l[controller]==rest, 0==u},
    {l[sensor1]==wait && l[controller]==wait1, 0==u},
    {l[sensor2]==wait && l[controller]==wait2, 0==u)}}
```

To compute the range of possible values for the clock c in the reachable states, we write:

```
InitialState = l[sensor1]==done && l[sensor2]==done && l[scheduler]==idle
    && l[controller]==rest && 0==c && 6==y1 && 6==y2 && 0==u
Bad = True
```

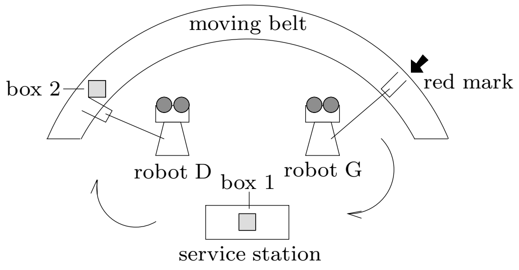


Figure 6: The two-robot manufacturing system

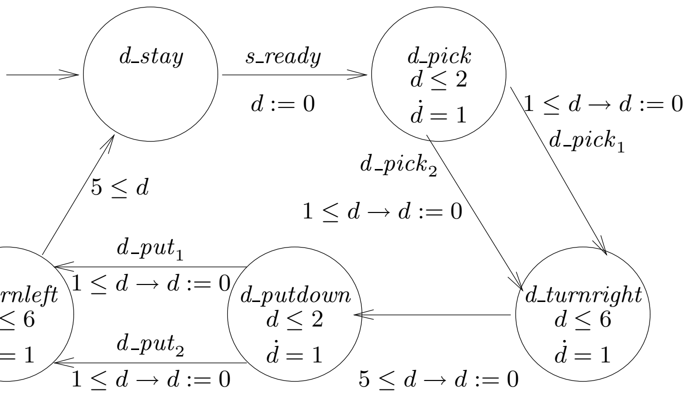


Figure 7: Robot D

```
EliminateLocList = {sensor1,sensor2,sched,gen}
EliminateVarList = {x1,x2,y1,y2,z,w,u}
```

Notice that, using the two projection operators, we ask HyTech to print only information about the clock c. Using forward analysis without approximation, HyTech returns, in 89.53 seconds of CPU time, the following answer:

```
0 <= c && -12 <= -5*c || -7 <= -2*c && 9 <= 10*c ||
-3 <= -c && 7 <= 10*c || -9 <= -2*c && 3 <= 2*c ||
12 <= 5*c && -28 <= -5*c || 5 <= 2*c && -18 <= -2*c ||
33 <= 10*c && -105 <= -10*c || 42 <= 5*c && -56 <= -5*c
```

From this result (the last disjunct is $42 \leq 5c \wedge -56 \leq -5c$), it follows that the maximum value of the clock c is 11.2; that is, a robot command is generated by the controller at least once every 11.2 milliseconds. We also applied HyTech to analyze the same system except that the priority of sensor 1 for using the processor is higher than the priority of sensor 2. In that case, a command is generated at least once every 11 milliseconds.

4.2 A Two-robot Manufacturing System

Puri and Varaiya [PV95] designed a manufacturing system that consists of a conveyor belt with two boxes, a service station, and two robots. The system is illustrated in Figure 6.

Robot D, one of the two robots, is modeled by the linear hybrid automaton of Figure 7. The clock d is used to measure the time needed for the actions performed by robot D. Initially robot D is looking at the service station (location *d_stay*). When it sees an unprocessed box in the service station, it picks up that box from the service station in 1 to 2 seconds (location *d_pick*), makes a right turn in 5 to 6 seconds (location *d_turnright*), puts the box at one end of the conveyor belt in 1 to 2 seconds (location *d_putdown*), makes a

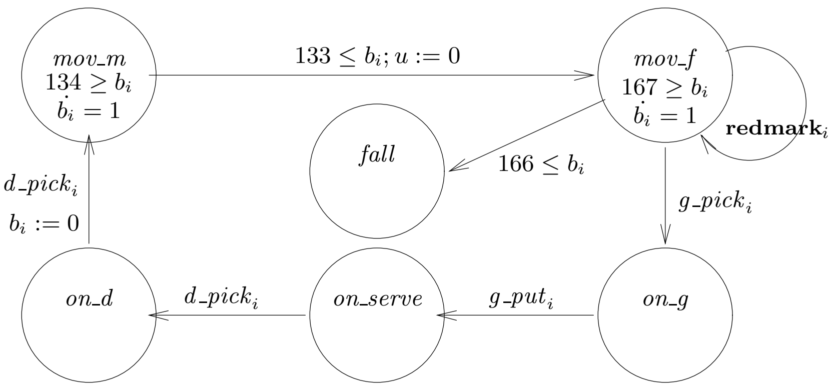


Figure 8: Box i

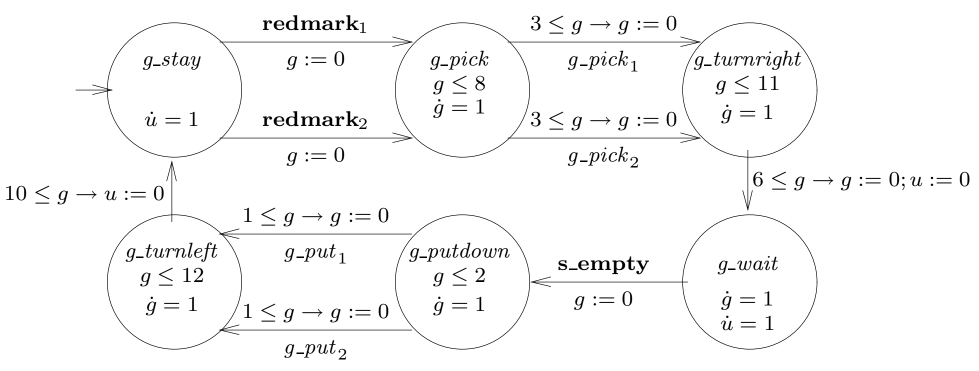


Figure 9: Robot G

left turn back to the service station in 5 to 6 seconds (location *d_turnleft*), and stays at there waiting for the next unprocessed box (location *d_stay*).

The two boxes, box 1 and box 2, are modeled by the indexed linear hybrid automaton in Figure 8, where the index i is either 1 (for box 1) or 2 (for box 2). A box may be in the service station (location *on_serve*), held by robot D (location *on_d*), moving on the conveyor belt before a red mark (location *mov_m*), moving on the conveyor belt beyond the red mark (location *mov_f*), held by robot G (location *on_g*), or falling off the end of the conveyor belt (location *fall*). A box on the conveyor belt is processed by the manufacturing system. The conveyor belt is moving at a certain speed from one end to the other. The clock b_i measures the total time that box i spends on the conveyor belt, and thus determines the position of box i on the belt. A box requires 133 to 134 seconds to reach the red mark after it is placed on the belt by robot D. If a box is not picked up by robot G before the end of the belt, then the box falls off the belt 166 to 167 seconds after it is placed on the belt.

Robot G at the end of the conveyor belt is modeled by the automaton in Figure 9. The clock g measures the time needed to perform the actions of robot G. Initially robot G is looking at the red mark next to the conveyor belt (location *g_stay*). When it sees a processed box moving beyond the red mark, it picks up that box from the belt in 3 to 8 seconds (location *g_pick*), makes a right turn in 6 to 11 seconds (location *g_turnright*), waits for the service station to be empty (location *g_wait*), puts the box into the service station in 6 to 11 seconds (location *g_putdown*), makes a left turn back to the conveyor belt in 1 to 2 seconds (location *g_turnleft*), and stays there watching the red mark (location *g_stay*).

The service station is modeled by the automaton in Figure 10. Whenever the service station receives a processed box, it pops up an unprocessed box for robot D to pick up. The service station takes 8 to 10

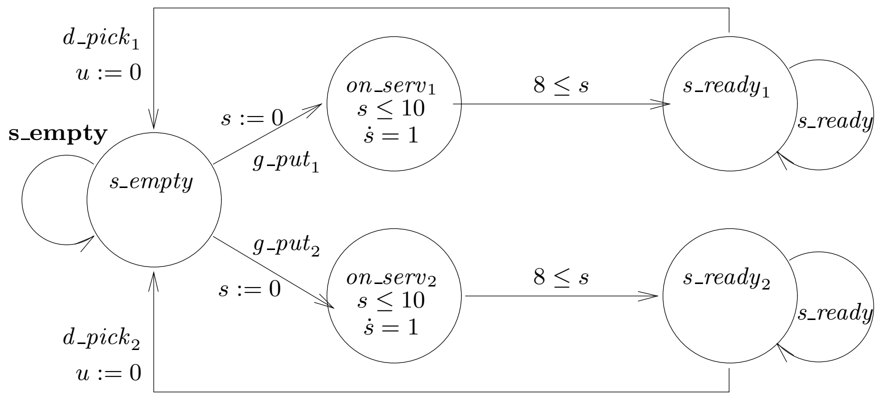


Figure 10: The service station

seconds to switch the processed and unprocessed boxes, which is measured by the clock s. Initially both boxes are on the conveyor belt before the red mark. There are at most two boxes on the belt at any time, because the service station pops up a new box only when it receives a processed box from robot G.

According to Puri and Varaiya's specification, the transitions with the synchronization letters *s_ready*, **redmark**$_1$, **redmark**$_2$, and **s_empty**, are urgent; that is, robot D picks up a box from the service station as soon as it is ready and sees a box in the service station, etc. We treat the *s_ready* transitions as ordinary transitions, because this assumption will not affect our analysis. We use the clock u and the following global invariants to model the urgent transitions:

```
GlobalInvar = {{l[grobot]==stay && l[box1]==movf, 0==u},
    {l[grobot]==stay && l[box2]==movf, 0==u},
    {l[grobot]==wait && l[station]==sempty, 0==u},
    {l[grobot]==wait && l[station]==sempty, 0==u}}
```

We want to check the safety requirement that no box will ever fall off the conveyor belt. This requirement clearly depends on the initial positions of the two boxes on the belt. We use the parameter *dist* such that $dist = b_1 - b_2$ represents the difference of the initial values of the clocks b_1 and b_2. Then we use HyTech to analyze the reachability problem $(A, \varphi_I, \varphi_F)$, where A is the product of all five automata and

$$\begin{aligned}\varphi_I \;&=\; (l[box_1] = mov\_m \wedge l[box_2] = mov\_m \wedge l[robot_G] = g\_stay \wedge l[robot_D] = d\_stay \wedge \\ &\quad l[servicestation] = s\_empty \wedge u = 0 \wedge dist = b_1 - b_2),\\ \varphi_F \;&=\; (l[box_1] = fall \vee l[box_2] = fall.\end{aligned}$$

Using forward analysis without approximation, HyTech returns, in 1718.73 second of CPU time, the following target region:

```
-1 <= -dist && -9 <= dist || -1 <= dist && -9 <= -dist
```

It follows that $-9 > dist \vee dist > 9$ is a necessary and sufficient condition on the parameter *dist* so that neither box will fall off the conveyor belt; that is, $|b_1 - b_2| > 9$.

References

[ACH+95] R. Alur, C. Coucoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[ACHH93] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, Lecture Notes in Computer Science 736, pages 209–229. Springer-Verlag, 1993.

[AHH93] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the 14th Annual Real-time Systems Symposium*, pages 2–11. IEEE Computer Society Press, 1993.

[AHV93] R. Alur, T.A. Henzinger, and M.Y. Vardi. Parametric real-time reasoning. In *Proceedings of the 25th Annual Symposium on Theory of Computing*, pages 592–601. ACM Press, 1993.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual Symposium on Principles of Programming Languages.* ACM Press, 1977.

[CC92] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP*, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, 1992.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Annual Symposium on Principles of Programming Languages.* ACM Press, 1978.

[Cor94] J.C. Corbett. Modeling and analysis of real-time Ada tasking programs. In *Proceedings of the 15th Annual Real-time Systems Symposium.* IEEE Computer Society Press, 1994.

[DW95] D.L. Dill and H. Wong-Toi. Verification of real-time systems by successive over- and underapproximation. To appear at CAV, 1995.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *CAV 93: Computer-aided Verification*, Lecture Notes in Computer Science 697, pages 333–346. Springer-Verlag, 1993.

[HH94] T.A. Henzinger and P.-H. Ho. Model-checking strategies for linear hybrid systems. Technical Report CSD-TR-94-1437, Cornell University, 1994. Presented at the Seventh International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, and at the Workshop on Hybrid Systems and Autonomous Control (Ithaca, NY).

[HH95] T.A. Henzinger and P.-H. Ho. Algorithmic analysis of nonlinear hybrid systems. To appear at CAV, 1995.

[HRP94] N. Halbwachs, P. Raymond, and Y.-E. Proy. Verification of linear hybrid systems by means of convex approximation. In B. LeCharlier, editor, *SAS 94: Static Analysis Symposium*, Lecture Notes in Computer Science 864. Springer-Verlag, 1994.

[NOSY93] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, Lecture Notes in Computer Science 736, pages 149–178. Springer-Verlag, 1993.

[PV95] A. Puri and P. Varaiya. Verification of hybrid systems using abstractions. To appear, 1995.

The Modal μ-Calculus, Model Checking, Equation Systems and Gauß Elimination

Angelika Mader[*]
Technische Universität München, Germany

1 Introduction

The modal μ-calculus [Koz83, Sti92] is a powerful logic. It is particularly useful for expressing properties of parallel processes with finite (or even infinite) state spaces; it finds application in process algebra [Wal89] and in Petri nets [Bra92]. Proving whether a property expressed in the modal μ-calculus holds for particular states of a process is called model checking [CE81, CES86]. Various algorithms are available. The main approaches are model checkers based on the fixpoint approximation [EmL86, CDS92, And92, BCMDH92, LBCJM94] and tableau based model checkers [StW89, Cle90, Lar92, Mad92]. One important technique consists of the transformation of a property and a model to a (Boolean) equation system [AC88, And92, CDS92, Lar92, VeL92]. Then model checking is equivalent to the computation of a certain fixpoint. In fact, various correctness problems may be represented in this way.

In this paper we present a novel, algebraic approach for solving Boolean equation systems. It does not use approximation techniques and therefore does not require backtracking. The method works straightforward by successively eliminating variables and reducing the Boolean equation system, similar to Gauß elimination for linear equation systems. Homogeneous, hierarchical and alternating fixpoints are treated uniformly. Contrary to other techniques Gauß elimination leads to both a global and a local model checking algorithm within one framework.

The elimination of a variable is based on a simple observation: the equation $X = A(X)$ (with monotone A) has the least fixpoint $A(\text{false})$ and the greatest fixpoint $A(\text{true})$. The reduction of a Boolean equation system is done by syntactical substitution of variables by expressions.

The difference between the global version of Gauß elimination and the local one can be characterized as follows: The global version solves the whole equation system, whereas the local version only takes a subset of equations into account which is necessary to determine the variable of interest. The selection of a suitable subset of equations is demand-driven. Whereas the global version is more of theoretical interest (approximation

[*]Institut für Informatik, Technische Universität München, Arcisstr.21, D-80333 München, Germany, email: mader@informatik.tu-muenchen.de. Supported by Siemens AG, Corporate Research and Development, and a grant from the Hochschulsonderprogramm II.

techniques have better worst case complexity), the local version has advantages in the context of model checking. It is closely related to the tableau methods, and can be interpreted as a combination of top-down strategy of the tableau method and bottom-up evaluation which avoids redundancy caused by recomputation of subtableaux. Therefore it is much more efficient than the tableau methods and has a better worst case complexity.
Section 2 introduces Boolean equation systems and their solution. Gauß elimination for Boolean equation systems is presented in section 3. Section 4 contains a short introduction into the modal μ-calculus, and the transformation of the model checking problem into a equation solving problem. Comparison with other work is discussed in section 5. Examples are in section 6. Section 7 is the conclusion. The appendix contains correctness proofs.

2 Boolean Equation Systems

In this section we define Boolean equation systems and what we regard as solution of a Boolean equation system.

Definition 1 Let $\mathcal{X} = \{X_1, \dots, X_n\}$ be a set of Boolean variables, $<$ a linear order on $\mathcal{X}$, and $\{A_1, \dots, A_n\}$ a set of negation free Boolean expressions containing variables from $\mathcal{X}$. Then the set of labelled equations $E_i : X_i \overset{\sigma_i}{=} A_i$, where $\sigma_i \in \{\mu, \nu\}$, is a Boolean equation system.

In the following we assume that the order on the variables is according to their indices.
As the Boolean expressions are negation free and therefore monotone the equation system (the plain one without order and labels) has a set of fixpoints. In the context here we are interested in a distinguished fixpoint which we call the solution of the Boolean equation system. Below we give the definition of the solution.
We introduce some notation first. The vector $(X_1, \dots, X_n)$ of Boolean variables will be abbreviated by $\underline{X}$; analogously $\underline{\sigma}$, $\underline{A}$ and $\underline{E}$ denote the vectors of labels, expressions and equations respectively. A Boolean equation system can now be written as: $\underline{E}$: $\underline{X} \overset{\underline{\sigma}}{=} \underline{A}(\underline{X})$.
Further abbreviations will be used. $\underline{Y}^{(i)}$ stands for the i-th rest $(Y_i, \dots Y_n)$ of the Boolean vector $\underline{Y}$, and again analogously $\underline{\sigma}^{(i)}$, $\underline{A}^{(i)}$, and $\underline{E}^{(i)}$ denote the i-the rests of the related vectors. By $\underline{E}^{(i)}[Y_j/X_j]$ we mean the equation system $\underline{E}^{(i)}$ where all unbound occurrences of X_j on the right-hand-side of the equations are substituted by Y_j.
Now for $\underline{\sigma} \in \{\mu, \nu\}^n$ we define the **lexicographic order** $<_{\underline{\sigma}}$ on the Boolean vector space $\mathbb{B}^n$. Let $<_\sigma$ on $\mathbb{B}$ for $\sigma \in \{\mu, \nu\}$ be such that $\mathsf{false} <_\mu \mathsf{true}$ and $\mathsf{true} <_\nu \mathsf{false}$.

Definition 2 Let $\underline{Y}, \underline{Y}' \in \mathbb{B}^n$, $\underline{\sigma} \in \{\mu, \nu\}^n$. The lexicographic order on $\underline{Y}, \underline{Y}'$ is defined as $\underline{Y} <_{\underline{\sigma}} \underline{Y}' :\Leftrightarrow \;\; \exists i, 1 \leq i \leq n : \; Y_i <_{\sigma_i} Y'_i$ and $\forall j, 1 \leq j < i : \; Y_j = Y'_j$.

Definition 3 $\underline{Y}^{(i)} \in \mathbb{B}^{n-i+1}$ is the solution of the equation system $\underline{E}^{(i)}[Y_1/X_1, \dots Y_{i-1}/X_{i-1}]$

$:\Leftrightarrow$ if $i = n$, then $\underline{Y}^{(i)}$ is the least fixpoint of $\underline{E}^{(i)}[Y_1/X_1, \dots Y_{i-1}/X_{i-1}]$ wrt. $<_{\sigma_n}$,

if $i < n$, then $\underline{Y}^{(i)}$ is the least one wrt. $<_{\underline{\sigma}^{(i)}}$ of those fixpoints of $\underline{E}^{(i)}[Y_1/X_1, \dots Y_{i-1}/X_{i-1}]$ which satisfy the following property:

$\underline{Y}^{(i+1)}$ is solution of $\underline{E}^{(i+1)}[Y_1/X_1, \dots Y_{i-1}/X_{i-1}, Y_i/X_i]$.

There exist several algorithms to determine the set of fixpoints of a Boolean equation system; for examples see [Rud74]. However, even if the set of fixpoints is given, it is not trivial to select the one fixpoint which satisfies the definition of the solution above. This indicates that the existing equation solving methods do not help in our case. We will illustrate this by two small examples.

The first example shows that the solution is not the lexicographic least fixpoint. The equation system $X_2 \overset{\mu}{=} X_2$ has two fixpoints true and false. With respect to the order $<_\mu$ false is the least one. Now consider the equation system $X_1 \overset{\nu}{=} X_2, X_2 \overset{\mu}{=} X_2$, where $X_1 < X_2$. The lexicographic least fixpoint is $(\mathsf{true}, \mathsf{true})$, whereas $(\mathsf{false}, \mathsf{false})$ is the solution as indicated by the first equation system and as defined above.

In the following example two Boolean equation systems are given, both having the same set of fixpoints and the same labels on the equations, but different solutions. The equation system $X_1 \overset{\nu}{=} X_2, X_2 \overset{\mu}{=} X_2$, where $X_1 < X_2$, has the fixpoints $(\mathsf{true}, \mathsf{true})$ and $(\mathsf{false}, \mathsf{false})$. The solution is $(\mathsf{false}, \mathsf{false})$ as in the previous example.

The equation system $X_1 \overset{\nu}{=} X_2, X_2 \overset{\mu}{=} X_1$, where $X_1 < X_2$, also has the fixpoints $(\mathsf{true}, \mathsf{true})$ and $(\mathsf{false}, \mathsf{false})$, but the solution here is $(\mathsf{true}, \mathsf{true})$.

3 Gauß Elimination

In this section we present two algorithms which determine the solution of a Boolean equation system as in definition 3. In contrast to other methods we do not make use of approximation and backtracking techniques. Instead we stepwise reduce a Boolean equation system to a Boolean equation systemconsisting one equation and one variable less. The steps of eliminating a variable from an expression and of substituting variables by expressions remind very much to Gauß elimination for linear equation systems.

The following propositions are the basis for the Gauß elimination. Proofs are contained in the appendix.

Proposition 1 For the solution Y of the equation system $X \overset{\sigma}{=} A(X)$ consisting of one single equation it holds:

$$Y = \begin{cases} A(\mathsf{false}) & \text{if } \sigma = \mu \\ A(\mathsf{true}) & \text{if } \sigma = \nu \end{cases}$$

Proposition 1 can be extended to expressions and equation systems. It allows a representation of the Boolean expression A_i with no occurrence of X_i. In the algorithm we will call this the Gauß division step.

Proposition 2 $\underline{Y}$ is the solution of the Boolean equation system $\underline{E}$ of the form $\underline{X} \overset{\sigma}{=} \underline{A}(\underline{X})$, iff $\underline{Y}$ also is the solution of the modified Boolean equation system $\underline{F}$, where the equations are of the following form:

$$\begin{array}{lcll} X_1 & \overset{\sigma_1}{=} & A_1(\mathsf{b}_1, X_2, \quad \ldots \quad, X_n) & \text{where} \\ \vdots & & & \\ X_i & \overset{\sigma_i}{=} & A_i(X_1, \ldots, X_{i-1}, \mathsf{b}_i, X_{i+1}, \ldots, X_n) & \mathsf{b}_i = \begin{cases} \mathsf{true} & \text{if } \sigma_i = \nu \\ \mathsf{false} & \text{if } \sigma_i = \mu \end{cases} \\ \vdots & & & \\ X_n & \overset{\sigma_n}{=} & A_n(X_1, \quad \ldots \quad, X_{n-1}, \mathsf{b}_n) & \end{array}$$

The next proposition shows that every occurrence of a variable X_n may be substituted by the expression A_n, in which X_n has been eliminated. We call this the Gauß elimination step.

Proposition 3 The Boolean vector $\underline{Y}$ is the solution of the equation system $\underline{E}$, iff it is the solution of the equation system $\underline{G}$, where $\underline{G}$ is the modified equation system:

$$\begin{array}{rcl} X_1 & \overset{\sigma_1}{=} & A_1 \quad (X_1, \ldots, X_{n-1}, A_n(X_1, \ldots, X_{n-1}, \mathsf{b}_n)) \\ \vdots & & \\ X_{n-1} & \overset{\sigma_{n-1}}{=} & A_{n-1}(X_1, \ldots, X_{n-1}, A_n(X_1, \ldots, X_{n-1}, \mathsf{b}_n)) \\ X_n & \overset{\sigma_n}{=} & A_n \quad (X_1, \ldots, X_{n-1}, \mathsf{b}_n) \end{array}$$

Based on these two Gauß steps we now propose two algorithms to determine the solution of a Boolean equation system. One algorithm operates on the whole equation system; this is the global version of Gauß elimination. The basic idea is that a Boolean equation system can be reduced to a Boolean equation system with the same solution, but one equation less. The reduction is performed by an elimination step, where in the last equation, say $X_j \overset{\sigma_j}{=} A_j(X_1, \ldots, X_j)$, all occurrences of X_j on the right hand side are instantiated by $\mathsf{b}_j = \mathsf{true}$ or $\mathsf{b}_j = \mathsf{false}$ depending on σ_j, and a substitution step, where in all other equations each occurrence of X_j is substituted by the expression $A_j(X_1, \ldots, X_{j-1}, \mathsf{b}_j)$. The result is an equation system with no free occurrence of X_j. Now the same reduction can be applied to the equation system consisting of the first $j-1$ equations and so on. In the end we get a variable free expression for the variable X_1.

```
Assume $\underline{X} \overset{\sigma}{=} \underline{A}(\underline{X})$ as input;
i := n;
**while not** ($A_1$ = true **or** $A_1$ = false)
    **do**
        Instantiate $X_i$ in $A_i$ to {true, false};          (Gauß-division)
        Substitute $A_i$ for $X_i$ in $A_1, \ldots, A_{i-1}$;            (Elimination step)
        $A_1$:=Eval($A_1$); ... ; $A_{i-1}$:=Eval($A_{i-1}$)       (Evaluation step)
        i := i - 1;
    **od**
```

Figure 1: Global Version of Gauß Elimination

In most contexts we are only interested in the first component of the solution, i.e. whether X_1 is true or false. Therefore the algorithm in figure 1 stops, if the solution of X_1 (A_1) is determined. If we are interested in the whole solution the Gauß division step and elimination step have to be applied n times giving an expression for every X_i where the variables $X_i, \ldots, X_n$ do not occur. A straight backward substitution leads to the whole solution.

If only the first variable is of interest, it suffices to consider only the subset of equations which is necessary to determine the solution for X_1. The relevant subset of equations is selected in a top-down manner. This observation leads to the local version of Gauß elimination given in figure 2. The idea is as follows. We start with the equation system $\underline{E}'$ consisting only of the equation $X_1 \overset{\sigma_1}{=} A_1(X_1, \ldots, X_j)$. As long as X_1 is not evaluated to true or false we select a free variable from A_1, insert its equation in $\underline{E}'$, apply the global version of Gauß elimination, and continue in the same way with the modified equation system $\underline{E}'$.

```
Create $E_1$ and let $E_1$ be $\underline{E}'$;
Instantiate $X_1$ in $A_1$;                          (Gauß-division)
$A_1 := \mathrm{Eval}(A_1)$;                         (Evaluation step)
while not ($A_1 = \mathsf{true}$ or $A_1 = \mathsf{false}$)
    do
        Select $X_j$, where $j$ is such that $E_j \notin \underline{E}'$;
        Create $E_j$, insert $E_j$ in $\underline{E}'$ and extend the order on $\underline{E}'$ to $E_j$;
        Apply Gauß-elimination on $\underline{E}'$
    od
```

Figure 2: Local Model Checking Algorithm

4 The Modal μ-Calculus and Model Checking

This section gives a brief introduction to the modal μ-calculus. For details see [Sti92].

The syntax of the modal μ-calculus is defined with respect to a set $\mathcal{Q}$ of atomic propositions including true and false, a denumerable set $\mathcal{Z}$ of propositional variables and a finite set $\mathcal{L}$ of action labels. The set μM of modal μ-calculus assertions is determined by the following grammar:

$\Phi ::= Z \mid Q \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid [a]\Phi \mid \langle a \rangle \Phi \mid \mu Z.\Phi \mid \nu Z.\Phi$

M denotes the set of variable and fixpoint free assertions, i.e., the expressions of the **propositional modal logic**, Π_0 denotes the set of fixpoint free assertions, $M \subset \Pi_0$. In the following an expression of the form $\sigma Z.\Phi$, where $\sigma \in \{\mu, \nu\}$, is called a **fixpoint expression**. Formulae of the modal μ-calculus with the set $\mathcal{L}$ of action labels are interpreted relative to a **labelled transition system** $\mathcal{T} = (\mathcal{S}, \{\xrightarrow{a} \mid a \in \mathcal{L}\})$, where $\mathcal{S}$ is a finite set of states and $\xrightarrow{a} \subseteq \mathcal{S} \times \mathcal{S}$ for every $a \in \mathcal{L}$ is a binary relation on states. A **valuation function** $\mathcal{V}$ assigns to every propositional variable Z and atomic proposition Q a set of states $\mathcal{V}(Z) \subseteq \mathcal{S}$ and $\mathcal{V}(Q) \subseteq \mathcal{S}$. Let V[S'/Z] be the valuation such that V[S'/Z](Z) = S', and otherwise as V. The pair $\mathcal{T}$ and $\mathcal{V}$ is called a **model** of the μ-calculus. The semantics of each μ-calculus formula Φ is the set of states $\|\Phi\|_{\mathcal{V}}^{\mathcal{T}}$ defined inductively as follows:

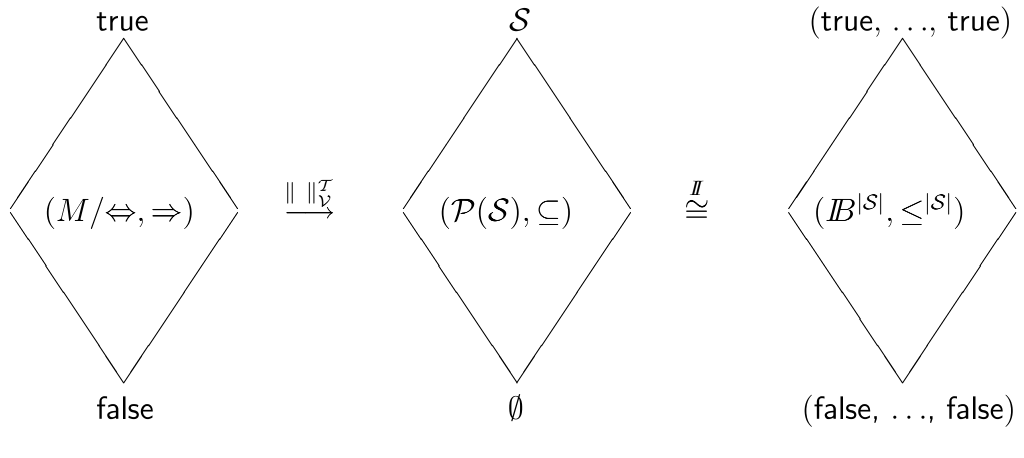
$$
\begin{array}{rclrcl}
\|Z\|^{\mathcal{T}}_{\mathcal{V}} & = & \mathcal{V}(Z) & \|\Phi_1 \vee \Phi_2\|^{\mathcal{T}}_{\mathcal{V}} & = & \|\Phi_1\|^{\mathcal{T}}_{\mathcal{V}} \cup \|\Phi_2\|^{\mathcal{T}}_{\mathcal{V}} \\
\|Q\|^{\mathcal{T}}_{\mathcal{V}} & = & \mathcal{V}(Q) & \|\Phi_1 \wedge \Phi_2\|^{\mathcal{T}}_{\mathcal{V}} & = & \|\Phi_1\|^{\mathcal{T}}_{\mathcal{V}} \cap \|\Phi_2\|^{\mathcal{T}}_{\mathcal{V}}
\end{array}
$$

$$
\begin{array}{rcl}
\|[a]\Phi\|^{\mathcal{T}}_{\mathcal{V}} & = & [\![a]\!]^{\mathcal{T}}\ \|\Phi\|^{\mathcal{T}}_{\mathcal{V}}, \quad \text{where } [\![a]\!]^{\mathcal{T}} S' = \{s \mid \forall s' \in S.\ \text{if } s \xrightarrow{a} s' \text{ then } s' \in S'\} \\
\|\langle a\rangle\Phi\|^{\mathcal{T}}_{\mathcal{V}} & = & \langle\!\langle a\rangle\!\rangle^{\mathcal{T}}\ \|\Phi\|^{\mathcal{T}}_{\mathcal{V}}, \quad \text{where } \langle\!\langle a\rangle\!\rangle^{\mathcal{T}} S' = \{s \mid \exists s' \in S'. s \xrightarrow{a} s'\} \\
\|\mu Z.\Phi\|^{\mathcal{T}}_{\mathcal{V}} & = & \bigcap\{S' \subseteq \mathcal{S} \mid \|\Phi\|^{\mathcal{T}}_{\mathcal{V}[S'/Z]} \subseteq S'\} \\
\|\nu Z.\Phi\|^{\mathcal{T}}_{\mathcal{V}} & = & \bigcup\{S' \subseteq \mathcal{S} \mid S' \subseteq \|\Phi\|^{\mathcal{T}}_{\mathcal{V}[S'/Z]}\}
\end{array}
$$

Given a model $\mathcal{M} = (\mathcal{T}, \mathcal{V})$ model checking is to examine the question whether a certain expression Φ holds for the initial state $s \in \mathcal{S}$ of the transition system $\mathcal{T}$, i.e., whether $s \in \|\Phi\|^{\mathcal{T}}_{\mathcal{V}}$. We transform the model checking problem into the problem of solving a Boolean equation system. This was already done by several authors, e.g. see [AC88, And92, Lar92, ClS91, CDS92, VeL92]. In contrary to their approaches here arbitrary negation free expressions are considered as right hand sides of the equations (no restriction to simple expressions). Furthermore we create one Boolean equation system for the whole problem with a partial order defined on its variables and equations.

Roughly the transformation is performed by the following steps: a fixpoint expression can be represented by an equation system with an additional ordering on the equations. On the semantic part we interpret the equation system with respect to a model, i.e., a fixpoint equation of modal logic becomes a fixpoint equation over the powerset of a state space. An isomorphic representation of a powerset of states is a Boolean vector space. This allows us to derive a Boolean equation system from the original fixpoint expression and its model.

A modal μ-calculus formula can be represented as an ordered equation system. For example the fixpoint expression $\nu Z_1.[a]\mu Z_2.[b]((Z_1 \wedge Q) \vee Z_2)$ is equivalent to the equation system $\mathcal{E}$: $Z_1 \overset{\nu}{=} [a]Z_2$, $Z_2 \overset{\mu}{=} [b]((Z_1 \wedge Q) \vee Z_2)$, where the variables are ordered by $Z_1 < Z_2$. Note that in general here the order on the variables is a partial order in contrast to the variable ordering as in definition 1.



true
$(M/\Leftrightarrow, \Rightarrow)$
false
$\xrightarrow{\|\ \|^{\mathcal{T}}_{\mathcal{V}}}$
$\mathcal{S}$
$(\mathcal{P}(\mathcal{S}), \subseteq)$
$\emptyset$
$\overset{I\!I}{\cong}$
(true, ..., true)
$(I\!B^{|\mathcal{S}|}, \leq^{|\mathcal{S}|})$
(false, ..., false)

Figure 3: The lattices and the mappings

The transformation is as follows: Recall that M is the set of variable and fixpoint free expressions of the modal μ-calculus, i.e., the expressions of the propositional modal logic. The equivalence classes of M together with the implication ordering form a lattice $(M/\Leftrightarrow, \Rightarrow)$. The powerset of the state space $\mathcal{S} = \{s_1, \ldots, s_n\}$ with the inclusion order forms a complete lattice $(\mathcal{P}(\mathcal{S}), \subseteq)$. The evaluation function $\| \,\|_{\mathcal{V}}^{\mathcal{T}} : M \to \mathcal{P}(\mathcal{S})$ is monotone (and continuous). The extension of the evaluation function from M to fixpoint equations over M maps modal operators $[a], \langle a \rangle$ to set operators $[\![a]\!]^{\mathcal{T}}, \langle\!\langle a \rangle\!\rangle^{\mathcal{T}}$, modal variables to set variables and the logical operators $\wedge, \vee$ to the set operators $\cap, \cup$. Thus we get an equation system over the powerset of the state space. The labels $\{\nu, \mu\}$ and the partial order on the equations remain the same as in the original equation system. Defining $\mathsf{false} \leq \mathsf{true}$ the Boolean lattice $(\mathbb{B}^{|\mathcal{S}|}, \leq^{|\mathcal{S}|})$ with pointwise ordering is isomorphic to $(\mathcal{P}(\mathcal{S}), \subseteq)$. The last step leads from a vector valued equation system in $\mathbb{B}^n$ to a Boolean equation system; every vector equation is split into n equations and the operators $[\![a]\!]^{\mathcal{T}}, \langle\!\langle a \rangle\!\rangle^{\mathcal{T}}$ are evaluated.

Altogether a μ-calculus equation $Z \overset{\sigma}{=} \Phi$ is mapped inductively to the (not ordered) set of n Boolean equations $\quad \mathbf{I}_Z(s_i) \overset{\sigma}{=} \mathbf{I}_\Phi(s_i) \quad$ for $1 \leq i \leq n$, where

$$
\begin{aligned}
\mathbf{I}_Q(s) &= \mathsf{true}\text{, if } s \in \|Q\|_{\mathcal{V}}^{\mathcal{T}} \\
\mathbf{I}_Q(s) &= \mathsf{false}\text{, if } s \notin \|Q\|_{\mathcal{V}}^{\mathcal{T}} \\
\mathbf{I}_{\Phi_1 \wedge \Phi_2}(s) &= \mathbf{I}_{\Phi_1}(s) \wedge \mathbf{I}_{\Phi_2}(s) \\
\mathbf{I}_{\Phi_1 \vee \Phi_2}(s) &= \mathbf{I}_{\Phi_1}(s) \vee \mathbf{I}_{\Phi_2}(s)
\end{aligned}
\qquad
\begin{aligned}
\mathbf{I}_Z(s) &= X_{Z,s} \\
\mathbf{I}_{[a]\Phi}(s) &= \bigwedge_{s \xrightarrow{a} s'} \mathbf{I}_\Phi(s') \\
\mathbf{I}_{\langle a \rangle \Phi}(s) &= \bigvee_{s \xrightarrow{a} s'} \mathbf{I}_\Phi(s')
\end{aligned}
$$

Note that the Boolean equations derived in this way do not contain negations or modal operators. A μ-calculus equation system of the size k determines a Boolean equation system of size $k * |\mathcal{S}|$. There is a partial order on the Boolean equations inherited from the partial order on the μ-calculus equations. Two Boolean equations derived from one vector equation are not ordered. Thus the tree-like order on the set equations becomes an acyclic order on the Boolean equations.

We now show that the two algorithms proposed in the previous section can be applied to the Boolean equation systems derived from a modal μ-calculus equation system and a model. There are remaining two open questions: first, whether the partial order on the Boolean equations here matches the linear order on the Boolean equations and variables as in definition 1, and second, whether the solution of a Boolean equation system coincides with the semantics of the modal μ-calculus.

Proposition 4 Given a μ-calculus expression and a transition system let $\underline{A}$ be the corresponding Boolean equation system. On its equations a partial order $<_{\mathcal{E}}$ is defined. For each two extensions of $<_{\mathcal{E}}$ to a linear order $<_l, <_{l'}$ it holds: $\underline{Y}$ is the solution of $\underline{A}$ with the order $<_l$, iff $\underline{Y}$ is the solution of $\underline{A}$ with the order $<_{l'}$.

Proof (Sketch of the Proof)
For unnested fixpoints an order of equations is not relevant for the solution (see [Bek84]). The order of the nested fixpoints is preserved by each extension of the partial order $<_{\mathcal{E}}$ to a linear order. □

Proposition 5 Given a fixpoint expression Φ of the modal μ-calculus and a transition system $\mathcal{T}$ with the initial state s, let $\underline{A}$ be the corresponding Boolean equation system. For the solution $\underline{Y}$ of $\underline{A}$ holds: $Y_{\Phi,s} = \mathsf{true}$, iff Φ holds at s.

Proof It is easy to see that the algorithm of Emerson and Lei [EmL86] calculates the solution as in Definition 2. □

5 Comparison to Other Work and Complexity

The model checking problem encoded as equation system was already treated by several authors [AC88, And92, CDS92, Lar92]. The method presented here differs from these approaches. Roughly speaking, the difference is that they get the solution by approximating sets. This approach was introduced by Emerson and Lei [EmL86] and continued for example by Cleaveland, Dreimüller and Steffen [CDS92]. In [And92] Andersen gives an algorithm based on Boolean equations. However, by representing a Boolean equation system as a graph his basic algorithm applies only for unnested fixpoints. The extension of the global version of his algorithm to the full calculus is due to the fixpoint approximation technique of Emerson and Lei. Also Larsen's algorithm [Lar92] deals with unnested fixpoints.

The Gauß elimination for model checking in its local version is more closely related to the tableau method of Stirling and Walker [StW89] and Cleaveland [Cle90]. In a Boolean equation system a variable is introduced for each pair of a state and a fixpoint formula. Each node in a tableau, a sequent, is a pair of a state and a formula. Hence a Boolean equation can be seen like a reduced form of a subtableau containing only sequents with fixpoint formulae, which is the only relevant part for the structure of the tableau. The top-down construction of the tableau can also be found in the local version of the Gauß elimination. While constructing a tableau the decision which path should be extended is equivalent to the selection of a variable from the top equation and creating the related equation. The condition for a leaf in the tableau of being successful or not corresponds to the Gauß division step: a cycle with a minimal fixpoint is regarded as unsuccessful (false), a cycle with a maximal fixpoint is regarded as successful (true). The advantage of the Gauß elimination over the tableau method has its roots in the bottom-up evaluation. On one hand it spares the introduction of different constants for the same fixpoint expression, on the other hand there is no redundant evaluation of identical subexpressions (subtrees). Altogether the local version of the Gauß elimination for model checking can be regarded as a combination of the top-down strategy of the tableau allowing to explore only the relevant part of the state space, and a bottom-up strategy which avoids recomputation of identical subtrees. The maximal size of a tableau is bounded by $O(b^{(|\Phi|*|\mathcal{S}|)^{f(\Phi)}})$, where b is the maximal branching degree of the transition system, $|\mathcal{S}|$ the number of states, $|\Phi|$ the size of the formula and $f(\Phi)$ the number of fixpoint operators in Φ. For the Gauß elimination the number of derived equations is determined by the size of the state space and the number of fixpoint operators in Φ. Substituting Boolean expressions leads to expressions exponential in the number of equations. The maximal size of the Boolean equation system constructed by the Gauß elimination is bound by $O((b^{a(\Phi)} * |\Phi|)^2 * 2^{|\mathcal{S}|*f(\Phi)})$, where additional to the abbreviations above $a(\Phi)$ is the maximal nesting depth of modal operators in the formula

Φ. Hence it is a natural idea to use the local version of the Gauß elimination for an implementation of the tableau method.

6 Examples

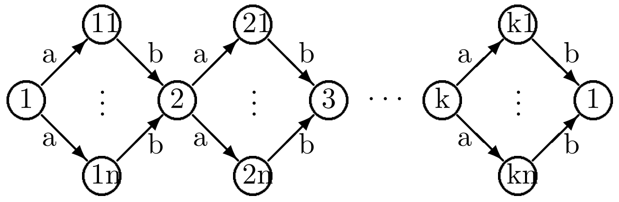
The aim of this section is to demonstrate the possible advantage of the local version of Gauß elimination over a tableau based model checker.

The first both examples are academic ones, without a special meaning. They do not show the advantage of local model checking, because the whole state space has to be traversed. However, they show how our algorithm avoids recomputation of subexpressions, or subtrees resp., whereas the tableau method does not.

In the third example we prove a fairness property for the mutual exclusion algorithm of Peterson.

A prototype of the local version of Gauß elimination was implemented in C++ using BDDs [Bry92] as data structure for Boolean expressions. In the examples here we compared our implementation with a tableau-based model checker as in [StW89] and with the tableau-based model checker incorporated in the Concurrency Workbench (CWB), which uses techniques for avoiding some recomputations. All implementations run on a SUN SPARC2.

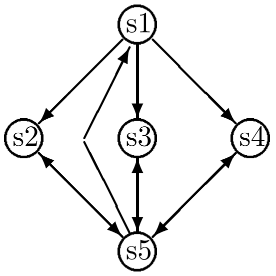
We wish to determine whether $s1 \models_{\mathcal{M}} \nu X.[a]\langle b\rangle X$ (every a-successor has a b-successor and this recursively) holds in:



The example consists of a scalable (n,k)-spindle where the final state is again identified with the start state. It has $kn+k$ states.

The local Gauß elimination creates k equations, each of the form: $X_i \overset{\nu}{=} \bigwedge_{j=1..n} X_{i+1 \ mod \ k}$ for $1 \leq i \leq k$ which can be reduced to $X_i \overset{\nu}{=} X_{i+1 \ mod \ k}$. It takes k elimination steps to determine the solution. The tableau based model checker as in [StW89] builds a tree with $1+2n+2n^2+\ldots+2n^k$ sequents. For this property the number of equations is thus linear in the variable k of the (n,k)-spindle, whereas the size of the tableau is exponential in the variable k.

Does the following $s1 \models_{\mathcal{M}} \nu Z_1.\langle a\rangle\mu Z_2.\langle a\rangle\langle a\rangle Z_2 \vee \langle a\rangle\langle a\rangle Z_1$ hold in (all transitions labelled with a):



This property was proved by our new local model checker with the following results: It created 6 equations and took one second time for the whole procedure. The tableau based model checker was interrupted after having generated more than 22 million (!) tableau sequents.

The model checker of the Concurrency Workbench could cope well with both examples: it "quickly" returned the result. The techniques for avoiding recomputation came in useful. This was not the case in the following example.

We considered the two process mutual exclusion algorithm of Peterson, given in [Wal89]. The property we proved is: "As long as process 1 proceeds, after a request it eventually enters the critical section." In order to detect progress we added "tick" and "tack" dummy actions which alternate each other when process 1 performs some action. Then the property to prove can be formulated as: "if a request comes, then along all paths where ticks and tacks alternate each other, eventually an enter will follow". The μ-calculus formula representing this property is:

$$\begin{aligned} \nu Z_1.([-]Z_1 \wedge \mu Z_2.(\quad & \nu Z_3.([\backslash\mathrm{enter}]((\langle\mathrm{tick}\rangle tt \vee \\ & \nu Z_4.([\backslash\mathrm{enter}]((\langle\mathrm{tack}\rangle tt \vee Z_2) \wedge Z_4)) \wedge Z_3))))) \end{aligned}$$

This rather difficult modal μ-calculus formula is of alternation depth 3 and nesting depth 4. Unfortunately the full discussion of this example exceeds the aim of the paper.

This property, together with our extended Peterson-2 algorithm, was fed to the model checker, with the following result: The model checker came back with a positive result after slightly more than 10 minutes of CPU time. The CWB model checker on the other hand could not compute an answer for the same input within 24 hours elapsed time. During execution our model checker created 156 out of a possible 240 Boolean equations. This example is typical of the results we got from an extensive investigation into several mutual exclusion algorithms with different liveness properties.

7 Conclusion

We presented a novel, algebraic approach for solving Boolean equation systems. As main application model checking in the full modal μ-calculus was intended. In contrast to other approaches using equation systems our method is not based on approximation techniques and backtracking. The method works straightforward by successively eliminating variables and reducing the Boolean equation system, similar to Gauß elimination for linear equation systems. Homogeneous, hierarchical and alternating fixpoints are treated uniformly. Contrary to other techniques Gauß elimination leads to both a global and a local model checking algorithm within one framework. The local version is closely related to the tableau methods, but has a better worst case complexity. An extension to model checking for infinite state spaces is in work.

There exists a prototype implementation of the Gauß elimination using BDDs for the representation of Boolean expressions. Several examples (e.g. fairness properties for mutex algorithms) showed that the local version of our algorithm beats existing tableau methods.

Acknowledgements Frank Wallner implemented the local version of the algorithm. The examples were elaborated together with Dieter Barnard. I thank Javier Esparza, Astrid Kiehn, Bernhard Steffen and Frank Wallner for proof reading and helpful comments.

References

[And92] H. Andersen. “Model Checking and Boolean Graphs.” In *Proc. of ESOP’92*, LNCS 582, 1992.

[AC88] A. Arnold, P. Crubille. “A Linear Algorithm to Solve Fixed-Point Equations on Transition Systems.” *Information Processing Letters*, vol. 29, 57-66, 1988.

[BCMDH92] J.R. Burch and E.M. Clarke and K.L. McMillan and D.L. Dill and L.J. Hwang. “Symbolic Model Checking: 10^{20} States and Beyond.” *Information and Computation*, Vol. 98, pp. 142-170, 1992.

[Bek84] H. Bekic. “Definable operations in general algebras, and the theory of automata and flow charts.” In C.B.Jones, editor, *Hans Bekic: Programming Languages and Their Definition*, LNCS 177, 1984.

[Bra92] J. Bradfield. “Verifying Temporal Properties of Systems.” Birkhäuser,1992.

[BS91] J. Bradfield, C. Stirling. “Verifying Temporal Properties of Processes.” *Proc. of CONCUR’90*, LNCS 458, 1991.

[Bry92] R.E. Bryant. “Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams.” *ACM Computing Surveys*, Vol. 24, No. 3, September 1992.

[CE81] E. M. Clarke and E. A. Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logics.” In LNCS 131, pp.52-71, 1981.

[CES86] E. M. Clarke, E. A. Emerson and A. P. Sistla. “Automatic verification of finite-state concurrent systems using temporal logic specifications.” In *ACM Trans. on Programming Languages and Systems 8* , pp. 244-263, 1986.

[Cle90] R. Cleaveland. “Tableau-based model checking in the propositional mu-calculus.” *Acta Informatica*, Vol. 27, pp. 725-747, 1990.

[ClS91] R. Cleaveland and B. Steffen. “A Linear-Time Model Checking Algorithm for the Alternation-Free Modal Mu-Calculus.” In *Proc. of CAV’91*, LNCS 575, 1992.

[CDS92] R. Cleaveland, M. Dreimüller and B. Steffen. “Faster Model Checking for the Modal Mu-Calculus.” In *Proc. of CAV’92*, LNCS 663, 1993.

[EmL86] E.A. Emerson and C.-L. Lei. “Efficient model checking in fragments of the propositional mu-calculus.” In *Proc. of LICS’86*, Computer Society Press, 1986.

[Koz83] D. Kozen. “Results on the Propositional μ-Calculus.” *TCS*, Vol. 27, 1983, pp. 333-354.

[Lar92] K. Larsen. “Efficient Local Correctness Checking.” In *Proc. of CAV’92*, LNCS 663, 1993.

[LBCJM94] D. E. Long, A. Browne, E. M. Clarke, S. Jha, W. R. Marrero. “An improved algorithm for the evaluation of fixpoint expressions.” In *Proc. of CAV’94*, LNCS 818, 1994.

[Mad92] A. Mader. “Tableau Recycling.” In *Proc. of CAV’92*, LNCS 663, 1993.

[Rud74] S. Rudeanu. “Boolean Functions and Equations.” North-Holland Publishing Company, 1974.

[Sti92] C. Stirling. “Modal and Temporal Logics.” In *Handbook of Logic in Computer Science*, Oxford University Press, 1992.

[StW89] C. Stirling and D. Walker. “Local model checking in the modal mu-calculus.” In *Proc. of TAPSOFT’89*, LNCS 351, 1989.

[Tar55] A. Tarski. “ A Lattice Theoretical Fixpoint Theorem and its Applications.” In *Pacific Journal of Mathematics*, 5, 1955.

[VeL92] B. Vergauwen and J. Lewi “A linear algorithm for solving fixed-point equations on transition systems” In *Proc. of CAAP’92*, LNCS 581, 1992.

[Wal89] D. Walker. “Automated Analysis of Mutual Exclusion Algorithms using CCS.” Technical Report ECS-LFCS-89-91, University of Edinburgh, 1991.

[Xin92] Liu Xinxin. “Specification and Decomposition in Concurrency.” PhD Thesis, Aalborg University Center, Denmark, 1992.

Appendix: Proofs

Proposition 1 For the solution Y of the equation system $X \overset{\sigma}{=} A(X)$ consisting of one single equation it holds:

$$Y = \begin{cases} A(\mathsf{false}) & \text{if } \sigma = \mu \\ A(\mathsf{true}) & \text{if } \sigma = \nu \end{cases}$$

Proof For $\sigma = \mu$ there are three cases:

1) $A(X) = \mathsf{true}$, i.e. the evaluation of the expression A is independent of the valuation of the free variable X. Then the solution is $Y = \mathsf{true}$.
2) $A(X) = \mathsf{false}$, i.e. the evaluation of the expression A is independent of the valuation of the free variable X. Then the solution is $Y = \mathsf{false}$.
3) In the remaining case because of monotonicity of $A(X)$ the following holds: $A(\mathsf{false}) = \mathsf{false}$ and $Y = \mathsf{false}$ is the solution.

Analogously it holds $Y = A(\mathsf{true})$ that is the solution of $X \overset{\nu}{=} A(X)$. $\square$

Often the solution of $X \overset{\sigma}{=} A(X)$ will be denoted by the expression $\sigma X.A(X)$.

For the proof of propositions 2 and 3 we need the following property of the solution.

Proposition 6 The solution $\underline{Y}$ of the Boolean equation system $\underline{E}$ is an extremal point of $\underline{E}$, i.e., for $1 \leq i \leq n$ holds:
Y_i is solution of $X_i \overset{\sigma_i}{=} A_i(Y_1, \ldots, Y_{i-1}, X_i, Y_{i+1}, \ldots, Y_n)$.

Proof trivial □

Proposition 2 $\underline{Y}$ is the solution of the Boolean equation system $\underline{E}$ of the form $\underline{X} \overset{\sigma}{=} \underline{A}(\underline{X})$, iff $\underline{Y}$ also is the solution of the modified Boolean equation system $\underline{F}$, where the equations are of the following form:

$$\begin{array}{lcl} X_1 & \overset{\sigma_1}{=} & A_1(\mathsf{b}_1, X_2, \quad \ldots \quad, X_n) \\ \vdots & & \\ X_i & \overset{\sigma_i}{=} & A_i(X_1, \ldots, X_{i-1}, \mathsf{b}_i, X_{i+1}, \ldots, X_n) \\ \vdots & & \\ X_n & \overset{\sigma_n}{=} & A_n(X_1, \quad \ldots \quad, X_{n-1}, \mathsf{b}_n) \end{array} \qquad \text{where} \quad \mathsf{b}_i = \begin{cases} \mathsf{true} & \text{if } \sigma_i = \nu \\ \mathsf{false} & \text{if } \sigma_i = \mu \end{cases}$$

Proof First we sketch the proof scheme, which will be the same for this and the next proof. For reasons of readability we introduce another abbreviation: $\underline{E}^{(i+1)}$ denotes $\underline{E}^{(i+1)}[Y_1/X_1, \ldots, Y_i/X_i]$, and analogously $\underline{E}^{(i)}$ denotes $\underline{E}^{(i)}[Y_1/X_1, \ldots, Y_{i-1}/X_{i-1}]$. The same holds for $\underline{F}^{(i+1)}$ and $\underline{F}^{(i)}$.
The proof is based on induction on i. The hypothesis for an induction step is: $\underline{Y}^{(i+1)}$ is solution of $\underline{E}^{(i+1)}$, iff $\underline{Y}^{(i)}$ is solution of $\underline{F}^{(i+1)}$. Then the conclusion of an induction step will be: $\underline{Y}^{(i)}$ is solution of $\underline{E}^{(i)}$, iff $\underline{Y}^{(i)}$ is solution of $\underline{F}^{(i)}$.
For this purpose we have to show the following: The solution of $\underline{E}^{(i)}$ is a fixpoint of $\underline{F}^{(i)}$, i.e. the solution of $\underline{F}^{(i)}$ is lower or equal to the solution of $\underline{E}^{(i)}$.
On the other hand we have to show that the solution of $\underline{F}^{(i)}$ is also a fixpoint of $\underline{E}^{(i)}$, i.e. the solution of $\underline{E}^{(i)}$ is at most smaller than the solution of $\underline{F}^{(i)}$. From these we can conclude that both solutions must be the same.
(i) induction basis: Y_n is the solution of $\underline{E}^{(n)}$, iff Y_n is the solution of $\underline{F}^{(n)}$. Suppose Y_n is the solution of $X_n \overset{\sigma_n}{=} A_n(Y_1, \ldots, Y_{n-1}, X_n)$. According to proposition 1 for the solution Y_n holds: $Y_n = A_n(Y_1, \ldots, Y_{n-1}, \mathsf{b}_n)$ and hence Y_n is also the solution of $\underline{F}^{(n)}$. With the same argument the converse holds.
(ii) induction step: for all i with $1 \leq i < n$ we now prove that the solution of the one equation system $\underline{E}^{(i)}$ is a fixpoint of $\underline{F}^{(i)}$ and conversely. Here we show a stronger proposition, namely that every extremal point of one equation system is also a fixpoint of the other one. Because the solution is an extremal point, we can conclude the desired proposition.
Suppose $\underline{Y}^{(i)}$ is an extremal point of $\underline{E}^{(i)}$. Then for $i \leq k \leq n$ holds:

$$\begin{array}{rcll} Y_k & = & \sigma_k X_k . A_k(Y_1, \ldots, Y_{k-1}, X_k, Y_{k+1}, \ldots, Y_n) & \underline{Y}^{(i)} \text{ is extremal point of } \underline{E}^{(i)} \\ & = & A_k(Y_1, \ldots, Y_{k-1}, \mathsf{b}_k, Y_{k+1}, \ldots, Y_n) & \text{proposition 1} \end{array}$$

Hence $\underline{Y}^{(i)}$ is also a fixpoint of $\underline{F}^{(i)}$.
Conversely suppose $\underline{Y}^{(i)}$ is an extremal point of $\underline{F}^{(i)}$. Then for $i \leq k \leq n$ it holds:

$$\begin{array}{rcll} Y_k & = & \sigma_k X_k . A_k(Y_1, \ldots, Y_{k-1}, \mathsf{b}_k, Y_{k+1}, \ldots, Y_n) & \underline{Y}^{(i)} \text{ is extrem. point of } \underline{F}^{(i)} \\ & = & A_k(Y_1, \ldots, Y_{k-1}, \mathsf{b}_k, Y_{k+1}, \ldots, Y_n) & X_k \text{ is not free in this expr.} \\ & = & \sigma_k X_k . A_k(Y_1, \ldots, Y_{k-1}, X_k, Y_{k+1}, \ldots, Y_n) & \text{proposition 1} \\ & = & A_k(Y_1, \ldots, Y_{k-1}, Y_k, Y_{k+1}, \ldots, Y_n) & \text{ev. extrem. point is a fixp.} \end{array}$$

Hence $\underline{Y}^{(i)}$ is also a fixpoint of $\underline{E}^{(i)}$.
With these arguments we now can apply the induction step n times, and altogether we get the proposition. □

Proposition 3 The Boolean vector $\underline{Y}$ is the solution of the equation system $\underline{E}$, iff it is the solution of the equation system $\underline{G}$, where $\underline{G}$ is the modified equation system:

$$\begin{array}{rcl} X_1 & \overset{\sigma_1}{=} & A_1 \quad (X_1, \ldots, X_{n-1}, A_n(X_1, \ldots, X_{n-1}, \mathsf{b}_n)) \\ \vdots & & \\ X_{n-1} & \overset{\sigma_{n-1}}{=} & A_{n-1}(X_1, \ldots, X_{n-1}, A_n(X_1, \ldots, X_{n-1}, \mathsf{b}_n)) \\ X_n & \overset{\sigma_n}{=} & A_n \quad (X_1, \ldots, X_{n-1}, \mathsf{b}_n) \end{array}$$

Proof The proof is based on the same scheme as the one of proposition 2.
We will use the analogous abbreviations as in proposition 2.
(i) induction basis:
Y_n is the solution of $\underline{E}^{(n)}$, iff Y_n is the solution of $\underline{G}^{(n)}$.
Suppose Y_n is the solution of $X_n \overset{\sigma_n}{=} A_n(Y_1, \ldots, Y_{n-1}, X_n)$. According to proposition 1 the following holds: $Y_n = A_n(Y_1, \ldots, Y_{n-1}, \mathsf{b}_n)$ and hence Y_n is also the solution of $\underline{G}^{(n)}$. The converse holds with the same argument.
(ii) induction step:
For all i with $1 \leq i < n$ each extremal point of $\underline{G}^{(i)}$ is a fixpoint of $\underline{E}^{(i)}$ and conversely.
Suppose $\underline{Y}^{(i)}$ is an extremal point of $\underline{G}^{(i)}$.

$$\begin{array}{rcll} Y_n & = & A_n(Y_1, \ldots, Y_{n-1}, \ \mathsf{b}_n) & \text{$\underline{Y}$ is a fixpoint of $\underline{G}$} \\ & = & \sigma_n X_n . A_n(Y_1, \ldots, Y_{n-1}, X_n) & \text{proposition 1} \\ (*) & = & A_n(Y_1, \ldots, Y_{n-1}, \ Y_n) & \text{every extremal point is also a fixpoint} \end{array}$$

and for $i \leq k \leq n$:

$$\begin{array}{rcll} Y_k & = & A_k(Y_1, \ldots, Y_{n-1}, A_n(Y_1, \ldots, Y_{n-1}, \ \mathsf{b}_n)) & \text{$\underline{Y}^{(i)}$ is a fixpoint of $\underline{G}^{(i)}$} \\ & = & A_k(Y_1, \ldots, Y_{n-1}, \ Y_n) & \text{with (*)} \end{array}$$

Hence $\underline{Y}^{(i)}$ is also a fixpoint $\underline{E}^{(i)}$.
Now suppose $\underline{Y}^{(i)}$ is an extremal point of $\underline{E}^{(i)}$.

$$\begin{array}{rcll} Y_n & = & \sigma_n X_n . A_n(Y_1, \ldots, Y_{n-1}, X_n) & \text{$\underline{Y}^{(i)}$ is an extremal point of $\underline{E}^{(i)}$} \\ (**) & = & A_n(Y_1, \ldots, Y_{n-1}, \ \mathsf{b}_n) & \text{proposition 1} \end{array}$$

and for $i \leq k \leq n$:

$$\begin{array}{rcll} Y_k & = & A_k(Y_1, \ldots, Y_{n-1}, Y_n) & \text{$\underline{Y}$ is a fixpoint of $\underline{E}$} \\ & = & A_k(Y_1, \ldots, Y_{n-1}, A_n(Y_1, \ldots, Y_{n-1}, \mathsf{b}_n)) & \text{with (**)} \end{array}$$

Hence $\underline{Y}^{(i)}$ is also a fixpoint of $\underline{G}^{(i)}$.
Now we can apply the induction step n times, and altogether we get the proposition. □

MONA: MONADIC SECOND-ORDER LOGIC IN PRACTICE[1]

JESPER GULMANN HENRIKSEN[2], JAKOB JENSEN[2], MICHAEL JØRGENSEN[2], NILS KLARLUND[3], ROBERT PAIGE[4], THEIS RAUHE[2], AND ANDERS SANDHOLM[2]

Abstract. The purpose of this article is to introduce Monadic Second-order Logic as a practical means of specifying regularity. The logic is a highly succinct alternative to the use of regular expressions. We have built a tool MONA, which acts as a decision procedure and as a translator to finite-state automata. The tool is based on new algorithms for minimizing finite-state automata that use binary decision diagrams (BDDs) to represent transition functions in compressed form. A byproduct of this work is an algorithm that matches the time but improves the space of Sieling and Wegener's algorithm to reduce OBDDs in linear time.

The potential applications are numerous. We discuss text processing, Boolean circuits, and distributed systems. Our main example is an automatic proof of properties for the "Dining Philosophers with Encyclopedia" example by Kurshan and MacMillan. We establish these properties for the parameterized case *without* the use of induction.

Our results show that, contrary to common beliefs, high computational complexity may be a desired feature of a specification formalism.

1. Introduction.

In computer science, *regularity* amounts to the concept that a class of structures is recognized by a finite-state device. Often phenomena are so complicated that their regularity either

- may be overlooked, as in the case of parameterized verification of distributed finite-state systems with a regular communication topology; or
- may not be exploited, as in the case when a search pattern in a text editor is known to be regular, but in practice inexpressible as a regular expression.

In this paper we argue that the *Monadic Second-Order Logic* or *M2L* can help in practice to identify and to use regularity. In M2L, one can directly mention positions and subsets of positions in the string. This feature distinguishes the logic from regular expressions or automata. Together with quantification and Boolean connectives, an extraordinary succinct formalism arises.

[1]This article is a heavily revised version of [10].

[2]BRICS, Centre of the Danish National Research Foundation for Basic Research in Computer Science, Department of Computer Science, University of Aarhus.

[3]The corresponding author is Nils Klarlund, who is with BRICS, Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Aarhus C. E-mail: klarlund@daimi.aau.dk.

[4]Department of Computer Science, CIMS, New York University, 251 Mercer St. New York, New York, USA; research partially supported by ONR grant N00014-93-1-0924, AFOSR grant AFOSR-91-0308, and NSF grant MIP-9300210.

Although it has been known for thirty-five years that M2L defines regular languages (see [17]), the translator from formulas to automata that we describe in this article appears to be one of the first implementations.

The reason such projects have not been pursued may be the staggering theoretical lower-bound: any decision procedure is bound to sometimes require as much time as a stack of exponentials that has height proportional to the length of the formula.

It is often believed that the lower the computational complexity of a formalism is, the more useful it may be in practice. We want to counter such beliefs in this article — at least for logics on finite strings.

Why use logic? Some simple finite-state languages easily described in English call for convoluted regular expressions. For example, the language L_{2a2b} of all strings over $\Sigma = \{a, b, c\}$ containing at least two occurrences of a *and* at least two occurrences of b seems to require a voluminous expression, such as

$$\begin{aligned}
&\Sigma^* a \Sigma^* a \Sigma^* b \Sigma^* b \Sigma^* \\
\cup\ &\Sigma^* a \Sigma^* b \Sigma^* a \Sigma^* b \Sigma^* \\
\cup\ &\Sigma^* a \Sigma^* b \Sigma^* b \Sigma^* a \Sigma^* \\
\cup\ &\Sigma^* b \Sigma^* b \Sigma^* a \Sigma^* a \Sigma^* \\
\cup\ &\Sigma^* b \Sigma^* a \Sigma^* b \Sigma^* a \Sigma^* \\
\cup\ &\Sigma^* b \Sigma^* a \Sigma^* a \Sigma^* b \Sigma^*.
\end{aligned}$$

If we added $\cap$ to the operators for forming regular expressions, then the language L_{2a2b} could be expressed more concisely as $(\Sigma^* a \Sigma^* a \Sigma^*) \cap (\Sigma^* b \Sigma^* b \Sigma^*)$. Even with this extended set of operators, it is often more convenient to express regular languages in terms of positions and corresponding letters. For example, to express the set $L_{a\mathrm{after}b}$ of strings in which every b is followed by an a, we would like a formal language allowing us to write something like

> "for every position p, if there is a b in p then for some position q after p, there is an a in q."

The extended regular languages do not seem to allow an expression that very closely reflects this description — although upon some reflection a small regular expression can be found. But in M2L we can express $L_{a\mathrm{after}b}$ by a formula

$$\forall p : {'b'}(p) \;\Rightarrow\; \exists q :\; p < q \;\wedge\; {'a'}(q)$$

(Here the predicate ${'b'}(p)$ means "there is a b in position p".) In general, we believe that many errors can be avoided if logic is used when the description in English does not lend itself to a direct translation into regular expressions or automata. However, the logic can easily be combined with other methods of specifying regularity since almost any such formalism can be translated with only a linear blow-up into M2L.

Often regularity is identified by means of *projections*. For example, if L_{trans} is regular on a cross-product alphabet $\Sigma \times \Sigma$ (e.g. describing a parameterized transition relation, see Section 5) and L_{start} is a regular language on Σ describing a set of start strings, then the set of strings that can be reached by a transition from a start string is $\pi_2(L_{trans} \cap \pi_1^{-1}(L_{start}))$, where π_1 and π_2 are the projections from $(\Sigma \times \Sigma)^*$ to the first and second component. Such language-theoretic operations can be very elegantly expressed in M2L.

Our results. In this article, we discuss applications of M2L to text processesing and the description of parameterized Boolean circuits. Our principal application is a new proof technique for establishing properties about parameterized, distributed finite-state systems with regular communication topology. We illustrate our method by showing safety and liveness properties for a non-trivial version of the Dining Philosophers' problem as proposed in [11] by Kurshan and MacMillan.

We present MONA, which is our tool that translates formulas in M2L to finite-state machines. We show how BDDs can be used to overcome an otherwise inherent problem of exponential explosion. Our minimization algorithm works very fast in practice thanks to a simple generalization of the unary apply operation of BDDs.

Comparisons to other work. Parameterized circuits are described using BDDs in [8]. This method relies on formulating inductive steps as finite-state devices and does not provide a single specification language. The work in [14] is closer in spirit to our method in that languages of finite strings are used although not as part of a logical framework. In [2], another approach is given based on iterating abstractions. The parameterized Dining Philosopher's problem is solved in [11] by a finite-state induction principle.

A tool for M2L on finite, binary trees has been developed at the University of Kiel [16]. Apparently, this tool has only been used for very simple examples.

In [7], a programming language for finite domains based on a fixed point logic is described and used for verification of non-parameterized finite systems.

Contents. In Section 2, we explain the syntax and semantics of M2L on strings. We recall the correspondence to automata theory in Section 3. We give several applications of M2L and the tool in Section 4: text patterns, parameterized circuits, and equivalence testing. Our main example of parameterized verification is discussed in Section 5. We give an overview of our implementation in Section 6.

2. The Monadic Second-order Logic on Strings.

Let Σ be an alphabet and let w be a string over Σ. The semantics of the logic determines whether a closed M2L formula ϕ holds on w. The language $L(\phi)$ denoted by ϕ is the set of strings that make ϕ hold. Assume now that w has length n and consists of letters $a_0 a_1 ... a_{n-1}$. The *positions* in w are then 0,...,$n-1$. We can now describe the three syntactic categories of M2L on strings.

A *position term* t is either

- the constant 0 (which denotes the position 0);
- the constant \$ (which denotes the last position, i.e. $n-1$);
- a position variable p (which denotes a position i);
- of the form $t \oplus i$ (which denotes the position $j + i \mod n$, where j is the interpretation of t); or
- of the form $t \ominus i$ (which denotes the position $j - i \mod n$, where j is the interpretation of t);

(Position terms are only interpreted for non-empty strings).

A *position set term* T is either

- the constant $\emptyset$ (which denotes the empty set);
- the constant **all** (which denotes the set $\{0, ..., n-1\}$);
- a position set variable P (which denotes a subset of positions);
- of the form $T_1 \cup T_2$, $T_1 \cap T_2$, or $\complement T_1$ (which are interpreted in the natural way);
- of the form $T + i$ (which denotes the set of positions in T shifted right by an amount of i); or

- of the form $T - i$ (which denotes the set of positions in T shifted left by an amount of i);

A *formula* ϕ is either of the form

- $'a'(t)$ (which holds if letter a_i in $w = a_0 a_1 \cdots$ is a, where i is the interpretation of t);
- $t_1 = t_2$, $t_1 < t_2$ or $t_1 \leq t_2$ (which are interpreted in the natural way);
- $T_1 = T_2$, $T_1 \subseteq T_2$, or $t \in T$ (which are interpreted in the natural way);
- $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \Rightarrow \phi_2$, or $\phi_1 \Leftrightarrow \phi_2$ (where ϕ_1 and ϕ_2 are formulas, and which are interpreted in the natural way);
- $\exists p : \phi$ (which is true, if there is a position i such that ϕ holds when i is substituted for p);
- $\forall p : \phi$ (which is true, if for all positions i, ϕ holds when i is substituted for p);
- $\exists P : \phi$ (which is true, if there is a subset of positions I such that ϕ holds when I is substituted for P); or
- $\forall P : \phi$ (which is true, if for all subsets of positions I, ϕ holds when I is substituted for P);

3. From M2L to Automata.

In this section, we recall the method for translating a formula in M2L to an equivalent finite-state automaton (see [17] for more details). Note that any formula ϕ can be interpreted, given a string w and a *value assignment* $\mathcal{I}$ that fixes values of the free variables. If ϕ then holds, we write $w, \mathcal{I} \models \phi$. The key idea is that a value assignment and the string may be described together as a word over an extended alphabet consisting of Σ and extra binary tracks, one for each variable. By structural induction, we then define for each formula an automaton that exactly recognizes the words in the extended alphabet corresponding to pairs consisting of a string and an assignment that satisfy the formula.

Example. Assume that the free variables are $\mathcal{P} = \{P_1, P_2\}$ and that $\Sigma = \{a, b\}$. Let us consider the string $w = abaa$ and value assignment

$$\mathcal{I} = [P_1 \mapsto \{0, 2\}, P_2 \mapsto \emptyset].$$

The set $\mathcal{I}(P_1) = \{0, 2\}$ can be represented by the bit pattern 1010, since the numbered sequence

$$\underset{0\,1\,2\,3}{1010}$$

defines that 0 is in the set (the bit in position 0 is 1), 1 is not in the set (the bit in position 1 is 0), etc. Similarly, the bit pattern 0000 describes $\mathcal{I}(P_2) = \emptyset$.

If these patterns are laid down as extra "tracks" along w, we obtain an *extended word* α, which may be depicted as:

a	b	a	a
1	0	1	0
0	0	0	0

Technically, we define $\alpha = \alpha_0 \cdots \alpha_3$ as the word $(a, 1, 0)(b, 0, 0)(a, 1, 0)(a, 0, 0)$ over the alphabet $\Sigma \times \mathbb{B} \times \mathbb{B}$ of *extended letters*, where $\mathbb{B} = \{0, 1\}$ is the set of truth values.

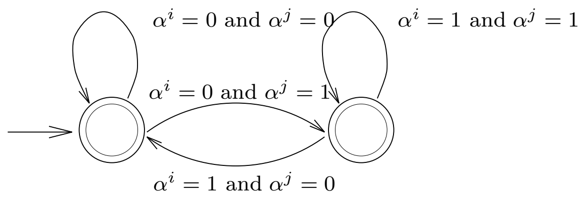
This correspondence can be generalized to any w and any value assignment for a set of variables $\mathcal{P}$ (which can all be assumed to be second-order).

By structural induction on formulas, we construct automata $A^{\phi,\mathcal{P}}$ over alphabet $\Sigma\times\mathbb{B}^k$—where $\mathcal{P}=\{P_1,\cdots,P_k\}$ is any set of variables containing the free variables in ϕ—satisfying the *fundamental correspondence*:

$$w,\mathcal{I}\models\phi \text{ iff } (w,\mathcal{I})\in L(A^{\phi,\mathcal{P}})$$

Thus $A^{\phi,\mathcal{P}}$ accepts exactly the pairs $(w,\mathcal{I})$ that make ϕ true.

Example. Let ϕ be the formula $P_i=P_j+1$. Thus when ϕ holds, P_i is represented by the same bit pattern as that of P_j but shifted right by one position. This can be expressed by the automaton $A^{\phi,\mathcal{P}}$:



$\alpha^i=0$ and $\alpha^j=0$
$\alpha^i=1$ and $\alpha^j=1$
$\alpha^i=0$ and $\alpha^j=1$
$\alpha^i=1$ and $\alpha^j=0$

In this drawing, α^i refers to the ith extra track. Thus, the automaton checks that the ith track holds the same bit as the jth track the instant before.

4. Applications.

4.1. **Text patterns.** The language L_{2a2b} of strings containing at least two occurrences of a and two occurrences of b can be described in M2L by the formula

$$\begin{array}{l}(\exists p_1,p_2 : 'a'(p_1)\ \wedge\ 'a'(p_2)\ \wedge\ p_1\neq p_2)\ \wedge\\ (\exists p_1,p_2 : 'b'(p_1)\ \wedge\ 'b'(p_2)\ \wedge\ p_1\neq p_2)\end{array}$$

Our translator yields the minimal automaton, which contains nine states, in a fraction of a second.

The language $L_{a\text{after}b}$ given by the formula

$$\forall p : 'b'(p)\ \Rightarrow\ \exists q :\ p<q\ \wedge\ 'a'(q)$$

is translated to the minimal automaton, which has two states, in .3 seconds.

A far more complicated language to express is $L_{<1\text{apart}}$ consisting of every string over $\{a,b\}$ such that for any prefix the number of a's and b's are at most one apart. When using regular expressions or M2L, one needs to struggle a bit, but in M2L there is a strategy for describing the functioning of the finite-state machine that comes to mind.

We observe that a position p may be used to designate a prefix; for example, 0 denotes the prefix consisting of the first letter and \$ (the last position) denotes the whole input string. We may now recognize a string in $L_{<1\text{apart}}$ by identifying three sets of positions: the set P_0 corresponding to prefixes with an equal number of a's and b's, the set P_{+1} corresponding to prefixes where the number of a's is one greater than the number of b's, and the set P_{-1} corresponding to prefixes where the number of a's is one less than the number of b's:

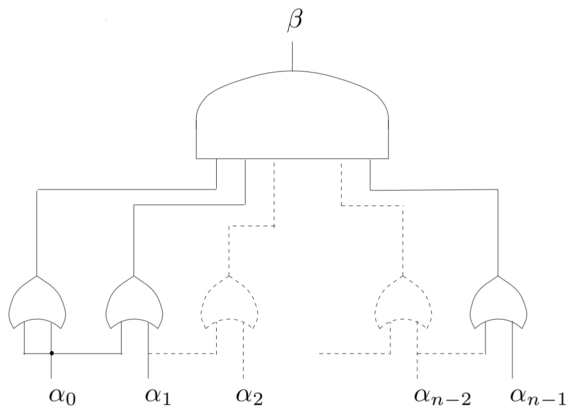


Figure 1. A parameterized circuit.

$$
\begin{aligned}
\exists P_0, P_{+1}, P_{-1} : P_0 \cup P_{+1} \cup P_{-1} = \mathbf{all}& \\
&\wedge\ 0 \notin P_0 \\
&\wedge\ 0 \in P_{+1} \Leftrightarrow {'a'}(0) \\
&\wedge\ 0 \in P_{-1} \Leftrightarrow {'b'}(0) \\
&\wedge\ \forall p : (p > 0 \ \Rightarrow \\
&\qquad\qquad p \in P_0 \ \Leftrightarrow ({'a'}(p) \ \wedge\ p \ominus 1 \in P_{-1}) \\
&\qquad\qquad\qquad\qquad \vee\ ({'b'}(p) \ \wedge\ p \ominus 1 \in P_{+1}) \\
&\qquad\quad \wedge\ p \in P_{+1} \ \Leftrightarrow\ {'a'}(p) \ \wedge\ p \ominus 1 \in P_0 \\
&\qquad\quad \wedge\ p \in P_{-1} \ \Leftrightarrow\ {'b'}(p) \ \wedge\ p \ominus 1 \in P_0)
\end{aligned}
$$

The resulting four-state automaton is calculated in a fraction of a second.

4.2. **Parameterized circuits.** Assume that we are given a drawing as in Figure 1 denoting a parameterized Boolean function.

How do we describe the language $L_{ex} \subseteq \mathbb{B}^*$ of input bit patterns that make the output true? From the drawing, no immediate description as a regular expression or finite-state automaton is apparent. In M2L, however, it is easy to model the outputs of the n or-gates as a second-order variable Q, which allows the language to be described from a direct interpretation of the drawing. Note that the or-gate at position $p > 0$ is true if either there is a 1 at $p - 1$ or p, or in other words: $p \in Q \Leftrightarrow {'1'}(p \ominus 1) \vee {'1'}(p)$. Since the output is 1 if and only if all or-gates are 1, i.e. if $Q = \mathbf{all}$, the language L_{ex} is given by the formula

$$
\begin{aligned}
\exists Q : (\forall p : \quad & (p = 0 \Rightarrow p \in Q \Leftrightarrow {'1'}(p)) \ \wedge \\
& (p > 0 \Rightarrow (p \in Q \Leftrightarrow {'1'}(p \ominus 1) \vee {'1'}(p))) \wedge Q = \mathbf{all})
\end{aligned}
$$

The resulting automaton has three states and accepts the language $(1 \cup 10)^*$, which is the regular expression that one would obtain by reasoning about the circuit. For more advanced applications to hardware verification, see [3].

4.3. **Equivalence testing.** A closed formula ϕ is a *tautology* if $L(\phi) = L(\Sigma^*)$, i.e. if all strings over Σ satisfy ϕ. The equivalence of formulas ϕ and ψ then amounts to whether $\phi \Leftrightarrow \psi$ is a tautology.

Example. That a set P contains exactly the even positions in a non-empty input string may be expressed in M2L by the following two rather different approaches: either by the formula *even1*$(P) \equiv$

$$\begin{aligned} 0 \in P \ \wedge \ \forall p : ((p \in P \ \wedge \ p < \$ \Rightarrow p \oplus 1 \notin P) \\ \wedge \ (p \notin P \ \wedge \ p < \$ \Rightarrow p \oplus 1 \in P)), \end{aligned}$$

or as a formula *even2*$(P) \equiv$

$$P \cup (P+1) = \mathbf{all} \ \wedge \ P \cap (P+1) = \emptyset \ \wedge \ P \neq \emptyset$$

To show the equivalence of the two formulas, we check the truth value of the bi-implication:

$$\forall P : even1(P) \Leftrightarrow even2(P)$$

The translation of this formula does indeed produce an automaton accepting Σ^*, and thus verifies our claim.

5. Dining Philosophers with Encyclopedia.

A distributed system is *parameterized* when the number n of processes is not fixed a priori. For such systems the state space is unbounded, and thus traditional finite-state verification methods cannot be used. Instead, one often fixes n to be, say two or three. This yields a finite state space amenable to state exploration methods. However, the validity of a property for $n = 2$, 3 does not necessarily imply that the property holds for all n.

A central problem in verification is automatically to validate parameterized systems. One way to attack the problem is to formulate induction principles such that the base case and the inductive steps can be formulated as finite-state problems. Kurshan and MacMillan [11] used such a method to verify safety and liveness properties of a non-trivial version of the Dining Philosophers example.

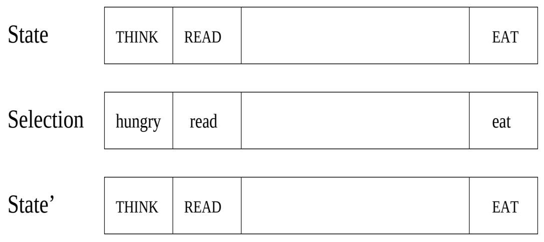


Figure 2. Dining Philosophers with Encyclopedia

In this system, symmetry is broken by an encyclopedia that circulates among the philosophers. Thus each philosopher is in one of three states: EAT, THINK, or READ. The global state can be described as a string *State* of length n over the alphabet $\Sigma_{\text{State}} = \{\text{EAT}, \text{THINK}, \text{READ}\}$, see Figure 2.

The system makes a transition according to external events that constitute a *selection* . Each process is presented with an event in the alphabet $\Sigma_{\text{Selection}} = \{\text{eat}, \text{think}, \text{read}, \text{hungry}\}$. Thus the selection can be viewed as a string *Selection* over $\Sigma_{\text{Selection}}$, see Figure 2. As shown, all processes make a synchronous transition to a new global *State*$'$ on a selection according to a transition relation trans$(State, State',$
$Selection)$, which is shown in Figure 3[1] together with an auxiliary predicate

[1]We use '#' in the beginning of a line to indicate that this line is a comment.

blocking(*Selection*) used in its definition. Thus the new state of each process is dependent on its old state and on the selection events presented to itself and its neighbors. The transition relation is so complicated that it is hard to grasp the functioning of the system.

Fortunately, the parameterized transition relation can be translated into basic M2L on strings. For example, we encode *State* using two second-order variables P and Q with the convention that

$$\begin{aligned}\mathrm{EAT}_p(\mathit{State}) &\equiv p \in P \ \wedge\ p \in Q \\ \mathrm{READ}_p(\mathit{State}) &\equiv p \notin P \ \wedge\ p \in Q \\ \mathrm{THINK}_p(\mathit{State}) &\equiv p \notin P \ \wedge\ p \notin Q\end{aligned}$$

Similarly, State' and *Selection* can also each be encoded using two second-order variables. Thus, the predicate $\mathrm{trans}(\mathit{State}, \mathit{State}', \mathit{Selection})$ becomes a formula with six free second-order variables.

For this distributed system there are two important properties to verify:

- *Safety Property*: The encyclopedia is neither lost nor replicated. Thus there is always exactly one process in state READ.
- *Liveness Property*: If no process remains in state EAT forever, then the encyclopedia is passed around over and over.

In [11] both properties are proved in terms of a complicated induction hypothesis. This hypothesis is itself a distributed system, where each process has four states. (The Liveness Property in [11] is technically different since it is modeled in terms of selections.)

Our strategy is fundamentally different. We cannot directly verify liveness properties. But we can easily verify properties about the transition relation in the parameterized case and *without* induction as follows.

Let ϕ be an M2L formula about the global state. For example, we might consider the property that if a philosopher eats, then his neighbors do not:

$$\phi_{\mathrm{mutex}}(\mathit{State}) \equiv \forall p : \mathrm{EAT}_p(\mathit{State}) \Rightarrow \neg\mathrm{EAT}_{p\ominus 1}(\mathit{State}) \ \wedge\ \neg\mathrm{EAT}_{p\oplus 1}(\mathit{State})$$

A property given as a formula ϕ can be verified using the *invariance principle*:

$$\forall \mathit{State}, \mathit{State}', \mathit{Selection} : \phi(\mathit{State}) \ \wedge\ \mathrm{trans}(\mathit{State}, \mathit{State}', \mathit{Selection}) \Rightarrow \phi(\mathit{State}'),$$

which is also a formula in M2L. In this way, we have verified for the parameterized case that both ϕ_{mutex} and the Safety Property that exactly one philosopher reads, i.e. $\exists! p : \mathrm{READ}_p(\mathit{State})$, are invariant. MONA verifies such a formula in approximately 3 seconds on a Sparc 20.

Note that this method does not rely on a state space exploration (which is impossible since the state space is unbounded). Instead, it is based on the Invariance Principle: to show that a property holds for all reachable states, it is sufficient to show that it holds for the initial state and is preserved under any transition.

Establishing the Liveness Property. The Liveness Property can be expressed in Temporal Logic as

$$\Box(\mathrm{READ}_{p\ominus 1} \ \Rightarrow\ \Diamond\mathrm{READ}_p), \tag{1}$$

that is, it always holds that if philosopher $p \ominus 1$ reads, then eventually philosopher p reads. We must prove this property under the assumption that no philosopher eats forever:

$$\Box(\mathrm{EAT}_p \ \Rightarrow\ \Diamond\neg\mathrm{EAT}_p). \tag{2}$$

blocking(*Selection*) $\equiv$
$\text{eat}_{p\oplus 1}(\mathit{Selection}) \;\vee\; \text{hungry}_{p\ominus 1}(\mathit{Selection})$
$\vee\; \text{eat}_{p\ominus 1}(\mathit{Selection})$

trans($\mathit{State}, \mathit{State}', \mathit{Selection}$) $\equiv$
$\forall p$:

#THINK $\rightarrow$ THINK :
$(\text{THINK}_p(\mathit{State}) \;\wedge\; \text{THINK}_p(\mathit{State}') \Rightarrow$
$\text{think}_p(\mathit{Selection}) \;\wedge\; \neg(\text{read}_{p\ominus 1}(\mathit{Selection}))$
$\vee$
$\text{hungry}_p(\mathit{Selection}) \;\wedge\; \text{blocking}(\mathit{Selection}))$

$\wedge$
#THINK $\rightarrow$ EAT :
$(\text{THINK}_p(\mathit{State}) \;\wedge\; \text{EAT}_p(\mathit{State}') \Rightarrow$
$\text{hungry}_p(\mathit{Selection}) \;\wedge\; \neg(\text{blocking}(\mathit{Selection})))$

$\wedge$
#THINK $\rightarrow$ READ :
$(\text{THINK}_p(\mathit{State}) \;\wedge\; \text{READ}_p(\mathit{State}') \Rightarrow$
$\text{think}_p(\mathit{Selection}) \;\wedge\; \text{read}_{p\ominus 1}(\mathit{Selection}))$

$\wedge$
#EAT $\rightarrow$ THINK :
$(\text{EAT}_p(\mathit{State}) \;\wedge\; \text{THINK}_p(\mathit{State}') \Rightarrow$
$\text{think}_p(\mathit{Selection}) \;\wedge\; \neg(\text{read}_{p\ominus 1}(\mathit{Selection})))$

$\wedge$
#EAT $\rightarrow$ EAT :
$(\text{EAT}_p(\mathit{State}) \;\wedge\; \text{EAT}_p(\mathit{State}') \Rightarrow$
$\text{eat}_p(\mathit{Selection}))$

$\wedge$
#EAT $\rightarrow$ READ :
$(\text{EAT}_p(\mathit{State}) \;\wedge\; \text{READ}_p(\mathit{State}') \Rightarrow$
$\text{think}_p(\mathit{Selection}) \;\wedge\; \text{read}_{p\ominus 1}(\mathit{Selection}))$

$\wedge$
#READ $\rightarrow$ THINK :
$(\text{READ}_p(\mathit{State}) \;\wedge\; \text{READ}_p(\mathit{State}') \Rightarrow$
$\text{read}_p(\mathit{Selection}) \;\wedge\; \text{think}_{p\oplus 1}(\mathit{Selection}))$

$\wedge$
#READ $\rightarrow$ EAT :
$(\text{READ}_p(\mathit{State}) \;\wedge\; \text{EAT}_p(\mathit{State}') \Rightarrow$
false)

$\wedge$
#READ $\rightarrow$ READ :
$(\text{READ}_p(\mathit{State}) \;\wedge\; \text{READ}_p(\mathit{State}') \Rightarrow$
$\text{read}_p(\mathit{Selection}) \;\wedge\; \neg(\text{think}_{p\oplus 1}(\mathit{Selection})))$

Figure 3. The transition relation

So assume that $\text{READ}_{p\ominus 1}$ holds. We must prove that $\Diamond\text{READ}_p$ holds. There are two cases as follows.

- Case EAT_p holds. By asssumption (2), there is an instant when $\text{EAT}_p \;\wedge\; \neg\circ\text{EAT}_p$ holds. Thus if

$$\text{READ}_{p\ominus 1} \;\wedge\; \text{EAT}_p \;\wedge\; \neg\circ\text{EAT}_p \Rightarrow \circ\text{READ}_p \tag{3}$$

  is a valid property of the transition system, $\Diamond\text{EAT}_p$ holds. In fact, we verified using MONA that (3) indeed holds.
- Case $\neg\text{EAT}_p$ holds. If EAT_p becomes true, then use the previous case. Otherwise, $\neg\text{EAT}_p$ continues to hold. Now, by the assumption (2) at some point $\neg\text{EAT}_{p\oplus 1}$ will hold. We then use the property

$$\text{READ}_{p\ominus 1} \;\wedge\; \neg\text{EAT}_p \;\wedge\; \neg\circ\text{EAT}_{p\oplus 1} \Rightarrow \circ\text{READ}_p \;\vee\; \circ\text{EAT}_p, \tag{4}$$

  which we have also verified using MONA, to show that eventually READ_p holds (or eventually EAT_p holds, which contradicts the assumption that $\neg\text{EAT}_p$ continues to hold).

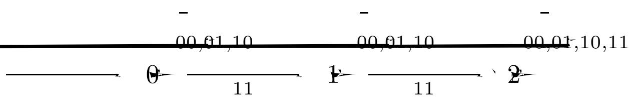
6. Implementation.

MONA is our implementation of the decision procedure, which translates formulas of M2L to finite-state automata as outlined in Section 3. Our tool is implemented in Standard ML of New Jersey. A previous version of MONA was written in C with explicit garbage collection and based on representing transition functions in a conjunctive normal form. Our present tool runs up to 50 times faster due to improved algorithms.

Representation of automata. Since the size of the extended alphabet grows exponentially with the number of variables, a straightforward implementation based on explicitly representing the alphabet would only work for very simple examples. Instead, we represent the transition relation using Binary Decision Diagrams (BDDs) [4, 5]. In this way, the alphabet is never explicitly represented. For the external alphabet of ASCII-characters, we choose an encoding based on seven extra tracks holding the binary representation. Thus, character classes such as `[a-zA-Z]` become represented as very simple BDDs.

A deterministic automaton A is represented as follows. The state space is $Q = \{0, 1, \ldots, n-1\}$, where n is size of the state space; $\mathbb{B}^k$ is the extended alphabet; $i_0 \in Q$ is the initial state; $\delta : Q \times \mathbb{B}^k \to Q$ is the transition function; and $F \subseteq Q$ is the set of accepting states. We use a bit vector of size n to represent F and an array containing n pointers to roots of multi-terminal BDDs representing δ. A leaf of a BDD holds the integer designating the next state. An internal node v is called a *decision node* and contains an *index* denoted $v.index$, where $0 \leq v.index < k$, and high and low successors $v.hi$ and $v.lo$. If b is a sequence of k bits, i.e. $b \in \mathbb{B}^k$, then $\delta(q, b)$ is found by looking up the qth entry in the array and following the decision nodes according to b until a leaf is reached (node v is followed by selecting the high successor if the $v.index$th component of b is 1 and the low successor if it is 0).

For example, the following finite automaton accepting all strings over $\mathbb{B}^2$ with at least two occurrences of the letter "11"



could be represented as in Figure 4.

The use of BDDs makes the representation very succinct in comparison to our earlier attempt to handle automata with large alphabets [10]. In most cases, we avoid the exponential blow-up associated with an explicit representation of the alphabet. We shall see that all operations on automata needed can be performed by means of simple BDD operations.

Another possibility would have been to use a two-dimensional array of ordinary BDDs. But that would complicate the operations on automata, because many more BDD operations would be needed.

Rewriting formulas. The first step in the translation consists of rewriting formulas so as to eliminate nested terms. Then all terms are variables and all formulas are among a small number of basic formulas.

Translating formulas. The translation is inductive. All automata corresponding to basic formulas have a small number of states (less than five!).

The composite formulas are translated by use of operations on automata. For $\neg\phi$, $\phi_1 \wedge \phi_2$ and $\exists P : \phi$, which are the ones left after rewriting, we need the operations of complement, product, projection, and determinization.

Complement. Complementation is done by simply negating the bit vector representing the set of final states.

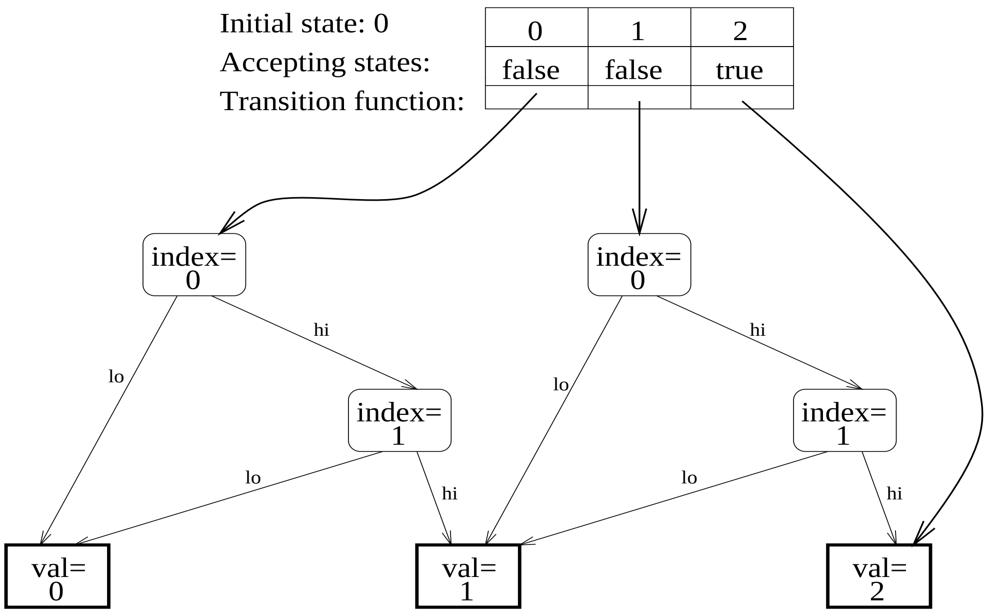


Figure 4. BDD automaton representation

Product. The product automaton A of two automata A_1 and A_2 is

$$(Q_1 \times Q_2, \mathbb{B}^k, (i_1, i_2), \delta, F_1 \times F_2),$$

where $\delta((q_1, q_2), b) = (\delta_1(q_1, b), \delta_2(q_2, b))$. We are careful, however, to consider only those states of A that are reachable from (i_1, i_2).

When considering a new state (q_1, q_2), we need to construct the BDD representing the corresponding part of the transition function δ. We use the binary apply operation on the BDDs corresponding to q_1 and q_2. For each pair of states (q', q'') encountered in a pair of leaves, we associate a unique integer in the range $\{0, 1, \ldots N-1\}$, where N is the number of different pairs considered so far. In this way, the new BDDs created conform with the standard representation.

Projection and determinization. Projection is the conversion of an automaton over $\mathbb{B}^{k+1}$ to a nondeterministic automaton over $\mathbb{B}^k$ necessary for translating a formula of the form $\exists P : \phi$. On any letter $b \in \mathbb{B}^k$, there are two transitions possible in the nondeterministic automaton corresponding to whether the P-track is 0 or 1. Therefore this automaton is not hard to construct using the projection (restriction) operation of BDDs.

Determinization is done according to the subset construction. The use of the apply operation is similar to that of the product construction except that leaves hold subsets of states.

Minimizing. Minimization seems essential in order to obtain an effective decision procedure. For example, if a tautology occurs during calculations, then it is obviously a good idea to represent it using a one-state automaton instead of an automaton with e.g. 10,000 states.

The difficulty in obtaining an efficient minimization algorithm stems from the requirement to keep our shared BDDs in reduced form. Recall that a reduced BDD has no duplicate terminals or nonterminals. Such a BDD is just a specialized form of directed acyclic graph that has been compressed by combining structurally isomorphic nodes (see Aho, Hopcroft, and Ullman [1] or Section 3.4 of Cai and Paige [6]). In addition, a reduced BDD has no redundant tests [4]. Such a BDD is obtained by repeatedly pruning every internal vertex v that has both outedges leading to the same vertex w, and redirecting all of v's incoming edges to w.

Suppose that the shared BDD had all duplicate terminals and nonterminals eliminated, but did not have any of its redundant tests eliminated. Then it would be easy to treat the deterministic finite automaton combined with its BDD machinery as a single automaton whose states were the union of the BDD nodes and the original automaton states, and whose alphabet were zero and one. If this derived automaton had n states, then it could be minimized in $O(nlogn)$ steps using Hopcroft's algorithm [9]. Unfortunately, such an automaton would be too big.

For our purposes, the space savings due to redundant test removal is of crucial importance. But the important 'skip' states that arise from redundant test removal complicates minimization. Our algorithm combines techniques based on [1] with new methods adapted for use with the shared BDD representation of the transition function. For a finite automaton with n states and a transition function represented by m BDD nodes, the algorithm presented here achieves worst-case running time $O(\max(n, m)n)$.

Terminology. A *partition* $\mathcal{P}$ of a finite set U is a set of disjoint nonempty subsets of U such that the union of these sets is all of U. The elements of $\mathcal{P}$ are called its *blocks.* A *refinement* $\mathcal{Q}$ of $\mathcal{P}$ is a partition of U such that any block of $\mathcal{Q}$ is a subset of a block of $\mathcal{P}$. If $q \in U$, then $[q]_{\mathcal{P}}$ denotes the block of partition $\mathcal{P}$ containing the element q, and when no confusion arises, we drop the subscript.

Let $A = (Q, \mathbb{B}^k, i_0, \delta, F)$ denote a deterministic finite automaton, and let $\mathcal{P}$ be a partition of Q, and $\mathcal{Q}$ a refinement of $\mathcal{P}$. A block B of $\mathcal{Q}$ *respects* the partition $\mathcal{P}$ if for all $q, q' \in B$ and for all $b \in \mathbb{B}^k$, $[\delta(q, b)]_{\mathcal{P}} = [\delta(q', b)]_{\mathcal{P}}$. Thus, δ cannot distinguish between the elements in B relative to the partition $\mathcal{P}$. A partition $\mathcal{Q}$ *respects* $\mathcal{P}$ if every block of $\mathcal{Q}$ respects $\mathcal{P}$. A partition is *stable* if it respects itself. The *coarsest, stable partition $\mathcal{Q}$ respecting* $\mathcal{P}$ is a unique partition such that any other stable partition respecting $\mathcal{P}$ is a refinement of $\mathcal{Q}$.

The refinement algorithm. The minimal automaton A' recognizing $L(A)$ is isomorphic to the automaton defined by the coarsest stable partition $\mathcal{Q}^A$ of Q respecting the partition $\{F, Q \setminus F\}$. The states of A' are $\mathcal{Q}^A$, the transition function δ' is defined by $\delta'([p], b) = [\delta(p, b)]$, the initial state is $[i_0]$, and the set of final states is $F' = \{[f] | f \in F\}$.

Now we are ready to sketch our minimizing algorithm, which works by gradually refining a current partition.

- First split Q into an initial partition $\mathcal{Q} = \{F, Q \setminus F\}$. Note that $\mathcal{Q}^A$ is a refinement of this partition.
- Now let $\mathcal{P}$ be the current partition. We construct the new current partition $\mathcal{Q}$ so that it respects $\mathcal{P}$ while $\mathcal{Q}^A$ remains a refinement of $\mathcal{Q}$.

  For each state q in Q consider the functions $f_q : \mathbb{B}^k \to \mathcal{P}$ defined by $f_q(b) = [\delta(q, b)]_{\mathcal{P}}$ for all q and b. Now let the equivalence relation $\equiv$ be defined as

$q \equiv q' \Leftrightarrow (f_q = f_{q'} \wedge [q]_{\mathcal{P}} = [q']_{\mathcal{P}})$. The new partition $\mathcal{Q}$ then consists of the equivalence classes of $\equiv$. By definition of the f_q's, $\mathcal{Q}$ respects $\mathcal{P}$ and is the coarsest such partition implying the invariant.

We repeat this process until $\mathcal{P} = \mathcal{Q}$.

It can be shown that the final partition $\mathcal{Q}$ is obtained in at most n iterations and equals $\mathcal{Q}^A$. The preceding algorithm is an abstraction of the initial naive algorithm presented in Section 4.13 of [1].

The difficult step in the above algorithm is the splitting according to the functions f_q. However, we can here elegantly take advantage of the shared BDD representation. The idea is to construct a BDD representing the functions f_q for each state. We represent a partition of the states Q, by associating with each state $q \in Q$ a *block id* identifying its block. The BDD for f_q is calculated by performing a unary apply on the collection of shared BDDs, where the value calculated in a leaf is the block id. By a suitable generalization of the standard algorithm, it is possible to carry out these calculations while visiting each node at most once (assuming that hashing takes constant time). Thus the split operation requires time $O(\max(n, m))$. Since we use shared BDDs, we may use the results of the apply operations directly as new block ids.

The Splitting Step Without Hashing. An alternative implementation of the splitting step is possible that achieves the same worst case time bound $O(\max(n, m))$ without hashing. It is instructive to first consider the case in which the shared BDDs are reduced only by eliminating redundant nodes but not by eliminating redundant tests. In this case the BDD may be regarded as an acyclic deterministic automaton D whose states are the BDD nodes, and whose alphabet is zero and one. Consider a partition $\mathcal{P}'$ of the BDD nodes defined by equivalence classes of the following relation. Two BDD leaves are equivalent iff their next states belong to the same block of partition $\mathcal{P}$. All decision nodes of the BDD are equivalent. The coarsest stable partition $\mathcal{Q}'$ that respects $\mathcal{P}'$ for automaton D can be solved in $O(m)$ worst case time by Revuz [13] and Cai and Paige [6], Sec. 3.4. Finding the equivalence classes of states in Q that point to BDD roots belonging to the same block of $\mathcal{Q}'$ (i.e., finding the coarsest partition $\mathcal{Q}$ that respects $\mathcal{P}$) solves the splitting step in the original automaton in time $O(n)$.

In the case of fully reduced BDDs, the splitting step is somewhat harder, and a closer look at the BDD structure is needed. For each decision node v, $v.index$ represents a position in a string of length k such that $v.index < (v.lo).index \wedge v.index < (v.hi).index$. For each BDD leaf v we have $v.index = k$, and let $v.lo = v.hi$ be an automaton state belonging to Q. For each BDD node v we define function $f_v : \mathbb{B}^k \rightarrow \mathcal{P}$ much like the way functions f_q were defined earlier on automaton states. For each nonleaf v, f_v is defined by the rule $f_v(b) = f_{v.lo}(b)$ if $b_{v.index} = 0$; $f_v(b) = f_{v.hi}(b)$ if $b_{v.index} = 1$. For each leaf v, f_v is a constant function that maps every argument into an element (i.e., a block) of partition $\mathcal{P}$.

If $q \in Q$ is an automaton state that points to a BDD root v, then, clearly, $f_q = f_v$. It is also not hard to see that for any two nonleaf BDD nodes v and v', $f_v = f_{v'}$ iff either of the following two conditions hold:

1. $v.index = v'.index \wedge f_{v.hi} = f_{v'.hi} \wedge f_{v.lo} = f_{v'.lo}$, or
2. $f_{v.hi} = f_{v.lo} = f_v \wedge v.hi = v'$.

This leads to the more concrete equivalence relation $\equiv$ on BDD nodes defined as $v \equiv v'$ iff $f_v = f_{v'}$ iff either,

1. $v.index = v'.index = k \wedge [v.lo]_{\mathcal{P}} = [v'.lo]_{\mathcal{P}}$, or
2. $v.index = v'.index < k \wedge v.hi \equiv v'.hi \wedge v.lo \equiv v'.lo$, or
3. $v.index < k \wedge v.lo \equiv v.hi \equiv v'$.

Note that two BDD nodes of different index can be equivalent only by condition (3). Note also, that we can strengthen condition (2) with the additional constraint $v.hi \not\equiv v.lo$ without modifying the equivalence relation. These two observations allow us to construct the equivalence classes inductively using a bottom-up algorithm that processes all BDD nodes of the same index in descending order, proceeding from leaves to roots. The steps are sketched just below.

1. In a linear time pass through all of the BDD nodes, place each node in a bucket according to its index. An array of $k + 1$ buckets can be used for this purpose.
2. Next, distribute the BDD leaves (contained in the bucket associated with index k) into blocks whose nodes all have *lo* successors that belong to the same block of $\mathcal{P}$. This takes time proportional to the number of leaves.
3. For $j = k - 1, ..., 0$ examine each node v with $v.index = j$. Both nodes $v.lo$ and $v.hi$ have already been examined, and have been placed into blocks. Hence, a streamlined form of multiset sequence discrimination [6] can be used to place v either in an old block (according to condition (3)) or a new block (according to condition (2)) for nodes whose children belong pair-wise to the same old block.

The preceding algorithm computes the equivalence classes as the final set of blocks in $O(m)$ time. As before, we can use these equivalence classes to find the coarsest partition $\mathcal{Q}$ that respects $\mathcal{P}$, which solves the splitting step in the original automaton, in time $O(n)$. Thus, the total worst-case time to solve the splitting step is $O(\max(n, m))$ (without hashing).

In an efficient implementation of finite-state automaton minimization, when the splitting algorithm above is is performed repeatedly, we only need to perform the first step of that algorithm (i.e., sorting BDD nodes according to index) once. Thus, the full DFA minimization algorithm runs in worst case time $O(\max(n, m)n)$ without hashing.

BDD Reduction Without Hashing. Sieling and Wegener[15] were the first to compress an arbitrary BDD into fully reduced form. Their result depended on a radix sort, which is closely related to the multiset discrimination technique that we use. However, their algorithm needs to maintain integer representations of BDD nodes, and it utilizes two arrays of size m. We can show how our algorithm just described can be modified to fully reduce an arbitrary BDD in worst case time linear in the number of BDD nodes (without hashing), but with expected auxiliary space k times smaller than Sieling and Wegener's algorithm.

Let $\mathcal{Q}'$ be the partition of BDD nodes produced by the algorithm. The states of the reduced BDD are the blocks in $\mathcal{Q}'$. For each block $B \in \mathcal{Q}'$, $B.index$ is the largest index of any BDD node contained in B. Let v' be any node belonging to B of maximum index. If v' is a BDD leaf, then B is a leaf in the reduced BDD (i.e., $B.index = k$), and $B.lo = B.hi = v'.lo$. Otherwise, $B.lo = [v'.lo]_{\mathcal{Q}'}$ and $B.hi = [v'.hi]_{\mathcal{Q}'}$. The *hi* and *lo* successor blocks can be determined during the multiset sequence discrimination pass when a new block is first created. The index of the first node placed in a newly created block is the index for that block.

What distinguishes our algorithm from that of Sieling and Wegener is that our buckets in steps (2) and (3) are associated with actual BDD nodes (inside the main BDD data structure). Their buckets are associated with components of two auxiliary arrays of size m each. If we replaced each equivalence class by a single witness (as they do) each iteration of step (3), then our auxiliary space would be bounded by the maximum number of BDD nodes that have the same index. If BDD nodes were uniformly distributed among indexes, then this number is m/k, which would give us a k-fold advantage in auxiliary space over their algorithm. We expect a minor constant factor advantage in time as well, because our BDD nodes are represented by their locations instead of by computed integer values, and because we avoid array access in favor of less expensive list and pointer processing.

Work is in progress for exploring the "processing the smaller half" idea found in e.g. [12]. We should mention, however, that the current implementation of the minimization algorithm in practice seems to run faster than the procedures for constructing product and subset automata.

MONA **features.** MONA is enriched by facilities similar to those of programming languages.

Predicates. The user may declare predicates that can later be instantiated. For example, if the predicate P is declared by $P(X, x) = (0 = x \wedge x \in X)$, then P can be instantiated as the formula $P(\complement Y, p \oplus 1)$ with the obvious meaning.

Libraries. MONA supports creation of user-defined libraries of predicates.

Separate translation. MONA automatically stores the automaton for a translated predicate. If there are n free variables, then there may be up to $n!$ different automata corresponding to different orderings of variables in the BDD representation.

To be done. In the current implementation, variables are ordered in their BDDs according to the level of syntactic nesting in the formula; i.e. innermost variables receive the highest index. This strategy is obviously often far from optimal and we are working on implementing heuristics to improve variable ordering. Another orthogonal optimization strategy is to reorder the product constructions by heuristics. In both cases, however, it is not hard to see that finding optimal orderings is NP-complete.

Acknowledgements. We are thankful to Vladimiro Sassone for comments on an earlier version, and to Andreas Potthoff for his advice based on the M2L implementation at the University of Kiel.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] F. Balarin and A.L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer Aided Verification, CAV '93, LNCS 697*, pages 29–40, 1993.

[3] D. Basin and N. Klarlund. Hardware verification using monadic second-order logic. Technical Report RS-96-7, BRICS, 1995.

[4] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing surveys*, 24(3):293–318, September 1992.

[5] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.

[6] J. Cai and R. Paige. Using multiset discrimination to solve language processing problems without hashing. *to appear Theoretical Computer Science*, 1994. also, U. of Copenhagen Tech. Report, DIKU-TR Num. D-209, 94/16, URL ftp://ftp.diku.dk/diku/semantics/papers/D-209.ps.Z.

[7] M-M Corsini and A. Rauzy. Symbolic model checking and constraint logic programming: a cross-fertilisation. In *5th. Europ. Symp. on Programming, LNCS 788*, pages 180–194, 1994.

[8] A. Gupta and A.L. Fisher. Parametric circuit representation using inductive boolean functions. In *Computer Aided Verification, CAV '93, LNCS 697*, pages 15–28, 1993.

[9] J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and Paz A., editors, *Theory of machines and computations*, pages 189–196. Academic Press, 1971.

[10] J. Jensen, M. Jrgensen, and N. Klarlund. Monadic second-order logic for parameterized verification. Technical report, BRICS Report Series RS-94-10, Department of Computer Science, University of Aarhus, 1994.

[11] B. Kurshan and K. MacMillan. A structural induction theorem for processes. In *Proc. Eigth Symp. Princ. of Distributed Computing*, pages 239–247, 1989.

[12] R. Paige and R. Tarjan. Three efficient algorithms based on partition refinement. *SIAM Journal of Computing*, 16(6), 1987.

[13] D. Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992.

[14] J-K. Rho and F. Somenzi. Automatic generation of network invariants for the verification of iterative sequential systems. In *Computer Aided Verification, CAV '93, LNCS 697*, pages 123–137, 1993.

[15] D. Sieling and I. Wegener. Reduction of obdds in linear time. *IPL*, 48:139–144, 1993.

[16] M. Steinmann. Übersetzung von logischen Ausdrücken in Baumautomaten: Entwicklung eines Verfahrens und seine Implementierung. Unpublished, 1993.

[17] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.

Computing Small Nondeterministic Finite Automata

Oliver Matz, Andreas Potthoff
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
D-24098 Kiel
e-mail: `oma@informatik.uni-kiel.de`

Abstract

We study the minimization problem for nondeterministic finite automata. Two approaches are discussed, based on the construction of two versions of "canonical" automata (for a given regular language), in which minimal automata occur as subautomata. We introduce a heuristic for this search, which has been implemented in the program AMoRE.

Introduction

Minimization of nondeterministic finite automata ("NFA") is more difficult than that of deterministic finite automata. The problem is PSPACE-complete (except for the case of one-letter-alphabets) [6], whereas deterministic finite automata can be minimized in time $O(n \cdot \log n)$ [4]. Moreover, there is in general not a unique minimal NFA recognizing a given regular language. Procedures for minimization of nondeterministic finite automata are investigated in several papers, e.g. Kameda and Weiner [10] (see also Indermark [7]) and Kim [11]. Further references are given in Brauer's monograph [2].
The purpose of this note is twofold. First, we describe a general technique of constructing minimal NFA which is implicit in older papers like [10] and has been suggested in a different version in a recent contribution of Arnold, Dicky and Nivat [1]. The idea is to construct a nondeterministic "canonical automaton" in which any NFA recognizing the considered language occurs (via a homorphism) as a subautomaton. This means that the construction of the minimal NFA can be done in two steps: construction of the canonical automaton and search for a minimal subautomaton therein which accepts the same language. We compare two versions of "canonical automata" and develop a method of computing one of these versions (called "fundamental automaton").
The second issue is the search of a minimal NFA within a given (canonical) automaton. We outline a heuristic which improves the one of [11]. It yields an algorithm with worst-case-behaviour time $\mathcal{O}(2^{(n^2)})$, where n is the number of states of the minimal deterministic automaton of the considered language. The algorithm is practical in the sense that in many non-trivial examples the actual implementation works in acceptable time. However, we show that the heuristic fails in some cases to produce

a minimal NFA. (This is true for both considered versions of canonical NFA.) It is open how to sharpen the heuristic in order to get an algorithm which always outputs a proper minimal NFA and is as "practical" as our heuristic.
In the first section basics and terminology on finite automata are recalled (powerset construction, minimization of deterministic finite automata). In the second section we construct the "fundamental automaton" $\mathcal{F}$ of a regular language L and show that a minimal NFA accepting L can be found as a subautomaton of $\mathcal{F}$. Then we give the announced sufficient (but not necessary) criterion for subautomata of $\mathcal{F}$ that accept L; the heuristic is then simply to find a subautomaton of $\mathcal{F}$ that satisfies this criterion and has a minimal number of states. Section 3 offers one of the rare examples where this heuristic fails to yield a minimal automaton.
Section 4 outlines an implementation for the described strategy.
In the last section, we give a different definition for the fundamental automaton in order to compare it to the "canonical automaton" as introduced in [1] (which contains as well any minimal NFA as subautomaton). We show that the fundamental and the canonical automaton may be different, and transfer the criterion mentioned above to the canonical automaton.
An implementation (written in C) of the heuristic, based on the fundamental automaton, is part of the AMoRE program (computing Automata, MOnoids, and Regular Expressions), which is available via FTP (ftp.informatik.uni-kiel.de:pub/kiel/amore).
We thank Wolfgang Thomas for helpful comments on previous versions of the paper.

1 Preliminaries

We use the standard notation of set theory and formal language theory. For a set P of sets we write $\bigcap P$ for $\bigcap_{p \in P} p$. We denote by $|Q|$ the number of elements of Q. $\emptyset$ is the empty set and 2^Q is the set of all subsets of Q. $\biguplus$ denotes a union of disjoint sets. By Σ we indicate a finite alphabet, by $a, b, c, \ldots$ letters and by $u, v, \ldots$ words. The empty word is written ϵ.
A *nondeterministic finite automaton (NFA)* is of the form $\mathcal{A} = (Q, \Sigma, I, \delta, F)$ with a finite non-empty set of states Q, a finite non-empty alphabet Σ, a non-empty set of initial states $I \subseteq Q$, a transition function $\delta : Q \times \Sigma \longrightarrow 2^Q$ and a set of final states $F \subseteq Q$. If $|I| = 1$ and $\forall q \in Q \quad \forall a \in \Sigma \quad |\delta(q, a)| = 1$, then $\mathcal{A}$ is a *deterministic finite automaton (DFA)*. A triple $(q, a, q') \in Q \times \Sigma \times Q$ with $q' \in \delta(q, a)$ is called a *transition*.
The transition function $\delta : Q \times \Sigma \longrightarrow 2^Q$ can be extended in a natural way to a function from $2^Q \times \Sigma^*$ to 2^Q, usually also denoted by δ. We write $p \xrightarrow[\mathcal{A}]{w} q$ to indicate that $q \in \delta(p, w)$ and say that there exists a *path* from p to q labelled w.
If the components of an NFA $\mathcal{A}$ are not listed explicitly, we will denote by $Q_{\mathcal{A}}$, $I_{\mathcal{A}}$, $F_{\mathcal{A}}$ and $\delta_{\mathcal{A}}$ its set of states (initial states, final states resp.) and its transition function. We will drop the subscripts only if the NFA is clear from the context.
A word $u \in \Sigma^*$ is *accepted by* $\mathcal{A}$, if $\delta(I, u) \cap F \neq \emptyset$. The *language accepted by* $\mathcal{A}$ is $L(\mathcal{A}) = \{ u \in \Sigma^* \mid \delta(I, u) \cap F \neq \emptyset \}$. Two NFA $\mathcal{A}$ and $\mathcal{B}$ are called *equivalent* if $L(\mathcal{A}) = L(\mathcal{B})$. An NFA $\mathcal{A}$ is called *minimal* if all NFA that are equivalent to $\mathcal{A}$ have at least as many states as $\mathcal{A}$.
We assign two languages to every state of an NFA.

Definition 1.1 *([10],[5])* Let $\mathcal{A}$ be an NFA and $q \in Q$. Then the *pre-language of* q and the *post-language of* q are given by $pre(\mathcal{A}, q) = L((Q_{\mathcal{A}}, \Sigma, I_{\mathcal{A}}, \delta_{\mathcal{A}}, \{q\}))$ and by $post(\mathcal{A}, q) = L((Q_{\mathcal{A}}, \Sigma, \{q\}, \delta_{\mathcal{A}}, F_{\mathcal{A}}))$, respectively. q is *reachable* if $pre(\mathcal{A}, q) \neq \emptyset$ and q is *productive* if q is reachable and $post(\mathcal{A}, q) \neq \emptyset$. Two states $p, q \in Q$ are *equivalent*, written $p \sim q$, if $post(\mathcal{A}, p) = post(\mathcal{A}, q)$. $[q] = \{\, p \mid p \sim q \,\}$ denotes the equivalence class to which q belongs.

In [1], pre- and post-language are called *history* and *prophecy*.
In the sequel all NFA have productive states only.
Now we introduce three classical operations on automata and analyze pre- and post-languages of the resulting automata.

Definition 1.2 *([10])* Let $\mathcal{A}$ be an NFA. Its *reversed NFA* is $\overline{\mathcal{A}} = (Q, \Sigma, F, \overline{\delta}, I)$ with $\forall a \in \Sigma \;\; \forall p, q \in Q : \; p \in \overline{\delta}(q, a) \Longleftrightarrow q \in \delta(p, a)$.
Given a word $u = a_1 \ldots a_n$ the word $\overline{u} = a_n \ldots a_1$ is its *reversed word*. Given a language L the language $\overline{L} = \{\, \overline{u} \mid u \in L \,\}$ is its *reversed language*.

Obviously $L(\overline{\mathcal{A}}) = \bigcup_{q \in F} post(\overline{\mathcal{A}}, q) = \overline{\bigcup_{q \in F} pre(\mathcal{A}, q)} = \overline{L(\mathcal{A})}$ for all NFA $\mathcal{A}$. Furthermore $\mathcal{A}$ is a minimal NFA accepting L iff $\overline{\mathcal{A}}$ is a minimal NFA accepting $\overline{L}$.
The next operation is the wellknown powerset construction.

Definition 1.3 *([12])* Let $\mathcal{A}$ be an NFA. The *subset automaton for* $\mathcal{A}$, denoted $D(\mathcal{A})$, is the DFA $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, I_{\mathcal{B}}, \delta_{\mathcal{B}}, F_{\mathcal{B}})$, where $Q_{\mathcal{B}} = \{\, \delta_{\mathcal{A}}(I_{\mathcal{A}}, u) \mid u \in \Sigma^* \,\} \subseteq 2^Q$, $I_{\mathcal{B}} = \{\, I_{\mathcal{A}} \,\}$, $F_{\mathcal{B}} = \{\, P \in Q_{\mathcal{B}} \mid P \cap F_{\mathcal{A}} \neq \emptyset \,\}$ and with $\delta_{\mathcal{B}}(P, a) = \delta_{\mathcal{A}}(P, a)$ for all $P \in Q_{\mathcal{B}}$ and for all $a \in \Sigma$.

Obviously $L(\mathcal{A}) = \bigcup_{q \in I} post(\mathcal{A}, q) = post(\mathcal{B}, I_{\mathcal{B}}) = L(D(\mathcal{A}))$.
Next we give the constructive definition of the minimal DFA for a given NFA.

Definition and Theorem 1.4 *([5],[10])*
Let $\mathcal{A}$ be an NFA and $\mathcal{B} = D(\mathcal{A})$. The *minimal DFA* $\mathcal{D} = (Q_{\mathcal{D}}, \Sigma, I_{\mathcal{D}}, \delta_{\mathcal{D}}, F_{\mathcal{D}})$ *for* $\mathcal{A}$, denoted $M(\mathcal{B})$ or $MD(\mathcal{A})$, can be constructed the following way: $Q_{\mathcal{D}} = \{[\,P\,] | P \in Q_{\mathcal{B}}\}$, $I_{\mathcal{D}} = [I_{\mathcal{B}}]$, $F_{\mathcal{D}} = \{[\,P\,] | P \in F_{\mathcal{B}}\}$ and for all states P of $Q_{\mathcal{B}}$ and for all $a \in \Sigma$, $\delta_{\mathcal{D}}([P], a) = [\delta_{\mathcal{B}}(P, a)]$.
Let q be a state of $\mathcal{A}$ and $[P]$ be a state of $MD(\mathcal{A})$. We write $q < [P]$ as an abbreviation for: $\exists P' \in [P] \; q \in P'$. We give some relations that hold between post- and pre-languages in $\mathcal{A}$ and $\mathcal{D}$.

$$q \in F \;\wedge\; q < [P] \quad \Rightarrow \quad [P] \in F_{\mathcal{D}} \tag{1}$$

$$q \in I \quad \Rightarrow \quad q < [P] \text{ where [P] is the initial state of } \mathcal{D} \tag{2}$$

$$p \xrightarrow[\mathcal{A}]{a} q \;\wedge\; p < [P] \quad \Rightarrow \quad q < \delta_{\mathcal{D}}([P], a) \tag{3}$$

Remark 1.5
For every regular language L, there is a unique (up to isomorphsm) minimal DFA $\mathcal{D}$ with $L(\mathcal{D}) = L$, which we call "the" minimal DFA for a language L.
The analogue is not true for minimal NFA since there are regular languages for which several non-isomorphic minimal NFA accepting these languages exist.
All states of a minimal DFA are reachable and at most one state can be non-productive. If it exists, this state is called the sink state or — because its post-language is the empty set — the empty state. Pre-languages of different states of a DFA are disjoint and post-languages of different states in a minimal DFA are different.
The minimal DFA $MD(\mathcal{A})$ for an NFA $\mathcal{A}$ with n states can have up to 2^n states.

The next theorem is interesting because it offers a way to compute the minimal DFA for a given language without computing the equivalence classes of states (as it is suggested by 1.4).

Theorem 1.6 (Brzozowski [3])
Let L be a regular language and $\mathcal{D}$ an arbitrary DFA accepting L. Then $\mathcal{E} := D(\overline{\mathcal{D}})$ is the minimal DFA accepting $\overline{L}$.

Definition 1.7 Let $\mathcal{A} = (Q, \Sigma, I, \delta, F)$ and $\mathcal{B}$ be nondeterministic automata. A mapping $h : Q \to Q_{\mathcal{B}}$ is called a *morphism* if $h(I) \subseteq I_{\mathcal{B}}$, $h(F) \subseteq F_{\mathcal{B}}$ and $h(\delta(q, a)) \subseteq \delta_{\mathcal{B}}(h(q), a)$ for all $q \in Q$ and $a \in \Sigma$.
$\mathcal{A}$ is called a *subautomaton* of $\mathcal{B}$ if $Q \subseteq Q_{\mathcal{B}}$, $F = F_{\mathcal{B}} \cap Q$, $I = I_{\mathcal{B}} \cap Q$ and $\delta(q, a) \subseteq \delta_{\mathcal{B}}(q, a) \cap Q$ for all $q \in Q$ and $a \in \Sigma$. If $\delta(q, a) \subseteq \delta_{\mathcal{B}}(q, a) \cap Q$, we call $\mathcal{A}$ the *subautomaton of* $\mathcal{B}$ *induced by* Q.
If h is a morphism from $\mathcal{A}$ to $\mathcal{B}$ then we denote by $h(\mathcal{A})$ the subautomaton of $\mathcal{B}$ induced by the set $h(Q)$ of states.

Remark 1.8 Let $\mathcal{A}$ and $\mathcal{B}$ nondeterministic automata and h a morphism from $\mathcal{A}$ to $\mathcal{B}$. Then $L(\mathcal{A}) \subseteq L(h(\mathcal{A})) \subseteq L(\mathcal{B})$.

Proof We have $p \underset{\mathcal{A}}{\overset{w}{\to}} q \Rightarrow h(p) \underset{h(\mathcal{A})}{\overset{w}{\to}} h(q) \Rightarrow h(p) \underset{\mathcal{B}}{\overset{w}{\to}} h(q)$ and $p \in F\,(I) \Rightarrow h(p) \in F_{\mathcal{B}}\,(I_{\mathcal{B}})$. Thus an accepting path in $\mathcal{A}$ is mapped to an accepting path in $h(\mathcal{A})$ which is an accepting path of $\mathcal{B}$ by definition. □

2 The Fundamental Automaton

In this section we construct for every regular language L an NFA $\mathcal{F}$, called the fundamental automaton of L, such that for every NFA $\mathcal{A}$ accepting L there exists a morphism from $\mathcal{A}$ in $\mathcal{F}$. We will use this fact to show that a minimal automaton accepting L can be found as a subautomaton of $\mathcal{F}$.
In order to simplify the notation we fix the language L. Let $\mathcal{A} = (Q, \Sigma, I, \delta, F)$ be an arbitrary NFA accepting L, $\mathcal{D} = (Q_{\mathcal{D}}, \Sigma, I_{\mathcal{D}}, \delta_{\mathcal{D}}, F_{\mathcal{D}})$ the minimal DFA accepting L and $\mathcal{E} = (Q_{\mathcal{E}}, \Sigma, I_{\mathcal{E}}, \delta_{\mathcal{E}}, F_{\mathcal{E}}) = D(\overline{\mathcal{D}})$. (Because of 1.6, $\mathcal{E}$ is the minimal DFA accepting $\overline{L}$.) If $\mathcal{E}$ has a sink state then this state is not productive in the reversed automaton

$\overline{\mathcal{E}} = (Q_{\overline{\mathcal{E}}}, \Sigma, I_{\overline{\mathcal{E}}}, \delta_{\overline{\mathcal{E}}}, F_{\overline{\mathcal{E}}})$ and we assume that this state and all transitions from this state are deleted in $\overline{\mathcal{E}}$.
$\overline{\mathcal{E}}$ has several important properties described in the following remark.

Remark 2.1 $\overline{\mathcal{E}}$ is a reversed automaton to a deterministic finite automaton. Therefore there exists for all words in L a unique accepting path in $\overline{\mathcal{E}}$, and deletion of any transition in $\delta_{\overline{\mathcal{E}}}$ yields an automaton which accepts a proper subset of L. Moreover, a post-language of an arbitrary NFA accepting L is a subset of a disjoint union of post-languages of $\overline{\mathcal{E}}$. $\mathcal{E}$ is isomorphic to $MD(\overline{\mathcal{A}})$ and thus we can associate to every state r of $\mathcal{E}$ an equivalence class of sets of states of $\mathcal{A}$ (cf. Definition 1.4). For every state q of $\mathcal{A}$ we denote by $\overset{\bullet}{q}$ the set of states r of $\mathcal{E}$ with $q < r$.

Now we arrive at the definition of the fundamental automaton, given $\mathcal{E} = D(\overline{\mathcal{D}})$.

Definition 2.2 (The Fundamental Automaton)
The *fundamental automaton* $\mathcal{F} = (Q_{\mathcal{F}}, \Sigma, I_{\mathcal{F}}, \delta_{\mathcal{F}}, F_{\mathcal{F}})$ is defined as follows:

$$Q_{\mathcal{F}} = \{\, P \subseteq Q_{\overline{\mathcal{E}}} \mid \bigcap P \neq \emptyset\}$$
$$I_{\mathcal{F}} = \{\, P \in Q_{\mathcal{F}} \mid P \subseteq I_{\overline{\mathcal{E}}}\,\}$$
$$F_{\mathcal{F}} = \{\, P \in Q_{\mathcal{F}} \mid P \cap F_{\overline{\mathcal{E}}} \neq \emptyset\,\}$$
$$\delta_{\mathcal{F}}(P,a) = \{\, P' \in Q_{\mathcal{F}} \mid P' \subseteq \delta_{\overline{\mathcal{E}}}(P,a)\,\} \text{ for all } P \text{ in } Q_{\mathcal{F}} \text{ and } a \text{ in } \Sigma.$$

We proceed in three steps. First, we show that the fundamental automaton accepts L. Then we verify that for each automaton $\mathcal{A}$ accepting L there exists a morphism from $\mathcal{A}$ into the fundamental automaton of L. This proves that a minimal automaton accepting L can be found as a subautomaton of $\mathcal{F}$. We conclude with a condition that describes how to find certain subautomata of $\mathcal{F}$ that accept L.

Lemma 2.3 The fundamental automaton $\mathcal{F}$ is equivalent to $\overline{\mathcal{E}}$.

Proof $L(\mathcal{F}) \supseteq L(\overline{\mathcal{E}})$ because $\overline{\mathcal{E}}$ is a subautomaton of $\mathcal{F}$ (take all subsets of $Q_{\mathcal{F}}$ of size 1). The following fact on the transiton function of $\mathcal{F}$ can be shown easily by an induction on the length of words w:

$$(4) \qquad \forall P, P' \in Q_{\mathcal{F}}\ \forall w \in \Sigma^*\ P \xrightarrow[\mathcal{F}]{w} P' \ \Rightarrow\ \forall p' \in P'\ \exists p \in P\ p \xrightarrow[\overline{\mathcal{E}}]{w} p'$$

Then the definition of initial and final states of $\mathcal{F}$ yields: if $P \xrightarrow[\mathcal{F}]{w} P'$ is an accepting path in $\mathcal{F}$ then there exists a state $p' \in P' \cap F_{\overline{\mathcal{E}}}$ and a state $p \in P \subseteq I_{\overline{\mathcal{E}}}$ with $p \xrightarrow[\overline{\mathcal{E}}]{w} p'$. Thus $L(\mathcal{F}) \subseteq L(\overline{\mathcal{E}})$. □

Lemma 2.4
Let $\mathcal{A}$ be an NFA accepting L and $\mathcal{F}$ the fundamental automaton of L. The mapping $q \mapsto \overset{\bullet}{q}$ is a morphism from $\mathcal{A}$ to $\mathcal{F}$.

Proof

Because of Theorem 1.6, we may regard $\mathcal{E}$ as $D(\overline{D(\mathcal{A})})$. The states of $\mathcal{E}$ are thus sets of states of $\overline{D(\mathcal{A})}$. For a state q' of $\overline{D(\mathcal{A})}$) and a state r of $\mathcal{E}$, we have then $q' < r$ iff $q' \in r$.

We will begin by showing that $q \mapsto \overset{\bullet}{q}$ is a mapping from $Q_{\mathcal{A}}$ in $Q_{\mathcal{F}}$.

Let q be a state of $\mathcal{A}$ and $r \in \overset{\bullet}{q}$. Since all states of $\mathcal{A}$ are supposed to be productive, there is a state q' of $D(\mathcal{A})$ such that $q \in q'$. Since $q < r$ and $\mathcal{E}$ is isomorphic to $MD(\overline{\mathcal{A}})$, there is a state R of $D(\overline{\mathcal{A}})$ such that $q \in R$ and $post(D(\overline{\mathcal{A}}), R) = post(\mathcal{E}, r)$. (Each state of $\mathcal{E}$ corresponds to a non-empty set of states of $D(\overline{\mathcal{A}})$ whose elements have the same post-language.)

Now we have

$post(\overline{D(\mathcal{A})}, q') = pre(D(\mathcal{A}), q') \subseteq pre(\mathcal{A}, q) = post(\overline{\mathcal{A}}, q) \subseteq \bigcup_{p \in R} post(\overline{\mathcal{A}}, p)$
$= post(D(\overline{\mathcal{A}}), R) = post(\mathcal{E}, r) = post(D(\overline{D(\mathcal{A})}), r) = \bigcup_{p' \in r} post(\overline{D(\mathcal{A})}, p')$

Since different pre-languages of a DFA are disjoint, so are different post-language of $\overline{D(\mathcal{A})}$. Thus we may conclude that $q' \in r$, i.e. $r \in \overset{\bullet}{q'}$. Since r was chosen arbitrarily from $\overset{\bullet}{q}$, we have shown $\overset{\bullet}{q} \subseteq \overset{\bullet}{q'}$.

We have $\overset{\bullet}{q'} = \{r \in Q_{\mathcal{E}} \mid q' \in r\}$, thus $q' \in \bigcap \overset{\bullet}{q'} \subseteq \bigcap \overset{\bullet}{q}$, showing that $\overset{\bullet}{q}$ is a state of $\mathcal{F}$. So we have shown that $q \mapsto \overset{\bullet}{q}$ is a mapping into $\mathcal{F}$.

To show that it is a morphism, we have to verify $q \in F\,(I) \Rightarrow \overset{\bullet}{q} \in F_{\mathcal{F}}\,(I_{\mathcal{F}})$ and $p \xrightarrow[\mathcal{A}]{a} q \Rightarrow \overset{\bullet}{p} \xrightarrow[\mathcal{F}]{a} \overset{\bullet}{q}$.

$q \in I_{\mathcal{A}}$	$q \in F_{\mathcal{A}}$	$p \xrightarrow[\mathcal{A}]{a} q$	
	$\Downarrow$		Definition of the reversed automaton
$q \in F_{\overline{\mathcal{A}}}$	$q \in I_{\overline{\mathcal{A}}}$	$q \xrightarrow[\overline{\mathcal{A}}]{a} p$	
	$\Downarrow$		Equations (1,2,3)
$\overset{\bullet}{q} \subseteq F_{\mathcal{E}}$	$\overset{\bullet}{q} \cap I_{\mathcal{E}} \neq \emptyset$	$\overset{\bullet}{p} \supseteq \delta_{\mathcal{E}}(\overset{\bullet}{q}, a)$	
	$\Downarrow$		Definition of the reversed automaton
$\overset{\bullet}{q} \subseteq I_{\overline{\mathcal{E}}}$	$\overset{\bullet}{q} \cap F_{\overline{\mathcal{E}}} \neq \emptyset$	$\overset{\bullet}{q} \subseteq \delta_{\overline{\mathcal{E}}}(\overset{\bullet}{p}, a)$	
	$\Downarrow$		Definition of the fundamental automaton
$\overset{\bullet}{q} \in I_{\mathcal{F}}$	$\overset{\bullet}{q} \in F_{\mathcal{F}}$	$\overset{\bullet}{p} \xrightarrow[\mathcal{F}]{a} \overset{\bullet}{q}$	

□

Theorem 2.5 A minimal automaton accepting L can be found as a subautomaton of $\mathcal{F}$.

Proof Let $\mathcal{A}$ be a minimal NFA accepting L and let $h : q \mapsto \overset{\bullet}{q}$ be the morphism of Lemma 2.4. The number of states of $h(\mathcal{A})$ is less or equal to the number of states of

$\mathcal{A}$. Furthermore we have: $L = L(\mathcal{A}) \subseteq L(h(\mathcal{A})) \subseteq L(\mathcal{F}) = L$. Thus $L(h(\mathcal{A})) = L$ and $h(\mathcal{A})$ is a minimal NFA accepting L and a subautomaton of $\mathcal{F}$. $\square$

With the next lemma we give a criterion for equivalent subautomata of $\mathcal{F}$ that improves the one given by Kim [11] and was suggested by Kahlert [9]. Kim claimed that his criterion was necessary, which is not true in general.
We start with an auxiliary definition. Let Q be a set, $\mathcal{P} \subseteq 2^Q$ and $R \subset Q$. We say $\mathcal{P}$ *covers R with subsets* iff $R = \bigcup\{P \in \mathcal{P} \mid P \subseteq R\}$.

Lemma 2.6
Let $\mathcal{F}_\mathcal{P}$ be a subautomaton of $\mathcal{F}$ induced by the set $\mathcal{P} \subseteq Q_\mathcal{F}$ of states. Assume

(i) $\mathcal{P}$ covers $I_{\overline{\mathcal{E}}}$ with subsets and

(ii) $\mathcal{P}$ covers $\delta_{\overline{\mathcal{E}}}(P, a)$ with subsets for all $P \in \mathcal{P}$, $a \in \Sigma$.

Then $\mathcal{F}_\mathcal{P}$ is equivalent to $\mathcal{F}$.

Proof Let $\mathcal{P}$ fulfill conditions (i) and (ii) and let $p_0 \underset{\overline{\mathcal{E}}}{\overset{a_1}{\to}} \dots \underset{\overline{\mathcal{E}}}{\overset{a_n}{\to}} p_n$ be an accepting path in $\overline{\mathcal{E}}$. We show that there exists an accepting path $P_0 \underset{\mathcal{F}_\mathcal{P}}{\overset{a_1}{\to}} \dots \underset{\mathcal{F}_\mathcal{P}}{\overset{a_n}{\to}} P_n$ in $\mathcal{F}_\mathcal{P}$ with $p_i \in P_i$. By $p_0 \in I_{\overline{\mathcal{E}}}$ and condition (i) there is a set $P_0 \in \mathcal{P}$ with $P_0 \subseteq I_{\overline{\mathcal{E}}}$ and $p_0 \in P_0$. Furthermore P_0 is initial in $\mathcal{F}_\mathcal{P}$. We find $P_1 \dots P_n$ by induction in the following way: if $p_i \in P_i$ then $p_{i+1} \in \delta_{\overline{\mathcal{E}}}(P_i, a_{i+1})$; by condition (ii) there exists $P_{i+1} \subseteq \delta_{\overline{\mathcal{E}}}(P_i, a_{i+1})$, $P_{i+1} \in \mathcal{P}$ with $p_{i+1} \in P_{i+1}$. By definition of $\mathcal{F}$ we get $P_i \underset{\mathcal{F}}{\overset{a_{i+1}}{\to}} P_{i+1}$. P_n is final in $\mathcal{F}$ because $p_n \in P_n$ and p_n is final in $\overline{\mathcal{E}}$. Thus $L(\overline{\mathcal{E}}) \subseteq L(\mathcal{F}_\mathcal{P})$. $\square$

3 An Example

In this section we present a simple example that shows that the criterion given in 2.6 is not necessary for subautomata of the fundamental automaton which are equivalent to the given NFA. Thus it does not yield an algorithm for determining a minimal NFA in general.
Consider the NFA $\mathcal{A}$ and its reversed NFA $\overline{\mathcal{A}}$ given by the following transition tables. Initial and final states are marked with “i” or “f”, respectively.

$\mathcal{A}$

	a	b
f0	0, 2	
1	3	2, 3
i2	1	0, 1
3	3	1

$\overline{\mathcal{A}}$

	a	b
i0	0	2
1	2	2, 3
f2	0	1
3	1, 3	1

The following tables show the minimal DFA $\mathcal{E}$ equivalent to $\overline{\mathcal{A}}$, the reverse of this DFA and the mapping $q \mapsto \overset{\bullet}{q}$ from $Q_\mathcal{A}$ into $2^{Q_\mathcal{E}}$. The leftmost column of the first table contains for each state of $\mathcal{E}$ the corresponding states of $\overline{\mathcal{A}}$.

		a	b
$\{0\}$	ie_0	e_0	e_1
$\{2\}$	fe_1	e_0	e_2
$\{1\}$	e_2	e_1	e_3
$\{2,3\}$	fe_3	e_4	e_2
$\{0,1,3\}$	e_4	e_5	e_5
$\{0,1,2,3\},\{1,2,3\}$	fe_5	e_5	e_5

	a	b
fe_0	e_0, e_1	
ie_1	e_2	e_0
e_2		e_1, e_3
ie_3		e_2
e_4	e_3	
ie_5	e_4, e_5	e_4, e_5

$$
\begin{aligned}
h(0) &= \{e_0, e_4, e_5\} \\
h(1) &= \{e_2, e_4, e_5\} \\
h(2) &= \{e_1, e_3, e_5\} \\
h(3) &= \{e_3, e_4, e_5\}
\end{aligned}
$$

The states of the fundamental automaton are certain subsets of $Q_{\mathcal{E}}$. In fact, a look at the correspondence between $\mathcal{E}$ and $D(\overline{D(\mathcal{A})})$, which is not presented here, would show that the fundamental automaton contains actually *all* subsets of $Q_{\mathcal{E}}$. That is why we do not present it here. But we show the transitition table of the image of $\mathcal{A}$ under the morphism h.

	a	b
f$\{e_0, e_4, e_5\}$	$\{e_0, e_1, e_3, e_4, e_5\}$	$\{e_4, e_5\}$
$\{e_2, e_4, e_5\}$	$\{e_3, e_4, e_5\}$	$\{e_1, e_3, e_4, e_5\}$
i$\{e_1, e_3, e_5\}$	$\{e_2, e_4, e_5\}$	$\{e_0, e_2, e_4, e_5\}$
$\{e_3, e_4, e_5\}$	$\{e_3, e_4, e_5\}$	$\{e_2, e_4, e_5\}$

This transition table has to be interpreted like this: if $Q \subseteq Q_{\mathcal{E}}$ is in the line corresponding to $P \subseteq Q_{\mathcal{E}}$ and letter a, then in the fundamental automaton there is a transition from P with a to each subset of Q. So the subautomaton of the fundamental automaton induced by the image of h is:

	a	b
f$h(0)$	$h(0), h(2), h(3)$	
$h(1)$	$h(3)$	$h(2), h(3)$
i$h(2)$	$h(1)$	$h(0), h(1)$
$h(3)$	$h(3)$	$h(1)$

The automaton is (up to isomorphism) the same as $\mathcal{A}$, except for one superflous transition.
The chosen set of states $\mathcal{P} = \{h(0), h(1), h(2), h(3)\}$ of the fundamental automaton covers indeed the set of initial states of $\overline{\mathcal{E}}$ with subsets, so the first criterion of 2.6 holds. But the transition table shows that the set $\delta_{\overline{\mathcal{E}}}(\{e_0, e_4, e_5\}, b) = \{e_4, e_5\}$ is not covered with subsets; so the second property does not hold. Thus $h(\mathcal{A})$ is a subautomaton of the fundamental automaton that accepts L, but does not fulfill the conditions (i) and (ii) of Lemma 2.6.
In fact, there is no set of states with the property of Lemma 2.6 that has less than 5 states, so for this example language the suggested heuristic fails to find a minimal NFA.

4 Some Comments on the Practical Computation

In this section we outline an algorithm to compute a small NFA. This algorithm is based on the results of Section 2.
An implementation of this strategy (with some improvements not described here) is part of the AMoRE-System, available via Anonymous FTP.

Main Idea

The main idea how to compute a small NFA for a given regular language L is to proceed in four steps:

1. Compute a minimal DFA $\mathcal{E}$ for $\overline{L}$.
2. Determine the *allowed* sets, i.e. those subsets of the state set of $\mathcal{E}$ that are states of the fundamental automaton.
3. Search for a selection $\mathcal{P}$ of allowed sets that satisfies the cover/closure property (i), (ii) of Lemma 2.6 and is minimal with that property.
4. Compute the subautomaton of the fundamental automaton induced by the selection $\mathcal{P}$.

The most important point is, of course, the step 3. For this step we will give (in Figure 1) a recursive procedure `cover` that explores in a backtracking strategy all reasonable extensions of a given selection to a selection satisfying the cover/closure property. `cover` receives three arguments called `allowed`, `selected`, `remain` for subsets of $Q_{\mathcal{E}}$. The intuitive meaning is the following: `allowed` contains all allowed sets that must be considered for possible extensions of the set `selected`. `selected` contains the subsets of $Q_{\mathcal{E}}$-states that have been selected so far. `remain` contains all subsets of $Q_{\mathcal{E}}$-states for which we will have to make sure (possibly by further selections) that `selection` covers them by subsets.
More precisely, the pre- and postcondition of `cover` are given by:

Precondition: $I_{\overline{\mathcal{E}}} \notin$ `remain` $\Rightarrow$ (`selected` covers $I_{\overline{\mathcal{E}}}$ by subsets) and
$\forall R \in$ `selected` $\forall a \in \Sigma : \delta_{\overline{\mathcal{E}}}(R, a) \notin$ `remain` $\Rightarrow$ (`selected` covers $\delta_{\overline{\mathcal{E}}}(R, a)$ by subsets)
Postcondition: If there is a $\mathcal{P} \supseteq$ `selected` with $\mathcal{P} \setminus$`selected` $\subseteq$ `allowed` such that $\mathcal{P}$ fulfills the cover/closure property of Lemma 2.6, then the global variable `best_solution` contains such a $\mathcal{P}$ of minimal size, otherwise `best_solution` remains unchanged. In any case, `best_known = |best_solution|`.

With this procedure `cover`, the algorithm for the computation of a small NFA looks like this:

1. Compute the minimal DFA $\mathcal{D}$ for L.
2. Apply the powerset constuction to $\overline{\mathcal{D}}$ and obtain $\mathcal{E}$.
3. Let `best_known` $= |Q_{\mathcal{E}}| + 1$.
4. `cover`$(\{P \subseteq Q_{\mathcal{E}} \mid \bigcap P \neq \emptyset\}, \emptyset, \{I_{\overline{\mathcal{E}}}\})$
5. Define the output automaton $\mathcal{A} = (Q, \Sigma, I, \delta, F)$ by

$Q =$ `best_solution`,
$I = \{P \in Q \mid P \subseteq I_{\overline{\mathcal{E}}}\}$,
$F = \{P \in Q \mid P \cap F_{\overline{\mathcal{E}}} \neq \emptyset\}$,
$\delta(R, a) = \{P \in Q \mid P \subseteq \delta_{\overline{\mathcal{E}}}(R, a)\}$ for all $R \in Q$, $a \in \Sigma$.

```
cover(allowed,selected,remain)
    /* allowed, selected, remain $\subseteq Q_{\mathcal{E}}$ fulfill the precondition */
    /* explores all reasonable extensions of selected
    to a set satisfying the cover/closure property of Lemma 2.6 */
    if remain $= \emptyset$ then
        /* no sets remain to be covered, we have a solution! */
        best_solution := selected
    else
        /* there are still sets that remain to be covered */
        for all current $\in$ remain do
            if selected covers current with subsets then
                /* current is already covered, treat the rest of remain */
                cover(allowed, selected, remain$\setminus\{$current$\}$ )
            else
                /* current has to be covered by further selections */
                if |selected| $+ 1 <$ best_known
                    /* stop backtracking if best known solution cannot be improved */
                    for all $P \subseteq$ current with $P \in$ allowed do
                        allowed := allowed$\setminus\{P\}$
                        cover(allowed, selected $\cup \{P\}$, remain $\cup \{\delta_{\overline{\mathcal{E}}}(P, a) \mid a \in \Sigma\}$)
                    rof
                fi
            fi
        rof
    fi
```

Figure 1: The recursive procedure cover

One important and simple way to improve this strategy is to compare the size of the state set of $\mathcal{E}$ and $\mathcal{D}$ after line 2 and exchange them if $\mathcal{D}$ is smaller. In this case, the algorithm computes a small NFA for the reversed language $\overline{L}$, thus the output NFA has to be reversed as well.

Note that the modification of allowed in the last for-loop is to avoid that one and the same selection of subsets is explored more than once: When inside one incarnation of cover the addition of a certain set P to the current selection has been fully explored, it is not necesarry to consider P any more unless this incarnation is exited and the current selection becomes smaller.

In the programm system AMoRE, a similar (but non-recursive) algorithm has been implemented. One great problem one has to deal with is how to represent the subsets of $Q_{\mathcal{E}}$ and $2^{Q_{\mathcal{E}}}$ without wasting too much space. We chose to represent the subsets of $Q_{\mathcal{E}}$ by 32-bit integers, so the available implementation does not work if both the minimal DFA for L and for $\overline{L}$ have more than 32 states.

Another problem is how to realize the last for-loop running over all subsets of a given set. Here, we have decided to trace the sets in the canonical ordering they have when represented by integers. Experience shows that this order influences the performance of the algotithm very much, so it might be advantageous to choose a more sophisticated ordering: For example, one could try to treat the large subsets before the small ones or vice versa.

5 Fundamental vs. Canonical Automaton

In this section, we will give a more abstract definition of the fundamental automaton, which is, however, less suitable for actual calculation. Using this characterization, we compare the fundamental automaton to a different automaton, the so-called *canonical automaton*, defined by Arnold, Dicky and Nivat in [1]. This one has similar properties as the fundamental automaton, but it may be smaller. Moreover, we transfer the mentioned sufficient cover/closure property (conditions (i),(ii) of Lemma 2.6) for equivalent subautomata from the fundamental to this canonical automaton.

5.1 Fundamental Automaton Revisited

Define the MDFA $\mathcal{D}'$ (minimal deterministic finite automaton) for a language L by

$$
\begin{aligned}
&Q_{\mathcal{D}'} = \{w^{-1}L \mid w \in \Sigma^*\}\\
&I_{\mathcal{D}'} = \{L\}\\
&\delta_{\mathcal{D}'}(w^{-1}L, a) = (wa)^{-1}L\\
&F_{\mathcal{D}'} = \{w^{-1}L \mid \epsilon \in w^{-1}L\}
\end{aligned}
$$

Observing the fact $\overline{w}^{-1}\overline{L} = \overline{Lw^{-1}}$, we get that the MDFA for $\overline{L}$ is isomorphic to the following DFA $\mathcal{E}'$ via the mapping $M \mapsto \overline{M}$.

$$
\begin{aligned}
&Q_{\mathcal{E}'} = \{Lw^{-1} \mid w \in \Sigma^*\}\\
&I_{\mathcal{E}'} = \{L\}\\
&\delta_{\mathcal{E}'}(Lw^{-1}, a) = L(aw)^{-1}\\
&F_{\mathcal{E}'} = \{Lw^{-1} \mid \epsilon \in Lw^{-1}\}
\end{aligned}
$$

With this definition we can define the fundamental automaton $\mathcal{F}'$ differently than in Section 2 by:

$$
\begin{aligned}
&Q_{\mathcal{F}'} = \{P \subseteq Q_{\mathcal{E}'} \mid \bigcap P \neq \emptyset\}\\
&I_{\mathcal{F}'} = \{P \in Q_{\mathcal{F}'} \mid \epsilon \in \bigcap P\}\\
&\delta_{\mathcal{F}'}(P, a) = \{R \in Q_{\mathcal{F}'} \mid R \subseteq \delta_{\overline{\mathcal{E}'}}(P, a)\}\\
&F_{\mathcal{F}'} = \{P \in Q_{\mathcal{F}'} \mid L \in P\}
\end{aligned}
$$

To see that this definition is indeed equivalent to the one given before, let $\mathcal{E} = D(\overline{\mathcal{D}'})$ with D' being the minimal DFA for the language L as described above, i.e. the states of D' are left quotients. (Note that D' is a possible choice for the D of the last section, so $\mathcal{E}$ is a possible choice for the $\mathcal{E}$ in SectionSecFunAut). Let $\mathcal{F}'$ be the fundamental automaton as defined in Definition 2.2 of Section 2, i.e. with states in $2^{Q_{\mathcal{E}}}$.

Lemma 5.1 $\mathcal{F}'$ is isomorphic to $\mathcal{F}$.

Proof It is a well known fact that there exists an isomorphism α from $\mathcal{E}$ onto $\mathcal{E}'$, namely

$$\alpha : Q_{\mathcal{E}} \longrightarrow Q_{\mathcal{E}'} \quad , \quad (\{u^{-1}L \mid uw \in L\}) \mapsto Lw^{-1}.$$

We claim that the extension from α to $Q_{\mathcal{F}} \subseteq 2^{Q_{\mathcal{E}}}$ into $2^{Q_{\mathcal{E}'}}$ is an isomorphism from $Q_{\mathcal{F}}$ onto $Q_{\mathcal{F}'}$. We will denote this extension again by α. Since the original α is injective, so is this extension.

Let $\emptyset \neq M \subseteq Q_{\mathcal{E}} = \{\{u^{-1}L \mid uw \in L\} \mid w \in \Sigma^*\}$. Then we have the following equivalence showing that α is indeed a mapping from $Q_{\mathcal{F}}$ into $Q_{\mathcal{F}'}$.

$$\begin{aligned}
M \in Q_{\mathcal{F}} &\iff \bigcap M \neq \emptyset \\
&\iff \exists u^{-1}L \in Q_{\mathcal{D}'} : \forall P \in M : u^{-1}L \in P \\
&\iff \exists u \in \Sigma^* : u^{-1}L \neq \emptyset \land \forall Lw^{-1} \in \alpha(M) : uw \in L \\
&\iff \bigcap \alpha(M) \neq \emptyset \\
&\iff \alpha(M) \in Q_{\mathcal{F}'}
\end{aligned}$$

Similarly, we have

$$\begin{aligned}
M \in I_{\mathcal{F}} &\iff M \subseteq I_{\overline{\mathcal{E}}} = F_{\mathcal{E}} \\
&\iff \forall P \in M : P \cap I_{\mathcal{D}'} \neq \emptyset \\
&\iff \forall P \in M : L \in P \\
&\iff \forall Lw^{-1} \in \alpha(M) : w \in L \\
&\iff \epsilon \in \bigcap \alpha(M) \\
&\iff \alpha(M) \in I_{\mathcal{F}'}
\end{aligned}$$

The correspondence of the set of final states is shown by the following equivalences:

$$\begin{aligned}
M \in F_{\mathcal{F}} &\iff \emptyset \neq M \cap F_{\overline{\mathcal{E}}} = M \cap I_{\mathcal{E}} \\
&\iff I_{\overline{\mathcal{D}'}} \in M \\
&\iff \{u^{-1}L \in Q_{\mathcal{D}} \mid \epsilon \in u^{-1}L\} \in M \\
&\iff L = \alpha(\{u^{-1}L \mid \epsilon \in u^{-1}L\}) \in \alpha(M) \\
&\iff \alpha(M) \in F_{\mathcal{F}'}
\end{aligned}$$

Finally, we check that α commutes with the transition relations. This is verified as follows:

$$\begin{aligned}
N \in \delta_{\mathcal{F}}(M, a) &\iff N \subseteq \delta_{\overline{\mathcal{E}}}(M, a) \\
&\iff \alpha(N) \subseteq \alpha(\delta_{\overline{\mathcal{E}}}(M, a)) \\
&\iff \alpha(N) \subseteq \delta_{\overline{\mathcal{E}'}}(\alpha(M), a) \\
&\iff \alpha(N) \in \delta_{\mathcal{F}'}(\alpha(N), a)
\end{aligned}$$

This completes the proof and establishes the claimed correspondence of the two definitions for the fundamental NFA. In the remainder of the paper we will write $\mathcal{F}$ instead of $\mathcal{F}'$. $\square$

5.2 The Canonical Automaton

We compare the fundamental automaton to a different NFA introduced by Arnold, Dicky and Nivat in [1]. They defined the *canonical automaton* $\mathcal{C}$ as described now. We start with an auxiliary definition. For $K \subseteq \Sigma^*$, let $\phi(K) = \{u \in \Sigma \mid uK \subseteq L\} = \bigcap_{v \in K} Lv^{-1}$.

Definition 5.2 The *canonical automaton* $\mathcal{C}$ is given by

$$\begin{aligned} Q_{\mathcal{C}} &= \{\phi(K) \mid K \subseteq \Sigma^* \text{ and } K, \phi(K) \neq \emptyset\} \\ &= \{\textstyle\bigcap P \neq \emptyset \mid P \text{ is a nonempty set of right quotients of } L\} \end{aligned}$$

$$\begin{aligned} &I_{\mathcal{C}} = \{\textstyle\bigcap P \in Q_{\mathcal{C}} \mid \epsilon \in \bigcap P\} \\ &\delta_{\mathcal{C}}(\textstyle\bigcap P, a) = \{\bigcap R \in Q_{\mathcal{C}} \mid (\bigcap P)a \subseteq \bigcap R\} \text{ for all } \bigcap P \in Q_{\mathcal{C}} \text{ and all } a \in \Sigma^* : \\ &F_{\mathcal{C}} = \{\textstyle\bigcap P \in Q_{\mathcal{C}} \mid \bigcap P \subseteq L\} \end{aligned}$$

The canonical and the fundamental automaton have similar properties. For example, it is shown in [1] that for any NFA $\mathcal{A}$ accepting L, there exists a morphism from $\mathcal{A}$ into $\mathcal{C}$, namely $q \mapsto \phi(post(\mathcal{A}, q))$. Thus any minimal NFA accepting L is isomorphic to some subautomaton of $\mathcal{C}$.
An interesting fact about the canonical automaton is that it is in some sense the smallest NFA with that property, because any epimorphism from $\mathcal{C}$ onto an equivalent NFA is injective.
Let us denote by $\mathcal{F}$ the fundamental automaton as defined in this section. The relation between the fundamental automaton and the canonical automaton is illustrated by the observation of [1] that a morphism f from $\mathcal{F}$ into $\mathcal{C}$ is given by

$$f : P \mapsto \phi(post(\mathcal{F}, P)) = \bigcap P$$

To see the last equality we observe that for any $w \in \Sigma^*$ and any $P \in Q_{\mathcal{F}}$ we have

$$\begin{aligned} w \in post(\mathcal{F}, P) &\iff \exists R \in Q_{\mathcal{F}} : R \in F_{\mathcal{F}} \wedge R \in \delta_{\mathcal{F}}(P, w) \\ &\iff \exists R \in Q_{\mathcal{F}} : L \in R \wedge \delta_{\mathcal{E}}(R, \overline{w}) \subseteq P \\ &\iff \exists R \subseteq Q_{\mathcal{E}} : \bigcap R \neq \emptyset \wedge L \in R \wedge \forall Lv^{-1} \in R (L(wv)^{-1} \in P) \\ &\iff Lw^{-1} \in P. \end{aligned}$$

(To see the last implication from right to left choose $R := \{L\}$.) This shows for any $u \in \Sigma^*$ and any $P \in Q_{\mathcal{F}}$

$$\begin{aligned} u \in \phi(post(\mathcal{F}, P)) &\iff \forall w \in post(\mathcal{F}, P) : uw \in L \\ &\iff \forall w \in \Sigma^* : Lw^{-1} \in P \Rightarrow uw \in L \\ &\iff \forall w \in \Sigma^* : Lw^{-1} \in P \Rightarrow u \in Lw^{-1} \\ &\iff u \in \bigcap P \end{aligned}$$

The morphism f is always surjective, and it is injective if there is no right quotient that is a superset of a nonempty intersection of other right quotients. There are

examples where there is such a right quotient. For example for $L = \{aa, ab, bb\}$, the automaton $\mathcal{F}$ has 5 states (namely $\{La^{-1}\}$, $\{Lb^{-1}\}$, $\{La^{-1}, Lb^{-1}\}$, $\{L\}$ and $\{\{\epsilon\}\}$), whereas $\mathcal{C}$ has 4 states (namely $\{a\}$, $\{a, b\}$, L and $\{\epsilon\}$); so the two automata $\mathcal{F}$ and $\mathcal{C}$ are in general not isomorphic.
The conditions of Lemma 2.6 leading to the minimization heuristic can be transfered to $\mathcal{C}$, yielding another heuristic for finding a small NFA for a given language.

5.3 A Criterion for Equivalent Subautomata of the Canonical NFA

Lemma 5.3 Let $\mathcal{P} \subseteq Q_{\mathcal{F}}$ and $\mathcal{P}_\cap \subseteq Q_{\mathcal{C}}$ with $\mathcal{P} = \{P \subseteq Q_{\mathcal{E}} \mid \bigcap P \in \mathcal{P}_\cap\}$. (That is, $\mathcal{P} = f^{-1}(\mathcal{P}_\cap)$, with f being the morphism $P \mapsto \bigcap P$ from above.) Let the following two properties be fulfilled:

(i) $\mathcal{P}$ covers $I_{\overline{\mathcal{E}}}$ with subsets and

(ii) For all $R \in \mathcal{P}$ and for all $a \in \Sigma$, $\mathcal{P}$ covers $\delta_{\overline{\mathcal{E}}}(R, a)$ with subsets.

We claim that the subautomaton $\mathcal{C}_{\mathcal{P}_\cap}$ of $\mathcal{C}$ induced by $\mathcal{P}_\cap$ recognizes L.

Proof The direct proof of this result is possible, but we can argue simply like this: Because of Lemma 2.6, the subautomaton $\mathcal{F}_{\mathcal{P}}$ of $\mathcal{F}$ induced by $\mathcal{P}$ is equivalent to $\mathcal{A}$. The subautomaton $\mathcal{C}_{\mathcal{P}_\cap}$ is simply the image of $\mathcal{F}_{\mathcal{P}}$ under the morphism $P \mapsto \bigcap P$, so we have $L \subseteq L(\mathcal{C}_{\mathcal{P}_\cap}) \subseteq L(\mathcal{C}) = L$, showing our claim. □

Unfortunately, this cover/closure property of Lemma 5.3 is not necessary for equivalent subautomata of $\mathcal{C}$, so also for $\mathcal{C}$ it does not give a heuristic that always produces a minimal NFA.
One way of making use of this lemma would be to proceed as described in Secion 4, but put together states that have the same image under $P \mapsto \phi(post(\mathcal{F}, P)$ before the backtracking is started.

6 Conclusion

The minimization of nondeterministic finite automata is a difficult problem. One possible approach is to search among the subautomata of either the fundamental or the canonical automaton of a language. Both of these do contain all minimal NFA as subautomata, but we have no efficiently testable criterion that characterizes those subautomata that accept the same language. All we have is a sufficient criterion, so one can at least search for subautomata fulfilling this. It remains an open question if and how this criterion can be modified so that it characterizes all equivalent subautomata of the fundamental or the canonical automaton. If this modification is not possible, it would be interesting at least to know for which languages the mentioned sufficient criterion will be necessary.
Apart from the minimization problem the definition and the properties of the fundamental and canonical automaton are interesting. Both of these are unique for a given regular language. Both contain homomorphic images of all automata equivalent to themselves, and the canonical automaton is in some sense the smallest with that property.

References

[1] A. Arnold, A. Dicky, M. Nivat, A note about minimal non-deterministic automata; in: *Bulletin of the EATCS* 47, June 1992, pp. 166-169.

[2] W. Brauer, *Automatentheorie*, Teubner, Stuttgart 1984.

[3] J.A. Brzozowski: Canonical regular expressions and minimal state graphs for definite events, in: *Proc. Symp. on Math. Theory of Automata*, Vol. 12, Brooklyn, N. Y.: Brooklyn Polytechnic Institute, 1963, pp. 529 -561.

[4] J.E. Hopcroft: An n log(n) algorithm for minimizing states in a finite automaton; in: Z. Kohavi, A. Paz (Eds.): *Theory of Machines and Computation*; Academic Press, New York etc., 1971, pp. 189-196.

[5] J.E. Hopcroft, J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation*; Addison-Wesley, 1979.

[6] M.R. Garey, D.S. Johnson: *Computers and Intractability – A Guide to the Theory of NP-Completeness*; Freeman, San Francisco, 1979.

[7] K. Indermark: Zur Zustandsminimisierung nichtdeterministischer erkennender Automaten; GMD Seminarberichte Bd. 33, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin – Bonn, 1970.

[8] T. Jiang, B. Ravikumar: Minimal NFA Problems Are hard (Extended Abstract); in: J.Leach Albert, B. Monien, M.Rodriguez Artalejo (Eds.): *Automata, Languages and Programming, 18th International Colloquium*, Madrid, July 1991, Proceedings, LNCS 510, Springer, Berlin etc., 1991, pp. 629-640.

[9] T. Kahlert: *Ein Verfahren für die Minimierung nichtdeterministischer endlicher Automaten und seine Implementierung*; Diplomarbeit, Christian-Albrechts-Universität, Kiel, 1991.

[10] T. Kameda, P. Weiner: On the State Minimization of Nondeterministic Finite Automata; *IEEE Trans. Comp. C–19*(1970), pp. 617-627.

[11] J. Kim: State minimization of nondeterministic machines; IBM Thomas J. Watson Res. Center Rep. RC 4896, 1974.

[12] M.O. Rabin, D. Scott: Finite Automata and their decision problems; *IBM J. Res. and Develop.* 3, April 1959, pp. 114-125.

Efficient Simplification of Bisimulation Formulas

Uffe H. Engberg
BRICS [*]Department of Computer Science, University of Aarhus, Denmark

Kim S. Larsen [†]
Department of Mathematics and Computer Science, Odense University, Denmark

Abstract

The problem of checking or optimally simplifying bisimulation formulas is likely to be computationally very hard. We take a different view at the problem: we set out to define a very fast algorithm, and then see what we can obtain. Sometimes our algorithm can simplify a formula perfectly, sometimes it cannot. However, the algorithm is extremely fast and can, therefore, be added to formula-based bisimulation model checkers at practically no cost. When the formula can be simplified by our algorithm, this can have a dramatic positive effect on the better, but also more time consuming, theorem provers which will finish the job.

1 Introduction

The need for validity checking or optimal simplification of first order bisimulation formulas has arisen from recent work on symbolic bisimulation checking of *value-passing calculi* [4, 9, 15]. The NP-completeness of checking satisfiability of propositional formulas [3] implies that validity checking of that class of formulas is co-NP complete. Additionally, checking of quantified formulas is P-space hard [7], so there is not much hope for a fast algorithm for deciding exactly when a bisimulation formula is valid.

Instead, we set out to solve the problem of what you can get for free, i.e., to what extent is it possible to decide validity simply while reading the formula? As it turns out, there is almost nothing that can be done in linear time. The most simple tasks of storing and retrieving information about variables will cost $O(n \log n)$. So, we allowed ourselves this extra log-factor and changed the question to what you can get *almost* for free. As we shall demonstrate in this paper, the algorithm we have designed is very fast. Not alone does it run in $O(n \log n)$; the constant is also very small. On average, we read through the formulas at a rate of about 75 Kbytes per second. Of course, this is only interesting if the algorithm outputs useful answers reasonably frequently, i.e., in the absence of an obvious notion of optimal simplification (to a minimal equivalent formula), if the algorithm can reasonably frequently guarantee that the formula is valid, rule out that the formula could be valid, or maybe simplify a huge formula to a much smaller equivalent one. It is not easy

[*] Basic Research in Computer Science, a Centre of the Danish National Research Foundation.
[†] The initial part of this work was done while this author was at Aarhus University.

to measure how often the algorithm produces a useful answer, but through examples, we show that there are families of process expressions which give rise to formulas, where our algorithm is successful.

The algorithm make a single pass over the formula making no assumptions about the variable names. Notice that this also implies that if formulas are passed on to our algorithm from another program, they do not have to be saved at any point, but can be passed on to our program via pipelining.

As already mentioned, in addition to validity checking, we can also simplify formulas. This greatly increases the usefulness of this work. Even if our algorithm fails to prove the validity of a formula, it will quite often simplify the formula so drastically that the validity of the formula (or the opposite) can easily be asserted by the user. Another possibility is to check the simplified formula for validity using other tools. Tools that would succeed more often, but the complexity of which would make it impossible to work on the original formula. The main advantage of simplifying formulas is that the algorithm can be built into the program generating the bisimulation formulas and simplify the formula on the fly. Also, intermediate simplifications can be used to prune the generation of other subformulas. This can be of vital importance since the formulas can become exponentially large.

Our algorithms work just as well on formulas with free variables as on closed formulas. This means that by simplifying a formula with free variables, we actually characterize the conditions, in terms of the free variables, under which two processes are bisimilar.

2 Bisimulation Formulas

In this section, we introduce bisimulation formulas and some general notions from formal logic. Due to lack of space we shall only briefly sketch how bisimulation formulas are obtained.

In [4, 9], bisimulation formulas are used in a three stage process of verifying bisimilarity of value-passing programs ([15] obtain similar formulas, albeit with a different approach).

In the first stage, symbolic transition graphs (a generalization of the standard notion of labeled transition graphs) are generated from terms of some value-passing language, say the full CCS calculus [14]. The two graphs are *symbolic bisimilar* [4, 9, 10] iff the two terms are bisimilar in traditional sense.

Below, two processes are shown together with their associated graphs. Following [14], $\mathbf{0}$ is the process having no actions, whereas the prefixed process $\tau.p$ can make an internal action and then act as p. Similarly, there are input (output) prefixes $\alpha?x.$ ($\alpha!x.$) for receiving (sending) values on channel α. The expression **if** E **then** p can do the actions of p, provided the condition E evaluates to true, T. The process $p + p'$ acts as either p or p'.

$A_1(x,y) \overset{\text{def}}{=} \tau.\mathbf{0} + \tau.\textbf{if } x \equiv y \textbf{ then } \tau.\mathbf{0}$ $\quad$ $A_3(x,y) \overset{\text{def}}{=} \tau.(\textbf{if } x \equiv y \textbf{ then } \tau.\mathbf{0}) + \textbf{if } x \equiv y \textbf{ then } \tau.\mathbf{0}$

$$\begin{tikzcd} & g_1 \arrow[dl, "{\mathsf{T},\tau}"'] \arrow[dr, "{\mathsf{T},\tau}"] & \\ {} & & {} \arrow[d, "{x \equiv y,\tau}"] \\ & & {} \end{tikzcd}
\qquad
\begin{tikzcd} & g_3 \arrow[dl, "{\mathsf{T},\tau}"'] \arrow[dr, "{x \equiv y,\tau}"] & \\ {} \arrow[d, "{x \equiv y,\tau}"'] & & {} \\ {} & & \end{tikzcd}$$

To shorten the presentation, all nodes except for root nodes, have been omitted from the graphs. Each edge is labeled with a guarding condition and a (symbolic) action. The initial conditions of the graph g_1 associated with A_1 are T, whereas the last condition is $x \equiv y$, i.e., x and y should have equivalent values.

In the next stage, an algorithm is used for finding a first order boolean expression *mgb*, called the most general boolean, characterizing the conditions for which two finite symbolic transition graphs are (symbolic) bisimilar.

Intuitively, two processes are bisimilar if, whenever one process can do an action, the other has a matching action such that the resulting two processes again are bisimilar. This is reflected in the bisimulation formula. For example, the fact that g_3 must match an action corresponding to the left edge of g_1, is captured in the mgb_{g_1,g_3} subformula

$$\mathsf{T} \to \bigvee \begin{matrix} \mathsf{T} \wedge [\mathsf{T} \wedge (x \equiv y \to \mathsf{F})] \\ x \equiv y \wedge [\mathsf{T} \wedge \mathsf{T}] \end{matrix}$$

If an instantiation of variables satisfies the guarding condition T of g_1, then it must satisfy a guarding condition, T or $x \equiv y$, of g_3 as well as the corresponding *mgb*, $[\mathsf{T} \wedge (a \equiv y \to \mathsf{F})]$ or $[\mathsf{T} \wedge \mathsf{T}]$.

When matching output actions, the values sent must be equal. An equality predicate captures this. By input, universal quantification is used to express that for all values received, the processes are bisimilar.

In the final stage, validity of the bisimulation formula is checked. If it is valid, the original programs are bisimilar under all instantiations. Otherwise, the formula expresses the weakest conditions on the instantiations for which they are bisimilar. Later, in section 7, we shall see that mgb_{g_1,g_3} is in fact valid.

Formally, the class of formulas, which we will work with in the rest of this paper, is defined by the syntax

$$E ::= P \mid E \wedge E \mid E \vee E \mid P \to E \mid \forall x\colon\ E \qquad \text{and} \qquad P ::= \mathsf{T} \mid \mathsf{F} \mid x \equiv y,$$

where x and y range over a set, V, of variables. As usual, formulas are closed if they have no free variables. Notice that bisimulation formulas only have universal quantification. The binary predicate symbol $\equiv$ is assumed be interpreted as an equivalence relation $\equiv_D$ over a nonempty domain D. It is then standard how to define when an *environment*, i.e., a function from V to D, *satisfies* a formula. An environment satisfies a set of formulas Γ, if it satisfies each formula of Γ. Γ *semantically entails* E, written $\Gamma \models E$, if E is satisfied by any environment satisfying Γ. E is *valid*, $\models E$, if $\emptyset \models E$.

The class of bisimulation formulas is a subset of the class of all quantified formulas and checking validity of bisimulation formulas is not P-space hard. An easy reduction shows that checking satisfiability with respect to an environment is NP complete, so presumably checking validity of bisimulation formulas is co-NP complete.

We now state some general properties of entailment relevant for the development of the algorithm. For simplicity, we write Γ as $E_1, \ldots, E_n$ when $\Gamma = \{E_1, \ldots, E_n\}$. Similarly, we write Γ, Δ for $\Gamma \cup \Delta$.

Theorem 1 (Entailment)

a) If $\Gamma \models E$ then $\Gamma, \Delta \models E$	(Ext)
b) If $E \in \Gamma$ then $\Gamma \models E$	(Rep)
c) If $\Gamma \models E$ and $\Gamma, E \models E'$ then $\Gamma \models E'$	(Cut)
d) If $\Gamma \models E$ and $E \models E'$ then $\Gamma \models E'$	(Trans)
e) $\Gamma \models E$ and $\Gamma \models E'$ iff $\Gamma \models E \wedge E'$	(Conj)
f) $\Gamma, E \models E'$ iff $\Gamma \models E \rightarrow E'$	(Imp)

Proof Standard, see e.g. [16]. □

In general, there is not any similar disjunction theorem allowing both introduction and elimination to the right. However, from the entailment theorems and a few tautologies, we get proposition 2. As a consequence of $\equiv_D$ being an equivalence relation we also have proposition 3.

Proposition 2 If $\Gamma \models E$ or $\Gamma \models E'$, then $\Gamma \models E \vee E'$.

Proposition 3 a) $\models x \equiv x$ b) $x \equiv y \models y \equiv x$ c) $x \equiv y, y \equiv z \models x \equiv z$

3 The Abstract Algorithm

We now set out to design an algorithm for checking validity of formulas with an equivalence predicate. We keep it as abstract as possible to allow for a large degree of freedom in the choice of data structures in the actual implementation.

Intuitively, the idea of the algorithm is to collect in a relation R (over variables) information about variables known to be equivalent when checking subformulas. For instance, checking the validity of a formula like $x \equiv y \rightarrow E$ is reduced to checking E under the assumption that x and y are equivalent, i.e. (x, y) is added to R and E then checked. To exploit that $\equiv$ is an equivalence relation, the symmetric and transitive closure, is taken before proceeding to E. However, when checking E of the formula $\forall x\colon E$ the situation is quite the opposite. Since a new scope is entered, all previous collected information in R concerning x, must be removed before E is checked.

Formally, for R denote the symmetric and transitive closure by $R^{\oplus}$, the reflexive closure, $R \cup \{(x,x) \mid x \in V\}$, by R^0 and the removal of x, $\{(y,z) \in R \mid y \neq x, z \neq x\}$, by $R \setminus x$. Notice, $R \setminus x \subseteq R$, and if R is symmetrically and transitively closed, then so is $R \setminus x$.

In order to connect with the logic, we associate with R the set of formulas $R_{\equiv} \overset{\text{def}}{=} \{x \equiv y \mid (x,y) \in R\}$. The notions of closures and removal extend to $R_{\equiv}$ in the natural way: $R_{\equiv} \setminus x$ is $(R \setminus x)_{\equiv}$ etc.

The algorithm is conveniently described using Kleene's three-valued logic [11], the three truth-values being **t** for "true", **f** for "false" and **u** for "undefined"/ "unknown". The Kleene truth tables for conjunction, $\wedge_K$, disjunction, $\vee_K$, and implication, $\rightarrow_K$, are:

$\wedge_K$	**t**	**f**	**u**
t	**t**	**f**	**u**
f	**f**	**f**	**f**
u	**u**	**f**	**u**

$\vee_K$	**t**	**f**	**u**
t	**t**	**t**	**t**
f	**t**	**f**	**u**
u	**t**	**u**	**u**

$\rightarrow_K$	**t**	**f**	**u**
t	**t**	**f**	**u**
f	**t**	**t**	**t**
u	**t**	**u**	**u**

The abstract algorithm is expressed in terms of a function, $\VDash$, which given a bisimulation formula E and a symmetrically and transitively closed relation R, returns **t** only if $R_{\equiv} \models E$, and **f** only if $R_{\equiv} \not\models E$. From now on, R is assumed to be symmetrically and transitively closed. Writing $\VDash$ infix, the definition is:

$$
\begin{array}{lrl}
R \VDash E \textbf{ is case } E \textbf{ of} & \mathsf{T} & : \mathbf{t} \\
 & \mathsf{F} & : \mathbf{f} \\
 & x \equiv y & : \textbf{if } x\ R^0\ y \textbf{ then t else u} \\
 & E' \wedge E'' & : R \VDash E' \wedge_K R \VDash E'' \\
 & E' \vee E'' & : R \VDash E' \vee_K R \VDash E'' \\
 & E' \rightarrow E'' & : R \VDash E' \rightarrow_K R' \VDash E'', \\
 & & \text{ where } R' = \begin{cases} (R \cup \{(x,y)\})^{\oplus}, & \text{if } E' \text{ is } x \equiv y \\ R, & \text{if } E' \text{ is } \mathsf{T} \text{ or } \mathsf{F} \end{cases} \\
 & \forall x\colon\ E' & : R \setminus x \VDash E'
\end{array}
$$

Notice that $\VDash$ is well-defined because we take the symmetric and transitive closure of $(R \cup \{(x,y)\})$.

Given a formula E, the initial call to this function will be $\emptyset \VDash E$, where $\emptyset$ is the empty relation.

Proving correctness is a matter of proving soundness of $\VDash$ relative to $\models$. First, we need a small result linking $R_{\equiv}$ to universal quantification.

Lemma 4 If $R_{\equiv} \setminus x \models E$, then $R_{\equiv} \models \forall x\colon\ E$.

Proof Assume that $R_{\equiv} \setminus x \models E$ and let an environment ρ satisfying $R_{\equiv}$ be given. Because $R_{\equiv} \setminus x \subseteq R_{\equiv}$, ρ must satisfy $R_{\equiv} \setminus x$ as well. Now x does not occur in any formula of $R_{\equiv} \setminus x$ so all environments differing from ρ only on the value of x, will then also satisfy $R_{\equiv} \setminus x$. By the assumption each such environment also satisfies E wherefore the original environment ρ satisfies $\forall x\colon\ E$. □

Writing $\VDash E$ for $\emptyset \VDash E$, we can now state the correctness of the algorithm.

Theorem 5 (Correctness) a) If $\VDash E = \mathbf{t}$, then $\models E$. b) If $\VDash E = \mathbf{f}$, then $\not\models E$.

Proof Part a) of the theorem follows from the stronger statement

$$\text{if } R \VDash E = \mathbf{t} \text{ then, } R_{\equiv} \models E$$

which we prove by induction on the structure of E. Assume $R \VDash E = \mathbf{t}$. We consider the forms of E:

T, F: In general, $\Gamma \models \mathsf{T}$, so also $R_{\equiv} \models \mathsf{T}$. The case of F is trivial, since $R \VDash \mathsf{F} \neq \mathbf{t}$.

$x \equiv y$: By definition of $\_^0$, it follows that $R \Vdash x \equiv y = \mathbf{t}$ iff either $x\ R\ y$ or $x = y$.
Now, $x\ R\ y$ is equivalent to $x \equiv y \in R_\equiv$. By (Rep), we get $R_\equiv \models x \equiv y$. In the case $x = y$, the situation is really that E is $x \equiv x$. By (Ext) and a) of proposition 3, we directly obtain $R_\equiv \models x \equiv x$.

$E' \wedge E''$: By definition, $R \Vdash E' \wedge E'' = \mathbf{t}$ implies $R \Vdash E' = \mathbf{t}$ and $R \Vdash E'' = \mathbf{t}$. By induction, we obtain that $R_\equiv \models E'$ and $R_\equiv \models E''$. Using (Conj), we obtain that $R_\equiv \models E' \wedge E''$.

$E' \vee E''$: Similar, using proposition 2 instead of (Conj).

$E' \to E''$: By the definition of $\longrightarrow_K$, we must have $R \Vdash E' = \mathbf{f}$ or $R' \Vdash E'' = \mathbf{t}$. The forms of E':

T: Then $R' = R$ and because $R \Vdash \mathsf{T} = \mathbf{t}$, it follows that $R \Vdash E'' = \mathbf{t}$. As above we deduce $R_\equiv \models E''$. Using (Ext), we get $R_\equiv, \mathsf{T} \models E''$ and therefore $R_\equiv \models \mathsf{T} \to E''$ follows by (Imp).

F: In general $\Gamma, \mathsf{F} \models E$, so in particular $R_\equiv, \mathsf{F} \models E''$. By (Impl), $R_\equiv \models \mathsf{F} \to E''$.

$x \equiv y$: We have $R \Vdash x \equiv y = \mathbf{t}$ or $R \Vdash x \equiv y = \mathbf{u}$. In either case, we must have $R' \Vdash E'' = \mathbf{t}$, where $R' = (R \cup \{(x, y)\})^\oplus$. By induction, we get $R'_\equiv \models E''$, which is the same as $(R_\equiv, x \equiv y)^\oplus \models E''$. Now any $z \equiv w$ in $(R_\equiv, x \equiv y)^\oplus$ can be deduced from $R_\equiv, x \equiv y$ so by repeated use of (Cut), each of these $z \equiv w$ can be removed from the hypothesis and we finally get $R_\equiv, x \equiv y \models E''$. Thus, by (Impl), $R_\equiv \models x \equiv y \to E''$.

$\forall x\colon E'$: By the induction hypothesis we get that $R_\equiv \setminus x \models E'$. The result follows from lemma 4.

Part b) follows similarly from the stronger statement that if $R \Vdash E = \mathbf{f}$, then $R_\equiv \models \neg E$.

$\square$

4 The Concrete Algorithm

In this section, we discuss the implementation of the abstract algorithm outlined in section 3. The function $\Vdash$, which is defined there, closely follows the structure of a formula E. The concrete implementation in this section will follow this structure in exactly the same way. So, the primary task is to find a representation of the relation R such that operations on this relation (union, closure, checking for equivalence, etc.) can be performed efficiently.

The primary operations are to make two variables equivalent and to test whether two variables are already equivalent. This is an instance of the so-called disjoint set problem, which is usually solved using rooted trees [6]. To obtain the best possible performance, path compression (McIllroy and Morris) and union by rank [17] (or similar schemes) are normally used to obtain an amortized complexity of $O(A^{-1}(n))$ per find operation [17, 19], where A^{-1} is the inverse of the (unary) Ackermann function [1].

However, when processing formulas like $(x \equiv y \to E) \wedge E'$, we need to first form the union of the equivalence classes of x and y, then process the expression E, and then deunion (undo) the last union before processing E'. Path compressions are impossible to undo without ruining the complexity, so we only use union by rank, and obtain a complexity of $O(\log n)$ per find [18]. In order to undo the unions, each union operation is registered on a stack. In this way, deunions can be done in constant time (unions are still constant time). These three operations, find, union, and deunion, can also be implemented such that the amortized complexity for the find operation becomes $O(\log n/(\log\log n))$. That proposal is from [12]. See [20] for the analysis. However, the size of the overhead is so large that for formulas that we consider (up to approximately 5Mbytes), this method is slower. For further details on disjoint set implementations, see [13]. We call the structure we use a *union-find-deunion* (UFD) *structure*.

For formulas without universal quantification, this would be all we would need. However, formulas like $(\forall x\colon\ E) \wedge E'$ require that the variable x is freed from previous unions while processing E. Afterwards, for the processing of E', all the old information on x must be restored. Having to keep track of several versions of variables means that the variables cannot be used directly in the *UFD* structure. Instead, we do the following: at any point during the processing of a formula, each variable, x, has an associated stack of pointers corresponding to the number of active versions of x. In greater detail, when a quantifier construction $\forall x\colon$ is encountered, a pointer is pushed onto x's stack. The pointer points to a new item in the *UFD* structure not related to anything, which was previously there. In this way, the old environment can be restored by simply popping the stack.

In order to access the stacks associated with variable names as fast as possible, variable names (along with the pointer to the stacks) are organized in a red-black tree [2, 8], which is one of the efficient implementations of dictionaries with a complexity of $O(\log n)$ per operation, where n is the number of elements in the tree.

To summarize, we use a red-black tree that has variable names as keys and stacks of pointers as values. All these pointers point into a common *UFD* structure. In addition, the *UFD* structure has its own stack of undo information. We refer to the structure consisting of all these other data structures as the *combined* structure.

In the following, we list the operations that the three data structures are assumed to be equipped with. The description is brief as all this is quite well known. However, it seems useful to introduce the names of the operations on the different structures.

A stack is a collection of values, which can be removed from the structure only in the reverse order of which they were inserted. Assume that S is a stack and v is a value. The following operations are supported: $\mathit{Push}(S, v)$, $\mathit{Pop}(S)$, $\mathit{Top}(S)$, $\mathit{Empty}(S)$, and $\mathit{InitStack}()$.

A dictionary implements a set of pairs (k, v), where k is a key value from a totally ordered domain and v is any value. We assume that each key value appears at most once in the dictionary. If T is a dictionary, then the following operations are supported: $\mathit{Insert}(T, k, v)$, $\mathit{Delete}(T, k)$, $\mathit{Member}(T, k)$, $\mathit{LookUp}(T, k)$, and $\mathit{InitTree}()$.

A *UFD* structure is a collection of elements some of which may be equivalent with other elements. The following operations are supported: $\mathit{Union}(U, p, q)$, $\mathit{Find}(U, p)$, $\mathit{Deunion}(U)$, and $\mathit{InitUFD}()$. Obviously, the implementation is basically the well-known union-find structure using a stack to save information about the unions.

A Kleene boolean is an implementation of Kleenes three-valued logic. The three Kleene truth-values *TRUE*, *FALSE*, and *UNKNOWN* correspond to **t**, **f**, and **u**, respectively. The operations *Kand*, *Kor*, and *Kimp* implement the operations $\wedge_K$, $\vee_K$, and $\longrightarrow_K$ as described in the tables of section 3. Furthermore, *Ktu* turns an ordinary boolean into the Kleene boolean *TRUE* if it is true and otherwise into *UNKNOWN*.

The Algorithm

In this section, we present the concrete algorithm, which implements the abstract algorithm from section 3. Basically, this is all about representing the relation R using advanced data structures. We assume that the formula E has a representation in the form of a syntax tree. There are well-developed standard techniques to define and manipulate syntax trees. For clarity, we leave out these details.

Also, to present the crucial parts of the algorithm as clearly as possible, we treat $E_1 \wedge E_2$ and $E_1 \vee E_2$ independently. In reality, as we want to process the formula using pipelining, we should process E_1 first and not until after that has been done can we decide whether a conjunction or a disjunction is been processed. Another reasonable assumption would be to require that the program generating the formula does this using a prefix notation like $\wedge(E_1, E_2)$ and $\vee(E_1, E_2)$.

For simplicity, we assume that the formulas are closed, i.e., they do not have any free variables. This is no serious simplification since free variables can be treated as if they were bound at the outermost level.

The concrete implementation follows the structure of the formula in the same way as $\Vdash$, except that the call for the left-hand operand of implication is unfolded and incorporated directly into the case analysis.

```
**function** check($E$: *formula*) $\rightarrow$ Kleene boolean;
  **var**
    r: Kleene boolean;
    p,q: pointers; (* into the *UFD* structure *)
  **case** $E$ **of**
    T:                      r := TRUE;
    F:                      r := FALSE;
    $x \equiv y$:                r := Ktu(Find(U,Top(LookUp(T,x))) = Find(U,Top(LookUp(T,y))));
    $E_1 \wedge E_2$:              r := Kand(check($E_1$),check($E_2$));
    $E_1 \vee E_2$:              r := Kor(check($E_1$), check($E_2$));
    $T \rightarrow E_1$:              r := check($E_1$);
    $F \rightarrow E_1$:              r := TRUE;
    $(x \equiv y) \rightarrow E_1$:      p := Find(U,Top(LookUp(T,x))); q := Find(U,Top(LookUp(T,y)));
                            **if** p $\neq$ q **then** Union(U,p,q);
                            r := Kimp(Ktu(p $\neq$ q),check($E_1$));
                            **if** p $\neq$ q **then** Deunion(U);
    $\forall x$: $E_1$:              **if** $\neg$Member(T,x) **then** Insert(T,x,InitStack());
                            **new**(p); Push(LookUp(T,x),p);
                            r := check($E_1$);
                            Pop(LookUp(T,x)); **free**(p);
                            **if** Empty(LookUp(T,x)) **then** Delete(T,x);
  **end**;
  **return** r;
**end**;
```

Before use, T is declared as a red-black tree and properly initialized using InitTree().

Similarly, U is declared as a *UFD* structure and initialized by a call to InitUFD().

Correctness

Proposition 6 The combined structure immediately after a call to the function check is exactly as it were immediately before the call to check.

Proof By induction in the number of calls to the function check. The base case is when this number is one, which means that check is not called recursively. Thus, we must be in one of the cases T, F, or $x \equiv y$. The result follows since the combined structure is not altered in any of these cases.

For the induction step, the result follows trivially from the induction hypothesis in the case where E is $E' \wedge E''$, $E' \vee E''$, $\mathsf{T} \to E'$, or $\mathsf{F} \to E'$, since the combined structure is not changed.

Assume that E is $(x \equiv y) \to E'$. By the induction hypothesis, the call check(E') leaves the structure unchanged. The claim follows as Deunion(U) will undo the last union not yet undone. This must be Union(U,p,q), as the combined structure after the call to check(E') is exactly as it were before the call.

Assume that E is $\forall x:\ E'$. By the induction hypothesis, the call check(E') leaves the structure unchanged. Since LookUp(T,x) is a stack, the statement Pop(LookUp(T,x)) will undo the effect of the statement Push(LookUp(T,x),p). Furthermore, if the stack LookUp(T,x) is empty, then this stack must have been inserted into T by this current invocation of check, so the empty stack should be deleted. □

Proposition 7 Let F be a bisimulation formula, and let E be a subexpression of F with x and y bound in the context of E. Immediately before the call check(E), the combined structure is an exact representation of R in the corresponding call $R \Vdash E$, i.e.,

$$x\ R^0\ y \Leftrightarrow \text{Find(U,Top(LookUp(T,x)))} = \text{Find(U,Top(LookUp(T,y)))}$$

Proof By induction in the structure of E. The base case is when $E = F$, in which case both R (and thus also R^0) and the combined structure are empty. For the induction step, we consider all possible forms that E could have.

If E is T, F, $x \equiv y$, $E' \wedge E''$, $E' \vee E''$, $\mathsf{T} \to E'$, or $\mathsf{F} \to E'$, then the combined structure remains unchanged and the same R is used in the recursive application of $\Vdash$.

Assume that E is $(x \equiv y) \to E'$. Then $\Vdash$ is called with the relation formed by adding (x, y) to R and taking the symmetric and transitive closure. In the combined structure, if x and y do not already belong to the same equivalence class, then the equivalence classes of x and y are joined. Notice that given the representation of the combined structure and the way it is used, it is automatically closed reflexively, symmetrically, and transitively.

Assume that E is $\forall x$: E'. Then $\Vdash$ is called with the relation formed from R by deleting all pairs that include x. In the combined structure, a new pointer into the *UFD* structure is created and placed on the top of x's variable stack, thus effectively hiding any pairs involving x; except that the pair (x, x) will belong to the structure ensuring reflexitivity. □

Lemma 8 The function, check, correctly implements $\Vdash$.

Proof From proposition 6, it follows that a function semantically equivalent to the function check can be written by letting the combined structure be a value-passing parameter to check. As the two algorithms are structurally equivalent modulo unfolding, it is sufficient to consider the use of the combined structure and R. From proposition 7, it follows that the combined structure is an exact representation of R^0. □

Theorem 9 Let E be a bisimulation formula, and let n be the size of E. Then the time-complexity of check(E) is $O(n \log n)$.

Proof The algorithm is recursive in the structure of E, and clearly, there are a constant number of statements per symbol in E. These statements either perform constant-time operations, or they operate on one of the data structures. As these are initially empty, and as they share the property that n operations are carried out in time $O(n \log n)$, the result follows. □

5 Extensions of the Algorithm

In this section, we consider various extensions of the algorithm. For each extension, we sketch the modifications of the abstract algorithm from section 3, and we discuss the correctness issues briefly.

Many more extension than the ones presented here are possible. However, we have decided only to present extensions according to the criteria:

a) the asymptotic complexity should not change.
b) the increase in the actual complexity should be very low (less than a factor of 10).
c) it should still be a one pass algorithm.

It is not hard to deal with constants and through the obvious transformation suggested by the equivalence

$$(E \wedge E') \to E'' \quad \dashv\models \quad E \to (E' \to E''),$$

the algorithm can easily cover implication subformulas with conjunctions of predicates to the left. It is straight forward to cope with multiple equivalence relations by letting the function $\Vdash$ work with multiple relations over variables.

The function $\Vdash$ is only able to return **t** (**f**) if the formula is valid (unsatisfiable). However, the algorithm, $\Vdash_c$, obtained from $\Vdash$ by returning the result of $(R \cup \{(x, y)\})^{\oplus} \Vdash_c E'$ in case of formulas of the form $(x \equiv y) \to E'$, is able to deal with *contingent* formulas as well.

Theorem 10 a) If $\Vdash_c E = \mathbf{t}$ then $\models E$. b) If $\Vdash_c E = \mathbf{f}$ then $\not\models E$.

Proof The proof of a) is almost exactly as the corresponding proof of theorem 5. Part b) is proved by showing that if $R \Vdash_c E = \mathbf{f}$ then E it is not satisfied by environments identifying all variables. □

$\Vdash_c$ is clearly as good as $\Vdash$. It is also strictly better because $\Vdash x \equiv y \to \mathsf{F} = \mathbf{u}$ and $\Vdash_c x \equiv y \to \mathsf{F} = \mathbf{f}$.

6 Simplifications

In this section, we discuss changes to the algorithm with the purpose of outputting a simplified formula equivalent to the original formula. The algorithm should contain the validity checking algorithm as a special case, i.e., if the validity checking algorithm deems a formula valid, then this new algorithm should simplify the formula to T. Also, we would like the algorithm to fulfill the criteria of the previous section.

Like the $\VDash$ function, the new function, $\VDash_r$, takes as arguments a relation, R, over variables and a first order formula, but now it returns a first order formula instead of a truth-value of three-valued logic. We use the same case analysis, but turn the Kleene truth tables into simplification tables, essentially by replacing **u** by the argument formula. Compare with the Kleene truth tables in section 3. However, this is not quite sufficient. In the $\forall x\colon\ E'$ case of $\VDash$, $\forall x$ is eliminated completely. This cannot be done here when E' does not simplify to T or F, so a simplification table for $\forall$ is also needed.

$\wedge_r$	T	F	E'
T	T	F	E'
F	F	F	F
E	E	F	$E \wedge E'$

$\vee_r$	T	F	E'
T	T	T	T
F	T	F	E'
E	T	E	$E \vee E'$

$\rightarrow_r$	T	F	E'
T	T	F	E'
F	T	T	T
E	T	$E \rightarrow \mathsf{F}$	$E \rightarrow E'$

$\forall_r\, x\colon$	
T	T
F	F
E'	$\forall x\colon\ E'$

We are now ready to define $\VDash_r$.

$$
R \VDash_r E \textbf{ is case } E \textbf{ of } \quad
\begin{array}{rl}
x \equiv y & : \textbf{if } x\, R^0\, y \textbf{ then } \mathsf{T} \textbf{ else } x \equiv y \\
E' \wedge E'' & : R \VDash_r E' \wedge_r R \VDash_r E'' \\
E' \vee E'' & : R \VDash_r E' \vee_r R \VDash_r E'' \\
E' \rightarrow E'' & : R \VDash_r E' \rightarrow_r \mathbf{Upd}(R, E') \VDash_r E'' \\
\forall x\colon\ E' & : \forall_r\, x\colon\ R \setminus x \VDash_r E' \\
E & : E
\end{array}
$$

The final case deals with T and F and for convenience, we have introduced an explicit update function:

$$
\mathbf{Upd}(R, E) = \begin{cases} (R \cup \{(x, y)\})^{\oplus}, & \text{if } E \text{ is } x \equiv y \\ R, & \text{otherwise} \end{cases}
$$

The simplified formula is logically equivalent with the original as stated in the following correctness theorem.

Theorem 11 E if and only if $\VDash_r E$.

Proof This follows from the statement below which is proved by induction. We omit the details.

$$
R_{\equiv} \rightarrow E \text{ iff } R_{\equiv} \rightarrow (R \VDash_r E) \qquad \square
$$

The next proposition expresses that $\VDash_r$ is at least as good as $\VDash$.

Proposition 12 If $\VDash E = \mathbf{t}$ ($\mathbf{f}$), then $\VDash_r E = \mathsf{T}$ (F).

The next section contains examples of simplifications using this algorithm.

A straightforward improvement of the simplification algorithm can be obtained from the semantic equivalence

$$E' \wedge E'' \quad \dashv\models \quad E' \wedge (E' \to E''). \tag{1}$$

Exploiting the simplification of E', the conjunction case is changed to:

$$\textbf{let } E'_r = R \Vdash_r E' \textbf{ in } E'_r \wedge_r \textbf{Upd}(R, E'_r) \Vdash_r E''.$$

In this way, the algorithm can simplify $(F \vee x \equiv y) \wedge (x \equiv y \to F)$, for example, to F.

Along the same lines, the algorithm can be improved further by using:

$$E \vee E' \quad \dashv\models \quad (\neg E) \to E'.$$

Writing $x \not\equiv y$ for the common occurring formula $x \equiv y \to \mathsf{F}$, we get as a special case:

$$x \not\equiv y \vee E \quad \dashv\models \quad x \equiv y \to E. \tag{2}$$

As the algorithm is formulated now, there is a priori nothing that prevents the algorithm from working with more predicates such as $x \not\equiv y$ and $x \leq y$. In fact, the simplification algorithm is still sound since the new predicates are not simplified and do not give rise to updates of R through $\textbf{Upd}(\_,\_)$. However, we can use R to simplify the new predicates in some cases, e.g., for $x \not\equiv y$, we can add the case

$$\textbf{if } x\ R^0\ y \textbf{ then } \mathsf{F} \textbf{ else } x \not\equiv y.$$

Now, we turn our attention to another type of simplification. The idea is that universal quantifications can be pushed inwards over conjunctions and that quantified predicates in some cases then can be simplified.

We use this observation to maintain a set, X, of variables corresponding to universal quantified variables met solely by simplification of conjunctions, and define a function $\Vdash_{re}$, which, compared to $\Vdash_r$, takes X as an extra argument.

$$\begin{array}{rrl} R \Vdash_{re}^{X} E \textbf{ is case } E \textbf{ of} & x \equiv y & : \textbf{if} \quad x\ R^0\ y \quad\quad\quad\quad \textbf{then } \mathsf{T} \\ & & \textbf{elseif } x \in X \ \textbf{ or } \ y \in X \ \textbf{then } \mathsf{F} \\ & & \textbf{else} \quad\quad\quad\quad\quad\quad\quad\quad\quad x \equiv y \\ & E' \wedge E'' & : R \Vdash_{re}^{X} E' \wedge_r R \Vdash_{re}^{X} E'' \\ & E' \vee E'' & : R \Vdash_{re}^{\emptyset} E' \vee_r R \Vdash_{re}^{\emptyset} E'' \\ & E' \to E'' & : R \Vdash_{re}^{\emptyset} E' \to_r \textbf{Upd}(R, E') \Vdash_{re}^{\emptyset} E'' \\ & \forall x\colon\ E' & : \forall_r\, x\colon\ R \setminus x \Vdash_{re}^{X \cup \{x\}} E' \\ & E & : E \end{array}$$

The soundness of $\Vdash_{re}$ follows from

$$\begin{array}{rcl} \forall x\colon\ E \wedge E' & \dashv\models & (\forall x\colon\ E) \wedge (\forall x\colon\ E') \\ \forall x\colon\ x \equiv y & \dashv\models & \mathsf{F}. \end{array}$$

Actually, the soundness of the latter requires the quotient set of the domain by the equivalence, $D/{\equiv_D}$, to have a size of at least two. However, for empty or singleton quotient sets,

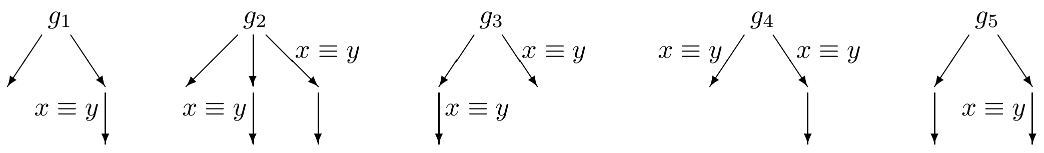
do not seem very useful, so the restriction should not be significant in practice.

Notice that with the exception of the extension concerning contingent formulas, all extensions can all be combined.

7 Examples

In the first half of this section, we focus on qualitative aspects of the simplification algorithm by means of five examples used to illustrate different simplification ideas. In the second half, we deal with some quantitative aspects of the simplification algorithm and the Kleene algorithm through time measures of concrete implementations applied to increasingly larger input.

Consider the following symbolic transition graphs:



All actions are internal, so τ has been omitted from the graphs together with the trivial guarding conditions T. Before proceeding, we invite the reader to try to see which graphs are bisimilar.

Now, applying $\models_r$ to the bisimulation formula mgb_{g_i,g_j} ($\leftrightarrow mgb_{g_j,g_i}$), we get the table of simplified formulas

$_i{}^j$	1	2	3	4
2	T			
3	$x \not\equiv y \vee x \equiv y$	$x \not\equiv y \vee x \equiv y$		
4	$x \equiv y \wedge E$	$x \equiv y \wedge E$	E	
5	$x \equiv y \wedge x \not\equiv y$	$(x \equiv y \vee x \equiv y)$ $\wedge\, x \not\equiv y$	$x \equiv y \wedge x \not\equiv y$	$x \equiv y \wedge E$ $\wedge\, x \not\equiv y$

where $E = ((x \equiv y \wedge x \not\equiv y) \vee (x \equiv y \wedge x \equiv y))$, and for sake of readability, $x \equiv y \rightarrow \mathsf{F}$ is written $x \not\equiv y$.

If, in stead, we apply $\models_r$ with the modifications corresponding to (1), many of the formulas are simplified considerably, some even completely as shown in table below to the left.

$_i{}^j$	1	2	3	4
2	T			
3	$x \not\equiv y \vee x \equiv y$	$x \not\equiv y \vee x \equiv y$		
4	$x \equiv y$	$x \equiv y$	$x \equiv y$	
5	F	$(x \equiv y \vee x \equiv y)$ $\wedge\, x \not\equiv y$	F	F

$_i{}^j$	1	2	3	4
2	T			
3	$x \not\equiv y \vee x \equiv y$	$x \not\equiv y \vee x \equiv y$		
4	F	F	F	
5	F	$(x \equiv y \vee x \equiv y)$ $\wedge\, x \not\equiv y$	F	F

If we are interested in knowing whether the graphs are bisimilar under all instantiations of x and y, we can check validity of the universal closure of the formulas, i.e. simplify the universally closed formulas to T or F if possible. The result of applying $\Vdash_{re}$ (with the modifications mentioned above) yields the table above to the right. We have omitted the quantifiers in the formulas different from T and F.

Adding to $\Vdash_{re}$ an extra case for $\not\equiv$, the formula in entry $(i, j) = (5, 2)$ would also simplify to F, and if the modification suggested from (2), i.e., transforming $x \not\equiv y \vee E$ to $x \equiv y \rightarrow E$, is incorporated into the algorithms as well, then the last two entries would also simplify completely, but this time to T.

Turning to the quantitative aspects of the concrete algorithms, we consider processes defined for $i \geq 0$ by

$$p_{i+2} \xrightarrow{\mathsf{T},\alpha?x_0} q_{i+1}, \qquad q_{i+1} \xrightarrow{\mathsf{T},\alpha?x_1} r^i_1, \qquad r^{i+1}_k \xrightarrow{c_k,\alpha?x_{k+1}} r^i_{k+1} \qquad \text{and} \qquad r^0_k \xrightarrow{c_k,\beta!x_k} \mathbf{0},$$

where c_k is the equality $x_{k-1} = x_k$. Initially two values are unconditionally received on α and then, iteratively, values are received on α provided the two most resently received values are equal. Finally, after i iterations and under the same proviso, the last value is send on β. Similarly, we define primed versions which only differ in that c'_k is the equality $x'_0 = x'_k$. That is, the last value received on α is compared with very first.

In order to give the reader examples of how concrete bisimulation formulas look like, we now describe the most general boolean, $mgb_{r^i_k, r'^i_k}$, characterizing those instantiations (environments) of r^i_k and r'^i_k for which they are late bisimilar.

$$mgb_{r^{i+1}_k, r'^{i+1}_k} = \bigwedge \begin{array}{l} c_k \rightarrow (c'_k \wedge \forall x_{k+1}\colon\ \forall x'_{k+1}\colon\ x_{k+1} = x'_{k+1} \rightarrow mgb_{r^i_{k+1}, r'^i_{k+1}}) \\ c'_k \rightarrow (c_k \wedge \forall x'_{k+1}\colon\ \forall x_{k+1}\colon\ x'_{k+1} = x_{k+1} \rightarrow mgb_{r^i_{k+1}, r'^i_{k+1}}) \end{array}$$

$$mgb_{r^0_k, r'^0_k} = \bigwedge \begin{array}{l} c_k \rightarrow (c'_k \wedge x_k = x'_k \wedge \mathsf{T}) \\ c'_k \rightarrow (c_k \wedge x'_k = x_k \wedge \mathsf{T}) \end{array}$$

The most general booleans for the p's and q's are similar to the first formula above, except that the conditions here are T.

For $2 \leq i \leq 13$, we have measured the average time of five runs of a C implementation of $\Vdash_c$ ($\Vdash_r$) processing $mgb_{p_{i+2}, p'_{i+2}}$ on a SPARC station ELC. To give a few examples, $mgb_{p_{13+2}, p'_{13+2}}$ of size 5.754 Mb was simplified to $\mathbf{t}$ (T) in 77284 (78912) milliseconds. Similarly, $mgb_{p_{13+2}, p'_{13+3}}$, which is 4.477 Mb large, was simplified to $\mathbf{f}$ in 62958 milliseconds. On average, $\Vdash_c$ and $\Vdash_r$ process input at a rate of about 75 Kbytes per second.

References

[1] W. Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Math. Annalen*, 99:118–133, 1928.

[2] R. Bayer. Symmetric Binary B-Trees. *Acta Inform.*, 1:290–306, 1972.

[3] S.A. Cook. The Complexity of Theorem-Proving Procedures. In *ACM STOC*, pages 151–158, 1971.

[4] U.H. Engberg. Simple Symbolic Bisimulations. In preparation.

[5] U.H. Engberg and K.S. Larsen. Efficient Reduction of Bisimulation Formulas. Preprint 47, Dept. of Math. and Computer Science, Odense University, 1993.

[6] B.A. Galler and M.J. Fischer. An improved equivalence algorithm. *Comm. ACM*, 7:301–303, 1964.

[7] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W. H. Freeman, 1979.

[8] L.J.Guibas and R.Sedgewick. A Dichromatic Framework for Balanced Trees. *IEEE FOCS*, 8–21, 1978.

[9] M. Hennessy and H. Lin. Symbolic Bismulations. Tech. Rep. 1/92, University of Sussex, 1992. To appear in *Theoretical Computer Science*, 1995.

[10] M. Hennessy and H. Lin. Proof systems for message-passing process algebras. *CONCUR '93*, pages 202–216, August 1993.

[11] G.J. Klir and T.A. Folger. *Fuzzy Sets, Uncertainty, and Information.* Prentice-Hall, 1988.

[12] H. Mannila and E. Ukkonen. The set union problem with backtracking. *LNCS 226*, 236–243, 1986.

[13] K. Mehlhorn and A. Tsakalidis. Data Structures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 301–341. Elsevier Science Publishers, 1990.

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

[15] Z. Schreiber. Verification of Value-Passing Systems. In *First North American Process Algebra Workshop*, pages 9.1–9.20. Tech. Rep. 92-15, Johns Hopkins University, 1992.

[16] D. Scott. Notes on the formalization of logic. Technical report, Sub-faculty of Phil., Oxford, 1981.

[17] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *JACM*, 22:215–225, 1975.

[18] R.E. Tarjan. *Data Structures and Network Algorithms.* Soc. for Industrial and Applied Math., 1983.

[19] R.E.Tarjan and J.v.Leeuwen. Worst-Case Analysis of Set Union Algorithms. *JACM*, 31:245–281, 1984.

[20] J. Westbrook and R.E. Tarjan. Amortized analysis of algorithms for set union with backtracking. *SIAM J. Comput.*, 18(1):1–11, 1989.

On Implementing Unique Fixpoint Induction for Value-passing Processes

H. Lin*
Laboratory for Computer Science
Institute of Software, Chinese Academy of Sciences
P.O Box 8718, Beijing 100080
E-mail: lhm@ios.ac.cn

Abstract

We examine the possible pitfalls in formulating the unique fixpoint induction proof rule in the setting of value-passing process calculi and describe how this rule is implemented in the verification tool *VPAM*. An argument is also given to justify the implementation.

1 Introduction

Computer-aided verification of concurrent systems has attracted considerable research efforts in the past decade. Many proof tools based on the theoretical results in process algebra have been developed and widely used: The Concurrency Work Bench [CPS89], TAV [GLZ89], AUTO [SV89], The Pisa tool [DIN89], PAM [Lin91], and LOTOS tools [BBV92], to name just a few. Some of these tools are automatic and work by searching state space generated according to operational semantics. Some are interactive and rely on direct manipulation of syntactic terms. These verification tools are only for *pure* process algebras where data exchange between processes is reduced to synchronization on signals. To apply them to problems involving value-passing one has to first transform the problems into pure calculi. Even for finite value domains such transformation could cause exponential explosion of states. When the value domains are infinite the problems would become infinite and thus beyond the scope of these tools.

The "symbolic" semantic theories for value-passing processes have recently been advocated, with avoiding such infinity in mind [HL92]. "Symbolic" style proof systems have also been formulated, allowing to reason about semantic equivalences at a syntactic level [HL93]. These proof systems are finitary in nature and are therefore

*Supported by the President Fund of Chinese Academy of Sciences, the National Science Foundation of China, and the EU KIT project SYMSEM.

amenable to implementation. In fact we have developed the proof assistant *VPAM* (for Value-passing Process Algebra Manipulator) [Lin93] based on such proof systems. The aim of this paper is to explain the way *VPAM* deals with *recursion*, a process construction needed in any non-trivial problems.

Let us start by considering a simple example (in full-*CCS* syntax):

$$\begin{array}{lcl} P & = & c?x.\ \mathit{if}\ x \geq 0\ \mathit{then}\ P'(x)\ \mathit{else}\ P \\ P'(x) & = & c!x.P \\ \\ Q & = & c?x.\ \mathit{if}\ x \geq 0\ \mathit{then}\ Q'(x)\ \mathit{else}\ Q \\ Q'(x) & = & c!|x|.Q \end{array}$$

where the data domain is integer and $|x|$ means the absolute value of x.

Clearly $P = Q$ when $=$ is interpreted as bisimulation equivalence. What kind inference rules do we need in order to infer $P = Q$? Since P, Q are recursively defined, some form of induction is required. The most-widely used induction rule in the process algebra community is *unique fixpoint induction*. In pure process algebras this rule has the form:

$$\frac{S_i = T_i[S_k/R_k|1 \leq k \leq m], \quad 1 \leq i \leq m}{S_i = R_i} \quad \begin{array}{l} R_i = T_i \\ \tilde{R} \text{ guarded in } \tilde{T} \end{array} \tag{1}$$

The premises of this rule verify that $\tilde{S}$ is a fixpoint of $\tilde{T}$. By definition $\tilde{S}$ is also a fixpoint of $\tilde{T}$. Since $\tilde{T}$ is guarded it has only one fixpoint. Hence the conclusion. But this rule is not applicable to our example because it does not deal with data variables and data expressions. To solve our problem we need to generalise it to the setting of value-passing processes.

Rule (1) uses process substitution $[S_k/R_k|1 \leq k \leq m]$ where S_k:s are process terms and R_k:s process identifiers. When applied to a term T it replaces R_k with S_k. This is fine for pure calculi because in such calculi a process identifier is a process term on its own-right, so replacing R_k with S_k in T is guaranteed to result in a syntactically well-formed term. In value-passing calculi, however, process identifiers may take data parameters and process terms may contain free data variables, so we can not simply substitute terms for identifiers. For instance applying substitution $[P'(x)/Q']$ to term $c?y.Q'(y)$ would result in $c?y.P'(x)(y)$ which is syntactically ill-formed. What is needed here is a device to take care of correct parameter-passing for substitutions. For this we can use lambda *abstraction* to close up substituting terms and *application* to realise parameter-passing: $T[\lambda\tilde{x}A/B]$, where $\tilde{x}$ is the set of free data variables of A, is the result of substituting $\lambda\tilde{x}A$ for identifier B in T followed by β-conversion, so that the $\tilde{x}$ in A get replaced by the actual parameters of B-occurences. For example

$$\begin{array}{ll} & (\mathit{if}\ x \geq 0\ \mathit{then}\ c!x.P(x)\ \mathit{else}\ c?z.P(z))[\lambda y Q(y)/P] \\ = & \mathit{if}\ x \geq 0\ \mathit{then}\ c!x.(\lambda y Q(y))(x)\ \mathit{else}\ c?z.(\lambda y Q(y))(z) \\ = & \mathit{if}\ x \geq 0\ \mathit{then}\ c!x.Q(x)\ \mathit{else}\ c?z.Q(z) \end{array}$$

With this notation a naive generalization of (1) to the value-passing setting might look like

$$\frac{S_i = T_i[\lambda\tilde{x}_k S_k/R_k | 1 \leq k \leq m], \quad 1 \leq i \leq m}{S_i = R_i(\tilde{x}_i)} \quad \begin{array}{l} R_i(\tilde{x}_i) = T_i \\ \tilde{R} \text{ guarded in } \tilde{T} \end{array} \tag{2}$$

Applying this rule to $P = Q$ in our example requires to prove the following two equations

$$\begin{array}{lcl} P & = & c?x.\ \mathit{if}\ x \geq 0\ \mathit{then}\ P'(x)\ \mathit{else}\ P \\ P'(x) & = & c!|x|.P \end{array}$$

The first equation can be settled immediately, but the second gives rise to (after unfolding $P'(x)$)

$$c!x.P = c!|x|.P$$

which can not (and should not) be proved.

It is not difficult to see what went wrong: $P'(x)$ is not bisimilar to $Q'(x)$ for *all values* of x; they are bisimilar only when $x \geq 0$ (i.e. $P'(x) \sim^{x\geq 0} Q'(x)$, using symbolic bisimulation notation). This is sufficient to establish the bisimilarity between P and Q, because in their definitions $P'(x)$ and $Q'(x)$ *appear only in contexts satisfying* $x \geq 0$.

To take such context information into account, instead of pure equations we use *conditional* equations of the form

$$b \rhd T = U$$

where b is a boolean expression on the value domain, as judgments of our inference system. Intuitively it means T and U are equivalent under the constraint b on the free data variables appearing in them. When $b \equiv \mathit{true}$ it becomes unconditional and can be abbreviated as $T = U$. The fact that $P'(x)$ and $Q'(x)$ are bisimilar over $x \geq 0$ can be expressed as

$$x \geq 0 \rhd P'(x) = Q'(x).$$

With conditional equation we could reformulate (2) as

$$\frac{b_i \rhd S_i = T_i[\lambda\tilde{x}_k S_k/R_k | 1 \leq k \leq m], \quad 1 \leq i \leq m}{b_i \rhd S_i = R_i(\tilde{x}_i)} \quad \begin{array}{l} R_i(\tilde{x}_i) = T_i \\ \tilde{R} \text{ guarded in } \tilde{T} \end{array} \tag{3}$$

Unfortunately this is unsound. Here is a counter-example: Let

$$\begin{array}{lcl} P_1 & = & c?x.\ \mathit{if}\ x \geq 0\ \mathit{then}\ P_1'(x)\ \mathit{else}\ P_1 \\ P_1'(x) & = & c!x.c?y.P_1'(y) \\ & & \\ Q_1 & = & c?x.\ \mathit{if}\ x \geq 0\ \mathit{then}\ Q_1'(x)\ \mathit{else}\ Q_1 \\ Q_1'(x) & = & c!|x|.c?y.Q_1'(y) \end{array}$$

Using (3) one can derive

$$x \geq 0 \rhd P_1'(x) = Q_1'(x) \tag{4}$$

This is wrong because although the first outputs from $P'_1(x)$ and $Q'_1(x)$, which are x and $|x|$ respectively, are equal due to the condition $x \geq 0$, after the input action $c?y$ further outputs from the two processes, which are y and $|y|$ now, can no longer match under this condition. This time the problem is with the premises of (3): The conclusion only asserts $S_i = R_i(\tilde{x}_i)$ *under the condition* b_i, but in the premise $\lambda\tilde{x}_i S_i$ are substituted for R_i *unconditionally.*

How to express the idea that S_i is substituted for $R_i(\tilde{x}_i)$ only over the value space constrained by b_i? We can use $\lambda\tilde{x}_i(b_i S_i)$ ($b_i S_i$ is a shorthand for *if* b_i *then* S_i *else NIL*) instead of just $\lambda\tilde{x}_i S_i$ as the substituting abstraction. This ensures that S_i will be constrained by b_i after substitution. With this observation one might change rule (3) to

$$\frac{b_i \rhd S_i = T_i[\lambda\tilde{x}_k(b_k S_k)/R_k | 1 \leq k \leq m], \quad 1 \leq i \leq m}{b_i \rhd S_i = R_i(\tilde{x}_i)} \quad \begin{array}{l} R_i(\tilde{x}_i) = T_i \\ \tilde{R} \text{ guarded in } \tilde{T} \end{array} \qquad (5)$$

Now (4) is no longer derivable. But the new rule creates a different problem. To see this let us change the definition clause for $P'_1(x)$ in the above example to

$$P'_1(x) = c!x.c?y.\mathit{if}\ y \geq 0\ \mathit{then}\ P'_1(y)\ \mathit{else}\ NIL$$

Apparently $P'_1(x)$ and $Q'_1(x)$ are not bisimilar over $x \geq 0$. But applying rule (5) to the goal $x \geq 0 \rhd P'_1(x) = Q'_1(x)$ results in the subgoal

$$\begin{aligned} x \geq 0 \rhd P'_1(x) &= (c!|x|.c?y.Q'_1(y))[\lambda x(x \geq 0 P'_1(x))/Q'_1] \\ &\equiv c!|x|.c?y.\mathit{if}\ y \geq 0\ \mathit{then}\ P'_1(y)\ \mathit{else}\ NIL \end{aligned}$$

which can be proved after unfolding $P'_1(x)$ with its new definition. So what is wrong with rule (5)? The problem is that the premises of this rule only checks whether $b_i S_i$:s constitute a fixpoint of T_i:s over b_i, but does not do so for R_i:s. Though by definition $\tilde{R}$ is a fixpoint of $\tilde{T}$, this does not imply that $b_i R_i$:s constitute a fixpoint of T_i:s over b_i:s. In this example one does not have

$$\begin{aligned} x \geq 0 \rhd Q'_1(x) &= (c!|x|.c?y.Q'_1(y))[\lambda x(x \geq 0 Q'_1(x))/Q'_1] \\ &\equiv c!|x|.c?y.\mathit{if}\ y \geq 0\ \mathit{then}\ Q'_1(y)\ \mathit{else}\ NIL \end{aligned}$$

To make the rule sound we must check such a "partial fixpoint" property for $\tilde{R}$ as well. This can be achieved by employing more premises:

$$\frac{\begin{array}{l} b_i \rhd S_i = T_i[\lambda\tilde{x}_k(b_k S_k)/R_k | 1 \leq k \leq m], \\ b_i \rhd R_i(\tilde{x}_i) = T_i[\lambda\tilde{x}_k b_k R_k/R_k | 1 \leq k \leq m], \end{array} \quad 1 \leq i \leq m}{b_i \rhd S_i = R_i(\tilde{x}_i)} \quad \begin{array}{l} R_i(\tilde{x}_i) = T_i \\ \tilde{R} \text{ guarded in } \tilde{T} \end{array} \qquad (6)$$

This rule is sound, but has the disadvantage of doubling the number of premises. This is particularly unsatisfactory from the implementation point of view. The *VPAM* implementation has avoided such duplication by exploiting "context conditions" implicitly associated with process identifiers. In the rest of this paper we shall explain how this can be achieved.

The next section briefly describes how proofs are constructed in *VPAM*, setting up the necessary background for further discussion. Section 3 explains the implementation of unique fixpoint induction in *VPAM*. It also includes a correctness proof for the implementation. The paper is concluded with Section 4.

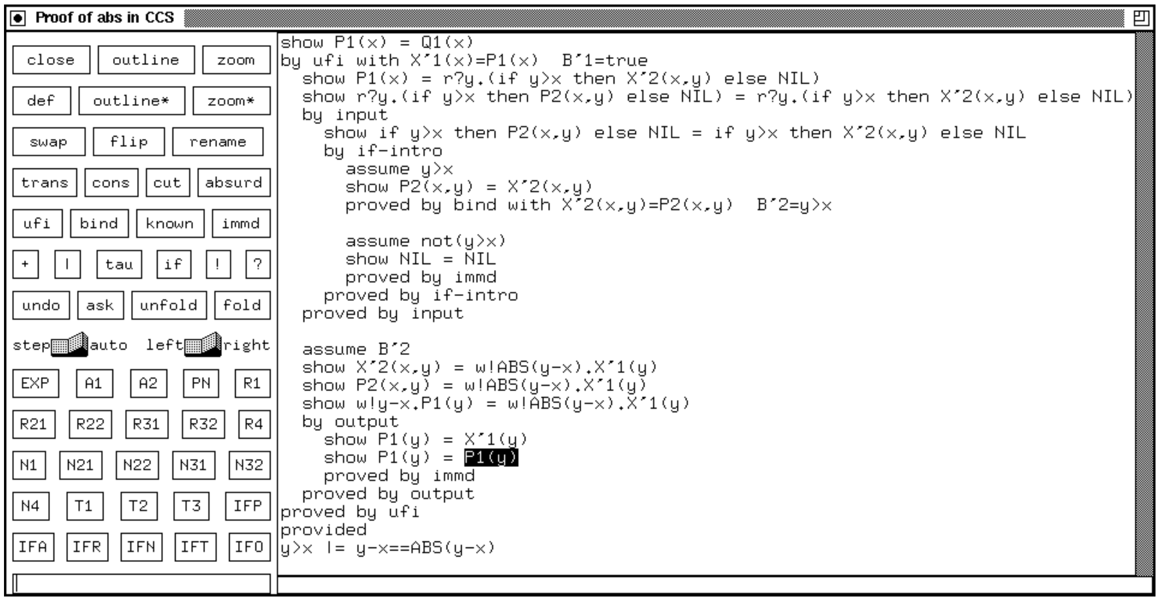
Some familiarity with value-passing process calculi such as full-*CCS* ([Mil89]) is assumed.

2 Proofs in *VPAM*

VPAM is an interactive verification tool for value-passing process calculi. It is an extension of *PAM* ([Lin91]) which can only deal with pure process algebras. *VPAM* implements a "symbolic" style proof system for value-passing processes, extending those proposed in [Hen91, HL93] for recursion-free languages with a version of unique fixpoint induction for value-passing calculi. The proof system consists of a set of inference rules together with some standard equational axioms. The inference rules, as listed in the Appendix A, are built into *VPAM* while axioms are provided by the users, allowing different equivalences to be reasoned about. The syntax of the calculus one wants to work with is also definable. Thus *VPAM* is a *parameterisable* tool. Both the syntax and axioms of a calculus are defined in a single file which can be compiled in *VPAM*'s top-level window. We will not get into details of calculus definition and compilation here. The interested readers are referred to [Lin93]. In the following discussion we will use full-*CCS* ([Mil89]) as the example language. So `.` is action prefixing, `+` external choice, `|` parallel composition, `if _ then _ else _` conditional, `c?x` input action, `c!e` output action, *NIL* the empty process, *etc.*. The full-*CCS* axioms, as defined in *VPAM*, are listed in Appendix B. We shall write $\vdash b \rhd T = U$ if $b \rhd T = U$ can be derived from the proof system. The set of free data variables of T will be denoted by $fv(T)$.

The problems to be submitted to *VPAM* for verification are described in *problem definition files*. An example follows:

```
function

      ABS _ :: Int -> Int

process

      P1 :: Int
      Q1 :: Int
      P2 :: Int Int
      Q2 :: Int Int

channel  r  w

conjecture
```



```
show P1(x) = Q1(x)
by ufi with X′1(x)=P1(x)  B′1=true
  show P1(x) = r?y.(if y>x then X′2(x,y) else NIL)
  show r?y.(if y>x then P2(x,y) else NIL) = r?y.(if y>x then X′2(x,y) else NIL)
  by input
    show if y>x then P2(x,y) else NIL = if y>x then X′2(x,y) else NIL
    by if-intro
      assume y>x
      show P2(x,y) = X′2(x,y)
      proved by bind with X′2(x,y)=P2(x,y)  B′2=y>x

      assume not(y>x)
      show NIL = NIL
      proved by immd
    proved by if-intro
  proved by input

  assume B′2
  show X′2(x,y) = w!ABS(y-x).X′1(y)
  show P2(x,y) = w!ABS(y-x).X′1(y)
  show w!y-x.P1(y) = w!ABS(y-x).X′1(y)
  by output
    show P1(y) = X′1(y)
    show P1(y) = P1(y)
    proved by immd
  proved by output
proved by ufi
provided
y>x |= y-x==ABS(y-x)
```

Figure 1: A Proof Window

```
        |- P1(x) = Q1(x)

where
        P1(x) = r?y. if y > x then P2(x,y) else NIL
        P2(x,y) = w!(y-x).P1(y)
        Q1(x) = r?y. if y > x then Q2(x,y) else NIL
        Q2(x,y) = w!ABS(y-x).Q1(y)
end
```

`Int` is a built-in type with the usual operators `>`, `<`, `+`, `-`, *etc.*. `ABS` is the absolute value function. `P2 :: Int Int` declares that `P2` is a process identifier with two integer parameters. Data parameters to a process identifier must be distinct variables. Recursive definitions are listed in the `where` part. The statement to prove is given after the keyword `conjecture`.

Given such a problem definition file *VPAM* creates a *proof window* for it (Figure 1). The left half of a proof window (see Figure 1) is a command panel where there is a command button for each inference rule. For example the buttons $\boxed{\text{if}}$, $\boxed{?}$ and $\boxed{!}$ correspond, respectively, to the rules IF, INPUT and OUTPUT in the Appendix A. Two switches (*auto-step* and *left-right*) control the behavior of the rewrite buttons (below them) which are labeled by axiom names appearing in the calculus definition file. Rules and axioms are invoked when the corresponding buttons are pressed. Proofs are displayed on the right subwindow. A conditional equation $b \rhd T = U$ is displayed as

```
    assum b
```

```
show T = U
```

As the inference system underlining *VPAM* consists of inference rules as well as equations, proofs in *VPAM* are displayed in a mixed way: invoking inference rules is goal-directed, while applying equations is implemented as rewriting (and does not generate subgoals). Some inference rules of this proof system use the semantics implication relation $\models$ between boolean conditions. These are where we appeal to an "oracle" to answer questions about the data domain. This is is a consequence of the fact that our process languages are parameterised on languages for data and boolean expressions. The inference system treats these expressions "symbolically", i.e. as uninterpreted strings. In fact, one of the main design ideas behind the proof system is to separate data reasoning from process reasoning as much as possible. For example, in order to infer $x = y \rhd c!(x - y).P(x) = c!0.P(x)$ we need to know $x = y \models x - y = 0$. Since we do not want to be involved in reasoning in any particular data domain (integer in this case), this question is left to the oracle. In *VPAM* the proof commands corresponding to these rules, such as `output` and `absurd`, will generate *proof obligations* appended at the end of the proof, after the keyword `provided`. Such proof obligations represent an interface to an auxiliary data domain theorem prover which "implements" the oracle.

In a proof window there is always a unique term, called the *current term*, which is highlighted. The equation and the subgoal containing the current term are called the *current equation* and *current goal* respectively. Each goal has an *assumption* which is a list of booleans (interpreted as connected by conjuncts). The *current global assumption* is the conjunction of the assumption of the current goal together with the assumptions of *all its ancestors*. *Goal equation* is the first equation of a goal.

3 UFI in VPAM

3.1 The Implementation

In *VPAM* the unique fixpoint induction is implemented by three related proof commands: **ufi**, **bind** and **known**. We assume parameters of any process identifiers are distinct data variables.

To apply **ufi**, the right hand side of the current equation must be a recursively defined identifier, possibly with parameters. For easy reference suppose the current conditional equation is

$$b_1 \rhd S_1 = R_1(\tilde{x}_1) \qquad (7)$$

and the definition clauses relevant to R_1 are

$$R_i(\tilde{x}_i) = T_i \qquad 1 \leq i \leq m.$$

where the set of identifiers appearing in the T_i:s is $\{ R_i \mid 1 \leq i \leq m \}$. When the **ufi** command is invoked the system automatically generates a pair of "unknowns" for each of these identifiers, one is "process unknown" of the form `X'n` and the other is "boolean unknown" of the form `B'n`, where `n` is a number. `X'1` is bound to the left hand side of the current equation and `B'1` to the global assumption. A subgoal is created for each of the relevant definition clauses in the following manner: The right hand side of the ith subgoal equation is the result of substituting $\lambda\tilde{x}_1 S$ for R_1 and X_k for R_k, $2 \leq k \leq m$, in T_i. The left hand side of the first subgoal equation is S, and the left hand side of the ith subgoal equation for $i \geq 2$ is $X'_i(\tilde{x}_i)$. The goal assumption of the first subgoal is empty and the ith subgoal assumption is B'_i. So for the absolute-value problem in the previous section, applying **ufi** to the top level goal `P1(x) = Q1(x)` will result in the following configuration:

```
show P1(x) = Q1(x)
by ufi with X'1(x)=P1(x) B'1=true
  show P1(x) = r?y.if y>x then X'2(x,y) else NIL

  assume B'2
  show X'2(x,y) = w!ABS(y-x).X'1(y)
```

The "unknowns" generated by the **ufi** command must be bound to some terms for the proof to be completed. In fact the **ufi** command does the initial binding: it binds R_1 to $X'1$ and the current global assumption to $B'1$. This is the "first push" allowing the proof to go ahead. Other "unknowns" must be bound using the **bind** command. To apply **bind**, one side of the current equation must be a "process unknown" which has not been bound before, and the other side must not contain any unknowns. **bind** binds this "process unknown", say $X'n(\widetilde{(x)}_n)$, to the term at the other side of the equation. At the same time it also binds $B'n$, the "boolean unknown" associated with $X'n$, to the current global assumption. This boolean will be referred as the *binding condition* of $X'n$.

An "unknown" that has been bound by **bind** can be "made known" using the command **known**. Similar to the case of **bind**, to apply **known** the current side of the current equation must be a "process unknown". **known** then replaces this "unknown" with the term previously bound to it by **bind**. At the same time **known** also generates a proof obligation which asserts that the current global assumption must imply the binding condition of this unknown. More precisely, suppose the current term is $X'n(\tilde{x}')$, and $X'n(\tilde{x}_n)$ and $B'n$ have previously bound to Sn and b, respectively. Then **known** will unfold $X'n(\widetilde{(x)}')$ to $Sn[\tilde{x}'/\tilde{x}_n]$ and generate $b' \models b[\tilde{x}'/\tilde{x}_n]$ as a proof obligation, where b' is the current global assumption. As said in the previous paragraph the **ufi** command automatically binds $X'1$ to R_1. It also automatically makes $X'1$ known: that is why the left hand side of the first subgoal equation is R_1 instead of $X'1$. The proof obligation of this implicit application of **known** is "the current global assumption implies the current global assumption" which holds trivially (hence is not generated).

3.2 Correctness of the Implementation

The *VPAM* implementation of the unique fixpoint induction, as described above, generates only m subgoals for the goal (7) which involves m mutual-recursive definition clauses while the unique fixpoint induction rule (6) has twice as many premises. In fact *VPAM* does not generates subgoals to check the context conditions of the identifiers, i.e. the m premises

$$b_i \rhd R_i(\tilde{x}_i) = T_i[\lambda\tilde{x}_k b_k R_k / R_k|k] \quad 1 \leq i \leq m$$

in rule (6). They are validated as a by-product. In this subsection we shall justify that the three commands **ufi**, **bind** and **known** correctly implement rule (6).

By α-conversion in the following discussions we assume different input actions in a term use different variables which are also different from the free variables of the term. We also assume the "unknowns" X'_n and B'_n are not used in the problem definitions.

An atom is a process term of the form $P(\tilde{x})$ where P is an identifier or an unknown. An occurrence of an atom A in a process term T is designated by a box surrounding it. When $A \equiv P(\tilde{x})$ we also call $\boxed{\text{A}}$ a P-occurrence. The *context condition* of an occurrence $\boxed{\text{A}}$ in T, denoted $C_T^{\boxed{\text{A}}}$, is defined inductively thus

- $C_{\boxed{\text{A}}}^{\boxed{\text{A}}} = true$
- $C_{if\ b\ then\ T\ else\ U}^{\boxed{\text{A}}} = \begin{cases} b \wedge C_T^{\boxed{\text{A}}} & \text{if } \boxed{\text{A}} \text{ in } T \\ \neg b \wedge C_U^{\boxed{\text{A}}} & \text{if } \boxed{\text{A}} \text{ in } U \end{cases}$
- $C_{\alpha.T}^{\boxed{\text{A}}} = C_T^{\boxed{\text{A}}}$
- $C_{T+U}^{\boxed{\text{A}}} = \begin{cases} C_T^{\boxed{\text{A}}} & \text{if } \boxed{\text{A}} \text{ in } T \\ C_U^{\boxed{\text{A}}} & \text{if } \boxed{\text{A}} \text{ in } U \end{cases}$
- $C_{T|U}^{\boxed{\text{A}}} = \begin{cases} C_T^{\boxed{\text{A}}} & \text{if } \boxed{\text{A}} \text{ in } T \\ C_U^{\boxed{\text{A}}} & \text{if } \boxed{\text{A}} \text{ in } U \end{cases}$
- $C_{T[f]}^{\boxed{\text{A}}} = C_T^{\boxed{\text{A}}}$
- $C_{T\backslash c}^{\boxed{\text{A}}} = C_T^{\boxed{\text{A}}}$

Intuitively, $C_T^{\boxed{\text{A}}}$ represents the boolean condition (implicitly) holds at $\boxed{\text{A}}$ in T. The usefulness of this notion lies in the fact that one can safely insert any boolean expression weaker than it in front of $\boxed{\text{A}}$ without change the behaviour of T, because such an expression will eventually be "absorbed" by $C_T^{\boxed{\text{A}}}$. More generally, a boolean expression can be added at every P-occurrence if it is implied by the context conditions of all such occurrences (module renaming of free data variables). For example, let T denote the term

$$\begin{array}{l} c!x.\mathit{if}\ x \geq 0\ \mathit{then}\ P(x)\ \mathit{else}\ Q\ + \\ d?y.\mathit{if}\ y \geq 1\ \mathit{then}\ P(y)\ \mathit{else}\ Q \end{array}$$

It is straightforward to calculate that $C_T^{\boxed{\text{P(x)}}} = x \geq 0$ and $C_T^{\boxed{\text{P(y)}}} = y \geq 1$. Consider the boolean $z \geq 0$. We have $x \geq 0 \models (z \geq 0)[x/z]$ and $y \geq 1 \models (z \geq 0)[y/z]$. Also

$$\begin{array}{rl} T[\lambda z(z \geq 0)P/P] \equiv & c!x.\mathit{if}\ x \geq 0\ \mathit{then}\ (\mathit{if}\ x \geq 0\ \mathit{then}\ P(x)\ \mathit{else}\ NIL)\ \mathit{else}\ Q\ + \\ & d?y.\mathit{if}\ y \geq 1\ \mathit{then}\ (\mathit{if}\ y \geq 0\ \mathit{then}\ P(y)\ \mathit{else}\ NIL)\ \mathit{else}\ Q \end{array}$$

It is easy to check that $\vdash T[\lambda z(z \geq 0)P/P] = T$.

In general, we have the following lemma:

Lemma 3.1 *If $C_T^{\boxed{\text{A}}} \models b$ then $\vdash T = T[bA/\boxed{A}]$.*

Proof: Straightforward induction on the structure of T. □

Corollary 3.2 *Let b be a boolean with $fv(b) \subseteq \{\tilde{x}'\}$, P an identifier or unknown. If $C_T^{\boxed{\text{A}}} \models b[\tilde{x}/\tilde{x}']$ for every P-occurrence $A \equiv P(\tilde{x})$ in T, then $\vdash T = T[\lambda\tilde{x}'bP/P]$.*

Applying `ufi` to goal (7) generates the following subgoals:

$$B_i' \rhd X_i'(\tilde{x}_i) = T_i', \qquad 1 \leq i \leq m \tag{8}$$

where $T_i' \equiv T_i[X_k'/R_k | 1 \leq k \leq m]$, $X_1'(\tilde{x}_1)$ and B_1' are bound to S_1 and b_1, respectively. From the description given in the previous subsection, an "unknown" can only be bound or made known when it is the top level term of one side of the current equation. In this case we say this unknown is *exposed*. The current global assumption when an unknown is exposed is called the *exposure condition* of the exposure. Note that although each exposed unknown has a unique occurrence in some T_i', an occurrence of an unknown in a T_i' may correspond to more than one exposures of this unknown due to applications of IF and CUT rules.

During a proof each term T in an equation $B \rhd U = T$ is associated with a global condition, namely the global assumption, say b, of the equation. We call $b \wedge C_T^{\boxed{\text{A}}}$ the *global context condition* of an atom occurrence $\boxed{\text{A}}$ in T. The following lemma says that such global context conditions are invariants of a proof.

Lemma 3.3 *Let $B \rhd X_j'(\tilde{x}) = T$ be a goal equation and $\boxed{A}$ with $A \equiv X_k'(\tilde{x}')$ an unknown occurrence in T. Suppose during the proof $X_k'(\tilde{x}')$ is exposed n times with exposure conditions b_i, $1 \leq i \leq n$. Then $B \wedge C_T^{\boxed{\text{A}}} = \bigvee_{1 \leq i \leq n} b_i$.*

Proof: By induction on the number of rule applications during the proof. We need only to check that each rule application preserves the global context condition of an unknown, i.e. $B \wedge C_T^{\boxed{\mathrm{A}}} = \bigvee_{1 \leq i \leq n}(B_i \wedge C_{T_i}^{\boxed{\mathrm{A}}})$, $n \in \{1, 2\}$ where $B \rhd U = T$ is a goal and $B_i \rhd U_i = T_i$, $1 \leq i \leq n$, $n \in \{1, 2\}$ are the subgoal(s) containing $\boxed{\mathrm{A}}$ generated from the goal by a rule application. For this purpose the only interesting rules are IF and CUT.

For the IF rule there are two cases. If T is of the form if b then T' else T'' then the two subgoals are $B \wedge b \rhd U = T'$ and $B \wedge \neg b \rhd U = T''$. If $\boxed{\mathrm{A}}$ is in T' then it is not in T'', and

$$B \wedge C_T^{\boxed{\mathrm{A}}} = B \wedge b \wedge C_{T'}^{\boxed{\mathrm{A}}}$$

If $\boxed{\mathrm{A}}$ is in T'' then it is not in T', and

$$B \wedge C_T^{\boxed{\mathrm{A}}} = B \wedge \neg b \wedge C_{T''}^{\boxed{\mathrm{A}}}$$

If U is of the form if b then U' else U'' then the two subgoals are $B \wedge b \rhd U' = T$ and $B \wedge \neg b \rhd U'' = T$, and

$$B \wedge C_T^{\boxed{\mathrm{A}}} = B \wedge (b \vee \neg b) \wedge C_T^{\boxed{\mathrm{A}}} = (B \wedge b \wedge C_T^{\boxed{\mathrm{A}}}) \vee (B \wedge \neg b \wedge C_T^{\boxed{\mathrm{A}}})$$

For the CUT rule the two subgoals are $B_1 \rhd U = T$ and $B_2 \rhd U = T$ with $B \models B_1 \vee B_2$. Hence

$$B \wedge C_T^{\boxed{\mathrm{A}}} = B \wedge (B_1 \vee B_2) \wedge C_T^{\boxed{\mathrm{A}}} = (B \wedge B_1 \wedge C_T^{\boxed{\mathrm{A}}}) \vee (B \wedge B_2 \wedge C_T^{\boxed{\mathrm{A}}})$$

$\square$

Now suppose during the proofs of the subgoals (8) $X_i'(\tilde{x}_i)$ is bound to S_i, and B_i' to b_i, for $2 \leq i \leq m$. Then we have

Lemma 3.4 *When all subgoals (8) are proved, and the proof obligations generated during the proofs are valid, the following hold:*

1. $\vdash b_i \rhd S_i = T_i'[\lambda \tilde{x}_k S_k / X_k' | 1 \leq k \leq m]$

2. *For each* T_i' *and each occurrence* $\boxed{\mathrm{A}}$ *with* $A \equiv X_j'(\tilde{x})$ *in* T_i', $b_i \wedge C_{T_i'}^{\boxed{\mathrm{A}}} \models b_j[\tilde{x}/\tilde{x}_j]$

Proof: *1* is a direct consequence of the fact that $X_i'(\tilde{x}_i)$ is bound to S_i for each i. To see *2*, suppose X_j' is exposed n times during the proof with exposure conditions b_k', $1 \leq k \leq n$. By Lemma 3.3

$$b_i \wedge C_{T_i'}^{\boxed{\mathrm{A}}} = \bigvee_{1 \leq k \leq n} b_k'$$

The first time $X_j'(\tilde{x}_j)$ is exposed it is bound to S_j and at the same time B_j' is bound to b_j. For each other exposure $X_j'(\tilde{x})$ is made known and a proof obligation $b_k' \models b_j[\tilde{x}/\tilde{x}_j]$

is generated. Hence $b_i \wedge C_{T_i'}^{\boxed{\mathrm{A}}} \models b_j[\tilde{x}/\tilde{x}_j]$, under the assumption that these proof obligations are valid. □

From this lemma and Corollary 3.2 we can derive the correctness of the implementation:

Theorem 3.5 *The three commands* `ufi`*,* `bind` *and* `known` *correctly implement the inference rule UFI. That is, when goal (7) is proved with* `ufi`*, and the proof obligations generated during the proofs are valid, the following hold:*

1. $\vdash b_i \rhd S_i = T_i[\lambda\tilde{x}_k(b_kS_k)/R_k|1 \leq k \leq m]$

2. $\vdash b_i \rhd R_i(\tilde{x}_i) = T_i[\lambda\tilde{x}_k b_k R_k/R_k|1 \leq k \leq m]$

Proof: In the following the range of k is omitted.

1.

$$\begin{array}{rcl} \vdash b_i \rhd S_i & \overset{3.4.1}{=} & T_i'[\lambda\tilde{x}_k S_k/X_k'|k] \\ & \overset{3.4.2,3.2}{=} & T_i'[\lambda\tilde{x}_k b_k X_k'/X_k'|k][\lambda\tilde{x}_k S_k/X_k'|k] \\ & = & T_i'[\lambda\tilde{x}_k(b_kS_k)/X_k'|k] \\ & = & T_i[X_k'/R_k|k][\lambda\tilde{x}_k(b_kS_k)/X_k'|k] \\ & = & T_i[\lambda\tilde{x}_k(b_kS_k)/R_k|k]. \end{array}$$

2.

$$\begin{array}{rcl} \vdash b_i \rhd R_i(\tilde{x}_i) & = & T_i \\ & = & T_i[X_k'/R_k|k][R_k/X_k'|k] \\ & = & T_i'[R_k/X_k'|k] \\ & \overset{3.4.2,3.2}{=} & T_i'[\lambda\tilde{x}_k b_k X_k'/X_k'|k][R_k/X_k'|k] \\ & = & T_i'[\lambda\tilde{x}_k b_k R_k/X_k'|k] \\ & = & T_i[X_k'/R_k|k][\lambda\tilde{x}_k b_k R_k/X_k'|k] \\ & = & T_i[\lambda\tilde{x}_k b_k R_k/R_k|k] \end{array}$$

□

4 Conclusions

We have discussed the formulation of unique fixpoint induction in the setting of value-passing process calculi and showed how this inference rule is implemented in the verification tool *VPAM*. By exploiting context conditions the implementation reduces the number of subgoals as required by the UFI rule by half. An argument has also been given to justify the implementation.

Theoretical results concerning the soundness and completeness of the proof system with the UFI rule will be reported in a separate paper.

Acknowledgment Thanks to an anonymous referee for detailed comments.

References

[BBV92] T. Bolognesi, E. Brinksma, and C.A. Vissers. *Proceedings of Third Lotosphere Workshop.* Pisa, CNR-CNUCE, 1992.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *Proceedings of the* 9th *International Symposium on Protocol Specification, Testing and Verification*, North Holland, 1989.

[DIN89] R. DeNicola, P. Inverardi, and M. Nesi. Using the axiomatic presentation of behavioural equivalences for manipulating CCS specifications. In *Proc. Workshop on Automatic Verification Methods for finite State Systems*, number 407 in Lecture Notes in Computer Science, 1989.

[GLZ89] J. Godskesen, K. Larsen, and M. Zeeberg. Tav user manual. Report R89-19, Aalborg University, 1989.

[Hen91] M. Hennessy. A proof system for communicating processes with value-passing. *Formal Aspects of Computing*, 3:346–366, 1991.

[HL92] M. Hennessy and H. Lin. Symbolic bisimulations. Technical Report 1/92, Computer Science, University of Sussex, 1992.

[HL93] M. Hennessy and H. Lin. Proof systems for message-passing process algebras. In *CONCUR'93*, number 715 in Lecture Notes in Computer Science, pages 202–216, 1993.

[Lin91] H. Lin. PAM: A process algebra manipulator. In *Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 136–146. Springer–Verlag, 1991.

[Lin93] H. Lin. A verification tool for value-passing processes. In *Proceedings of* 13th *International Symposium on Protocol Specification, Testing and Verification*, IFIP Transactions. North-Holland, 1993.

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

[SV89] R. De Simone and D. Vergamimi. Aboard auto. Report RT111, INRIA, 1989.

Appendices

A Inference Rules

EQUIV $$\frac{}{true \rhd T = T} \qquad \frac{b \rhd T = U}{b \rhd U = T} \qquad \frac{b \rhd T = U, U = V}{b \rhd T = V}$$

CONGR $$\frac{b \rhd T_i = U_i \quad i = 1, 2}{b \rhd op(T_1, T_2) = op(U_1, U_2)} \quad op \notin \{., ?, !\}$$

EQN $$\frac{}{true \rhd T\sigma = U\sigma} \quad T = U \text{ is an axiom}$$

RENAME $$\frac{}{true \rhd c?x.T = c?y.T[y/x]} \quad y \notin fv(T)$$

L-INPUT $$\frac{b \rhd T = U}{b \rhd c?x.T = c?x.U} \quad x \notin fv(b)$$

OUTPUT $$\frac{b \models e = e',\ b \rhd T = U}{b \rhd c!e.T = c!e'.U}$$

TAU $$\frac{b \rhd T = U}{b \rhd \tau.T = \tau.U}$$

IF $$\frac{b \wedge b' \rhd T = V \quad b \wedge \neg b' \rhd U = V}{b \rhd \text{if } b' \text{ then } T \text{ else } U = V}$$

UFI $$\frac{\begin{array}{c} b_i \rhd S_i = T_i[\lambda \tilde{x}_k(b_k S_k)/R_k | 1 \leq k \leq m], \\ b_i \rhd R_i(\tilde{x}_i) = T_i[\lambda \tilde{x}_k(b_k R_k(\tilde{x}_i))/R_k | k], \end{array} \quad 1 \leq i \leq m}{b_1 \rhd S_1 = R_1(\tilde{x}_i)} \quad \begin{array}{l} R_i(\tilde{x}_i) = T_i \\ \tilde{R} \text{ guarded in } \tilde{T} \end{array}$$

REC $$\frac{}{true \rhd P_i(\tilde{x}_i) = T_i[\lambda \tilde{x}_k T_k / P_k | 1 \leq k \leq m]} \quad P_i(\tilde{x}_i) = T_i,\ 1 \leq i \leq m$$

CUT $$\frac{b \models b_1 \vee b_2,\ b_1 \rhd T = U \quad b_2 \rhd T = U}{b \rhd T = U}$$

ABSURD $$\frac{}{false \rhd T = U}$$

B Value-passing *CCS* Axioms in *VPAM*

```
A1    x + x = x
A2    x + NIL = x
PN    x | NIL = x

R1    (x + y)\A = x\A + y\A
R21   (c?v.x)\A = c?v.(x\A)           if not(c in A)
R22   (c?v.x)\A = NIL                 if c in A
R31   (c!e.x)\A = c!e.(x\A)           if not(c in A)
R32   (c!e.x)\A = NIL                 if c in A
R4    NIL\A = NIL

N1    (x + y)[a/b] = x[a/b] + y[a/b]
N21   (c?v.p)[a/b] = c?v.(p[a/b])     if not(c eq b)
N22   (c?v.p)[a/b] = a?v.(p[a/b])     if c eq b
N31   (c!e.p)[a/b] = c!e.(p[a/b])     if not(c eq b)
N32   (c!e.p)[a/b] = a!e.(p[a/b])     if c eq b
N4    NIL[a/b] = NIL

T1    a.tau.x = a.x
T2    x + tau.x = tau.x
T3    a.(x + tau.y) + a.y = a.(x + tau.y)

IFP   (if b then x else y) | z = if b then (x | z) else (y | z)
IFA   (if b then x else y) + z = if b then (x + z) else (y + z)
IFR   (if b then x else y)\A = if b then x\A else y\A
IFN   (if b then x else y)[a/b] = if b then x[a/b] else y[a/b]
IFT   tau.(if b then x else y) = if b then tau.x else tau.y
IFO   c!e.(if b then x else y) = if b then c!e.x else c!e.y

expansion law

EXP let x = a1.x1 + ... + an.xn      y = b1.y1 + ... + bm.ym
    then
     (x|y)\A = NIL  if sync_move(x,y) eq nil and async_move(x,y) eq nil
     (x|y)\A = Sum(+,async_move(x,y))         if sync_move(x,y) eq nil
     (x|y)\A = Sum(+,sync_move(x,y))          if async_move(x,y) eq nil
     (x|y)\A = Sum(+,async_move(x,y)) + Sum(+,sync_move(x,y))
                                              otherwise
    with
      async(c?x) = true            if  not(c in A)
      async(c!e) = true            if  not(c in A)
```

Translating a Process Algebra with Symbolic Data Values to Linear Format

Doeko Bosscher
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Alban Ponse
Programming Research Group, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam

The first author was supported by Esprit BRA 7166 concur 2.

Abstract

Historically, process algebras have been studied mostly *without* data. In this paper the transformation is described of the valued process algebra μCRL [GP95] to a symbolic transition system in the spirit of [Sch94]. The data oriented specifications thus obtained, seem to be in a better format for checking modal properties.

1. Introduction

Historically, much effort has been put into understanding the theories of pure process algebra calculi. Also, process algebra tools concentrate on process calculi with explicit data values as an input language. A few front ends have been developed which translate a calculus with symbolic values into a pure calculus, such as the value passer [Bru91], which translates value-passing CCS [Mil89] into pure CCS. In the LOTOS community several tools have been constructed which can translate (valued) LOTOS to C programs or labeled transitions system, e.g. [FGM$^+$92, KBG93].

We aim at building a tool which can do model checking for μCRL [GP95]. μCRL is a specification language for a process algebra with symbolic data values, where the process part is based on ACP [BW90] and the data part on algebraic specifications as in [EM85]. Until now μCRL has been mainly used for manual verification proving equivalences between processes (e.g. [BG94a]), but we are thinking of checking properties in a suitable logic. Sometimes we know only part of the desired behavior, as in the case of safety criteria for railroads [GKvV94].

In this respect [Sch94] is highly interesting, which treats a calculus with symbolic data values as a first class citizen. In [Sch94] value-passing CCS is mapped onto a data structure called *parametrized graphs*, which are essentially symbolic transition systems. This has two advantages over the conventional procedure of translating the valued calculus to the pure calculus and then perform modal checking. First, the structure of the processes is still visible in the parametrized graphs. Second, part of the state explosion is avoided because data is not expanded.

Whereas [Sch94] is mainly focused on checking various equivalences, we are interested in checking modal formulas as in [GvV94] or [HL93]. We think that translating the language μCRL to parametrized graphs is an interesting experiment in itself and a signal for model checking. We refer to the experience that Hennessy-Milner Logic seems to be checked more efficiently on a restricted form of pure CCS [Hol89]. A second point of interest is that the parameterized graphs are a kind of *data oriented* specifications. In several case studies verification starts by transforming specifications to such a form by hand and then performing further analysis, see e.g. [Bru95, GS95].

We describe the transformation of μCRL to parametrized graphs, which we will define syntactically.

We start with a translation of a fragment of μCRL to single-linear format by means of typical examples [1]. This format is a direct translation of a graph grammar and can be seen also as a fragment of value-passing CCS. Next we explain how a larger part of the μCRL specifications in the full calculus can be translated to this format. We end with some conclusions on the implementation of the transformation in the ASF+SDF system [Kli93].

2. Translating a Fragment of μCRL to a Single-Linear Specification

The specification language μCRL has come out of the SPECS project, as the essence of the language CRL [BDE$^+$93]. It has been developed under the assumption that a study of the basic concepts of specification languages will yield more fundamental insights then studying the complete language.

The data part contains equational specifications. The process part contains processes described in the style of CCS, CSP or ACP, where the syntax has been taken from the last. It basically consists of a set of uninterpreted actions that may be parametrized by data. These actions represent various activities, depending on the usage of the language. There are sequential composition, alternative and parallel composition operators. Furthermore, recursive processes are specified in a simple way. See for a complete definition of jargon, syntax and semantics [PVvV95].

In this section we describe the translation of a basic fragment of μCRL to linear format. It is similar to BPA, in that it contains only alternative and sequential composition [2] It extends BPA by the presence of data and the if-then-else and sum construct. First we define this fragment and linear specifications in a precise way. Next we describe the translation by means of examples.

2.1 Specifications in BPS Format

A well-formed μCRL *specification* E is a *specification* in Basic Process Syntax, BPS for short, iff all *process-declarations* occurring in E have in their right-hand sides *process-expressions* that are in BPS:

Definition 2.1 The syntactical category BPS that constitutes the class of processes in BPS has the following BNF syntax.

$$\begin{array}{lcl}
\textit{process-expression} & ::= & \textit{process-expression} + \textit{process-expression} \\
& | & \textit{process-expression} \lhd \textit{data-term} \rhd \textit{process-expression} \\
& | & \textit{process-expression} \cdot \textit{process-expression} \\
& | & \sum(\textit{single-variable-declaration}, \textit{process-expression}) \\
& | & \delta \\
& | & \textit{name} \\
& | & \textit{name}(\textit{data-term-list}) \\
& | & (\textit{process-expression}).
\end{array}$$

In the above defintion $+$ is the choice operator, $\cdot$ sequential composition and $\lhd\ \rhd$ is the notation for the if-then-else construct in μCRL . $\sum$ is the notation for a summation over data. δ is the deadlocked process. The precedence is in the order $\cdot, \lhd\ \rhd, +$ (as can be seen seen from definition above).

Example 2.2 Consider the following well-formed μCRL *specification* in BPS, E of the sender in the Alternating Bit Protocol of [GP95]:

[1]Formal approaches to μCRL proof theory are e.g., [GP93],[BG94b].

[2]In linear formats, sequential composition can actually be replaced by action prefixing.

$$
\begin{array}{lll}
E & \equiv & \textbf{sort} \quad bit, D, error, \textbf{Bool} \\
 & & \textbf{func} \quad T, F :\rightarrow \textbf{Bool} \\
 & & \qquad\quad 0, 1 :\rightarrow bit \\
 & & \qquad\quad e :\rightarrow error \\
 & & \qquad\quad invert : bit \rightarrow bit \\
 & & \qquad\quad d_1, d_2, d_3 :\rightarrow D \\
 & & \textbf{act} \quad r1 : D \\
 & & \qquad\quad r6 : error \\
 & & \qquad\quad r6 : bit \\
 & & \qquad\quad s2 : D \times bit \\
 & & \qquad\quad c : \textbf{Bool} \\
 & & \textbf{rew} \quad invert(0) = 1 \\
 & & \qquad\quad invert(1) = 0 \\
 & & \textbf{proc} \quad S = S(0) \cdot S(1) \cdot S \\
 & & \qquad\quad S(n : bit) = sum(d : D, r1(d) \cdot S(d, n)) \\
 & & \qquad\quad S(d : D, n : bit) = s2(d, n) \cdot ((r6(invert(n)) + r6(e)) \cdot S(d, n) + r6(n))
\end{array}
$$

We will use the terms *process* and *action* as follows: let E be a μCRL *specification* and q a *process-expression* that is Statically and Semantically Correct (SSC, [GP95]) with respect to E and has no free data variables, then p **from** E is called a process from E. We will use the term *parametrized process name* for the name in the left-hand side of a process specification, which has a type given by the parametrization [3]. Furthermore an *action* is a process that refers directly to an *action-specification* in E and has no free data variables. So in the example above $S(0)$ is a process from E, and $r6(invert(0)), r6(e)$ are actions from E. If E is fixed, we just speak of "the process p".

We will restrict our attention to a decidable class of guarded specifications in BPS. We will admit only those specifications where the defining right hand side of every process name is such that the process name occurs only guarded, i.e. either directly or indirectly in the scope of an action.

Definition 2.3 Let P be the set of process names occurring in the specification E and $p, p_1, ..., p_n, q \in P$ (parametrized) process names. Let $UG(p, E)$ be a set of tuples of the form $< p, q >$ where q is a (parametrized) process name occurring unguarded in the declaration of p, i.e. not in the scope of a preceding action. E is *syntactically guarded* iff $\bigcup_{p \in P} UG(p, E)$ contains no cycle, i.e. a subset of the form $\{< p_1, p_2 >, < p_2, p_3 > ... < p_{n-1}, p_n >\}$ so that $p_1 \equiv p_n$.

Given a μCRL *specification* E, we associate with each process from E a (referential) transition system that describes its meaning. The intended semantics of a process p from a μCRL *specification* E is a transition system $\mathcal{A}(\boldsymbol{A}_{N_E}, p\ \textbf{from}\ E)$ where $\boldsymbol{A}_{N_E}$ is the canonical term algebra of E, and where the labels of transitions may be parameterized by the fixed representations of the elements of $\boldsymbol{A}_{N_E}$. These transition systems are considered modulo bisimulation equivalence, notation $\leftrightarrows_{\boldsymbol{A}_{N_E}}$, as this is the coarsest congruence that respects operational behaviour.

Now processes from syntactically guarded μCRL *specifications* in BPS constitute the **source language** for the translation described in the sequel.

Conventions. For readability we adopt the following conventions.

- Binary operations associate to the right, brackets are omitted if possible.

[3]So in Example 2.2 the three process declarations have a *different* parametrized process name, although their name is the same.

- Instead of repeatedly denoting μCRL *specifications* in a syntactically correct way (as was done in the example above), we often only write down a *process-specification* without the keyword **proc**, and assume that it is part of some well-defined μCRL *specification*. In doing so we use $a, b, c, ...$ as syntactic variables for action *names* and $X, Y, Z, ...$ as syntactic variables for process *names*.

- Whenever convenient, we assume that any μCRL *specification* under consideration contains the (standard) functions $\neg$ and $\wedge$ on the standard sort **Bool**. Applications of the function $\wedge$ will always be written in an infix manner. Note that from the point of view of describing *processes* this convention causes no loss of generality, as we can always extend *specifications* with these functions. □

2.2 Single-Linear Process Specifications
In this section we define the syntax of "single-linear" *process-specifications* that play a crucial role in our canonical translation.

We start by introducing the following two archetypes of μCRL *process-specifications* in BPS. In their definition we use the symbol Σ also as a shorthand to denote **finite** sums (not to be confused with the *sum operator* of μCRL): let $p_1, p_2, ...$ be *process-expressions*, then the expression

$$\sum_{i=1}^{k} p_i$$

abbreviates δ in case $k = 0$, and $p_1 + p_2 + ... + p_k$ otherwise.

Definition 2.4 A *process-specification* of the form $pd_1 \, ... \, pd_m$ with $m \geq 1$ from some μCRL *specification* E is in *normal form* iff for all $1 \leq i \leq m$ the declaration pd_i has a right-hand side of the form

$$\sum_{j=1}^{k_i} p_{ij}$$

where each of the *process-expressions* p_{ij} [4] is of the form

$$\begin{array}{l}(\sum_{k=1}^{k_{ij}} \Sigma(d_{ijk} : D_{ijk}, a_{ijk} \cdot X^1_{ijk} \cdot X^2_{ijk}) + \\ \sum_{l=1}^{l_{ij}} \Sigma(d_{ijl} : D_{ijk}, b_{ijl} \cdot X^3_{ijl}) + \\ \sum_{m=1}^{m_{ij}} \Sigma(d_{ijm} : D_{ijm}, c_{ijm})) \lhd t_{ij} \rhd \delta\end{array}$$

with the d_{ijk} single variables over data types D_{ijk}, $a_{ijk}, b_{ijk}, c_{ijk}$ (possibly parameterized) *process-expressions* over the *names* in the *action-specifications* from E, and the $X^1_{ijk}, X^2_{ijk}, X^3_{ijk}$ (possibly parameterized) *process- expressions* over the *names* in the left-hand sides of the declarations $pd_1, ..., pd_m$.
In the special case that $k_{ij} = 0$ for all appropriate i, j we say that the *process-specification* $pd_1 \, ... \, pd_m$ is in *linear form*.

Now we can define what is meant by a "single-linear" *process-specification*.

Definition 2.5 Let E be a μCRL *specification*. A *process-specification* occurring in E is *single-linear* iff it is in linear form and contains exactly one *process-declaration*.

[4]We use of course the axiom $\sum(d : D, p) = p$, d *not* free in p, to remove summations.

Example 2.6 Consider the following specification:

$$\begin{array}{lll} E \equiv & \textbf{sort} & \textbf{Bool}, S \\ & \textbf{func} & T, F :\to \textbf{Bool} \\ & & C :\to S \\ & & f : \textbf{Bool} \to S \\ & & g : S \to \textbf{Bool} \\ & \textbf{act} & a, d \\ & & b : \textbf{Bool} \\ & & c : S \times \textbf{Bool} \\ & \textbf{proc} & X(x : \textbf{Bool}, y : S) = \quad (a \cdot X(x, f(x)) + b(x)) \lhd x \rhd \delta \\ & & \qquad\qquad\qquad\qquad\quad +(c(y, g(y)) \cdot X(g(y), f(x)) + d) \lhd g(y) \rhd \delta \end{array}$$

that has a single-linear *process-specification*.

2.3 The Translation
Given a syntactically guarded well-formed μCRL *specification* E in BPS and a process p **from** E, we describe in this section the construction of a syntactically guarded μCRL *specification* E' such that

- E' is a μCRL *specification*, obtained from E by the (possible) addition of *sort-*, *function-*, *rewrite-* and *process-specifications* in such a way that p **from** $E \leftrightarrows_{\boldsymbol{A}_{N_{E'}}} p$ **from** E'.
- there is a process p' from E' such that
  - p' satisfies p' **from** $E' \leftrightarrows_{\boldsymbol{A}_{N_{E'}}} p$ **from** E', i.e. p and p' behave the same,
  - p' is a process that is specified in a single-linear way, i.e. the *name* of p' is declared in a single-linear *process-specification* contained in E'.

We just describe the construction of E' by means of examples, and refrain from formal descriptions which are required for a correctness proof. We hope that the suggestion of provability is sufficiently clear.

We distinguish six consecutive steps in this type of construction, each of which should be applied in case its conditions hold. Application of such a step extends the *specification* with at least a *process-specification*. We assume that these extensions always yield a μCRL *specification*, so in particular we assume that the newly added *sort-*, *function-* and *process-specifications* have fresh *names*.

1. Introducing a process expression as a new declaration. Let p **from** E be the object for translation. This step applies whenever p is not of the form n or $n(t_1, ..., t_k)$ for some *process name* n. In this case we extend E to E_1 by adding a *process-specification* that specifies a process p_1 of the form n or $n(t_1, ...t_k)$ that behaves the same as p **from** E_1.

Example of step 1. Let $p \equiv X(t) + b(u)$ where $X(x : S)$ is specified as follows:

$$X(x : S) \quad = \quad a(x) \cdot X(x) + a(x)$$

and the *action-specification* **act** $b : S'$ is also contained in E. We extend E to E_1 by adding the *process-specification*

$$X'(x : S, y : S') \quad = \quad X(x) + b(y)$$

Note that

$$X(t) + b(u) \textbf{ from } E_1 \leftrightarrows_{\boldsymbol{A}_{N_{E_1}}} X'(t,u) \textbf{ from } E_1.$$

(End example.)

2. Translating the process declarations to normal form. Let p_1 **from** E_1 satisfy $p_1 \equiv n$ or $p_1 \equiv n(t_1, ..., t_k)$. This step applies whenever the *process-specification* of p_1 is not in normal form. In this case we extend E_1 to E_2 by adding a *process-specification* in normal form of a process p_2 that behaves the same as p_1 **from** E_2.

Example of step 2. Let $p_1 \equiv X(t)$ where $X(d : D)$, is specified as follows, with $d_0 \in D$ a constant:

$$X(d : D) \quad = \quad \sum(e : D, a(d) \cdot X(d_0) \cdot X(e) \cdot X(d)) + b$$

We sketch the technique to obtain a *process-specification* in normal form that defines the same process(es) as $X(d : D)$. The main problem here is the summand $\sum(e : D, a(d) \cdot X(d_0) \cdot X(e) \cdot X(d))$, as it is essentially different from the 'normal form syntax'. We start by replacing this subterm by the term $\sum(e : D, a(d) \cdot X_1(d, e))$. We add the new process declaration

$$X_1(d : D, e : D) \quad = \quad X(d_0) \cdot X(e) \cdot X(d)$$

and thus obtain the specification

$$\begin{array}{lcl} X(d : D) & = & \sum(e : D, a(d) \cdot X_1(d, e)) + b \\ X_1(d : D, e : D) & = & X(d_0) \cdot X(e) \cdot X(d). \end{array}$$

The process declaration for X is now essentially in normal form. We repeat the same step for the process declaration for X_1. The new specification becomes

$$\begin{array}{lcl} X(d : D) & = & \sum(e : D, a(d) \cdot X_1(d, e)) + b \\ X_1(d : D, e : D) & = & X(d_0) \cdot X_2(d, e) \\ X_2(d : D, e : D) & = & X(e) \cdot X(d). \end{array}$$

Having done this, we can replace the specification using the *new* declaration for X, i.e.,

$$\begin{array}{lcl} X(d : D) & = & \sum(e : D, a(d) \cdot X_1(d, e)) + b \\ X_1(d : D, e : D) & = & (\sum(e : D, a(d_0) \cdot X_1(d_0, e)) + b) \cdot X_2(d, e) \\ X_2(d : D, e : D) & = & (\sum(e' : D, a(e) \cdot X_1(e, e')) + b) \cdot X(d). \end{array}$$

Using the axioms for the sum operator distributivity and the conditional construct this gives a specification which is in normal form. From this sketch it follows in what we can extend E_1 to E_2 with a *process-specification* in *normal form* that defines a process behaving like $X(t)$:

$$\begin{array}{lcl} X'(d : D) & = & \sum(e : D, a(d) \cdot X'_1(d, e)) + b \lhd T \rhd \delta \\ X'_1(d : D, e : D) & = & \sum(e : D, a(d_0) \cdot X'_1(d_0, e) \cdot X'_2(d, e)) + b \cdot X'_2(d, e) \lhd T \rhd \delta \\ X'_2(d : D, e : D) & = & \sum(e' : D, a(e) \cdot X'_1(e, e') \cdot X'(d)) + b \cdot X'(d) \lhd T \rhd \delta. \end{array}$$

We remark that a *process-specification* in normal form has a syntax comparable to the *restricted Greibach Normal form* (rGNF) as defined in [BBK87]. They do not give an explicit method to obtain this form but give a sketch in the proof. We believe that their method is more difficult to implement than the method presented above, as we restrict ourselves to syntactically guarded specifications.

(End example.)

3. Disambiguate the formal parameters. Let p_2 **from** E_2 be specified in a *process-specification* that is in normal form. This step applies whenever it is the case that the *process-specification* of p_2 has overloading of variable *names*. By definition of E_2 being Statically Semantically Correct (SSC), this can only be the case if the *process-specification* of p_2 contains more than one declaration. In this case we extend E_2 to E_3 by adding a *process-specification* in normal form that has uniquely typed variable *names*, and that defines a process p_3 that behaves like p_2 **from** E_3.

Example of step 3. Let $p_2 \equiv X(t)$ where $X(x:S)$ is specified as follows:

$$\begin{array}{rcl} X(x:S) & = & (a \cdot Y(f(x)) + b) \triangleleft t \triangleright \delta \\ Y(x:S') & = & (c \cdot X(g(x)) + d(x)) \triangleleft h(x) \triangleright \delta \end{array}$$

We extend E_2 to E_3 by adding the *process-specification*

$$\begin{array}{rcl} X'(x:S) & = & (a \cdot Y'(f(x)) + b) \triangleleft t \triangleright \delta \\ Y'(y:S') & = & (c \cdot X'(g(y)) + d(y)) \triangleleft h(y) \triangleright \delta \end{array}$$

Note that

$$X(t) \textbf{ from } E_3 \leftrightarrows_{\boldsymbol{A}_{N_{E_3}}} X'(t) \textbf{ from } E_3.$$

(End example.)

4. Globalize formal parameters. Let p_3 **from** E_3 be specified in a *process-specification* that is in normal form and that has uniquely typed variable *names*. This step applies whenever it is not the case that the *process-specification* of p_3 has *global parameterization*:

> **Definition 2.7** A *process-specification* in normal form with uniquely typed variable *names* has *global parameterization* iff each occurring variable *name* is declared in *all* of its declarations, that is in all occurring process parameter lists.

Note that a single-linear *process-specification* has by definition global parameterization. If step 4 applies, we extend E_3 to E_4 by adding a *process-specification* in normal form and with uniquely typed variables that has global parameterization, and that defines a process p_4 that behaves like p_3 **from** E_4. The next step will show the purpose of this extension.

Example of step 4. Let $p_3 \equiv X(t)$ and let $X(x:S)$ be specified as follows:

$$\begin{array}{rcl} X(x:S) & = & (a \cdot Y(f(x)) \cdot X(g(x)) + b(x)) \triangleleft t_1 \triangleright \delta \\ Y(y:S') & = & (c \cdot Y(h(y)) + d(y)) \triangleleft t_2 \triangleright \delta \end{array}$$

We extend E_3 to E_4 by adding the *process-specification*

$$\begin{array}{lcl} X'(x:S,y:S') & = & (a\cdot Y'(x,f(x))\cdot X'(g(x),y)+b(x))\lhd t_1 \rhd \delta \\ Y'(x:S,y:S') & = & (c\cdot Y'(x,h(y))+d(y))\lhd t_2 \rhd \delta \end{array}$$

Note that x and y being different *names* is essential for application of this step. This extension has the following property:

$$X(t)\ \mathbf{from}\ E_4 \leftrightarroweq_{\boldsymbol{A}_{N_{E_4}}} X'(t,u)\ \mathbf{from}\ E_4$$

for any closed *data-term* u of sort S'.
(End example.)

5. Form single declaration. Let p_4 **from** E_4 be specified in a *process-specification* in normal form that has uniquely typed variable *names* and global parameterization. This step applies whenever the *process-specification* of p_4 contains more than one *process-declaration*. In this case we extend E_4 to E_5 by adding a *sort-specification*, a *function-specification* and a *process-specification* containing only one declaration that defines a process p_5 which behaves the same as p_4 **from** E_5. The following example also shows how the data-part of μCRL may be used, and the purpose of global parameterization (step 4).

Example of step 5. Let $p_4 \equiv X'(t,u)$ where $X'(x:S,y:S')$ is specified as in the example of step 4:

$$\begin{array}{lcl} X'(x:S,y:S') & = & (a\cdot Y'(x,f(x))\cdot X'(g(x),y)+b(x))\lhd t_1 \rhd \delta \\ Y'(x:S,y:S') & = & (c\cdot Y'(x,h(y))+d(y))\lhd t_2 \rhd \delta \end{array}$$

We extend E_4 to E_5 by adding a new sort *Sort* with constants X', Y', an equality function on *Sort* (we use infix notation) and the *process-specification*

$$\begin{array}{rl} Z(n:Sort,x:S,y:S') = (a\cdot Z(Y',x,f(x))\cdot Z(X',g(x),y)+b(x)) & \lhd\ t_1\wedge n=X' \rhd \delta \\ +(c\cdot Z(Y',x,h(y))+d(y)) & \lhd\ t_2\wedge n=Y' \rhd \delta \end{array}$$

The summands $b(x)$ and $d(y)$ show the purpose of global parameterization: the process Z has to be parameterized with both the sorts S and S' in order to have the *specification* E_5 SSC. Note that indeed

$$X'(t,u)\ \mathbf{from}\ E_5 \leftrightarroweq_{\boldsymbol{A}_{N_{E_5}}} Z(X',t,u)\ \mathbf{from}\ E_5.$$

(End example.)

6. Linearize the process declaration. Let p_5 **from** E_5 be specified in a *process-specification* in normal form containing one *process-declaration*. This step applies whenever the *process-specification* of p_5 is not linear. In this case we extend E_5 to E_6 by adding *sort-*, *function-* and *rewrite-specifications*, and a single-linear *process-specification* that defines a process p_6 that behaves the same as p_5 **from** E_6.

Example of step 6. Let $p_5 \equiv Z(X',t,u)$ where $Z(n:Sort,x:S,y:S')$ is specified as in the example of step 5:

$$\begin{array}{rl} Z(n:Sort,x:S,y:S') = (a\cdot Z(Y',x,f(x))\cdot Z(X',g(x),y)+b(x)) & \lhd\ t_1\wedge n=X' \rhd \delta \\ +(c\cdot Z(Y',x,h(y))+d(y)) & \lhd\ t_2\wedge n=Y' \rhd \delta \end{array}$$

We add two sorts to E_5. First a sort *Unproper* over which the *data-terms* are of the form X', t', u' and Y', t', u', for all *data-terms* t', u' over the sorts S and S', respectively. Note that this cannot be proper μCRL syntax, as *names* may not contain commas. However, for the purpose of readability we do not care for the moment and underline the elements of the unproper sort.

Next we add a sort *Stack* defined over *Unproper* and the constant λ for the empty stack, and the functions *pop*, *push*, *rest* and *is-empty* with rewrite rules as expected. We extend E_5 to E_6 by also adding the *process-specification*

$$\begin{array}{ll}
\multicolumn{2}{l}{Z'(n : S, x : S, y : S', s : \mathit{Stack}) =} \\
\quad (a \cdot Z'(Y', x, f(x), \mathit{push}(\underline{X', g(x), y}, s)) + b(x)) & \lhd\, t_1 \wedge n = X' \wedge \mathit{is\text{-}empty}(s) \rhd \delta \\
+\ (a \cdot Z'(Y', x, f(x), \mathit{push}(\underline{X', g(x), y}, s)) + b(x) \cdot Z'(\mathit{pop}(s), \mathit{rest}(s))) & \\
 & \lhd\, t_1 \wedge n = X' \wedge \neg(\mathit{is\text{-}empty}(s)) \rhd \delta \\
 & \\
+\ (c \cdot Z'(Y', x, h(y), s) + d(y)) & \lhd\, t_2 \wedge n = Y' \wedge \mathit{is\text{-}empty}(s) \rhd \delta \\
+\ (c \cdot Z'(Y', x, h(y), s) + d(y) \cdot Z'(\mathit{pop}(s), \mathit{rest}(s))) & \lhd\, t_2 \wedge n = Y' \wedge \neg(\mathit{is\text{-}empty}(s)) \rhd \delta
\end{array}$$

Note that

$$Z(X', t, u) \textbf{ from } E_6 \leftrightarrows_{\boldsymbol{A}_{N_{E_6}}} Z'(X', t, u, \lambda) \textbf{ from } E_6.$$

(End example.)
The general idea behind step 6 is that we can define a sort that has a class of (properly encoded) *process-expressions* as its closed *data-terms*, and a sort *Stack* of stacks over this sort. Upon a summand of the form $a \cdot X \cdot Y$ we stack the subprocess Y, and upon a non-recursive summand of the form a and a non-empty stack, we pop the first subprocess for execution.

3. From μCRL to Single-Linear Specifications

The Basic Process Syntax of the previous section is a concise way to specify processes with data, but somewhat inconvenient to specify protocols. Usually protocols are specified as a parallel composition of processes. Therefore we reintroduce more involved operators (merge, encapsulation etc.) into the syntax. This will make specifying easier, but at the same time we have to be attentive that the specifications we allow can be translated to a linear format.

It is well-known that regularity (and hence linearity) is undecidable when the occurrence of parallelism in the syntax is unrestricted [BK89]. Moreover finiteness conditions as in the case of process algebra *without* data such as in [MV90] become undecidable if processes and data interact.

It will be sufficient for our purposes to exclude specifications like

$$X(n : \mathit{Int}) = a(n) \parallel a(n+1) \cdot X(n+2)$$

where a merge operator is used in the scope of the recursion. For convenience the above mentioned operators will only be used to compose processes which are in BPS, or can be translated to it. Such a strategy is straightforward and is used in e.g. the AUTO tool [SR91] to specify processes. In [Sch94] syntactic conditions similar to ours are formulated and motivated with examples.

We formalize the restriction to a specification with a safe use of parallel operators with the aid of syntactic guardedness.

Definition 3.1 Let E be a well-formed μCRL *specification*. E is *safely linearizable* iff

1. E is the extension of a syntactically guarded μCRL *specification* E_{syng} with (parametrized) process names N_{syng} and,
2. All right hand sides of process declarations in the extension $E - E_{syng}$ are process expressions in which only (parametrized) process names in N_{syng} occur.

Without proof we state that well-formed μCRL *specifications*, which are safely linearizable are bisimilar to linear process specifications (see e.g. [BP94]). The receipt to obtain such a specification is obvious. We translate in an innermost-outermost fashion all process declarations to single-linear format, starting with the declarations in BPS. The other operators are eliminated in the usual way, by expansion and straight forward data parametric substitution, using the recursive specification principle RSP [GP93].

Of course the conditions of Definition 3.1 can be relaxed to allow more nesting. For this an iteration á la syntactic guardedness suffices.

4. Conclusions and Future Work

In this paper we aim at arriving at a single-linear format. We believe that this is a natural format for a parametrized graph or a symbolic transition system. Of course other formats are possible. The use of steps 3–5 can be avoided if we had aimed at a *linear format*, i.e. several coexisting linear declarations. One could say that this is a matter of taste, but we feel that Step 6 becomes more difficult and the resulting specification is less insightful. If several (mutually dependent) process declarations remain, process calls are not uniform and explicit list access has to be introduced, instead of implicit bindings. Also extra control information has to be supplied to process calls, to allow correct selection of the called process. Also in some way or another, process calls have to be stacked with varying types of parameters. The data structure needed will be a list of lists of varying types, and hence be complicated.

At the moment the first author is implementing the above described translation in the ASF+SDF system [Kli93]. This general purpose term rewriting system has several built-in possibilities, among them the possibility to compile rewriting systems to C code. We can make ample use of the fact that μCRL data and process specification is ASF like. We are aiming to integrate this “linearizer” with the well-formedness checker [HK95] developed for μCRL .

We see several next steps. A first (conservative) next step is to build an “instantiator”, a front end which can translate single-linear specifications to labeled transition systems. These can then be interfaced with the tools in the Concur 2 project, which offer various model checking facilities for pure calculi. Of course it will then be essential that all data types are finitary.

A second, more ambitious step is to implement a part of the logic of [GvV94], which is tailored to the syntax of μCRL . An obvious strategy would be to expand modal formulas, to instantiate data and check the pure formulas on a labeled transition system.

Third, we can make a detailed investigation of the complexity of the various steps and suggest optimizations. Furthermore we can look for a class of specifications for which the stacking of processes in Step 6 of Section 2 can be avoided, using the results of [MM94].

Acknowledgements. Jan Bergstra, David Griffioen, Jan-Friso Groote, Wilco Koorn, Eric Madelaine and Frits Vaandrager are thanked for helpful suggestions.

References

[BBK87] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *Proceedings PARLE conference,* Eindhoven*, Vol. II (Parallel Languages)*, volume 259 of *Lecture Notes in Computer Science*, pages 94–113. Springer-Verlag, 1987.

[BDE$^+$93] W. Bouma, M. Dauphin, G.D. Evans, M. Michel, and R. Reed, editors. *SPECS-Specification and Programming Environment for Communicating Software.* North Holland, 1993.

[BG94a] M.A. Bezem and J.F. Groote. A correctness proof of a one bit sliding window protocol in μCRL. *The Computer Journal*, 37(4):289–307, 1994.

[BG94b] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR'94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 401–416. Springer-Verlag, 1994.

[BK89] J.A. Bergstra and J.W. Klop. Process theory based on bisimulation semantics. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency,* Noordwijkerhout, The Netherlands, May/June 1988, volume 354 of *Lecture Notes in Computer Science*, pages 50–122. Springer-Verlag, 1989.

[BP94] M.A. Bezem and A. Ponse. Two finite specifications of a queue. Report P9424, Programming Research Group, University of Amsterdam, 1994.

[Bru91] Glenn Bruns. A language for value-passing CCS. Technical Report ECS-LFCS-91-175, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[Bru95] J.J. Brunekreef. Process Specification in a UNITY format. In Ponse et al. [PVvV95], pages 319–337.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications I*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.

[FGM$^+$92] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodríguez, and Joseph Sifakis. A toolbox for the verification of lotos programs. In *Proceedings of the 14th International Conference on Software Engineering ICSE'14 Melbourne, Australia*, 1992.

[GKvV94] J.F. Groote, J.W.C. Koorn, and S.F.M. van Vlijmen. The Safety Guaranteeing System at Station Hoorn-Kersenboogerd. Report 121, Logic Group, Preprint Series, Utrecht University, 1994.

[GP93] J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computer Science, pages 231–250. Springer-Verlag, 1993.

[GP95] J.F. Groote and A. Ponse. The Syntax and Semantics of μCRL. In Ponse et al. [PVvV95], pages 26–62.

[GS95] J.F. Groote and A. Sellink. Confluence for Process Verification, 1995. To be published in the proceedings of *CONCUR'95.*

[GvV94] J.F. Groote and S.F.M. van Vlijmen. A Modal Logic for μCRL. Research Report 114, Dept. of Philosophy, Utrecht University, May 1994. Also in: *Modal Logic and Process Algebra*, A. Ponse, Y. Venema, and M. de Rijke, editors, CSLI Publications, to appear.

[HK95] J.A. Hillebrand and H. Korver. A Well-formedness Checker for μCRL. Report P9501, Programming Research Group, University of Amsterdam, February 1995.

[HL93] M. Hennessy and X. Liu. A modal logic for message passing processes. Research Report 3/93, University of Sussex, January 1993.

[Hol89] Uno Holmer. Translating Static CCS Agents into Regular Form. PMG report 51, Department of Computer Science, Chalmers University of Technology and the University of Göteborg, 1989.

[KBG93] G. Karjoth, C. Binding, and J. Gustafsson. LOEWE: A LOTOS engineering workbench. *Computer Networks and ISDN Systems*, 25:853–874, 1993.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

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

[MM94] Sjouke Mauw and Hans Mulder. Regularity of BPA-Systems is Decidable. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR'94: Concurrency Theory*, volume 836, pages 34–47. Springer-Verlag, 1994.

[MV90] E. Madelaine and D. Vergamini. Finiteness Conditions and Structural Construction of Automata for all Process Algebras. In *Proceedings of CAV '90*, New Brunswick (NJ), 1990.

[PVvV95] A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors. *Algebra of Communicating Processes, Utrecht '94*, Workshops in Computing, Utrecht, 1995. Springer-Verlag.

[Sch94] Marcel Zvi Schreiber. *Value-passing Process Calculi as a Formal Method.* PhD thesis, University of London, 1994.

[SR91] R. de Simone and V. Roy. Auto/Autograph. In E.M. Clarke and R.P. Kurshan, editors, *Proceedings of the 2nd International Conference on Computer-Aided Verification,* New Brunswick, NJ, USA, volume 531 of *Lecture Notes in Computer Science*, pages 65–75. Springer-Verlag, 1991.

A UNITY-based Algorithm Design Assistant

Michel Charpentier[†] Abdellah El Hadri[‡]
Gérard Padiou[†]

[†]ENSEEIHT-IRIT
2, rue Charles Camichel
31071 TOULOUSE cedex
e-mail: {charpov, padiou}@enseeiht.fr

[‡]EMI
Avenue Ibn Sina BP 765
RABAT (MAROC)
e-mail: elhadri@emi.ma

Abstract

We address the problem of the automatic verification of reactive systems. For such algorithms, parallelism, non-determinism and distribution, lead to frequent design flaws and make debugging difficult. Proving programs with respect to their specification may solve both these problems. In this framework, we describe the implementation of an algorithm design assistant based upon the `UNITY` formalism. A theorem prover and a Presburger formulas calculator are used to perform the underlying proofs. We illustrate the main difficulties encountered with representative examples.

Key words: Program verification, reactive programs, `UNITY` formalism, parallelism, distribution, theorem proving.

I Introduction

Concurrency and distribution generate two further difficulties with respect to sequential programming. Concurrency leads to a drastic increase in program states and distribution results in a knowledge loss of both any global state and time. Therefore, program debugging becomes especially difficult.

To cope with these problems, we need to improve and formalize the methods and tools to design such programs. In this framework, two approaches may be investigated. Firstly, the development process itself may be assisted and supervised rigorously. Several projects are currently devoted to such computer assisted software engineering (*CASE*) tools. Secondly, the development operations may be improved too. In this case, the use of well formalized methods should enable to produce correct programs more quickly. However, without assistance, proofs are long, tedious and error prone. Therefore, computer assisted proofs appear a worthwhile requirement. We investigate this approach.

Stepwise refinement programming is a well known method. The development of a program consists of a sequence of pairs $(Spec, Prog)$ in which *Spec* is a specification and *Prog* its associated implementation. Starting from a high level abstract version, successive refinements generate new pairs to lead to a final program validated with respect to its specification. Each refinement must have well defined semantics in order to check its validity in the context of a specific development. The handling of both the specification and the implementation of a program helps to grasp the problem insofar as these representations complement each other.

However, a parallel calculus model is required to experiment such an approach. This model should:

- provide both a specification and programming language with a well formalized foundation;
- allow a stepwise refinement design with the same formalism all along the development process;
- include parallel features.

According to these requirements, we have chosen the `UNITY` model from Chandy and Misra [CM88] as an experimental basis. Its specification language allows to describe the safety properties and liveness properties of programs and the programming language provides both synchronous and asynchronous statements to express parallelism. Furthermore, this model is general enough to carry out a stepwise refinement approach.

In this framework, the formalization of a development involves two tasks:

- the research and the definition of generic refinements;
- the consistency verification of the pair $(Spec, Prog)$. More precisely, the program *Prog* must be validated with respect to its specification *Spec*.

About the first task, the refinement method allows to prove only the initial pair $(Spec, Prog)$ if the further performed refinements are sound. Although this point will not be further considered in this paper, we actually search how to refine a centralized algorithm so that it can be efficiently mapped to a target distributed architecture.

The second task raises an automatic proof problem. We study this question in the following sections more precisely and its implementation in a `UNITY` based environment.

II The UNITY formalism

The UNITY formalism is based on fair transition systems. It consists of a programming language to describe the state transitions and a specification language based upon linear temporal logic. We only outline the main features. Readers familiar with UNITY may skip this section.

II.1 UNITY programs

A UNITY program consists of four sections:

- the **declare** section defines a set of typed variables.
- the **initially** section assigns initial values to the variables. A variable may be initialised at most once with an expression of its type. Non-initialized variables have arbitrary values at the beginning of the program execution.
- the **always** section defines some variables as functions of others.
- lastly, the **assign** section contains the program statements. More precisely, a set of multiple-assignment statements are listed. The symbol $[\!]$ is used to separate these assignments.

A program execution runs forever starting from any state satisfying the initial conditions. At any step, a statement is arbitrarily selected for execution. However, each statement is executed infinitely often (weak fairness hypothesis).

Terminating programs are interpreted as programs that reach a fixed point, namely, a stable state. Any further execution step leaves the (stable) state unchanged.

The only UNITY statement is the assignment with several forms: conditional, quantified and enumerated. For instance, a conditional multiple assignment has the following syntax:

$$\begin{array}{lclcl} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & if & g_1 \\ & \sim & e_1^2, e_2^2, \cdots, e_n^2 & if & g_2 \\ & \cdots & \cdots & if & \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p & if & g_p \end{array}$$

Unlike Dijkstra's guarded commands, when several conditions are simultaneously true, the assigned values must be the same. Each assignment is deterministic. The assignment leaves the variable values unchanged if none of the conditions is true.

Enumerated assignments are a syntactic feature to cut long multiple assignments. For instance, $x := a \parallel y, z := b, c \parallel t := d$ is an equivalent notation for $x, y, z, t := a, b, c, d$.

Quantification is allowed, especially with parallel or exclusive assignments. For instance, the quantified assignment $\langle \parallel i : 1 \leq i \leq N :: A[i] := B[i] \rangle$ assigns the array B to the array A.

II.2 UNITY logic

A specification language is associated to the former operational language. It allows to specify safety and liveness properties as logical expressions. The program variables are operands of such predicates. These properties may be proved with respect to the program component by means of a set of theorems.

The UNITY logic is based on Hoare triples $\{P\}\ s\ \{Q\}$ [Hoa69] in which P and Q are predicates and s a statement. Such a triple has the following interpretation: if the statement s is performed from a program state satisfying P and terminates, the resulting state will verify the predicate Q.

Safety property syntax

Three basic relations allow to express safety properties, namely: **Unless**, **Stable** and **Invariant**. For a program *Prog* and predicates P and Q, we have the following definitions:

$$P\ \mathbf{Unless}\ Q \ \equiv\ \langle \forall s : s\ in\ Prog :: \{P \wedge \neg Q\} s \{P \vee Q\}\rangle$$

(if P holds at some point, P remains true at least until Q holds.)

$$\mathbf{Stable}\ P \ \equiv\ P\ \mathbf{Unless}\ false$$

(if P becomes true, it remains true forever.)

$$\mathbf{Invariant}\ P \ \equiv\ (initial\ conditions \Rightarrow P) \wedge (\mathbf{Stable}\ P)$$

(P is true at the program start and all over the execution.)

Liveness property syntax

The basic liveness relation is **ensures** with the following definition: for a program *Prog* and predicates P and Q

$$P\ \mathbf{Ensures}\ Q \ \equiv\ (P\ \mathbf{Unless}\ Q) \wedge \langle \exists s : s\ \mathbf{in}\ Prog :: \{P \wedge \neg Q\} s \{Q\}\rangle$$

(if P holds at some point, P remains true at least until Q becomes true and there exists one statement to eventually validate Q.)

This property allows to define a more powerful relation $\mapsto$ (**leads-to**). $P \mapsto Q$ means that if P holds at some point, Q eventually becomes true.

II.3 Example

We consider a program with two counters i and j. Their behaviour depends on each other.

```
**initially**  $i, j = 0, 10$;
**assign**
               $i := i + 1$
$[\!]$         $j := i$          **if** $i \neq 5$
```

This program verifies the following safety and liveness properties:

$\forall\, k$ **Stable** $i \geq k$	– i cannot decrease
$\forall\, k$ $i = k$ **Unless** $i = k + 1$	– i can only increase by 1 steps
$\forall\, k$ $i = k$ **Ensures** $i = k + 1$	– i actually increases by 1 steps
Invariant $i \geq 0$	– i is always non negative
Invariant $i \geq 0 \wedge j \geq 0$	– i and j are always non negative
$\forall\, k$ $j = k \mapsto j > k$	– j increases indefinitely

Note on invariant and "always true" properties

As defined in UNITY, a property I is invariant if it holds in the initial state of the program and it is maintained by any transition of the program (stable).

Thus, from a state where I holds, a transition must lead to a state where I still holds. The only knowledge about the starting state is that I holds. In particular, no assumption is made about the reachability of the state. If there exists a state from which I is not maintained, I is not invariant, even if this state is unreachable.

Therefore, a property may hold for all possible executions of a program and not be invariant (i.e. not be maintained from some unreachable state). For instance, in the example above, we do not have **Invariant** $j \geq 0$ although j is never less than zero for all possible executions of the program. Actually, from the (unreachable) state $(i, j) = (-1, 2)$, the second statement does not maintain $j \geq 0$.

A weaker form of invariants has been introduced that exactly fits the notion of *always true* properties [San91]. It is based on the idea of *strongest invariant.* The *strongest invariant* of a program is the strongest (for the imply relation) invariant property of that program. It is defined as the conjunction of all the invariants of the program. In other words, the *strongest invariant* holds in any reachable state and only in the reachable states. It may be used to characterize the reachable states of a program, and thus avoids the above difficulty. An *always true* property is then a property implied by the *strongest invariant.*

In practice, the main difficulty in proving that a given property is *always true* in a program is to find an invariant strong enough to imply the property, that is to specify precisely enough which are the reachable states. Some tools used in proving programs automatically allow to strengthen a given invariant (see VI.2) and even to compute the *strongest invariant* in particular cases (see VI.1).

III The design assistant

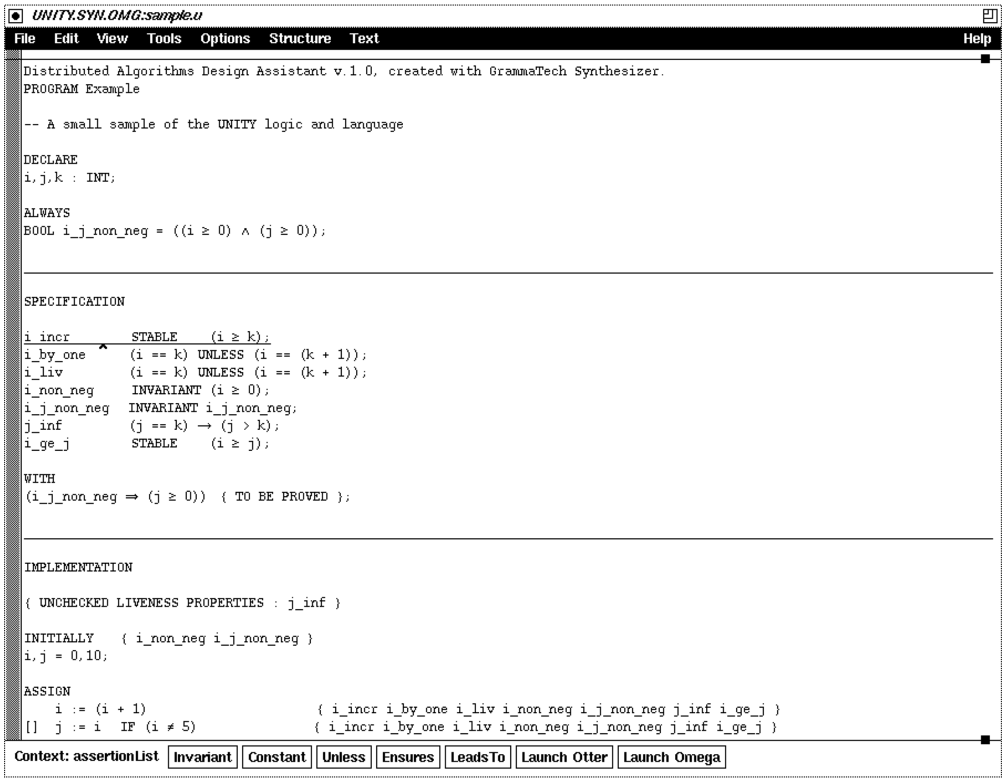
III.1 Description

The assistant appears as a syntax-directed editor with an X Window System interface for both the UNITY language and logic. It has been generated by the Grammatech Synthesizer Generator and provides all the functionalities of the editors created with this tool, described in [RT89].

The system handles a two-part text:

- an algorithm description in the UNITY language,
- a specification for both the safety and liveness properties of this algorithm in the UNITY logic. Each property is labelled with a name.

The goal is to control automatically the program with respect to all the specified properties.



```
Distributed Algorithms Design Assistant v.1.0, created with GrammaTech Synthesizer.
PROGRAM Example

-- A small sample of the UNITY logic and language

DECLARE
i,j,k : INT;

ALWAYS
BOOL i_j_non_neg = ((i ≥ 0) ∧ (j ≥ 0));

SPECIFICATION

i_incr        STABLE    (i ≥ k);
i_by_one      (i == k) UNLESS (i == (k + 1));
i_liv         (i == k) UNLESS (i == (k + 1));
i_non_neg     INVARIANT (i ≥ 0);
i_j_non_neg  INVARIANT i_j_non_neg;
j_inf         (j == k) → (j > k);
i_ge_j        STABLE    (i ≥ j);

WITH
(i_j_non_neg ⇒ (j ≥ 0))  { TO BE PROVED };

IMPLEMENTATION

{ UNCHECKED LIVENESS PROPERTIES : j_inf }

INITIALLY   { i_non_neg i_j_non_neg }
i,j = 0,10;

ASSIGN
    i := (i + 1)                      { i_incr i_by_one i_liv i_non_neg i_j_non_neg j_inf i_ge_j }
[]  j := i   IF (i ≠ 5)               { i_incr i_by_one i_liv i_non_neg i_j_non_neg j_inf i_ge_j }
```

Figure 1: User interface of the assistant

Figure 1 shows the user interface of the environment. The user can edit a syntactically correct text in the UNITY formalism. The editor translates this text into an attributed abstract tree. Incremental analysis is performed by updating attribute values throughout the current tree in response to each editing transaction (text editing, system command, ...).

In this UNITY editor, the attribute evaluation performs the following semantic actions:

- static semantic checking (type control);
- generation of formulas that express the consistency between the specification part and the implementation part;
- calls to a theorem prover to verify the previous formulas. Currently, the user can choose between two tools: the Otter theorem prover (see IV) and the Omega calculator (see V).

The labels of the properties to be checked are displayed after each program statement. In order to check a property, the user must first select this property with the mouse, and then call a theorem prover by clicking on the corresponding button. If the proof of the property succeeds, the corresponding label disappears. For instance, in figure 1, the **Stable** property *i_incr* is selected and its proof may be launched by clicking on a button located at the bottom of the window.

Currently, the environment only manages a subset of the UNITY formalism. The language accepts integers and booleans but neither arrays nor statement quantification are allowed. At the logical level, the temporal operator $\mapsto$ is not handled. Such proofs involve too many practical and theoretical difficulties (transitive closure calculus, for example).

In addition to the usual sections of a UNITY program (**declare**, **always**, **initially** and **assign**) a **with** section provides a direct interface with the theorem provers. This feature is useful for proving (non temporal) predicates with program variables as operands.

III.2 Example

The tool has been tested in different areas including distributed systems, hardware design or robotics. We detail an infinite-state synchronisation problem, namely the parking problem.

The parking model [AHV83] illustrates the problem of critical resource allocation to concurrent processes. The parking size C represents the number of available resources. A car arrival event simulates a resource request. An effective car entrance into the parking represents a resource allocation and an exit stands for a resource release.

The specification is based upon event counters respectively representing the number of arrivals A, entrances E and departures D. The basic invariant property expresses that the number of cars inside the parking always remains less then or equal to the parking capacity, that is $E - D + F = C$ where F is the number of free places and E, D and F are non negative.

A distributed implementation can be seen as a parking with several access points. Each access point manages its own local counters of events and free places, namely a_i, e_i, d_i and f_i with the representation invariant:

$$A = \sum_i a_i, \quad E = \sum_i e_i, \quad D = \sum_i d_i, \quad F = \sum_i f_i$$

In the following program, we consider a three access point parking. UNITY assignments update event counters and the different access points exchange free places by means of a virtual ring topology.

```
Program Parking

declare
     C : INT;
     a1,a2,a3,e1,e2,e3,d1,d2,d3,f1,f2,f3 : INT;
     ka,ke,kd : INT
```

```
always
    INT D = ((d1 + d2) + d3),
    INT E = ((e1 + e2) + e3),
    INT A = ((a1 + a2) + a3),
    INT F = ((f1 + f2) + f3)
```

```
Specifications
    Inv_gl_Impl    invariant  (((E − D) + F) = C),
    f123_ge_0      invariant  (((f1 ≥ 0) ∧ (f2 ≥ 0)) ∧ (f3 ≥ 0)),
    a123_ge_0      invariant  (((a1 ≥ 0) ∧ (a2 ≥ 0)) ∧ (a3 ≥ 0)),
    e123_ge_0      invariant  (((e1 ≥ 0) ∧ (e2 ≥ 0)) ∧ (e3 ≥ 0)),
    d123_ge_0      invariant  (((d1 ≥ 0) ∧ (d2 ≥ 0)) ∧ (d3 ≥ 0)),
    a1_Croit       (a1 = ka)  unless  (a1 > ka),
    e1_Croit       (e1 = ke)  unless  (e1 > ke),
    d1_Croit       (d1 = kd)  unless  (d1 > kd)

with
    (((((E − D) + F) = C) ∧ (F ≥ 0)) ⇒ ((E − D) ≤ C))
```

```
implementation

initially
    a1,a2,a3,e1,e2,e3,d1,d2,d3 = 0,0,0,0,0,0,0,0,0;
    f1,f2,f3,C = 2,2,1,5

assign
    a1 := (a1 + 1)
[]
    e1,f1 := (e1 + 1),(f1 − 1)              if  ((e1 < a1) ∧ (f1 > 0))
[]
    d1,f1 := (d1 + 1),(f1 + 1)              if  (d1 < e1)

[]
    a2 := (a2 + 1)
[]
    e2,f2 := (e2 + 1),(f2 − 1)              if  ((e2 < a2) ∧ (f2 > 0))
[]
    d2,f2 := (d2 + 1),(f2 + 1)              if  (d2 < e2)

[]
    a3 := (a3 + 1)
[]
    e3,f3 := (e3 + 1),(f3 − 1)              if  ((e3 < a3) ∧ (f3 > 0))
[]
    d3,f3 := (d3 + 1),(f3 + 1)              if  (d3 < e3)

[]
    f1,f2 := (f1 − 1),(f2 + 1)              if  (f1 > 0)
[]
    f2,f3 := (f2 − 1),(f3 + 1)              if  (f2 > 0)
[]
    f3,f1 := (f3 − 1),(f1 + 1)              if  (f3 > 0)

end {Parking}
```

With a 64 MB SUN SPARCstation 5, the proof of all the properties in the **Specifications** section requires 114 calls to `Otter` taking a total time of 2 minutes and 20 seconds, or 102 calls to `Omega` taking a total time of 20 seconds.

IV The theorem prover Otter

IV.1 Presentation

By means of attribute grammars, the editor part of the tool performs an attribute evaluation based on *weakest precondition* (*wp*) [Dij76] and generates a set of predicates. These predicates have to hold for the program to respect its specification.

Therefore, we need to use a theorem prover to check the predicates validity. After having studied different tools ([KZ87, GG91, Ast92]), based on rewriting or resolution, we have chosen the `Otter` (*Organized Techniques for Theorem-proving and Effective Research*) theorem prover [McC90, McC91], mainly because of its simplicity (that makes easier the interfacing with the editor), its speed, and its ability to work *entirely* automatically. We emphasize that the formulas to prove automatically are numerous but quite simple.

IV.2 Checking specifications

The `UNITY` specification language is based on the respect of state formulas at different points of a program (*Hoare-triples*). This kind of logic is very useful, and Dijkstra introduced a useful tool for reasoning about it, known as *weakest precondition* (*wp*) [Dij76]. For a state formula Q and a statement s, $wp\,(s, Q)$ is the *weakest* state formula that must hold *before* s to ensure that Q holds *after* s. Therefore, a state formula P is a sufficient precondition for Q with respect to s if and only if it is a *stronger* formula than $wp\,(s, Q)$, that is:

$$\{P\}\ s\ \{Q\} \quad \textit{iff} \quad P \Rightarrow wp\,(s, Q)$$

The function $Q \rightarrow wp\,(s, Q)$, from predicates to predicates, expresses the semantics of the statement s, interpreted as a predicate transformer.

While reasoning about `UNITY` programs, wp is very useful because of its computability. One can automatically compute $wp\,(s, Q)$ and transform a specification problem $\{P\}\ s\ \{Q\}$ into the logical formula $P \Rightarrow wp\,(s, Q)$. This deductive approach can deal with infinite state spaces (in contrast with model checking), as, for example, in the parking program (see III.2).

Application to UNITY:

As we have seen, `UNITY` only provides a single kind of statement, namely the multiple assignment. Therefore, we only have to define wp for such assignments. A formula Q will hold after an assignment s if the formula it comes from through s held before executing s. Thus, the wp for a formula Q is obtained by variable substitution in Q:

$$wp\,(v_1, \cdots, v_n := e_1, \cdots, e_n \ \ , \ \ Q) \quad \equiv \quad Q|^{v_1,\cdots,v_n}_{e_1,\cdots,e_n}$$

This method still works with guarded assignments. An assignment with n guards can be seen as n different assignments, each one having to fit the specification. The wp has to be sufficient whichever assignment is actually performed, but the guard can strengthen the precondition for the selected assignment. Finally, one must consider the *nop* statement in the case where no guard holds (no assignment is performed).

For an assignment s of the form:

$$\begin{array}{rcll} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & if \;\; g_1 \\ & \sim & e_1^2, e_2^2, \cdots, e_n^2 & if \;\; g_2 \\ & \cdots & \cdots & if \;\; \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p & if \;\; g_p \end{array}$$

we obtain:

$$\{P\}\ s\ \{Q\}\ \textit{ iff }\ \bigwedge_{i=1}^{p}(P \wedge g_i \Rightarrow wp\,(\alpha_i, Q))\ \bigwedge\ (P \wedge \neg g_1 \wedge \cdots \wedge \neg g_p \Rightarrow Q) \quad (1)$$

where α_i is the assignment $v_1, v_2, \cdots, v_n \;:=\; e_1^i, e_2^i, \cdots, e_n^i$ and where each $wp\,(\alpha_i, Q)$ is computed by substitution over Q.

In the UNITY logic, a constraint of the form X **Unless** Y requires to check the Hoare triple $\{X \wedge \neg Y\}\ s\ \{X \vee Y\}$ for all assignments s. For a *nop* statement, this boils down to verify $X \wedge \neg Y \Rightarrow X \vee Y$ which is a tautology.

Therefore, for any **Unless** property, the last conjunction of formula 1 needs not be computed, since this part of the formula expresses the *nop* case. This simplification remains possible with the **Stable**, **Constant** and **Invariant** constraints which are special cases of the **Unless** relation.

V The Presburger calculator Omega

V.1 Presentation

Reactive programs, like those we consider, often use integer counters. However, classical theorem provers, like Otter, are not well suited to symbolic arithmetic calculations. Therefore, we decided to try another kind of tool.

Omega [Pug92, KMP$^+$94] is a calculator for Presburger formulas. It is not a theorem prover, but manipulates on sets and tuple relations based on Presburger arithmetic, a class of logical formulas built from affine constraints over integer variables. Such a formula is of the form:

$$\begin{array}{llll} & \phi \wedge \psi & \textit{or} & \phi \vee \psi \\ \textit{or} & \neg\phi & \textit{or} & \exists\, v_1, v_2, \cdots, v_n : \phi \\ \textit{or} & \forall\, v_1, v_2, \cdots, v_n : \phi & \textit{or} & C \end{array}$$

where ϕ, ψ are Presburger formulas and C is a logical constraint of the form $e_1, e_2, \cdots, e_n \;\; op \;\; e'_1, e'_2, \cdots, e'_n$ with *op* among $=, \neq, <, >, \leq, \geq$. The e'_i are affine expressions on the variables.

A set is defined as follow:

$$\{[u_1, u_2, \cdots, u_n] : \phi\};$$

where ϕ is a Presburger formula, and a tuple relation is of the form:

$$\{[u_1, u_2, \cdots, u_n] \rightarrow [v_1, v_2, \cdots, v_n] : \phi\};$$

`Omega` then offers a variety of set and relational operators to handle these sets and relations, such as **union**, **subset**, **join**, etc.

V.2 Checking specifications

We have added a second interface to the syntax directed editor that allows us to check `UNITY` specifications with `Omega`. We translate preconditions and postconditions into sets of state, and the multiple assignment into a tuple relation.

To check an assertion like $\{P\}\ s\ \{Q\}$, one can use the notion of *strongest postcondition* (*sp*). For a statement s and a predicate P, $sp\,(s, P)$ is the *strongest* state formula that holds after having executed s from a state in which P holds. Therefore, a formula Q is a necessary postcondition of P with respect to s if and only if it is a *weaker* formula than $sp\,(s, P)$, that is:

$$\{P\}\ s\ \{Q\}\ \ \textit{iff}\ \ sp\,(s, P) \Rightarrow Q$$

For an assignment s of the form:

$$\begin{array}{rcll} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & if\ \ g_1 \\ & \sim & e_1^2, e_2^2, \cdots, e_n^2 & if\ \ g_2 \\ \cdots & & \cdots & if\ \ \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p & if\ \ g_p \end{array}$$

we obtain:

$$\{P\}\ s\ \{Q\}\ \ \textit{iff}\ \ \bigwedge_{i=1}^{p} (sp\,(\alpha_i, P \wedge g_i) \Rightarrow Q)\ \bigwedge\ (P \wedge \neg g_1 \wedge \cdots \wedge \neg g_p \Rightarrow Q)$$

where α_i is the assignment $v_1, v_2, \cdots, v_n\ :=\ e_1^i, e_2^i, \cdots, e_n^i$.

Instead of handling a predicate directly, one can consider the set of the states for which the predicate holds. A state is the tuple built from the values of the variables in the program. An assignment is a function from states to states. The formula $sp\,(s, P) \Rightarrow Q$ is handled as the set relation $\mathcal{S}\,(\mathcal{P}) \subset \mathcal{Q}$ where $\mathcal{S}$ is the function corresponding to the statement s and $\mathcal{P}$ and $\mathcal{Q}$ are the sets of states corresponding to the formulas P and Q. This approach avoids to compute the *strongest postcondition* explicitly and allows to compute neither *weakest precondition* nor *strongest postcondition* at the editor level.

VI Related work

We distinguish two main approaches in the program proof domain.

On the one hand, the programmer has to describe a complete and formal proof by hand. Afterwards, this proof is automatically checked by a verification system. In this case, no assistance is provided in generating the proof. Systems like *HOL* [GM93], *Deva* [CJL$^+$89] and *PVS* [SOR93] are used to perform such verifications. The tool does not handle both specification and program, but only the hand proof resulting from such a confrontation. `UNITY` proof systems have been described with *HOL* [And92] and with the *Boyer-Moore logic* [Gol92].

On the other hand, the proof construction from a program and its specification is handled at the tool level. The *TLP* system [Eng94], based on the temporal logic of actions (*TLA*) [Lam94], illustrates this one approach. We now focus on two others projects with a closer relationship to our work. The first one, called *UNITY Verifier* [Kal94a, Kal94b], is also based on the `UNITY` programming notation, but uses model checking to perform the proofs. The second one, called ST$_{\mathrm{e}}$P [Ma94], allows to validate a program with respect to a linear temporal logic specification.

VI.1 The UNITY Verifier

Model checking is a traditional way for automatically proving programs. It consists in checking that a model (a program) is correct with respect to a formula (a specification) usually by exhaustively checking all possible transitions.

The *UNITY Verifier* is a model checker intended for interactively verifying concurrent programs written in `UNITY`. Through a `X Window System` interface, it operates on a restricted part of the `UNITY` formalism (no array, no statement quantification, no **always** section) and, due to its model checking algorithm, it only handles finite state programs.

Its main attractive feature is its ability to generate the *strongest invariant* (see II.3) for a finite state program and thus, to directly handle *always true* properties. However, *always true* properties raise compositional issues [Col94] and non finite state programs cannot be handled, especially those with integer variables.

VI.2 The ST$_{\mathrm{e}}$P system

The ST$_{\mathrm{e}}$P project (*Stanford Temporal Prover*) is a system being developed to support the computer-aided formal verification of concurrent and reactive systems based on fair state transition models and temporal logic specifications. Thus, it handles both a linear temporal logic specification and a program written in a parallel language called *SPL*.

The proofs either can be entirely automatic or can be user directed. The system combines model checking and deductive methods to handle non finite state programs. The programming language *SPL* is more complete and deterministic than `UNITY`, and provides synchronisation facilities (such as semaphores, for instance), but `UNITY` remains sufficient to express parallelism with a high degree of non determinism.

Finally, an interesting feature of the system is its ability to generate invariants from the program text and to strengthen user given invariants.

VII Conclusions

With regard to this experiment, the use of several provers with their own capabilities seems to be a more practical way than looking for a very general tool. The ST$_{\mathrm{e}}$P system also adopts this approach. Therefore, we think of using other provers, especially the Boyer and Moore theorem prover [BM88].

Automatic invariant generation (as, for instance, in the *UNITY Verifier* and the STeP system) provides an important proof assistance. However, the *strongest invariant* cannot be generated by computation for non finite state programs. In our approach, refinements may allow a stepwise invariant derivation.

Hundred of proofs were necessary to check the small programs described above. Large programs remain out of reach. In front of this difficulty, we are studying two possibilities:

- investigate modularity. Modules are proved independently and further linked with each other. Consequently, sound compositional operators must be available. `UNITY` provides such operators and allows this research direction.

- define generic sound refinements.

In both cases, the assistant will have to check if the refinements are allowed. We investigate both these directions. Generic refinements have already been defined for mapping a centralized algorithm to a distributed architecture [FMP93].

References

[AHV83] F. André, D. Herman, and J.-P. Verjus. *Synchronisation de programmes parallèles.* Dunod Informatique, 1983.

[And92] F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, 1992.

[Ast92] O. Astrachan. Exploring Model Elimination Theorem Proving. Technical Report CS-1992-22, Duke University, Durham, ,NC USA, December 1992.

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[CJL$^+$89] J. Cazin, R. Jacquart, M. Lemoine, P. Michel, and P. Maurice. *Method driven programming.* G.X. Ritter Editor, Information Processing 89. Elsevier Science Publishers B.V. (North-Holland), 1989.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design : A Foundation.* Addison-Wesley, 1988.

[Col94] P. Colette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications. Application to UNITY.* Thése de docteur en sciences appliquées, Faculté des Sciences Appliquées, Université Catholique de Louvain, June 1994.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.

[Eng94] U. Engberg. *TLP Manual.* Computer Science Department, Aarhus University, Ny Munkegade, Building 540, DK-8000 Aarhus C, DENMARK, release 2.5a edition, May 1994.

[FMP93] M. Filali, Ph. Mauran, and G. Padiou. Raffiner pour répartir. Technical Report 93-29/R, Institut de Recherche en Informatique de Toulouse, France, November 1993.

[GG91] S. J. Garland and J. V. Guttag. A Guide to LP, The Larch Prover. Technical Report 82, Systems Research Center, Digital Equipment Corporation, December 1991.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL : A Theorem Proving Environnement for Higher Order Logic.* Cambridge University Press, 1993.

[Gol92] D. M. Goldschlag. *Mechanically Verifying Concurrent Programs.* PhD thesis, University of Texas at Austin, May 1992.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580,583, 1969.

[Kal94a] M. Kaltenbach. *Model Checking for UNITY : The UV System.* Department of Computer Sciences, The University of Texas at Austin, revision 1.10 edition, May 1994.

[Kal94b] Markus Kaltenbach. *The UV System. User Interface Manual.* The University of Texas at Austin, revision 1.13 edition, December 1994.

[KMP$^+$94] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Calculator and Library, version 0.90.* Dept. of Computer Science and Institute for Advanced Computer Studies, Univ. of Maryland, College Park, MD 20742, November 1994.

[KZ87] Deepak Kapur and Hantao Zhang. *RRL : Rewriting Rule Laboratory User's Manual.* Department of Computer Science, State University of New York at Albany, The University of Iowa, June 1987.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Ma94] Z. Manna and al. SteP : The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Department of Computer Science, June 1994.

[McC90] William W. McCune. OTTER 2.0 Users Guide. Technical Report ANL-90/9, Mathemetics and Computer Science Division, Argonne National Laboratory, Illinois, March 1990.

[McC91] William W. McCune. What's New in OTTER 2.2. Technical Report ANL/MCS-TM-153, Mathemetics and Computer Science Division, Argonne National Laboratory, Illinois, July 1991.

[Pug92] W. Pugh. The Omega Test : a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, August 1992.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[San91] B. A. Sanders. Eliminating the Substitution Axiom from UNITY Logic. *Formal Aspects of Computing*, 3(2):189–205, April-June 1991.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. A tutorial on Specification and Verification Using PVS (Beta Release). Technical report, Computer Science Laboratory, SRI International,, Menlo Park, CA, March 1993.

Implementing FS_0 in Isabelle: adding structure at the metalevel

Seán Matthews
Max-Planck-Inst., Im Stadtwald
66123 Saarbrücken, Germany
<sean@mpi-sb.mpg.de>

April 30, 1995

Abstract Often the theoretical virtue of simplicity in a theory does not fit with the practical necessities of working with it. We present as a case study an implementation in a generic theorem prover (Isabelle) of a theory (FS_0) which at first sight seems to lake all the facilties needed to be practically usable. However, we show that we can use the facilties available in Isabelle to provide all the structuring facilities (modules, abstraction, etc.) that are needed without compromising the simplicity of the original theory in any way, resulting in a thouroghly practical implementation. We further argue that it would be difficult to build a custom implemenation as effective.

§1 Introduction

A great many logics have been proposed as tools in computer science, especially for all sorts of formal, machine checked reasoning. However, if we try to implement these theories in some practical manner, we find that what has been proposed by theoreticians as a practical tool has to be augmented in all sorts of ways before it really becomes a practical tool. Essentially, the basic tools of structured programming and other facilties need to be imported. Unfortunately, this means that a proof theory which originally could be summarised on a page or so, grows to fill a manual, and is augmented with descision procedures an other extras, which can be difficult to verify.

A suggested solution to this problem is what have become known as 'Logical frameworks': systems designed to be suitable for implementing a wide range of different logics easily, so that they can be presented in a uniform manner to users, allowing the same theorm proving facilties to be reused across a wide range of implemented theories, instead of having to be rebuilt from scratch each time. The basic idea of a logical framework (we will be concentrating in particular here on the Isabelle system) is to make it easier to implement logics, via some sort of equation of the Syntax + Axioms + Rules = Theorem prover form; but in fact we get more than that. Because a logical framework based system is generic, its implementors can afford to invest in much more powerful facilties, since they are likely to be reused in a range of different contexts (in fact, since they don't know what those contexts will be, they have to provide powerful and general tools so as to improve the possibility that they are usable any particular context). But since this machinery is developed prior to the implementation of any particular theory it must be independent of the details of such theories. Thus given an initial proposal for a theory to be implemented, a logical framework based system should not only be able to provide a much quicker implemenation (via the equation given above) but also might be able to provide some, or all of the structuring facilities that are needed for practical proof development, so that they are actually independent of the theory to implemented. If this is so, then the theory itself does not have to be extended, and thus can be implemented in a way that is closer to the original proposal.

Feferman's FS_0 [3] is such a theory, in this case as nominated as a suitable vehicle for machine checked metatheory. While it is simple (20 or so axioms in a three sorted first-order logic), it so primitive that it is not at all clear that it is practically usable (especially in the form that Feferman gives it). It comes with none of the structuring facilities which we usually depend on when developing large theories (modularity, abstraction, etc.), but these have to be provided somehow in the implementation. We show how we can we can get these directly from Isabelle.

§1.1 *Contributions* We see this paper as making the following contributions. We show the effectiveness of a generic theorem prover such as Iabelle for dealing with an unusual logic and how its facilties can be exploited to provide a great deal of high-level structuring of development in such a theory, without having to introduce such structuring facilties into the logic to be implemented itself, and thus complicating it unnecessarily. We also claim that this case example is an argument that implementation in a generic system can in the end sometimes be more effective than a custom implementation, since so many of the facilties our implementation provides are exploit the powerful facilities that a generic system has to provide (in fact the design is directly driven by the facilities Isabelle provides).

Secondly, and independently We demonstrate that FS_0 is a plausible theory for real computer supported theory development, by presenting the first practical implementation[1]

§1.2 *Outline of paper* The outline of the rest of this paper is as follows: In §2 and §3 we briefly describe FS_0 as background, in §4 we discuss the facilities that we want to provide for theory definition in our implementation, in §5 we describe how we have provided these, in §6 we describe some of the tools for proving theorems in FS_0, in §7 we then briefly outline some of the theory development we have performed, and in §8 we present our conclusions.

§2 The theory FS_0

FS_0 is a theory in the tradition of Gödel's incompleteness results: one can think of it as a 'rational reconstruction' of the results of the preliminary development that that Gödel did in arithmetic (i.e. building tools for doing 'gödel-numbering') to prove his theorems, and is a conservative extension of primitive recursive arithmetic. The details can be found in [3].

A first impression is that the theory is very simple, and, as we have said, there are various reasons for not adulterating that simplicity, so it should be implemented pretty much as it stands: as a three-sorted classical first order, finitely axiomatisable theory of s-expressions, primitive recursive functions and recursively enumerable classes, that resembles Pure Lisp. A summary outline of Feferman's definition is as follows:

- There is the sort S of s-expressions. This is contains a leaf object nil, and is closed under a pairing function $(\_,\_)$; equality is defined in the obvious way over s-expressions.
- There is the sort F of functions. All functions are of the form $S \to S$ and function application is denoted by ‘, so that if f is a function and t is an s-expressions then $f`t$ is a function application of sort S. we have a small set of basic functions on s-expressions, Id (identity), π_1, π_2 (car and cdr), and Dec (decide). Most of these should be well known, apart, perhaps, from the last which behaves as follows:

$$Dec`(((a,b),c),d) = \begin{cases} a = b & \to c \\ a \neq b & \to d \end{cases}$$

There are also constant functions $K(a)$ of sort $S \to S$, where $K(a)`b = a$.

The basic functions can be combined using combinators of the form $F \times F \to F$, of which there are three, as follows. Composition, $\_ * \_$, where $(f * g)`t = f`(g`t)$; pairing, $[\_,\_]$, where $[f,g]`t = (f`t, g`t)$; and structural recursion, $Rec(\_,\_)$, where,

[1]So far as we, or Feferman, know.

if $h = Rec(f, g)$,

$$\begin{aligned} h`nil &= nil \\ h`(a, nil) &= f`a \\ h`(a,(b,c)) &= g`((((a,b),c), h`(a,b)), h`(a,c)) \end{aligned}$$

Finally, equality on functions is defined extensionally.

- There is the sort C of classes. We are given the class containing only nil, i.e. $\{nil\}$ and have binary intersection and union $\cup$ and $\cap$, as well as the inverse image of a class c under a function f, $f^{-1}c$ where $t \in f^{-1}c \leftrightarrow f`t \in c$.
  More complicated, we can also build recursively enumerable classes $Ic(a, b)$, which is the class containing a and closed under the rule $t_1, t_2/t$, where $((t, t_1), t_2) \in b$. Equality, and the subset relation, on classes are defined extensionally.
- Finally, we have induction. Over the universe,

  $$nil \in c \rightarrow \forall a, b(a, b \in c \rightarrow (a, b) \in c) \rightarrow \forall x(x \in c)$$

  and over inductively defined sets,

  $$c' \subset c \rightarrow \forall a, b, c(b, c \in c \rightarrow ((a, b), c) \in c'' \rightarrow a \in c) \rightarrow Ic(c', c'') \subset c.$$

FS_0 is intended for building Gödel-encoding of formal languages and theories, and this is done by building classes that define well formed, or provable, formulae; e.g. we could define the class of all well-formed formulae of first order logic (encoded as s-expressions). When we try to do this, however, it soon becomes clear that enormous and painstaking effort is needed to build these by hand and at the end the definitions are not are not intuitive; further, when we try to prove that what we have produced has the properties that we want, we find, almost invariably, that it is full of hard to correct errors. Further, there is no way to structure developments very effectively.

We do, however, have a theorem that characterises which classes we are able, in theory, to define, in the form of a comprehension principle. Given the definition

Definition 1 *We define the class of Σ^0_1-formulae to be the class containing equalities and inequalities between S sorted terms, and set membership, and closed under disjunction, conjunction and existential quantification of S sorted variables.*

then we have a comprehension theorem,

Theorem 1 (Feferman) *Given a Σ^0_1-formula $P[x]$ with one free variable x of type S, there exists a class c, such that $FS_0 \vdash x \in c \leftrightarrow P[x]$.*

Σ^0_1-formulae provide an expressive specification language: with them we can define any recursively enumerable set of s-expressions (which includes sets of provable formulae, of course). But this comprehension result is a meta-theorem; it is neither a schematic axiom nor a theorem of FS_0, and while we could add it as an axiom to the theory directly, that is precisely the sort of extension that we want to avoid. How we provide comprehension is in fact one of the main facilities that we document.

§3 Isabelle

The Isabelle pds [5] comes as a collection of extensions for an SML programming system. It cannot accurately be described as a program that just happens to be written in SML; the relationship between the two is much closer than that. We work with Isabelle directly through the SML command line, meaning that we also have direct access to SML to program extensions, or as a tactic language; a powerful but safe facility—the strong typing acting as an effective prophylaxis against accidental, unnoticed damage being done to an implemented theory.

The system is based on the observation dating back to the Automath [2] project, that a good foundation for a generic pds is type-theory, or typed lambda calculus, since it is possible to encode the proof systems of many logics (particularly those that can be reasonably expressed in a natural deduction style) directly in the $\forall, \rightarrow$ fragment of such a theory and that in doing this we finesse the traditionally ubiquitous problems with variable binding or capture, and substitution.[2] For details of the type theory provided by Isabelle see [5]. We can think of Isabelle as a collection of tools for deriving and manipulating terms in type theory.

It is important to stress several unusual points about Isabelle. The terms that are derived in it are terms in a typed lambda calculus, some of which happen to encode terms in a declared logic, but these are not the only terms Isabelle works with: other terms represent rules in declared logic, and these can be derived too. In fact perhaps the best way to think of Isabelle is as basically a system for deriving rules rather than theorems; theorems are simply a degenerate case of rules with no premises and no schematic variables. Also, Isabelle is not, based on rewriting; lambda terms (i.e. encoded rules) are combined together rather by (higher order) unification and proofs are thus built by resolving rules represented as implicational terms in the type-theory against forumalae to be proved (also terms in type theory). This has some interesting effects; for instance we can use the same rule for both forward and backwards proof (since unification is 'bidirectional'), and it is possible to have metavariables in formulae that are to be proven, which can be instantiated in the course of the proof, picking up information from rules as they are used.

The details of how Isabelle allows access to the type theory through SML are as follows:

- A new data-type `theory` for packaging the collections of constant definitions that make up a declared theory.
- Functions for combining and extending `theory`s; i.e. if we have defined a `theory` encoding first-order logic and call it `Pred`, we can extend it with definitions for natural numbers to generate a new theory which we might call `Nat`, or combine `Nat` with, say, `List` to generate a theory of naturals and lists which inherits all the theorems that have been proven for either.
- A new data-type `thm` for the axioms of the `theory`s we have defined, and also for the terms we have derived in the 'theory's we have defined. In fact, and importantly, in Isabelle there is no distinction between derived and basic theorems of a theory, they are all just `thm`s.

Along with the the basic system, we also get some tools for building things like rewrite systems, and a few predefined logics, including sorted classical first-order logic.

Since classical first order logic is already available, we can immediately define, as an extension of it, a basic system for FS_0, all we have to do is declare sorts S, F, C then we can type in the axioms literally (allowing for the restrictions that a typewriter imposes compared to a typesetting system) as we find them in [3]. And we have a naïve implementation of FS_0.

§4 The basic implementation

As we have already said, a naïve implementation of FS_0 is not usable, but we can take stock of what it does make available (in Isabelle).

What this mostly amounts to is an effective method for modularising development. We have said that in Isabelle we can define a theory as an extension of another; thus the idea of a theory simply as a definitional extension of another is very natural: we simply add a new constant, and make it equivalent to the formula or term that it is abbreviating in the new extended theory.

Now, since a theory in FS_0 is basically a collection of FS_0 terms, we can take over this facility for abbreviation as definitional extension in a new theory and use it

[2]Of course, FS_0 is in a completely different tradition to this, and we have to build our own binding mechanism, but is intended for different purposes—we do find it pleasing that one framework logic should be so effective for implementing another.

for defining new FS_0 theories, each of which *is* a new Isabelle theory, albeit only a definitional extension. For a collection of classes, functions and s-expressions that we want to define as a module, we make a definitional extension of some earlier theory (which might be root FS_0, or itself some extension) and package these together as a new theory. Then we can prove the basic theorems that show that the definitions have the properties we want. From then on, all the the messy details of the implementation work can be hidden behind the abbreviations for the definitions, and theorems about them. And since we can combine these theories in Isabelle, with the result inheriting the theorems of its ancestors, we have a simple but effective tool for structuring the development of large theories as collections of small ones.

There are two ways to define a new theory as an extension of an old. The the first, 'basic' way is to use the `extend_theory` function that comes with the system. This is messy, since it takes a large and complicated collection of half a dozen or more arguments. The second is to use the Isabelle front end; this is a preprocessor that takes files which contain the equivalent of the information needed by `extend_theory`, but in a much more readable format, generates the theory and packages it, along with the various new axioms and other declarations that have been added, as an SML structure. Unfortunately neither of these methods is really suitable. The second method is not suitable because to use it we need to type the details of terms in by hand, and we have already explained why we want to avoid that). The first is unsuitable not only because here too, we would need to type the terms in by hand, but because it is simply too complicated (since we are only interested in definitional extensions).

§4.1 *What do we need?* At this point we have to consider in more detail exactly what we need to be able to do in an implementation. We automatically have a way to structure theories declared in FS_0, but anything else has to be built. And we know that we want to avoid having to construct, by hand, large terms to be assigned to abbreviations.

If a collection of definitions is constructed by hand (as happened in [4]) then the first thing that has to be done is to prove a collection of theorems describing (and checking) what those definitions actually do; i.e. translating them back into logical propositions. For instance we know from the comprehension theorem (theorem 1) that there is a close relationship between FS_0 classes and a certain class of logical formulae, and we have explained that this relationship is central to how we use the theory, so, if possible, we would like direct access to it. But as well as classes, we also want to define new functions, where there is no clear relationship like for classes. However, the idea that allows us to provide comprehension can be generalised (if not so elegantly) to provide a mechanism for generating terms from statements of their extensional properties.

The facilities for constructing classes and functions that we have implemented are very effective for connecting a defined object to a formula giving its properties. However they are also 'bottom up', since they build objects out of basic components. This is very safe, or course, since it ensures that everything we define *is* a definitional extension of FS_0. This means that the 'top down' approach to development is not possible. Thus we also provide a way to add, temporarily, new constants to the theory, along with new axioms, instead of just definitional extensions perhaps so that development on it can be postponed, or maybe continued in parallel.

We shall describe each of these in turn in the next section.

§5 Building definitions

Thus we have defined a new function `extend_FS0_theory` (by analogy with `extend_theory`) which takes the theory to be extended and a list of definitions and returns a package of a new theory and an association list of useful theorems that we have been able to generate automatically. Currently four sorts of definitions can be put on this list, and we discuss them one at a time.

§5.1 *Simple constants* The first sort of definition allows the definition of simple constants where we can type the body of the definition in whole.

For instance, in a theory of natural numbers, the natural numbers can be modeled as lists of *nils*; i.e. $0 \rightsquigarrow nil$, $s(0) \rightsquigarrow (nil, nil)$, $s(s(0)) \rightsquigarrow (nil, (nil, nil))$ etc. Then on the list of definitions would be the declarations

```
def("zero", "nil"),
def("s", "[const(nil),Id]")
```

which results in the new theory containing the new name *zero* for *nil* and the new function s, where $s`t = (nil, t)$. Note that this last fact is a theorem that we have to prove, `def` declarations are so general that it is not possible to extract any extra information from them; however we can see the beginning of an abstract interface here: we can try to provide a set of theorems that talk about the abstract behaviour of s and *zero* in the natural numbers, without regard to their implementation.

§5.2 *Comprehension* A `def` declaration is not really different from the sort of declarations that the standard Isabelle front end can handle. More interesting is comprehension. We would like to make this available in some convenient way. The secret to doing this is to examine the proof (on paper) of the comprehension theorem. This shows, by induction on the structure of Σ_1^0-formulae, that it is always possible to construct a suitable class. Most of the cases are easy; for instance there is an obvious equivalence between $\cap, \cup$, and $\wedge, \vee$. The most tricky case is for $\exists$, were we have to show that given $x \in c \leftrightarrow P[\pi_1`x, \pi_2`x]$, then there is some construction $Ex(c)$ such that $x \in Ex[c]$ iff $\exists y.P[x, y]$. The construction of Ex is a bit tedious but not impossible (see [3] for the details).

Unfortunately the type theory of Isabelle is too weak to formalise all of this argument, since it does not support induction. The induction is the only thing that cannot be formalised though; all the step cases are provable; i.e. we can derive rules for each possible reduction step needed by the proof. Then, since rules in Isabelle are implicational formulae in higher order logic, and proofs are built by resolving those rule against the formula to be proven, the equivalent of the induction can be implemented as a simple backchaining algorithm (which is, in fact, deterministic, since there is exactly one rule that resolves against each case in the definition of Σ_1^0-formulae.

Thus for existential quantifiers, we can prove the rule

$$\frac{\forall x(x \in z \leftrightarrow P[\pi_1`x, \pi_2`x])}{\forall x(x \in Ex[z] \leftrightarrow \exists a(P[a, x]))}$$

(given some z such that..., then there is some z' (actually $Ex[z]$) such that ...).

Thus, if we wanted to define the class of the 'less than' relation, we could start with a goal of the form

$$\forall x(x \in ?\mathbf{c} \leftrightarrow \exists w, y, z(x = (y, z) \wedge plus`(x, w) = z))$$

(where $?\mathbf{c}$ is a metavariable hole in the goal). We can immediately apply the rule for the existential case, which reduces the goal to

$$\forall x(x \in ?\mathbf{d} \leftrightarrow \exists y, z(\pi_2`x = (y, z) \wedge plus`(\pi_2{}'x, \pi_1`x) = z))$$

now $?\mathbf{c}$ has been instantiated with $Ex[?\mathbf{d}]$. We can repeat this step twice more, then we change to the rule for $\wedge$ and so on. Eventually we have only goals of the form

$$\forall x(x \in ?\mathbf{e} \leftrightarrow x \in d)$$

which can be made true by unifying with

$$\forall x(x \in y \leftrightarrow x \in y)$$

which sets $?\mathbf{e}$ equal to d, then we can look at $?\mathbf{c}$ to see what it has been instantiated with, which gives us the class term we are looking for. This way, not only do we generate the class term that we want from the property we want it to satisfy, but we get, for free a theorem that states that it satisfies that property.[3] This approach is similar to the idea that Basin suggests in [1], as a general method for program synthesis.

We have implemented this so that we can write a definition, directly, as

```
comp("ltC", "(y,z)", "EX w. plus‘(y,w)=z")
```

(i.e. the class of all instances of the term `(y,z)` such that...) Then the whole procedure just described is performed automatically: first an equivalent term, with just one free variable, is constructed, which has the form

```
x:?c <-> EX y z w. x = (y,z) & plus‘(y,w)=z
```

then this is set as the goal to be proven and the proof procedure we have just described is run on it. The class is generated and attached to a name, then a version of the theorem defining the comprehension relation, only with the new name substituted for the generated class term, is proven and returned.

Thus we have solved both problems at once: the constant defining the class has been generated, automatically from a clear specification, and at the same time, a theorem connecting the specification and the class together by a logical equivalence has been proved, so it should never, in future, be necessary to unfold this class definition to get at what is inside it.

§5.3 *general synthesis* We have described a powerful method for building classes in FS_0 as conservative extensions, but general though it is, it is not always suitable, and anyway we also want to define, similarly, new functions as conservative extensions; and unfortunately no similarly elegant solution is available for that.

However the situation can be improved far beyond having to piece functions together by hand out of primitives, and then proving that the result has the right properties. In fact, we can extend the idea that we have just used to implement the comprehension theorem to a much larger class of synthesis problems.

Above, we have a uniform procedure which when given theorem-with-a-hole of a particular form, can fill out that hole. But in general no such uniform procedure is available; instead, a custom proof has to be provided. As an example, consider the 'less than' ordering again, only this time we want to define it as a function, not as a relation. We can adopt the same method to synthesise it, starting with a formula-with-a-hole that we try to prove, as follows:

$$\begin{gathered}?\mathbf{lt}`nil = false \wedge ?\mathbf{lt}`(a, zero) = false \wedge \\ ?\mathbf{lt}`(a, s`a) = true \wedge (a \neq b \rightarrow ?\mathbf{lt}`(a, s`b) = ?\mathbf{lt}`(a, b))\end{gathered}$$

The ?**lt** here can be filled in in exactly the same way as ?**c** above, resulting in a definition that can be read off, and theorem giving properties of that definition. The difference here is that we have to find a proof ourselves.

This can still be partially automated though. We just have to arrange to tell the system somehow what the right way to go about proving this goal, apart from the results are the same. Thus we find the entry

```
sch_def("lt",
   "?lt‘nil=true & ?lt‘(a,zero)=false &
    ?lt‘(a,s‘a)=true) & (a~=b --> ?lt‘(a,s‘b)=?lt‘(a,b))"
   ltsynthtac)
```

where the extra argument, `ltsynthtac`, is a program to prove the goal, that replaces the uniform proof procedure for comprehension. Assuming that `ltsynthtac` doesn't fail, the result of running it is exactly like before: the new function is synthesised and attached to a new constant, and the theorem that has been proved is returned, with the new abbreviation replacing the synthesised term.

[3]Essentially the induction which was not possible in Isabelle has been added informally, using SML.

In fact, we could have defined `ltC` in the same way, by giving the definition

```
sch_def("ltC",
        "x:?ltC <-> EX y z w. x = (y,z) & plus‘(y,w)=z",
        comprehension_tac)
```

As we said, this is clearly not quite as elegant as that for comprehension, for all sorts of reasons: it requires a user to build the tactic that is to be given as a parameter, which can be quite tricky, and it does not guarantee that the theorem returned is an exact and complete description of the object that has been synthesised, but it is, nonetheless, very effective, and, of course, very general.

§5.4 *top down specification* It is part of the received wisdom that large systems should, at least in part, be developed 'top down'; i.e. the implementation should be developed by repeatedly refining the abstract definition into somethingc oncrete. This, it is argued, helps to keep the development under intellectual control. We cannot do this with the facilities we have defined so far: anything we define has be be built from the ground up. We add a fourth sort of entry on the range of possible definitions, to allow a top down style. This looks very like `sch_def`, but has one less parameter.

```
abstract("lt",
   "lt‘nil=true & lt‘(a,zero)=false &
    lt‘(a,s‘a)=true) & (a~=b --> lt‘(a,s‘b)=lt‘(a,b))")
```

(notice that the formula has no holes) With `comp` and `sch_def` a definitional extension of the given theory is generated and a theorem about it is proven. With `abstract` this process is short-circuited: no effort is made to try to generate a suitable definition, or prove a theorem. The new theory is extended with a new constant `lt`, and a new *axiom* defining its behaviour. This is dangerous because this extension is not definitional and the associated theorem is not a theorem of FS_0 (there is nothing to stop us adding a false axiom), but as was said earlier, `abstract` is supposed to be used only as a temporary, stop-gap, measure, and removed before the end.

§5.5 *Taking stock* If we take stock of what we have done in this section we see a single idea, presented in a variety of ways. We have tried constantly to hide the details of FS_0 definitions behind abbreviations, which we treat as new objects, with new defining axioms, added to the theory. FS_0 is, in fact really being used only as an underlying foundation to the extension we define. However, if we really need to, we can, at any time, strip these levels of abstraction away, leaving the unadorned theory, since everything is, in the end, just a definitional extension. We will extend this theme in the next section, when we talk about how Isabelle allows us to implement rewriting.

§6 In use

We have described a modification of Isabelle that we have found to be a practical way of building theories as definitional extensions of FS_0. We now show that it is a practical way of working in those theories. There are several aspects to this work, which we will discuss in turn.

§6.1 *Induction* As we have said earlier, a lot of the work involved in using FS_0, is building classes. This work is more ubiquitous even than we have implied, since the typical method of proof in the theory is induction, of one sort or anther, and induction is only available over classes, in spite of the fact that we almost never want actually to do this. Again, as a result of the comprehension theorem we know that there is an equivalent class for any Σ^0_1 predicate, and thus we have induction over this class of formulae (which is enough in for most practical things). The metatheory of Isabelle will not allow us to prove derived rules of this form; e.g. we cannot prove as a single metatheorem that

$$\frac{P[nil] \quad \forall x, y(P[x] \to P[y] \to P[(x, y)])}{\forall x P[x]}$$

(where P is a Σ^0_1-formula). We are, however able to prove something almost identical in the rule:

$$\frac{\exists c \forall x(x \in c \leftrightarrow P[x]) \quad P[nil] \quad \forall x, y(P[x] \to P[y] \to P[(x, y)])}{\forall x P[x]}$$

(no side condition). This produces an extra goal, of course, which corresponds to the side condition on the previous rule since we know that it is exactly these classes that is defineable. And the extra goal is not a problem, since it can be disposed of immediately and automatically with the same tactic that we use to implement generation by comprehension in `comp`.

§6.2 *Rewriting* The other large and ubiquitous problem-in-use for FS_0 is term simplification, since many theorems are proven mostly by selecting a suitable induction then simplifying the resulting terms. But this work of term simplification is tedious and difficult to do by hand.

It maybe not immediately clear why this is a problem. The functions we are able to define have a well defined structure with obvious rewrites (described in 2); surely we need only implement this, and arrange for abbreviations to be unfolded as necessary.

But consider a simple example: we need a function to number the elements of a list with their positions. We can specify this as follows:

$$\begin{array}{rcl} label`l & = & labeld`(zero, l) \\ labeld`(n, nil) & = & nil \\ labeld`(n, (f, r)) & = & ((n, f), labeld`(s`n, r)) \end{array}$$

and it is not hard, though tedious, to define first-order primitive recursive (i.e. using *Rec*) versions of *label* and *labeld* to satisfy this specification. But if we try to evaluate the resulting program using the strategy we have just described, we discover that the path the evaluation takes looks nothing like the specification suggests it should be. If the term to be normalised is ground, we get the result we want, but as 'raw s-expressions'; that is, many of the abbreviations in subterms will have been unfolded (and without these abbreviations, the term is an incomprehensible s-expression built of nothing but *nil*).[4] If the term is not ground, the results will be much worse. In fact the naïve term reduction strategy suggested by the axioms is useless.

What is needed is a rewriting system that respects the specifications, and abstract properties of what has been defined, not the concrete details of the implementation. However, Isabelle, provides a general rewrite package (`term_simp_tac`) that can be effectively used for our purpose. This takes, as a parameter, a set of equational theorems that are to be used as rewrite rules. These theorems need not describe the actual normalisation path of a set of terms (the package accept a diverging set of rules); they must only be provably correct. Thus once we have verified that *label* and *labeld* behave as they should, according to their specification above (we need induction for this) we can use the equations that specify their behaviour as their rewrite rules. Such equations are always available, since whenever a function is defined, the first thing we have to do is check that it does what it should. In fact if, like in the case of `lt` above, which is defined with a `sch_def`, we get the desirable rewrite properties at the same time as we synthesise it.

[4]It worth mentioning that in this case even the technique that has been used in some systems, where unfolded abbreviations are tracked, so to be folded back afterwards when possible, does not work.

Thus we have continued the the theme of abstraction that we started in the last section. There we provided facilities for defining objects in abstract terms, trying as much as possible to hide the underlying structures used to build them. Here we bring that process to a conclusion by implementing rewrite for the functions we define in those same abstract terms, again hiding (and thus allowing a user in future to ignore) the underlying structure. We are thus able to provide efficient, and completely abstract interfaces for the various theories that we build, and we can combine them using just these interfaces.

§7 Development experience

We have described the basic facilities that we have implemented for working with FS_0. We now describe some of the development that we have actually carried out.

FS_0 is designed for doing proof theory, and therefor for formalising mathematical theories. But most theories have some notion of binding, and substitution, and (unlike type theoretic logics for encoding) it does not have these, so they have to be developed inside the theory. However, with a sufficiently general idea of what sort of facilities are needed, the work of implementing this should be reusable for any theory. I.e. rather than define binding for first-order logic, the lambda calculus, and higher order logic all separately, we could define the term structure of each of these on top of some more abstract theory, and this is what we have done. The largest theory we have implemented is for the 'binding structures' proposed by Talcott [6]. However this is a large and complicated theory (the implementation details will appear in another paper) so the development has been structured. In fact we have developed in all the following theories: natural numbers, lists, finite functions (from lists), and a general system of binding structures. However the intention of this paper has been to concentrate on the implementation of FS_0 that we have built, rather than to report on the details of how theories are developed in it.

§8 Conclusions

At the centre of any formal proof development system are two things: an underlying logic which can be used to build various theories, and a way to make that work of building as easy as possible, by allowing users to impose various sorts of structure on the development. This latter might provide (among other things)

- Modularity: it should be possible to develop parts of a theory as separate chunks, which can be combined as necessary.
- Top down development: it should be possible to assume that certain theories are available, even if the supporting development has not been finished, allowing that effort to be postponed or performed concurrently.
- Abstraction: it should be possible to present an interface to a theory that hides the (possibly messy) details of the implementation behind an abstract description of its behaviour.

We have shown that in a system like Isabelle, which provides sophisticated structuring and development facilities of the sort we have listed, that are provided prior to, and therefore independent of, any theory we might implement, we can take a simple (= primitive) theory, in this case FS_0, lacking any such facilities and provide them, while preserving the simplicity of the theory. We have been able to provide a very practical and usable system which is implemented using exactly the axioms that we find in Feferman's original paper; all development in the system can easily be unwound to that level.

We even believe it would be difficult to build a custom theorem prover as effective as what we have produced, since many of the more useful facilities are inspired directly by what Isabelle provides, and these facilities are so much more powerful than we can imagine trying to program 'from scratch'.

System availability Parties interested in getting a copy of the code for the implementation should send e-mail to the address given at the beginning of this paper); we hope to have it ready for release in the immediate future.

Appendix: !An example development

In this appendix we list some code, and some sample development from a theory (the of lists).

The Isabelle development of a theory can be packaged inside a structure, which means that it can present a fairly abstract interface, which is of the following form:

```
signature LISTS =
    sig val thy : theory
        val thms : (string * thm) list
        val list_ax1 : thm
        val list_ax2 : thm
        val list_ax3 : thm
        val ListRec1 : thm
        val ListRec2 : thm
        val ListRecTyping : thm
        val inject1 : thm
        val inject2 : thm
        val injectTyping : thm
        val map1 : thm
        val map2 : thm
        val mapTyping : thm
        val ListIndg : thm
        val List_ss :simpset
    end;
```

i.e., the rest of the system should see the development as consisting of a theory `thy`, an association list `thms` which contains a bunch of information about comprehension generated by the comprehension tactics, that we do not want to use explicitly (and could not really, even if we wanted to) and a list of theorems which together (should) present to the rest of the system an abstract view of the concrete structures that we have developed. Unfortunately the typing of ML is not enough to make the details of a `thm` explicit (i.e. what exactly it is a theorem about), but only enough to ensure that it *is* a theorem of some sort. Thus the typing details of signature cannot give all the details that we would like. Then, inside a `lists:LISTS=struct`, `end` pair we can build concrete objects in FS_0, prove theorems about them, and then assign some of these theorems to the names listed in the signature.

First we build the basic theory, which is a collection of constant definitions, as follows:

```
val (thy,thms) =
  extend_FS0_theory basics.thy "sorted lists" []
      [def("empty_List", [], "nil"),
       comp("base_List", [], "a", "a=empty_List"),
       comp("step_List", ["sort"], "re2((h,t),t)", "h:sort"),
       def("List", ["sort"], "I2(base_List, step_List(sort))"),
       sch_def("ListF", ["f"],
         "?ListF=Rec(Id,andF * [f*arg1,rc2]) * [trueF, Id] &\
         \?ListF‘nil=true & ?ListF‘(a,b)=andF‘(f‘a,?ListF‘b)",
         (fn _ => fn _ =>
            [...])),
       sch_def("ListRec", ["base", "step"],
         "?ListRec=Rec(base, step * [[[arg0,arg1],arg2],rc2]) &\
         \?ListRec‘(a,empty_List)=base‘a &\
         \?ListRec‘(a,((b,c)))=step‘(((a,b),c),?ListRec‘(a,c))",
         (fn thy => fn _ =>
             [...])),
       comp("step_List_x", ["sort"],
              "re2((h,t),t)", "h:sort & t:List(sort)"),

       def("Listarg0", [], "P1 * P1 * P1"),
       def("Listarg1", [], "P2 * P1 * P1"),
       def("Listarg2", [], "P2 * P1"),
       def("Listrc", [], "P2"),

       def("inject", ["f"],
              "ListRec(P2, f *\
              \[[P1 * Listarg0, Listarg1], Listrc])"),
       def("map", ["f"],
              "inject([f*P1,P2]) * [[P1, const(empty_List)],P2]")
      ];
```

This constucts a new theory (with the name `"sorted lists"` as an extension of the earlier `basics.thy` and assigns it to `thy` while at the same time assigning details about how the various classes generated by the comprehension tactic correspond to the given first-order predicates, to `thms`.

Then, using these two, we are able to start developing the theorems that give the abstract properties of the objects we have just defined. For instance we can write

```
    val List_ax1 =
        prove_goal thy
            "empty_List : List(D)"
            (fn _ =>[...]);
```

which proves that `empty_List` is a member of the class of lists generated from any class `D` (the details of the tactic used to prove it have been replaced with an ellipsis `...`).

Or, more complex, we can prove one of the axioms that gives the properties of the `inject` combinator.

```
    val inject2 =
        prove_goal thy
            "inject(f)‘((k,b),(h,t))=f‘((k,h),inject(f)‘((k,b),t))"
            (fn _ => [...])
```

Then finally we have the theorem for induction over lists,

```
    val ListIndg =
        prove_goal thy
            "[| x:List(C); !! x. f‘x = true <-> x:C;
                !! x. x:X <-> P(x); P(empty_List);
                !! h s. [| h:C ; s: List(C); P(s)|] ==> P((h,s))
             |]==> P(x)"
            (fn [h1, h2, h3, h4, h5] =>
                [...]);
```

and we finish by building a suitable rewriting system `List_ss`, which rewrites in terms of these theorems (though it, of course, *doesn't* actually have to use them, or work through them).

Now, if we have chose our theorems well, we have a complete theory of lists which as far as the rest of the system is concerned, has been effectively abstracted away from impementational details. We are able to treat the set of theorems provided by the structure `list` as though they were simply axioms. The isolation is not quite complete, we can always go around this abstract interface, but there should not be much temptation to do so if the theorems are well chosen, since it is hard and messy to do. (If we wanted to be absolutely sure about what we have done, we could, of course, check for implementational bias).

References

1. D. Basin. Logic frameworks for logic programs. In *4th International Workshop on Logic Program Synthesis and Transformation, (LOPSTR'94)*, Pisa, Italy, June 1994. Springer-Verlag. To Appear.
2. N. G. de Bruijn. A survey of the project Automath. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, New York, 1980.
3. S. Feferman. Finitary inductive systems. In D. Gabbay, editor, *What is a Logical System?*, Oxford, 1994. Oxford University Press. (also appeared in Logic Colloquium '88).
4. S. Matthews. *Metatheoretic and Reflexive Reasoning in Mechanical Theorem Proving*. PhD thesis, University of Edinburgh, 1992.
5. l. C. Paulson. *Isabelle: A generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, Berlin, 1994.
6. C. Talcott. A theory of binding structures, and applications to rewriting. *Theoret. Comput. Sci.*, 112:99–143, 1993.

A framework for parallel program refinement

J.-P. Bodeveix, M. Filali
IRIT
Université Paul Sabatier
118 Route de Narbonne
F-31062 Toulouse cédex France
email: bodeveix@irit.fr filali@irit.fr

April 26, 1995

Abstract

This paper presents a formal framework for developing provably correct parallel programs through refinements. We formalize rigorously, with respect to typing, some imperative programming constructs and propose refinements of them. Moreover, we try to make the semantics denotation as close as possible to the usual programming notation; the aim being to make easy the reasoning at the semantics level. This formalization is embedded within the HOL interactive theorem prover and consequently reusable as a framework for general programming by refinement.

1 Introduction

This paper presents a formal framework for developing provably correct parallel programs through refinements. We formalize rigorously, with respect to typing, some imperative programming constructs and propose refinements of them. After the study of some existing state representations in programming logics, we propose a new state representation allowing strong typing. Then, we adapt the semantics of usual sequential and parallel statements to our representation and show that classical or intuitive properties are still valid. In a third step, after introducing refinements in the context of transition systems, we propose and validate some refinement properties concerning sequential and parallel statements. In the last part of the paper, we show how these results have been embedded within the HOL interactive theorem prover and can be used to establish refinement properties on concrete problems like cache coherency protocols on multiprocessors.

2 Program state logic

2.1 Overview of some state representations

In this section, we first review some existing approaches for representing program states. Then, we present a "genuine" state representation, its motivation being to allow the use of well types notations of Pascal like programming languages. In some way, we try to reduce the gap between the syntactic domain where we are used to reason and the semantic domain where we can rigorously prove program properties.

2.1.1 States as mono valued functions

In this approach, the space of program variables `Var` and a space of values `Val` are assumed. A program state is a mapping from variables to values. Its type is

$$\mathtt{Var} \rightarrow \mathtt{Val}$$

For instance, in their programing logics theory[Tre92, vW90], `Var` is the type `string` and `Val` is the set of natural numbers. Then in order to reuse such logics, one has to encode all its variables as naturals.

2.1.2 States as type union valued functions

This representation has been used by [APP93] to describe UNITY[CM88] logics. A state is represented by a function from an enumerated of variable representatives to the union of variable values types. Then a variable is defined as a function from states to its type domain.

Suppose we have the following declaration in a PASCAL like language:

```
VAR i, j: INTEGER; b: BOOLEAN;
```

Then, we define the abstract data type `Rep` of variable representatives by the enumeration:

```
type Rep = I | J | B
```

Then, we define the union of variable types `Types` by the enumeration:

```
type Types = INT integer | BOOL boolean
```

Destructors are associated to this type in order to down-cast union-typed expressions to their effective type. Here, two functions are defined:

```
dest_int:  Type → integer
dest_int (INT n) = n

dest_bool:  Type → boolean
dest_bool (BOOL b) = b
```

A state σ is represented by a function from `Rep` to `Types`.

Now, a variable is defined as a function from states to its type. For instance, the the variable `i` is defined as follows:

$$\begin{array}{l} i : (\texttt{Rep} \ \rightarrow \texttt{Types}) \rightarrow \text{ integer} \\ i(\sigma) = \texttt{dest\_int}\ (\sigma(I)) \end{array}$$

This representation raises a problem concerning the definition of assignment. An expression must be converted to the union type before being assigned. Thus, the type checker cannot detect that an expression is assigned to a variable of a different type. Type checking will indeed occur at proof time: an access to a badly assigned variable cannot be reduced to its value as destructor functions are partial.

2.2 Another state representation

A state is represented by a boolean valued function over *variable- value* associations. The space of well typed variable−value associations is introduced as an abstract data type of which constructors are the variables of the program. Consequently, a variable can also be considered as a function from well−typed values to the previously defined abstract data type. In a given state, the variable values are such that the association variable−value is mapped to true through the state function. Then, such a state function can also be considered as a set of associations variable−value.

In order to give a concrete view of such a representation we consider two examples concerning scalar and array variables.

2.2.1 Scalar variables

Suppose we have the following declaration in a PASCAL like language:

```
VAR i, j: INTEGER; b: BOOLEAN;
```

Then, we define the abstract data type `declarations` of well typed variable−value associations by its constructors:

```
type declarations =
         i INTEGER
       | j INTEGER
       | b BOOLEAN
```

This type declaration introduces three functions (the constructors of the data type):

```
i :  INTEGER → declarations
j :  INTEGER → declarations
b :  BOOLEAN → declarations
```

The state where the variables `i, j` and `b` have respectively the values i_0, j_0 and b_0 is represented by the function $\texttt{state}_0$: `declarations` $\rightarrow$ `bool` such that[1]:

$\texttt{state}_0(i(x)) = (x = i_0)$

$\texttt{state}_0(j(x)) = (x = j_0)$

$\texttt{state}_0(b(x)) = (x = b_0)$

In our approach, the operation which consists in the creation of a "pair" (variable,value) defined by the application of the variable to the value is not injective if the variable is not supposed to be a type constructor. More precisely, this application can give the same association for different pairs. The definition of injectivity would require here quantification over types and a generalized equality ($\doteq$) between elements of different types. The expression of such a property would state the existence of a set of variables $\mathcal{V}$ such that [2]:

$$\begin{array}{l} \forall (x_1 : *x_1 \rightarrow *s)\ (v_1 : *x_1)\ (x_2 : *x_2 \rightarrow *s)\ (v_2 : *x_2) \\ x_1 \mathrel{\dot\in} \mathcal{V} \wedge x_2 \mathrel{\dot\in} \mathcal{V} \Rightarrow x_1(v_1) = x_2(v_2) \Rightarrow (x_1 \doteq x_2) \wedge (v_1 \doteq v_2) \end{array}$$

This formula raises two typing problems:

- the set $\mathcal{V}$ contains elements of different types (x_1, x_2).
- equality is applied between objects of different types $(x_1 \doteq x_2)$.

In fact, we would like to specify that $\mathcal{V}$ is the set of constructors of some type. In order to overcome this problem, we define weaker properties on individual variables by introducing the two polymorphic predicates `IS_VAR` and `D_VAR`.

`IS_VAR` $(v : *v \rightarrow *s) = \forall\ a\ b.\ (v(a) = v(b)) = (a = b)$
`D_VAR` $(x : *x \rightarrow *s, y : *y \rightarrow *s) = \forall\ vx\ vy.\ (x(vx) \neq y(vy))$

2.2.2 Array variables

The same formalism can be used to represent arrays. For the following declaration:

```
VAR t: ARRAY [INDEX] of INTEGER;
```

we associate the data type:

```
type declarations =
        t INDEX INTEGER
```

In the same way, this type declaration introduces the function:

```
t :  INDEX → INTEGER → declarations
```

The state $\texttt{state}_0$ where all the elements of the array `t` are `0` except the one at index i_0 where the value is v_0 is encoded as follows.

$\forall i\ x.\ \texttt{state}_0(t(i, x)) = \text{if } (i = i_0) \text{ then } x = v_0 \text{ else } x = 0$

[1]We note that such a representation allows for multivalued variables. However in this paper we do not use such a feature.
[2]type variables are prefixed by *

2.2.3 Expressions and variable access

Although in our representation a variable may be multivalued, we define expressions as *functions* over states:

```
expression :  *state → *exp_type
```

Then the standard operators (boolean and arithmetic operators) can be lifted to such expressions. For this purpose, we define a unary and a binary polymorphic lifting functions. For instance, we define the `Bop_Lift` function as follows:

`Bop_Lift` $op = \lambda\ e_1\ e_2\ st.\ op\ (e_1(st), e_2(st))$

We use the same convention as [APP93] where the name given to the lifted operator is the name of the original operator suffixed by `*`. For instance, `Bop_Lift (<) = <*`.

Note that the multivaluation of variables could have been extended to expressions. However, in this framework, the definition of boolean algebra operators becomes tricky and some classical properties are lost.

As specific expressions, we define the variable access function `val`. This function must choose *one* of the values associated to the variable by the state. Moreover, in a given state, this choice must be the same for every access in the same expression. For this purpose, we use the Hilbert choice function[GM94] denoted ϵ. The term $\epsilon\ x : \sigma.\ P(x)$ denotes an arbitrary but fixed variable of type σ. This term verifies the predicate P if a term verifying P exists. Then, the polymorphic function `val` is defined as follows:

```
val :  (*v → *decl)  →  (*decl → bool)  →         *v
            v        ,         st        ↦   ϵ  x. st(v(x))
```

Note that `val(v)` is a function from states of type `*decl → bool` to variable values of type `*v`.

3 Statements logic

In order to allow for non determinism, we define statements as binary relations over states[Tre92]. If we consider states as sets of associations variable-value, a statement can also be interpreted as a function which consumes some elements of a set and produces new ones. However, we have not pursued further in this direction. An interesting study would be to explore the links with linear logic [Gir87].

In the following, we give the semantics of some basic statements like assignments, conditional, alternative and parallel statements. In the last section, we establish some well known results and give some examples.

3.1 Assignments

The assignment statement `x := e` establishes a relation between two states st_1 and st_2 such that the value of the variable `x` in st_2 is the value of the expression `e` in st_1, any other variable binding remaining unchanged. The assignment is defined as follows:

$$\begin{aligned} x := e \ = \ \lambda\ st_1\ st_2.\quad & \forall\ y.\ st_2(x(y)) = (y = e(e1)) \wedge \\ & \forall\ s.\ (\forall\ y.\ s \neq x(y)) \Rightarrow st_2(s) = st_1(s) \end{aligned}$$

From this definition, we can prove the following theorems which state explicitly the usual properties of the assignment statement:

$$\begin{array}{rl} & (x := e)(st_1, st_2) \Rightarrow \text{val}\ (x)(st_2) = e(st_1) \\ \text{D\_VAR}\ (x, y) \Rightarrow & (x := e)(st_1, st_2) \Rightarrow \text{val}\ (y)(st_2) = \text{val}\ (y)(st_1) \end{array}$$

We have only defined the single assignment statement. The multiple assignment statement will be defined in section 3.4.3 through the parallel constructor $\|$.

3.2 Alternatives

The definition of statements as binary relations over states yields a straightforward definition of non deterministic statements as in CSP[Hoa85]. We consider the binary operator $[\!]$ and the generalized one indexed by a set of alternatives A: $[\!]_A$.

$$\begin{array}{lcl} i_1 \mathbin{[\!]} i_2 & = & \lambda\ st_1 st_2.\ i_1(st_1, st_2) \ \lor\ i_2(st_1, st_2) \\ [\!]_A i & = & \lambda\ st_1 st_2.\ \bigvee_{a \in A} i_a(st_1, st_2) \end{array}$$

The associativity and commutativity of the binary alternative operator are easily proved.

3.3 Conditional Statements

We introduce successively two conditional statements. The first one ($\longrightarrow$) is similar to a guarded statement[Dij76], while the second one (`IF`) is similar to the usual `if-then` statement.

$$\begin{array}{ll} i \longrightarrow c & = \lambda\ st_1\ st_2.\ c(st_1) \land i(st_1, st_2) \\ i \text{ IF } c & = \lambda\ st_1\ st_2.\ \textbf{if}\ c(st_1)\ \textbf{then}\ i(st_1, st_2)\ \textbf{else}\ (st_1 = st_2) \end{array}$$

3.4 Parallel statements

The basic idea of the parallel construct is to *superpose* the changes performed by several statements. A variable binding is changed by a parallel statement if it is changed by one of the composing statements; otherwise it remains unchanged. We should note that such a definition of parallelism cannot be reduced to interleaving and non-determinism. For instance the exchange of two variables can be achieved by a parallel composition (§3.6.2) of the two symmetric assignments which is not equivalent to a non deterministic choice between the sequential compositions of the two assignments.

3.4.1 The binary PAR constructor

From the intuitive definition of parallelism, we first define a binary parallel operator $\parallel$ between two statements i_1 and i_2. The parallel statement updates a state so that a binding is present in the final state if it is present in the final state of one of the statements modifying it. This is encoded as follows:

$$\begin{array}{lll} i_1 \ \parallel\ i_2 = \lambda\ st\ st'.\ \exists\ st_1\ st_2. & i1(st, st_1) \land i2(st, st_2) \land & \\ & \forall s.\ st'(s) = & (st(s) \neq st_1(s) \land st_1(s)) \lor \\ & & (st(s) \neq st_2(s) \land st_2(s)) \lor \\ & & (st(s) = st_1(s) \land st(s) = st_2(s) \land st(s)) \end{array}$$

The previous definition can be simplified to the following one:

$$\begin{array}{lll} i_1 \ \parallel\ i_2 = \lambda\ st\ st'.\ \exists\ st_1\ st_2. & i1(st, st_1) \land i2(st, st_2) \land & \\ & \forall s.\ st'(s) = \ \textbf{if}\ st(s) & \textbf{then}\ st_1(s) \land st_2(s) \\ & & \textbf{else}\ st_1(s) \lor st_2(s) \end{array}$$

We should remark that the $\parallel$ constructor requires a point by point interpretation of states contrary to other constructors such as conditional alternatives.

3.4.2 The generalized PAR constructor

The previous definition can be easily extended to an indexed set of statements ins_i in the following way:

$$\begin{array}{ll} \text{PAR}\ ins = \lambda\ st\ st'.\ \exists\ e'. & \forall\ i.\ ins_i(st, e'_i) \land \\ & \forall\ s.\ st'(s) = \ \textbf{if}\ st(s)\ \textbf{then}\ \forall\ i.\ e'_i(s)\ \textbf{else}\ \exists\ i.\ e'_i(s) \end{array}$$

3.4.3 Multi-assignments

The parallel operator lets us introduce multi-assignment statements as a parallel combination of single assignments. Then, single assignment theorems (§3.1) can be extended, for instance, to two-assignment statements.

$$\begin{array}{l} \text{D\_VAR } (x,y) \ \wedge \ (x := ex \parallel y := ey)(st_1, st_2) \Rightarrow \\ \text{val } (x)(st_2) = ex(st_1) \wedge \text{val } (y)(st_2) = ey(st_1) \end{array}$$

$$\begin{array}{l} \text{D\_VAR } (x,z) \ \wedge \ \text{D\_VAR } (y,z) \ \wedge \ (x := ex \parallel y := ey)(st_1, st_2) \Rightarrow \\ \text{val } (z)(st_2) = \text{val } (z)(st_1) \end{array}$$

The previous two rules illustrate the idea of the superposition of changes made by the components of a parallel statement. However, the behaviour of a multi-assignment to a single variable (which is usually syntactically forbidden) is counter intuitive; for instance, let us consider the multi-assignment $x := 1 \parallel x := 2$. If in the initial state the value of x is 0, then in the final state x is multivalued. But, if the initial value of x is 1, then the final value of x is 2! Another semantics could be to allow a non-deterministic behaviour in such cases, or to state explicitly how values are combined[BC84]. Nevertheless, we have done with this definition for its conciseness and since we do not consider parallel updates of the same variable.

At last, general multi-assignment rules cannot be stated because of typing problems: each variable has its own type and a set of such variables cannot be defined. However, we will see in section 5.3 how to overcome this problem by introducing a kind of meta theorem.

3.5 Assessment of the definitions

In order to assess our definitions, we establish some results of well known programming logics within our framework. We consider the weakest precondition semantics of Dijkstra [Dij76]. The following definition expresses the weakest precondition (or predicate transformer) of a statement i and a predicate p:

$$\text{wp } (i,p) = \lambda\ e.\ \forall e'.\ i(e,e') \Rightarrow p(e')$$

Then, by defining the substitution of variable x in expression e_1 by expression e_2 as follows:

$$\text{subst } (x, e_1, e_2) = \lambda\ st.\ e_2(\lambda\ s.\ \textbf{if } \exists z.\ s = x(z) \textbf{ then } s = x(e_1(st)) \textbf{ else } st(s))$$

We have the following theorems[3]:

$$\begin{array}{l} \forall x.\ \text{IS\_VAR } (x) \Rightarrow \ \text{wp } (x := e, \text{FALSE\_P}) = \text{FALSE\_P} \\ \forall x\ p.\ \text{IS\_VAR } (x) \Rightarrow \ \text{wp } (x := e, p) = \ \text{subst } (x, e, p) \end{array}$$

3.6 Some results and examples

3.6.1 Independent parallel statements

Parallel constructs may be transformed into sequential constructs given some independence hypothesis expressed using the following `e_indep_x` predicate:

An expression is said to be independent of a variable x if any assignment to x does not modify its value.

$$\texttt{e\_indep\_x (exp, x)} = \forall st_1\ st_2\ e.\ (x := e)(st_1, st_2) \Rightarrow \texttt{exp}\ (st_1) = \texttt{exp}\ (st_2)$$

The following theorem states the equivalence between a sequential and a parallel assignment.

$$\begin{array}{l} \forall x\ y\ ex\ ey.\ \texttt{IS\_VAR}\ (x) \wedge \texttt{IS\_VAR}\ (y) \wedge \texttt{D\_VAR}\ (x,y) \wedge \text{e\_indep\_x } (ey, x) \Rightarrow \\ \quad (x := ex \parallel y := ey) = \text{Seq } (x := ex, y := ey) \end{array}$$

3.6.2 Exchange of two variables

Under the hypothesis that x and y are different variables, the following result is a straightforward application of the multi-assignment rules given in section 3.4.3.

$$\text{D\_VAR } (x,y) \Rightarrow (x := y \parallel y := x)(st_1, st_2) \Rightarrow \text{val } (x)(st_2) = \text{val } (y)(st_1) \wedge \text{val } (y)(st_2) = \text{val } (x)(st_1)$$

[3]FALSE_P is the predicate identically false

4 Program refinements

Several formulations of refinements have been proposed[Nip92]. For our work on the protocols for multi-processor memories [BFR94], we have found the definition given by [LT87] well suited since it considers transition systems[Arn92]. In the following, we recall the basic definition of refinement then we present some of the results we have established.

4.1 Definitions

4.1.1 Transition systems

We represent a generic transition system as a pair consisting in a set of initial states and a next relation between its states.

Definition 1 (Transition system) *A transition system on an alphabet A is a triplet (Σ, Q_0, T) where*

- *Σ is a state space,*
- *Q_0, the set of initial states, is a subset of Σ,*
- *T, the set of labelled transitions, is a subset of $A \times \Sigma \times \Sigma$.*

Notation 1 *A triplet (l, e, e') of the set of labelled transitions of a system S will be denoted $e \xrightarrow{l}_S e'$.*

Definition 2 (Initial states) *Let $S = (\Sigma, Q_0, T)$ be a transition system. Then $\texttt{Init}_S$ characterizes the initial states of the transition system: for any state e of Σ, $\texttt{Init}_S\ e$ is true iff $e \in Q_0$.*

Definition 3 (Next Relation) *Let $S = (\Sigma, Q_0, T)$ be a transition system. Then $\texttt{Next}_S$ (also denoted $\rightarrow_S$) is a relation defined over Σ such that $e \rightarrow_S e'$ iff there exits a label* `l` *such that $e \xrightarrow{l}_S e'$.*

4.1.2 Refinement relation

Definition 4 (Refinement relation) *Given two transition systems* `s` *and* `s'`*,* `s` *simulates* `s'` *iff there exits a* simulation relation *such that:*

- *to each initial state of* `s` *at least one initial state of* `s'` *corresponds,*
- *if from a state* `e1`*,* `s` *can move to* `e2`*, and* `e1` *corresponds to the state* `e1'` *of* `s'` *then there exists a state* `e2'` *such that* `e2'` *corresponds to* `e2` *and* `s'` *can move from* `e1'` *to* `e2'`*.*

We formalize such a definition as follows:

```
∀ s s'. s Sim s' = ∃ R.
    ∀ e. Init s e ⟹ ∃ e'. R e e' ∧ Init s' e' ∧
    ∀ e1 e1' e2. Next_s e1 e2 ∧ R e1 e1' ⟹ ∃ e2'. R e2 e2' ∧ Next_s' e1' e2'
```

4.2 Refinements through operators

In the previous section, the simulation relation is defined in terms of the initial states and the `Next` relation of a transition system. However, refinement is usually proved independently for each label (also called operator in the following).

Definition 5 (Binary relation refinement) *A relation R (denoted $\xrightarrow{R}$) is a refinement of a relation R' through a relation φ ($R \sqsubseteq_\varphi R'$) iff*

$$\forall e1\ e2\ e1'.\ e1 \xrightarrow{R} e2\ \wedge\ \varphi\ e1e1' \Longrightarrow \exists e2'.\ e1' \xrightarrow{R'} e2'\ \wedge\ \varphi\ e2e2'$$

We extend this binary refinement relation to transition systems where labels are interpreted as binary relations over the system states. We can now link transition system simulation and refinement between their respective labeled transitions.

Theorem 1 (Refinement through operators) *Given two transition systems $S^1(\Sigma^1, Q_0^1, T^1)$ and $S^2(\Sigma^2, Q_0^2, T^2)$ with the same alphabet A, a sufficient condition for S^2 to simulate S^1 is that there exists a relation φ such that:*

- *for each label $l \in A$, $\xrightarrow{l}_{S^2} \sqsubseteq_\varphi \xrightarrow{l}_{S^1}$*
- *$\forall e \in Q_0^2$, $\exists e' \in Q_0^1$ such that $\varphi\, e\, e'$*

4.3 Refinement of sequential statements

In [BFR94], we have already presented refinement rules for the conditional and alternative constructs. In this paper, we investigate two new refinement rules for the sequential construct.

- The first one allows the independent refinement of the elements of a sequence.
- The second one allows the refinement of a sequence by a single statement that refines independently each element of the sequence.

4.3.1 Independent refinement

A sequence of statements can be refined by refining each of its components separately:

$$\frac{i_1 \sqsubseteq_\varphi i'_1 \quad i_2 \sqsubseteq_\varphi i'_2}{\text{Seq } (i_1, i_2) \sqsubseteq_\varphi \text{Seq } (i'_1, i'_2)}$$

Sketch of the proof:

The figure 1 represents the states connected by the sequential construct and its components as well as the states connected by the refinement relation φ.

Given st_1, st_3, st'_1 such that $Seq(i_1, i_2)(st_1, st_3)$ and $\varphi(st_1, st'_1)$, we look for a state st'_3 such that $st'_1 \xrightarrow{\text{Seq}(i'_1, i'_2)} st'_3$ and $\varphi(st_3, st'_3)$.

By the first refinement hypothesis: $i_1 \sqsubseteq_\varphi i'_1$, there exists a state st'_2 such that $st'_1 \xrightarrow{i'_1} st'_2$ and $\varphi(st_2, st'_2)$ and finally, by the second refinement hypothesis: $i_2 \sqsubseteq_\varphi i'_2$, there exists a state st'_3 such that $st'_2 \xrightarrow{i'_2} st'_3$ and $\varphi(st_3, st'_3)$.

$$\begin{tikzcd}
st'_1 \arrow[r, dashed, "i'_1"] & st'_2 \arrow[r, dashed, "i'_2"] & st'_3 \\
st_1 \arrow[u, "\varphi"'] \arrow[r, "i_1"] \arrow[rr, bend right, "\text{Seq } i_1 i_2"'] & st_2 \arrow[u, dashed, "\varphi"'] \arrow[r, "i_2"] & st_3 \arrow[u, dashed, "\varphi"']
\end{tikzcd}$$

Figure 1: Independent SEQ refinement

4.3.2 Joint refinement

We first introduce the predicate `stable` (φ, i):

Definition 6 (Stability) *The refinement relation φ is invariant through the statement i.*

$$stable\ (\varphi, i) = \forall st\ st'_1\ st'_2.\ \varphi(st, st'_1) \wedge i(st'_1, st'_2) \Rightarrow \varphi(st, st'_2)$$

The idea of the joint refinement rule is to split the refinement of a sequence of statements by a single statement into the refinement of each component through a *projection* of the initial refinement operator.

$$\frac{i \sqsubseteq_{\varphi_1} i'_1 \quad i \sqsubseteq_{\varphi_2} i'_2 \quad \text{stable } (\varphi_1, i'_2) \quad \text{stable } (\varphi_2, i'_1)}{i \sqsubseteq_{\varphi_1 \wedge \varphi_2} \text{Seq } (i'_1, i'_2)}$$

Sketch of the proof:

The figure 2 represents the states connected by the sequential construct and its components as well as the states connected by the refinement relation φ.

Given st_1, st_2, st'_1 such that $st_1 \xrightarrow{i} st_2$, $\varphi_1(st_1, st'_1)$ and $\varphi_2(st_1, st'_1)$, we look for a state st'_3 such that $st'_1 \xrightarrow{\text{Seq}(i'_1, i'_2)} st'_3$ and $\varphi_i(st_2, st'_3)$.

- By the refinement hypothesis $i \sqsubseteq_{\varphi_1} i'_1$, there exists a state st'_2 such that $st'_1 \xrightarrow{i'_1} st'_2$ and $\varphi_1(st_2, st'_2)$.
- By the stability hypothesis `stable` (φ_2, i'_1), we have $\varphi_2(st_1, st'_2)$.
- By the refinement hypothesis $i \sqsubseteq_{\varphi_2} i'_2$, there exists a state st'_3 such that $st'_2 \xrightarrow{i'_2} st'_3$ and $\varphi_2(st_2, st'_3)$.
- By the stability hypothesis `stable` (φ_1, i'_2), we have $\varphi_1(st_2, st'_3)$.
- Thus there exists a state st'_3 satisfying the refinement requirements.

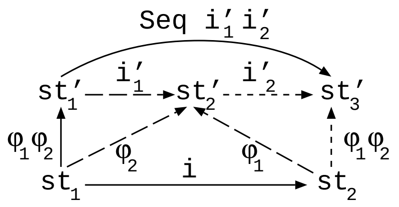


Figure 2: Joint SEQ refinement

4.4 Refinement of parallel constructs

4.4.1 Binding invariance

A binding is invariant or unchanged through a statement i if it has the same status in any states connected by i.

$$\texttt{unchanged}\ (s, i) = \forall st_1\ st_2.\ i(st_1, st_2) \Rightarrow st_1(s) = st_2(s)$$

We can prove the intuitive result concerning invariance through a parallel statement: if i_1 and i_2 are functional, we have:

$$\texttt{unchanged}\ (s, i_1 \parallel i_2) = \texttt{unchanged}\ (s, i_1) \wedge \text{unchanged}\ (s, i_2)$$

4.4.2 Projections

We define the projection of a relation φ with respect to a statement i (denoted by φ_i) as a relation which only depends on bindings unchanged by i.

$$\varphi_i = \lambda\ st\ st'.\ \exists\ st''.\ \varphi(st, st'') \wedge \forall s.\ \texttt{unchanged}\ (s, i) \Rightarrow st'(s) = st''(s)$$

Then, the projected relation is easily shown to be stable by the statement i.

$$\text{stable}\ (\varphi_i, i)$$

Furthermore, if φ is functional and if the statements i_1 and i_2 are independent (they cannot change a binding together[4]), then φ is the conjunction[5] of its projections over i_1 and i_2.

$$\texttt{IsF}\ (\varphi) \wedge \texttt{indep}\ (i_1, i_2) \Rightarrow \varphi = \text{And}\ (\varphi_{i_1}, \varphi_{i_2})$$

4.4.3 A restricted parallel construct

In order to allow a smooth proof of the following theorem, we introduce a new parallel construct denoted $|||$. We define it using the `unchanged` global predicate.

$$\begin{aligned}(i_1 \mathbin{|||} i_2)(st_1, st_2) = \exists st'\ st''.\quad & i_1(st_1, st') \wedge i_2(st_1, st'') \wedge \\ \forall s\ .\ st_2(s) = \quad & (\texttt{unchanged}\ (s, i_1) \wedge st''(s)) \vee \\ & (\texttt{unchanged}\ (s, i_2) \wedge st'(s))\end{aligned}$$

The following theorem establishes the relation between the two parallel constructs:

$$\forall st_1\ st_2\ i_1\ i_2.\ \texttt{indep}(i_1, i_2) \Rightarrow (i_1 \mathbin{|||} i_2)(st_1, st_2) \Rightarrow (i_1 \parallel i_2)(st_1, st_2)$$

Under the same independence hypothesis, the constructor $|||$ can also be defined as follows:

$$\begin{aligned}\texttt{indep}\ (i_1, i_2) \Rightarrow\quad & (i_1 \mathbin{|||} i_2)(st_1, st_2) = \\ & \exists st'\ st''.\quad i_1(st_1, st') \wedge i_2(st_1, st'') \wedge \\ & \qquad \forall s\ .\ st_2(s) = \texttt{if unchanged}\ (s, i_1)\ \texttt{then}\ st''(s)\ \texttt{else}\ st'(s)\end{aligned}$$

4.4.4 Refinement theorem

The following refinement rule states that the refinement of a parallel construct can be split into the refinement of its components given some hypothesis.

$$\frac{\texttt{IsF}\ (\varphi) \quad \texttt{indep}\ (i'_1, i'_2) \quad i \sqsubseteq_{\varphi_{i'_2}} i'_1 \quad i \sqsubseteq_{\varphi_{i'_1}} i'_2}{i \sqsubseteq_{\varphi} i'_1 \parallel i'_2}$$

Sketch of the proof:

The figure 3 represents the states connected by the parallel construct and its components, as well as the refinement relation and its projections over the two statements i'_1 and i'_2. The refinement hypothesis provides the states st'_2 and st''_2 and the refinement relations $\varphi_{i'_1}(st_2, st'_2)$ and $\varphi_{i'_2}(st_2, st''_2)$. Now, the state st'''_2 can be introduced as the superposition of the states st'_2 and st''_2, as defined in the $|||$ construct:

$$\forall s.\ st'''_2(s) = \text{if unchanged}\ (s, i'_1)\ \text{then}\ st''_2(s)\ \text{else}\ st'_2(s)$$

Thus, we have the relation $(i'_1 \mathbin{|||} i'_2)(st'_1, st'''_2)$ and by the theorem relating the two parallel constructs, we have $(i'_1 \parallel i'_2)(st'_1, st'''_2)$.

Then, we prove that the two projected refinement relations remain valid between the states st_2 and st'''_2. So is their conjunction φ.

[4] indep $(i_1, i_2) = \forall s.$ unchanged $(s, i_1) \vee$ unchanged (s, i_2)

[5] And $(\varphi_1, \varphi_2) = \lambda\ x\ y.\ \varphi_1(x, y) \wedge \varphi_2(x, y)$

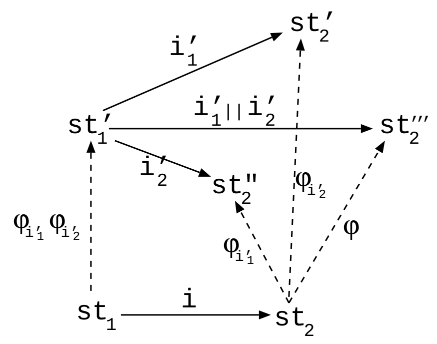


Figure 3: Parallel independent refinement

5 Refinements rules and tactics

We use a theorem prover first to validate the previously presented results and second to offer a framework to support the development of correct parallel algorithms.

5.1 The HOL interactive theorem prover

HOL[GM94] is an interactive theorem prover based on higher order logic. The formulation of this logic relies on a small number of axioms, inference and definition rules. Although HOL was initially intended for hardware design, it is now widely used for general program proving[CT94, Bac92].

5.2 HOL refinement theories

All the refinement theory we have developed is definitional. Starting with the initial theory, we have elaborated the presented framework without introducing any axiom. Consequently, since the initial HOL theory is consistent, the developed one is consistent as well. The figure 4 illustrates the hierarchy of the theories.

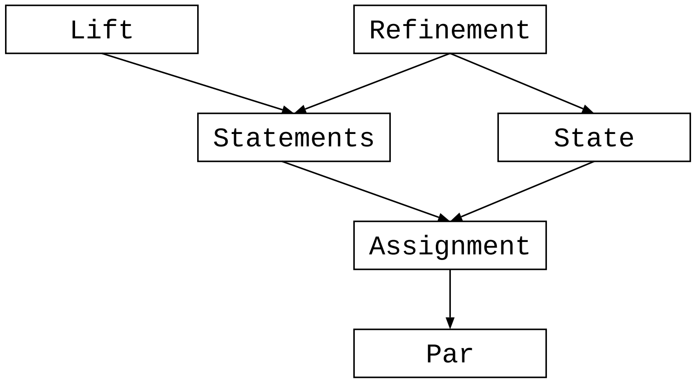


Figure 4: HOL hierarchy of theories

- The Lift theory (§2.2.3) defines the standard operators lifted to functions over states as well as the proof of lifted standard theorems on booleans, sets ...
- The Refinement theory (§4) defines basic relational algebra, the refinement relation and basic refinement theorems.
- The State theory (§2.2) defines environments and variables.

- The Statements theory (§3) defines basic sequential control structures and refinement theorems on them.
- The Assignment theory (§3.1) defines the assignment statement, theorems on environments modified through assignments.
- The Par theory (§3.4.3) defines the parallel operator, theorems concerning independent refinements of parallel statements as well theorems and conversions on multi-assignments.

5.3 Meta theorems in HOL

The theorems stated in this paragraph assume that we have a data type defined by an indexed set of constructors. Such a data type cannot be characterized in HOL since this would require at least sets with elements of different types. Consequently generals theorems about these data types cannot be stated rigorously in HOL. However, HOL *conversions* provide a way to generate specific instances of these theorems for a given data type.

In the following, we suppose that we have a data type `Decl` and an indexed family of typed constructors $(C_i : T_i \rightarrow \texttt{Decl})_{i \in I}$ such that:

- $\forall i \in I.\ \texttt{IS\_VAR}\ (C_i)$
- $\forall i\ j \in I.\ i \neq j \Rightarrow \texttt{D\_VAR}(C_i, C_j)$

An example of such a data type `Decl` coming from the study of the refinement of multiprocessor memory models, describing an array of cache and status registers, and a global memory is given below:

```
C_decls = VAL *ind *val
        | STATE *ind *state
        | MEM *val
```

This type declaration defines three families of constructors: `VAL (i)`, `STATE (i)` and `MEM`.

5.3.1 Assignments

The single assignment meta-rule applied to an assignment statement expresses the value of each variable in the new state in terms of expressions evaluated in the current state.

$$\bigwedge_{i \in I} \left\{ \begin{array}{ll} \forall e : T_i.\ (C_i := e)(st_1, st_2) = & \forall x : T_i.\ st_2(C_i(x)) = (x = e(st_1)) \wedge \\ & \bigwedge_{j \neq i}\ \forall x : T_j.\ st_2(C_j(x)) = st_1(C_j(x)) \end{array} \right.$$

We illustrate the application of this meta rule on a data type on the data type introduced in the previous paragraph. The HOL conversion associated to the single assignment meta rule proves the following theorem:

```
|- (∀ x0 ex e1 e2.
     ((VAL x0) := ex)e1 e2 =
       (∀ x0′ x1. e2(VAL x0′ x1) = ((x0′ = x0) => (x1 = ex e1) | e1(VAL x0′ x1))) ∧
       (∀ x0′ x1. e2(STATE x0′ x1) = e1(STATE x0′ x1)) ∧
       (∀ x. e2(MEM x) = e1(MEM x))) ∧

   (∀ x0 ex e1 e2.
     ((STATE x0) := ex)e1 e2 =
       (∀ x0′ x1. e2(VAL x0′ x1) = e1(VAL x0′ x1)) ∧
       (∀ x0′ x1. e2(STATE x0′ x1) = ((x0′ = x0) => (x1 = ex e1) | e1(STATE x0′ x1))) ∧
       (∀ x. e2(MEM x) = e1(MEM x))) ∧

   (∀ ex e1 e2.
     (MEM := ex)e1 e2 =
       (∀ x0 x1. e2(VAL x0 x1) = e1(VAL x0 x1)) ∧
       (∀ x0 x1. e2(STATE x0 x1) = e1(STATE x0 x1)) ∧
       (∀ x. e2(MEM x) = (x = ex e1)))
```

5.3.2 Multi-Assignments

The multi-assignment meta rule is similar to the single assignment one except that the effects of the different assignments are superposed. The meta rule is formulated as follows:

$$\begin{array}{ll}
\forall e_1 : T_1 \cdots e_n : T_n & \\
\quad (C_{i_1} := e_1 \parallel \cdots \parallel C_{i_n} := e_n)(st_1, st_2) = & \\
& \forall x : T_{i_1}.\ st_2(C_{i_1}(x)) = (x = e_1(st_1)) \land \\
& \vdots \\
& \forall x : T_{i_n}.\ st_2(C_{i_n}(x)) = (x = e_n(st_1)) \land \\
\bigwedge_{i \notin \{i_1, \cdots, i_n\}} & \forall x : T_i.\ st_2(C_i(x)) = st_1(C_i(x))
\end{array}$$

We illustrate the application of the multi-assignment rule on the same data type. Here, the HOL conversion needs the variables assigned to. The generation of all combinations of assignments is not realistic. Here, the conversion proves the following theorem defining the multi-assignment of the three variables `MEM`, `STATE p` and `VAL q`:

```
|- (MEM := m) || (STATE p := s) || (VAL q := v) =
   (λ e1 e2.
     (∀ x0 x1. e2 (VAL x0 x1) = ((x0 = q) => (x1 = v e1) | e1(VAL x0 x1))) ∧
     (∀ x0 x1. e2 (STATE x0 x1) = ((x0 = p) => (x1 = s e1) | e1(STATE x0 x1))) ∧
     (∀ x. e2 (MEM x) = (x = m e1)))
```

5.3.3 Refinements tactics

Refinement theorems concerning the sequential and parallel constructs have been also implemented in HOL as proof tactics. Thus, these tactics can be used by the backward proof engine to reduce a goal into subgoals. To give the flavor of the application of a tactic, we have peeked a fragment from the proofs developed for the study of refinements between multiprocessor memory models.

Suppose we have to prove the following refinement property:

```
∀ p st ost states. st IN states ∧ ¬ ost IN states ⇒
  REF_Op
  ((((STATE p) := (CST st)) || ((VAL p) := (val MEM))) IF ((val(STATE p)) =* (CST ost)))
  ((A_STATE := (REPLACE_B_P(val A_STATE)(CST ost)(CST st))) || (A_VAL := ((val A_MEM) INSERT_P (val A_VAL))))
  (λ e1 e2. e2 = C2Af states e1)
```

The application of the parallel-sequential transformation of independent assignments (section 3.6.1) yields the following subgoal:

```
% Goal %
REF_Op
 ((((STATE p) := (CST st)) || ((VAL p) := (val MEM))) IF ((val(STATE p)) =* (CST ost)))
 (Seq (A_STATE := (REPLACE_B_P(val A_STATE)(CST ost)(CST st))) (A_VAL := ((val A_MEM) INSERT_P (val A_VAL))))
 (And (Pr2_Phi_X(λ x y. y = C2Af states x)A_VAL) (Pr2_Phi_X(λ x y. y = C2Af states x)A_STATE))"

% Hypothesis %
  2  ["st IN states" ]
  1  ["¬ ost IN states" ]
```

Then, the application of the tactic associated to the joint sequential refinement theorem of section 4.3.2 splits the current goal into four subgoals:

```
% Subgoal 4 %
REF_Op
 ((((STATE p) := (CST st)) || ((VAL p) := (val MEM))) IF ((val(STATE p)) =* (CST ost)))
 (A_VAL := ((val A_MEM) INSERT_P (val A_VAL)))
 (Pr2_Phi_X(λ x y. y = C2Af states x)A_STATE)
  2  ["st IN states" ]
  1  ["¬ ost IN states" ]

% Subgoal 3 %
REF_Op
 ((((STATE p) := (CST st)) || ((VAL p) := (val MEM))) IF ((val(STATE p)) =* (CST ost)))
 (A_STATE := (REPLACE_B_P(val A_STATE)(CST ost)(CST st)))
 (Pr2_Phi_X(λ x y. y = C2Af states x)A_VAL)
  2  ["st IN states" ]
  1  ["¬ ost IN states" ]

% Subgoal 2 %
stable
 (Pr2_Phi_X(λ x y. y = C2Af states x)A_STATE)
 (A_STATE := (REPLACE_B_P(val A_STATE)(CST ost)(CST st)))
  2  ["st IN states" ]
  1  ["¬ ost IN states" ]

% Subgoal 1%
stable
 (Pr2_Phi_X(λ x y. y = C2Af states x)A_VAL)
 (A_VAL := ((val A_MEM) INSERT_P (val A_VAL)))
  2  ["st IN states" ]
  1  ["¬ ost IN states" ]
```

6 Conclusion

The start point of this paper is the proposal of a new state representation. We believe that it should reduce the gap between the syntactic and the semantic levels. Proofs do not need any new information with respect to the syntactic specification of the program, such as type informations. Within this framework, we have restated and validated the usual programming constructs. Another important aspect of our work has been concerned with the refinement of sequential and parallel constructs. All these results have been validated by their embedding in the HOL theorem prover.

The next step of our work is to use the results presented here to validate consistency cache protocols. We are especially interested in generic models and their relations.

This paper was mainly concerned with local properties of programs. A further study would be to consider state sequences for reasoning about behavioral program properties.

References

[APP93] F. Andersen, K. D. Petersen, and J.S. Pettersson. Program verification using HOL-UNITY. In *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantiques des processus communicants*. Etudes et recherches en informatique. MASSON, 1992.

[Bac92] R. J. R. Back. Refinement calculus, lattices, and higher order logic. In *Program design calculi*, volume 118 of *NATO ASI Series F. Computer and systems sciences*, pages 53–72. Springer Verlag, 1992.

[BC84] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. volume 197 of *Lecture Notes in Computer Science*, pages 389–448, Berlin, Germany, 1984. Springer-Verlag.

[BFR94] J.-P. Bodeveix, M. Filali, and P. Roché. Towards a HOL theory of memory. In *Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 49–64. Springer-Verlag, sep 1994.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[CT94] C. Ching-Tsun. Mechanical verification of distributed algorithms in higher order logic. In *Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 158–176. Springer-Verlag, 1994.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Englewood Cliffs New Jersey: Prentice Hall, 1976.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Comp. Science*, 50:1–102, 1987.

[GM94] M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1994.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM symposium on principles of distributed computing*, pages 137–151, aug 1987.

[Nip92] T. Nipkow. Formal verification of data type refinement - theory and practice. In *Stepwise refinement of distributed systems*, volume 430 of *Lecture Notes in Computer Science*, pages 561–591. Springer Verlag, 1992.

[Tre92] G. Tredoux. Mechanizing execution sequence semantics in HOL. *South African Computer Journal*, (7), July 1992.

[vW90] J. von Wright. *A lattice-theoretical basis fro program refinement*. PhD thesis, Abo Akademi Finland, 1990.

Formal Verification of Concurrent Programs using the Larch Prover

Boutheina Chetali

CRIN-CNRS and INRIA-Lorraine, University of Henri Poincaré,Nancy I
Campus Scientifique — B.P. 101, 54602 Villers-les-Nancy — FRANCE
email: chetali@loria.fr

Abstract. This paper describes, by means of an example, how one may mechanically verify concurrent programs on the theorem prover LP. The chosen specification environment is UNITY, a subset of ordinary temporal logic for specifying and verifying programs. We present the proof of a lift-control program, we explain how we can use the theorem proving methodology to prove safety and liveness, and to get semi-automated proofs.

1 Introduction

UNITY [CM88], a fragment of ordinary temporal logic, is a theory to specify and verify concurrent programs. It provides a formalism to express the relevant properties of a program, an appropriate language to construct well-founded formulae and a proof system to construct proofs. Nevertheless, despite the simplicity of the model, formal reasoning about UNITY program correctness is a complex and error prone task. Therefore, a theorem prover is required to provide a very high degree of confidence in the correctness of the verification.

UNITY offers a logic and a notation in which abstract specification of the computation are expressed, without any mention to execution sequences. This *static* view of the program recall the LARCH style of specification [GHG$^+$93], which emphasizes brevity and clarity rather than executability. Moreover UNITY logic is built from a number of modalities called temporal predicates that allow a concise specification of many interesting aspects of programs. These predicates are applied to nonmodal formulae and logical operations or nesting of modalities are not part of the logic. This is interesting aspect for the encoding of UNITY logic on a first-order theorem prover.

In this paper, by means of an example, we show how we mechanize formal proofs of concurrent programs specified in UNITY and the use of a general-purpose theorem prover for first order logic like LP, to verify safety and liveness properties. Our aim is to show that we can take advantage of both the power and the simplicity of UNITY and LP, in order to get semi-automated proofs of safety properties.

The chosen example is taken from [APP93]. It describes a lift-control program specified in UNITY. Its formal proofs of correctness was made with the HOL-UNITY system, which implements UNITY theory in the higher order theorem prover HOL. This example was interesting for several reasons: First, it is a good archetypical problem, second it allows us fruitful comparisons on the use of a theorem prover in mechanizing the proof of correctness of a concurrent programs described in UNITY. Moreover, this example is well-suited to illustrate our approach to get semi-automated proofs of safety properties.

The paper is organized as follows. We start by a brief description of the mechanized implementation of UNITY in LP. Then we present the lift-control program as described in [APP93], and we give the main steps of the mechanization of the correctness proof. For that we focus on one of its safety properties, we present our informal hand proof and the corresponding automated proof in LP. Finally, we conclude with a discussion about this study and future work.

We suppose little familiarity with LP (as few as possible) and we recommand [GG91] as an introduction. This experiment has been conducted with the release 3.1 of LP.

2 About LP

In this section, we give some hints about LP, in order to introduce the formalism used in the remainder of this paper.

The design and development of the Larch proof assistant LP have been motivated primarily to debug LSL specification [GHG+93] but it has been also used to establish the correctness of hardware designs [SGG89] [CL92], or to reason about algorithms involving concurrency [SGG88].

It is a general-purpose theorem prover for multisorted first-order logic. It is based on equational term rewriting [GG91], and supports proofs about axiomatic specifications. All proofs are carried out by applying rewrite rules, proofs by case splitting, induction, contradiction and application of inference rules.

LP does not contains any predefined theory, but automatically declares the sort *bool* with the corresponding logical operators. A simple specification in LP looks like:

```
 declare sorts Elt, Set
 declare variables e, e1: Elt, x, y: Set
 declare operators
   {}:                        -> Set
   {__}:            Elt       -> Set
   insert:          Elt, Set -> Set
   __ \union __:    Set, Set -> Set
   __ \in __:       Elt, Set -> Bool
   ..
```

Sample axioms for this specification are:

```
set name Axioms
 assert
   sort Set generated by {}, insert;
   {e} = insert(e, {});
   ~(e \in {});
   e \in insert(e1, x) <=> e = e1 \/ e \in x;
   e \in (x \union y) <=> e \in x \/ e \in y
   ..
```

The first axiom is an *induction rule*, used in proofs by induction. These axioms (except the first one) are oriented into rewrite rules, for which LP uses the predefined ordering:

```
Axioms.2: {e} -> insert(e, {})
Axioms.3: e \in {} -> false
Axioms.4: e \in insert(e1, x) -> e = e1 \/ e \in x
Axioms.5: e \in (x \union y) -> e \in x \/ e \in y
```

A sample conjecture is: `prove e \in {e}`

3 About the encoding of UNITY in LP

A UNITY program consists of a **declare** section that declares the variables used in the program, an **initially** section that describes the initial values of the variables, and an **assign** section that consists of a non-empty set of assignment statements. It could be an additional section named **always** section used to define shorthands for complicated expressions used in the program.

Execution of every UNITY statement (assignment) terminates in every program state and a program execution consists of an infinite number of steps in which each statement is executed infinitely often. A UNITY program terminates by reaching a *fixed point*, which is equivalent to termination in standard sequential-programming terminology. Correctness of concurrent programs is defined in terms of properties of execution sequences. There are two basic kinds of properties of concurrent programs: *Safety*, the property must always be true, and *Liveness*, the property must eventually be true [Lam77]. There are five relations on predicates in UNITY theory: *Unless*, *Stable*, *invariant*, *Ensures* and *leads_to*. The first three are used for stating *safety* properties whereas the last two are used to express *progress* properties, a subset of liveness properties.

The reader can find the specifications of the UNITY logic and methodology in LP in [Che95b], with the encoding of the syntax of UNITY, the wp-calculus and the syntax of first order predicates. In the following, we give only the definitions needed in this paper.

The Unity predicates

Unless, *ensures* and *leads_to* are higher-order functions applied to predicates, which are nonmodal formulae. As LP is a first-order language, we had to encode these higher-order functions by first-order functions.

We define a sort Bexp, which is the sort of boolean expressions (boolean structures[DS89]). Two boolean expressions connected by a relational operator, such as $A{=}B$ or $A < B$, is not interpreted as statement of fact but denotes a boolean expression as general as the connected operands. Therefore, as the set Bexp is not (and cannot be) an extension of bool (a basic type provided by LP, we had to define *imply, or, and, true, false* as `\=>`, `\or`, `^`, `T` and `F` different from LP's built-in operators `=>`, `\/`, `/\`, `true`, `false` for implication, disjunction and conjunction. So as for `nnot`, `\=` which are different from LP operators `~`, `=`. Then in our system, `A=B` is interpreted as the fact that `A` and `B` are equal "everywhere", and `A \= B` stands for a boolean expression.

Moreover, expressions in program variables have types, there are *boolean* expressions or *integer* expressions. Integer expressions are the arithmetic expression built with the operators `+`, `-` , and `*`.

Proofs in UNITY are based on assertions of type $\{P\}s\{Q\}$, equivalent to $P \Rightarrow wp(s,Q)$, where $wp(s,Q)$ is the weakest pre-condition for the post-condition Q [Dij76][1] We formalize the wp-calculus for assignment statements using a function: wp:Act,Bexp→Bexp, where act is the sort of actions. We also define list of actions (alist), pairs of identifiers and expressions (id, exp) or $(id, bexp)$ and finally sets of pairs (pset) that we think of as being unordered lists of pairs of identifiers and expressions.

Here is the axiomatization of wp:

```
Set name wp
assert
   wp(assg(i1.ex),p) = sub_bexp(p(i1.ex),p)
   wp(cond_assg(i1.ex,b),p) = (b \=> wp(assg(i1.ex),p)) ^ (nnot(b) \=> p)
   wp(mult_assg(pl),p) = sub_bexp(pl,p)
   wp(cond_mult_assg(pl,b),p) = (b \=> wp(mult_assg(pl),p)) ^ (nnot(b) \=> p)
```

[1] $\{P\}s\{Q\}$ means that "starting from a state where P holds, the execution of s in that state results in a state where Q holds if s terminates". As UNITY statements are assignments, we assume their termination, i.e. $P \Rightarrow wlp(s,Q)$ is equivalent to $P \Rightarrow wp(s,Q)$.

The function sub_bexp(list,p), where list is a list of pairs (id,exp), substitutes each occurrence of id by exp in the boolean expression p.

Let us now give the definition of the temporal predicates.

```
Set name unless
assert
    unless(p, q, anil) = true
    unless(p, q, a) = (((p ^ nnot(q)) \=> wp(a, p \or q))=T)
    unless(p, q, cons(a,act_l)) = unless(p, q, a) /\ unless(p, q, act_l)

Set name ensures
assert
    ensures(p, q, anil) = false
    ensures(p, q, pgm) = unless(p, q, pgm) /\ exists_act(p, q, pgm)

Set name exist_act
assert
    exists_act(p, q, anil) = false
    exists_act(p, q, a) = (((p ^ nnot(q)) \=> wp(a, q))=T)
    exists_act(p, q, cons(a,act_l)) = exists_act(p, q, a) \/ exists_act(p, q, act_l)

Set name leads_to
assert
    (1) when ensures(p, q, pgm) yield leads_to(p, q, pgm)
    (2) leads_to(p, r, pgm) /\ leads_to(r, q, pgm)) ⇒ leads_to(p, q, pgm)
    (3) leads_to(p, r, pgm) /\ leads_to(q, c, pgm)) ⇒ leads_to(p \or q, r \or c, pgm)

Set name Invariant
assert
Inv(p, pgm, init_cond) = ((init_cond \=> p)=T) /\ unless(p,F,pgm)
```

In these definitions, cons(a,act_l) is the list of the actions of the program. For an atomic statement a, unless(p,q,a) means that if p holds and q does not in a program state, then after executing a either q or p holds; hence, by induction on the number of statement executions, p keeps holding as long as q does not hold.
The function exists_act checks whether there is a in pgm such that $\{p \wedge \neg q\}a\{q\}$ is valid. For a given program pgm, ensures(p,q,pgm) implies that unless(p,q,pgm) holds, and if p holds at any point in the execution of the program then q holds eventually.
invariant(p,pgm,cond_init) states that if p holds at every initial state and p is stable, then p holds at every state during any execution of pgm.
In order to infer *p leads_to q* once we have *p ensures q* in the current system, we use what LP calls a **deduction** rule and which it notes when *hypotheses* yield *conclusions*: the *when* clause contains the hypotheses of the rule and the *yield* clause, the fact inferred when the hypotheses are true. In the definition of the predicate leads_to [CM88], the disjunction rule is really an infinite set of rules, one for each integer n. Indeed, all of these rules are consequences of special case of the rule for $n = 2$. Furthermore, we encode the disjunction rule for $n = 2$.

An Induction principle for *leads_to*

The following induction rule has is crucial since it involves the fundamental predicate, *leads_to*, which usually helps to specify progress properties of programs. In most of the proofs of concurrent programs, in order to prove a progress property, we have to exhibit something which "decreases". We give the rule as it was in [CM88]:

$$\textbf{Induction}: \quad \frac{\forall m : m \in W :: \mathit{leads\_to}(p \wedge M = m, (p \wedge M <_w m) \vee q)}{\mathit{leads\_to}(p, q)}$$

W is a set well-founded under the relation $<_w$ and M is a function (the *metric*) from program states to W. For simplicity, in this rule, M (without its argument) denotes the function value when the program state is understood from the context. The hypothesis of this rule is that from any program state in which p holds, the program execution eventually reaches a state in which q holds, or it reaches a state in which p holds with a lower value of the metric M. Since the metric value cannot decrease forever, eventually a state is reached in which q holds.

The function M and the ordering depend on the example or the program to prove. So M and the corresponding order should be considered as parameters to be instancied. That it is not possible in Lp, as in almost all theorem provers. For example, M could be a binary function $M \equiv f(x, y) = (x, y)$, with the lexicographic ordering among pairs of integers, or an unary function $f(x) = x + 2$ with the ordering $<$.

So we have formalized this principle using a lexicographic ordering on list of arithmetic expression. We define this ordering as follow:

```
    lexico(nil,nil)=T
    lexico(cons(e1,list1),cons(e2,list2)) = (e1<e2) \or ((e1 \= e2) ^ lexico(list1,list2))
```

where list1 and list2 are lists of expressions, e1 and e2 expressions. We define $<: \mathit{exp}, \mathit{exp} \rightarrow \mathit{bexp}$ as an extension of $<: \mathit{nat}, \mathit{nat} \rightarrow \mathit{bool}$.

In Lp, we state the induction principle as:

```
Set name IND_leads_to
assert
leads_to((p ^ equal_exp_list(exp_list1,exp_list2)),(p ^ lexico(exp_list1,exp_list2)) \or q,pgm)
    \=> leads_to(p,q,pgm)
```

where eq is a function testing "equality" of two expression lists.

With this principle, we prove this following rule :

$$\textbf{IND\_coroll}: \quad \frac{\mathsf{leads\_to(pq(l1, l2), lexico(l1, l2), pgm)}}{\mathsf{leads\_to(T, nnot(p), pgm)}}$$

4 A lift-control program

The aim of this section is to show, by means of an example, the mechanization of safety properties with Lp and how we deal with a program manipulating naturals, booleans and abstract data type as arrays. The following example is taken from [APP93], which describes a lift-control program. The proof was made with the HOL-Unity system, which implements Unity theory in the higher order theorem prover HOL. Therefore we want to investigate the mechanization of the proof on a first order theorem prover like Lp.

The lift program describes a lift that moves between a number of floors to serve requests on these. In [APP93] the bottom and top floors are specified with two constant parameters *min* and *max*, we simplify without loss of generality considering three floors : $min = 0$ and $max = 2$.

```
Program {Lift}
Declare
  $floor : integer,$
  $up, move, stop, open : bool,$
  $req : array[0..2]\ of\ bool,$
initially
  $floor = 0\ []\ up, move, stop, open = false, true, true, false$
  $req[0], req[1], req[2] = false, false, false$
Always
  $above = \exists i : floor < i \leq 2 \wedge req[i]$
  $below = \exists i : 0 \leq i < floor \wedge req[i]$
  $queueing = above \vee below$
  $goingup = above \wedge (up \vee \neg below)$
  $goingdown = below \wedge (\neg up \vee \neg above)$
  $ready = stop \wedge \neg open \wedge move$
assign
    $\{request\_act\}$  $stop, move$              $:=$ $true, false$              $if\ \neg stop \wedge req[floor]$
  $[]\{open\_act\}$     $open, req[floor], move$  $:=$ $true, false, true$        $if\ stop \wedge \neg open \wedge req[floor]$
                                                                                     $\wedge \neg(move \wedge queueing)$
  $[]\{close\_act\}$    $open$                    $:=$ $false$                    $if\ open$
  $[]\{req\_up\}$       $stop, floor, up$         $:=$ $false, floor + 1, true$   $if\ ready \wedge goingup$
  $[]\{req\_down\}$     $stop, floor, up$         $:=$ $false, floor - 1, true$   $if\ ready \wedge goingdown$
  $[]\{move\_up\}$      $floor$                   $:=$ $floor + 1$                $if\ \neg stop \wedge up \wedge \neg req[floor]$
  $[]\{move\_down\}$    $floor$                   $:=$ $floor - 1$                $if\ \neg stop \wedge \neg up \wedge \neg req[floor]$
end{lift}
```

The state space of the lift is represented by six variables, $floor$ denotes the current position of the lift, *open* (resp. *stop*) whether the door is open (resp. stopped) at $floor$, $req[i]$ whether the lift is requested at the floor i, *up* denotes the current direction of the movement, and *move* whether moving the lift takes precedence over opening of the doors.

5 Translation in LP

Much of the power of the deductive system of LP relies upon *normalization*, e.g. computation of normal forms by rewriting, then the basis for proofs is a logical system, which consists of signatures, equations, rewrites rules, operators theories, ...etc. In addition, LP provides inference rules and tactics for proving theorems, and requires guidance in the proof strategy, e.g. the chaining of elementary proof tactics, and in the introduction of lemmas if necessary. These lemmas are proved and later used as axioms in proofs. In our proof, the suggestions we made were about the proof methods, or explicit instantiations of variables in lemmas, equations, ...etc. In the following, we explain the main steps in the translation of the UNITY program $lift$ in LP.

5.1 State variable and Proof variable

We consider two kinds of UNITY variables. The *identifiers* appear in the program text, they are *state* or *dynamic* variables (so called because they represent quantities that can vary with time). These variables are encoded as constant in LP of sort Id. The *proof variables* appear in the UNITY proof, are *static* variables encoded as LP variables of sort Id_of_var[2].

The variables $floor$, *up*, *move*, *stop*, and *open* are declared in LP as unary operators of sort Id. The array *req* is defined as an array of Id.

[2] The difference in the encoding of state variables and proof variables comes from the fact that the former are involved in substitution. Substitutions are needed to encode the wp-calculus [Che95b]

5.2 The program

A Unity program is defined as a list of actions and will be encodded in Lp as a constant of sort Actlist: lift : $\to$ Actlist.

A boolean expression init represents the **Initially** section of the program *lift*:

```
assert
init =
 (id_to_exp(floor) |= nat_to_exp(0))
 ^ bool_to_bexp(id_to_bexp(up) = F) ^ bool_to_bexp(id_to_bexp(move) = T)
 ^ bool_to_bexp(id_to_bexp(stop) = T) ^ bool_to_bexp(id_to_bexp(open) = F)
 ^ bool_to_bexp(id_to_bexp(req[0]) = F)^ bool_to_bexp(id_to_bexp(req[1]) = F)
 ^ bool_to_bexp(id_to_bexp(req[2]) = F)
```

The term `bool_to_bexp(id_to_bexp(up) = F)` represents the initial condition $up = false$, the function id_to_bexp embeds the sort Id in the sort bexp and bool_to_bexp transforms a boolean into a boolean expression.

Identifiers and naturals are themselves expressions. As the logic of Lp does not has subsorts, we explicitly embedded Id, Id_of_var and nat into exp using functions id_to_exp and nat_to_exp.
The **always** section is translated as a set of axioms:

```
set name always
assert
queueing = (above \/ below);
goup     = (above /\ ((id_to_bexp(up)=T) \/  ~(below)));
godown   = (below /\ ((nnot(id_to_bexp(up))=T) \/ ~(above)));
ready    = ((id_to_bexp(stop) ^ (nnot(id_to_bexp(open)) ^ id_to_bexp(move)))=T);
above    = (\E ex_i ((((id_to_exp(floor)<id_to_exp(ex_i))
                     ^(id_to_exp(ex_i)<=nat_to_exp(2)))
                     ^(id_to_bexp(req[id_to_nat(ex_i)])))=T));

below    = (\E ex_j ((((nat_to_exp(0)<=id_to_exp(ex_j))
                     ^(id_to_exp(ex_j)<id_to_exp(floor)))
                     ^id_to_bexp(req[id_to_nat(ex_j)]))=T))
..
```

The **assign** section is encoded as a list of actions:
lift = cons(*request_act*, cons(*open_act*, ..., cons(*move_down_act*, nil)))

For example, the action *request_act* is translated in Lp using the function cond_mult_assg(plist, bexp) which encodes conditional multiple assignments:

```
 cond_mult_assg({stop.T} \u ({move.F}\u {}),
     (nnot(id_to_bexp(stop)) ^ (id_to_bexp(req[id_to_nat(floor)]))))
```

6 Proof Obligations

In this section, we present the proof as it is described in [APP93], in order to show its mechanization in our system.

The proof obligations describe properties that the *lift* program must satisfy. For that, the main property to prove is that any request for service will eventually be served:

$$\forall n : min \leq n \leq max \Rightarrow req[n] \;\; \textbf{leads\_to} \;\; open \wedge floor = n \qquad (1)$$

As in HOL-Unity, a property *leads_ to* must be satisfied for any state in the total state space spanned by the program variables. Hence, a predicate *valid* is introduced, which characterizes a subset of the total state space including the reachable states of the program *lift* :

$$\forall n : min \leq n \leq max \Rightarrow req[n] \wedge valid \;\; \textbf{leads\_to} \;\; open \wedge floor = n \qquad (2)$$

This property is decomposed into simpler UNITY properties [APP93], from which it can be deduced using *leads_ to* inference rules.

Note: In the following, $\mapsto$ stands for *leads_ to*, and p **op** q for $\textbf{op}(p, q, lift)$. We use s for the variable *stop*, o for *open*, m for *move*, f for *floor*, q for *queueing*, a for *above*, b for *below*, *goup* for *goingup*.

The decomposition of (2) leads to the following properties, the proof of each one follows from the corresponding *ensures* properties:

$$s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid \mapsto s \wedge o \wedge m \wedge (f = n) \wedge req[n] \qquad (P_1)$$
$$s \wedge \neg o \wedge \neg m \wedge (f \neq n) \wedge req[n] \wedge valid \mapsto s \wedge o \wedge m \wedge (f \neq n) \wedge req[n] \qquad (P_2)$$
$$s \wedge o \wedge m \wedge (f \neq n) \wedge req[n] \wedge valid \mapsto s \wedge \neg o \wedge m \wedge \neg(f = n \wedge \neg q) \wedge req[n] \qquad (P_3)$$
$$s \wedge o \wedge m \wedge (f = n \wedge \neg q) \wedge req[n] \wedge valid \mapsto s \wedge o \wedge m \wedge (f = n) \wedge req[n] \qquad (P_4)$$
$$s \wedge \neg o \wedge m \wedge \neg(f = n \wedge \neg q) \wedge req[n] \wedge valid \mapsto \neg s \wedge \neg o \qquad (P_5)$$
$$\neg s \wedge \neg o \wedge req[n] \wedge valid \mapsto s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \qquad (P_6)$$

To illustrate the next section, let us explain the main steps of the proof of P_1, which states that if the lift is moving and there is a request at the floor n then the lift will stop at that floor and the door will open:

$$s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid \mapsto s \wedge o \wedge m \wedge (f = n) \wedge req[n] \qquad (P_1)$$

$\Leftarrow$*{Definition of leads_ to}*

$$s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid \; \textbf{ensures} \; s \wedge o \wedge m \wedge (f = n) \wedge req[n] \qquad (P_1)$$

$\Leftarrow$*{Definition of ensures}*

$$[s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid \; \textbf{unless} \; s \wedge o \wedge m \wedge (f = n) \wedge req[n]] \qquad (P_{1.1})$$
$$\wedge[s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid \; \textbf{exit\_act} \; s \wedge o \wedge m \wedge (f = n) \wedge req[n]] \; (P_{1.2})$$

$\Leftarrow$*{Definition of unless}*

$$\begin{aligned}
&[\forall act \in lift \; [(s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid) \\
&\qquad \Rightarrow \textbf{wp}(act, (s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid) \\
&\qquad\qquad \vee (s \wedge o \wedge m \wedge (f = n) \wedge req[n]))]] \qquad (P_{1.1}) \\
&\wedge[s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid \; \textbf{exit\_act} \; s \wedge o \wedge m \wedge (f = n) \wedge req[n]] \; (P_{1.2})
\end{aligned}$$

Proof of $(P_{1.1})$: This subgoal leads to six subsubgoals, each per action of the program. Therefore, in order to keep the length of the proof within a reasonable size, we present the main steps using the first action of the *lift* program.

$$\begin{aligned}
&[(s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid) \\
&\quad \Rightarrow \textbf{wp}(req\_act, (s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid) \\
&\qquad\qquad \vee (s \wedge o \wedge m \wedge (f = n) \wedge req[n]))]
\end{aligned}$$

{Definition of wp and assignments}

$$\begin{aligned}
&(s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid) \\
&\quad \Rightarrow [((\neg s \wedge req[f]) \Rightarrow \; \textbf{sub\_bexp}(\{(s.true), (m, false)\}, \\
&\qquad\qquad\qquad (s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid)
\end{aligned}$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad \vee(s \wedge o \wedge m \wedge (f=n) \wedge req[n]))) \\
\qquad \wedge(\neg(\neg s \wedge req[f]) \Rightarrow ((s \wedge \neg o \wedge \neg m \wedge (f=n) \wedge req[n] \wedge valid) \\
\qquad\qquad \vee(s \wedge o \wedge m \wedge (f=n) \wedge req[n]))))]
\end{array}
$$

{Definition of substitution}

$$
\begin{array}{l}
(s \wedge \neg o \wedge \neg m \wedge (f=n) \wedge req[n] \wedge valid) \\
\quad \Rightarrow [((\neg s \wedge req[f]) \Rightarrow ((true \wedge \neg o \wedge \neg false \wedge (f=n) \wedge req[n] \wedge valid) \\
\qquad\qquad\qquad\qquad \vee(true \wedge o \wedge false \wedge (f=n) \wedge req[n]))) \\
\qquad \wedge(\neg(\neg s \wedge req[f]) \Rightarrow ((s \wedge \neg o \wedge \neg m \wedge (f=n) \wedge req[n] \wedge valid) \\
\qquad\qquad\qquad\qquad \vee(s \wedge o \wedge m \wedge (f=n) \wedge req[n]))))]
\end{array}
$$

{Definitions of $\wedge, \vee$ *and booleans }*

$$
\begin{array}{l}
(s \wedge \neg o \wedge \neg m \wedge (f=n) \wedge req[n] \wedge valid) \\
\quad \Rightarrow [((\neg s \wedge req[f]) \Rightarrow (\neg o \wedge (f=n) \wedge req[n] \wedge valid)) \\
\qquad \wedge((s \vee \neg req[f]) \Rightarrow ((s \wedge \neg o \wedge \neg m \wedge (f=n) \wedge req[n] \wedge valid) \\
\qquad\qquad\qquad\qquad \vee(s \wedge o \wedge m \wedge (f=n) \wedge req[n]))))]
\end{array}
$$

$[true]$ □

Proof of $P_{1.2}$:

$[s \wedge \neg o \wedge \neg m \wedge (f=n) \wedge req[n] \wedge valid\ \mathbf{exit\_act}\ s \wedge o \wedge m \wedge (f=n) \wedge req[n]]$

{Definition of exis_act}

$$
\begin{array}{l}
[(s \wedge \neg o \wedge \neg m \wedge (f=n) \wedge req[n] \wedge valid) \\
\quad \Rightarrow \mathbf{wp}(req\_act, (s \wedge o \wedge m \wedge (f=n) \wedge req[n]))] \\
\vee[(s \wedge \neg o \wedge \neg m \wedge (f=n) \wedge req[n] \wedge valid) \\
\quad \Rightarrow \mathbf{wp}(open\_act, (s \wedge o \wedge m \wedge (f=n) \wedge req[n]))] \\
\vee \ldots \\
\vee[(s \wedge \neg o \wedge \neg m \wedge (f=n) \wedge req[n] \wedge valid) \\
\quad \Rightarrow \mathbf{wp}(move\_down\_act, (s \wedge o \wedge m \wedge (f=n) \wedge req[n]))]
\end{array}
$$

{only open_act verifies the *wp* property}

$$
\begin{array}{l}
[false] \\
\vee[(s \wedge \neg o \wedge \neg m \wedge (f=n) \wedge req[n] \wedge valid) \\
\quad \Rightarrow [((s \wedge \neg o \wedge req[f] \wedge \neg(m \wedge q)) \Rightarrow (s \wedge true \wedge true \wedge (f=n) \wedge req[n])) \\
\qquad \wedge(\neg(s \wedge \neg o \wedge req[f] \wedge \neg(m \wedge q)) \Rightarrow (s \wedge o \wedge m \wedge (f=n) \wedge req[n]))]] \\
\vee[false]
\end{array}
$$

{Simplifications}

$$
\begin{array}{l}
[false] \\
\vee[(s \wedge \neg o \wedge \neg m \wedge (f=n) \wedge req[n] \wedge valid) \\
\quad \Rightarrow [((s \wedge \neg o \wedge req[f] \wedge \neg(m \wedge q)) \Rightarrow (s \wedge (f=n) \wedge req[n])) \\
\qquad \wedge(\neg(s \wedge \neg o \wedge req[f] \wedge \neg(m \wedge q)) \Rightarrow (s \wedge o \wedge m \wedge (f=n) \wedge req[n]))]] \\
\vee[false]
\end{array}
$$

{Using $((f=n) \wedge req[n] \wedge \neg q) \equiv false)$}

$[true]$ □

In the next section, we illustrate the mechanized proof by showing how Lp is used to prove the progress property P_1. Precisely, Lp automatically discharges the appropriate subgoals in the informal proof and saves us from supplying the tedious details in manipulating the logic parts of the proof goals.

7 The mechanized Proof

In LP, to prove that *p unless q* holds for the lift program, we must prove that the term `unless(p,q,lift)` boils down to `true`.

We introduce the conjecture typing:

```
prove unless(p,q,lift)
```

Using the definition of `unless`, LP expands the conjecture in:

```
unless(p, q, request_act) /\ unless(p, q, cons(open_act,cons(..,nil))).
```

Each conjunct is expanded, introducing `wp`, the normalization goes on and leads to the term[3]:

```
((p ^ nnot(q)) \=> wp(request_act,p \or q))
  /\ ((p ^ nnot(q)) \=> wp(open_act,p \or q))
   /\ ...
      /\ ((p ^ nnot(q)) \=> wp(move_down_act,p \or q))
```

Following the informal hand proof of the previous section, we must divide this conjecture in six subgoals, one per action. We supply LP with two forms of user guidance:

The command `resume by /\-method` directs LP to prove each of the subgoal of the current conjecture as a separate subgoal. It generates the appropriate subgoals of the form

```
((p ^ nnot(q))=T) => (wp(act,p \or q)=T).
```

For each subgoal, the command `resume by =>-m` directs LP to use the implication method. It generates the hypothesis `((p ^ nnot(q))=T)` and automatically discharges the subgoal `wp(act,p \or q)=T)`. To complete the proofs, it uses the axioms and rewrite rules which define the `^`, `\or` and properties of the boolean expressions.

In this proof, we made no mention to the formulae `p` or `q`. Actually, what appears in the following script is the entire interaction with LP for the proof of the *unless* part of the property $P_{1.1}$:

```
set name property1
prove
  unless(stopped^(id_to_exp(floor)|=id_to_exp(nn))
          ^id_to_bexp(req[nn])
          ^(id_to_exp(nn)<=nat_to_exp(2))^(nat_to_exp(0)<=id_to_exp(nn)),
         opened^(id_to_exp(floor)|=id_to_exp(nn))
          ^id_to_bexp(req[nn])
          ^(id_to_exp(nn)<=nat_to_exp(2))^(nat_to_exp(0)<=id_to_exp(nn)),
         lift)
..
   resume by /\-method
```

[3] Actually, we do not write the normal form of the subterms `wp(..)`

```
       resume by =>-method
       resume by =>-method
       resume by =>-method
       resume by =>-method
       resume by =>-method
       resume by =>-method
       resume by =>-method
qed
```

To prove the properties P_2,P_3, and P_4 we use the same script[4] than the one of P_1. To make the proving of the property P_5 easier, we first prove an additional property:

```
   (~above /\ ~below /\ (req[n]=T) /\ (0<=n<=2))  => (floor = n)
```

which states that "if there is no request at the floors upper than the *floor* where the lift is currently located, nor at the floors lower than *floor* and there is a request at the floor n then $floor = n$". The predicate *valid* is proved as an invariant in our system and implicitly added to each property to prove.

More technically The general tactic is the following: Let k be the number of statements in a given program *pgm*. For each proof of an *unless* property, we use one command `resume by /\-m` and k commands `resume by =>-m`. The proofs of an *invariant* use one command `resume by /\-m` and $k+1$ commands `resume by =>-m`. The additional subgoal is `(init \=> I)` given by the definition of `invariant(p,init,pgm)`.

The proofs of the progress property *ensures* involve a `resume by  /\-method` which leads to a number of subgoals equal to $(k+1)$, where k is as above the number of actions in the program. The additional subgoal corresponds to the *exist_act* part in the definition of `ensures(p,q,pgm)`. This subgoal is proved by cases, `(p ^ nnot(q)) = T` and `(p^nnot(q)) = F`. In order to derive the corresponding *leads_to* properties, LP uses the deduction rule (1) in the definition of the *leads_to* predicate.

Note that we can define a general proof strategy using the command:

```
        set proof-methods normalization, /\, =>
```

This commands directs LP to attempt to prove conjectures by the first applicable proof method specified in the given list. So to prove an *unless* property, LP uses the `/\` method to split the conjecture after rewriting it to its normal form, and uses the `=>` method for each subgoal. As the proof obligations of *p unless q* or *invariant p* are the same for all p and q, the mechanized proof could be fully automated with a single command and the corresponding script will be

```
   prove unless(p,q,pgm)
   qed
```

Although this strategy leads to less interaction with LP, we do not use it in order to keep control on the lower steps of the proof. Indeed, as our goal is primarily to check proofs and safety properties, the full mechanization of a proof leads us, as often in such an experiment, to enter low-level detail of the specification. In most of the cases, the low-level steps carry the deeper aspect of the problem. Nevertheless, low steps in hand proofs are usually achieved using intuitive arguments. As those arguments are often based on critical aspects of the problem, they must be formalized and mechanized in order to retrieve errors and mistakes[Che95a].

[4] The command `qed` requests LP to confirm that the proof is indeed complete.

8 Discussion and Conclusion

In this paper, we have proved the correctness of a lift-control protocol. We have used this example as an illustration of our approach of fully mechanizing the correctness using the theorem prover LP. This process requires first a specification in UNITY. Unlike others, our approach lies on a reasoning on the text of the program which we feel more intuitive and closer to the problems we have to solve. Moreover our mechanized LP proof follows as closely as possible a hand proof a user could make, adding to the hand proof the scrutinity of the theorem prover which should take out all mistakes of the original specification.

For a first comparison with the HOL proofs, we note that we made no mention to *state* and *execution* in our proofs. In Andersen, Petersen, and Pettersson [APP93] proofs, the variable names, the program constructs and the logical connectors are state lifted. The organization of the proof is also differents. The proof of a property that must be satisfied by the program is derived from smaller proofs, each per action, when our proof deals with the whole program. They obtain proofs of minimum size but a number of proofs linear in the number of program actions. In our proofs have exactly the opposite behaviour, e.g the number of commands used in each proof is linear in the number of program actions. This approach allows to handle larger examples, since the proofs of safety properties have the same architecture, e.g the number of commands used depends on the number of actions in the given program.

The whole tactics used for the progress property *leads_ to* in both proofs are similar. In both formalizations, the proof of a *leads_ to* requires a decomposition of the given property into simpler ones in order to use the derived inference rules. This decomposition is based most of the time on intuitive arguments and the use of the *cancellation rule* of the predicate *leads_ to* [Che95a].

The extra amount of time needed to prove the arithmetic parts of each subgoal could be advocated against our approach. But these additional lemmas could be proved and put in a separate library and credit should be given to HOL for its rich arithmetic library.

Our implementation of UNITY contains all definitions, theorems, corollaries defined in [CM88], but the axiom of substitution has been removed in order to preserve soundness [B.A91]. All the theorems (derived rules) about *unless* and *ensures* are proved in LP [Che95b], whereas the rules about *leads_ to* are not since they require structural induction on the length of proof that is not available in LP.

Our future efforts will concentrate on the generalization of the presented approach and the design of a translator, which will translate automatically a UNITY program in a script LP, and proof obligations as input to LP.

References

[APP93] F. Andersen, K.D. Petersen, and J.S. Pettersson. Program verification using HOL-Unity. In J.J. Joyce and C.H. Seger, editors, *Proceedings sixth International Workshop on Higher Order Logic theorem proving and its applications*, volume 780 of *Lecture Notes in Computer Science*, pages 1–15, Vancouver, Canada, August 1993. Springer-Verlag.

[B.A91] Sanders B.A. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3:189–205, 1991.

[Che95a] B. Chetali. A formal proof of a protocol for communications over faulty channels using the Larch Prover. Research Report 2476, Inria-Lorraine, 1995. submitted.

[Che95b] B. Chetali. Formal verification of concurrent programs: How to specify UNITY using the Larch Prover. Research Report 2475, Inria-Lorraine, 1995.

[CL92] B. Chetali and P. Lescanne. An exercice in LP: The proof of the non restoring division circuit. In U. martin and J.M. Wing, editors, *Proceedings First International Workshop on Larch*, volume 780 of *Workshops in Computing*, pages 55–68, Dedham, Boston, August 1992. Springer-Verlag.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Fundation.* Addison-Wesley, 1988. ISBN 0-201-05866-9.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DS89] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989.

[GG91] S. V. Garland and J. V. Guttag. A guide to LP, the larch prover. Technical Report 82, Digital Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, USA., 1991.

[GHG$^+$93] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[SGG88] J. Staunstrup, S. J. Garland, and J. V. Guttag. Verification of VLSI circuits using LP. In *Proceedings of the IFIP WG 10.2 Conference on the Fusion of Hardware Design and Verification*, pages 329–345. Elsevier Science Publishers B. V. (North-Holland), 1988.

[SGG89] J. Staunstrup, S. J. Garland, and John V. Guttag. Localized verification of circuit descriptions. In J. Sifakis, editor, *Proceedings of a Workshop on Automatic Verification Methods for Finite State Systems, Grenoble (France)*, volume 407 of *Lecture Notes in Computer Science*, pages 349–364. Springer-Verlag, June 1989.

This article was processed using the LaTeX macro package with LLNCS style

Hierarchical compression for model-checking CSP *or* How to check 10^{20} dining philosophers for deadlock

A.W. Roscoe, P.H.B. Gardiner, M.H. Goldsmith,
J.R. Hulance, D.M. Jackson and J.B. Scattergood

1 Introduction

FDR (Failures-Divergence Refinement) [4] is a model-checking tool for CSP [5]. Except for the recent addition of determinism checking [12, 14] (primarily for checking security properties) its method of verifying specifications is to test for the refinement of a process representing the specification by the target process. The presently released version (FDR 1) uses only *explicit* model-checking techniques: it fully expands the state-space of its processes and visits each state in turn. Though it is very efficient in doing this and can deal with processes with approximately 10^7 states in about 4 hours on a typical workstation, the exponential growth of state-space with the number of parallel processes in a network represents a significant limit on its utility. A new version of the tool (FDR 2) is at an advanced stage of development at the time of writing (February 1995) which will offer various enhancements over FDR 1. In particular, it has the ability to build up a system gradually, at each stage compressing the subsystems to find an equivalent process with (hopefully) many less states. By doing this it can check systems which are sometimes exponentially larger than FDR 1 can – such as a network of 10^{20} (or even 10^{1000}) dining philosophers.

This is one of the ways (and the only one which is expected to be released in the immediate future) in which we anticipate adding direct *implicit* model-checking capabilities to FDR. By these means we can certainly rival the sizes of systems analysed by BDD's (see [2], for example) though, like the latter, our implicit methods will certainly be sensitive to what example they are applied to and how skillfully they are used. Hopefully the examples later in this paper will illustrate this.

The idea of compressing systems as they are constructed is not new, and indeed it has been used to a very limited extent in FDR for several years (at the boundary between its low and high-level processes). What we believe is new is the choice of models, obtaining far better compressions in some cases than can be achieved using other, stronger, equivalences. The most similar work to our own is that of Valmari, for example [7, 16].

The main ideas behind FDR were introduced in a paper in the Hoare *Festschrift* [11] as, indeed, was part of the theory behind this compression.

In this paper we will introduce the main compression techniques used by FDR2 and give some early indications of their efficiency and usefulness.

A.W. Roscoe and J.B. Scattergood: Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK
and Formal Systems (Europe) Ltd,3 Alfred Street, Oxford;
other authors all at Formal Systems as above.

2 Two views of CSP

The theory of CSP has classically been based on mathematical models remote from the language itself. These models have been based on observable behaviours of processes such as traces, failures and divergences, rather than attempting to capture a full operational picture of how the process progresses.

On the other hand CSP can be given an operational semantics in terms of labelled transition systems. This operational semantics can be related to the mathematical models based on behaviour by defining *abstraction* functions that 'observe' what behaviours the transition system can produce. Suppose Φ is the abstraction function to one of these models. An abstract operator op and the corresponding concrete/operational version $\mathbf{op}$ are congruent if, for all operational processes $\mathbf{P}$, we have $\Phi(\mathbf{op}(\mathbf{P})) = op(\Phi(\mathbf{P}))$. The operational and denotational semantics of a language are congruent if all constructs in the language have this property, which implies that the behaviours predicted for any term by the denotational semantics are always the same as those that can be observed of its operational semantics. That the standard semantics of CSP are congruent to a natural operational semantics is shown in, for example, [10].

Given that each of our models represents a process by the set of its possible behaviours, it is natural to represent refinement as the reduction of these options: the reverse containment of the set of behaviours. If P refines Q we write $Q \sqsubseteq P$, sometimes subscripting $\sqsubseteq$ to indicate which model the refinement it is respect to.

In this paper we will consider three different models – which are the three that FDR supports. These are

- The *traces* model: a process is represented by the set of finite sequences of communications it can perform. $traces(P)$ is the set of P's (finite) traces.
- The *stable failures* model: a process is represented by its set of traces as above and also by its stable failures (s, X) pairs where s is a finite trace of the process and X is a set of events it can refuse after s which (operationally) means coming into a state where it can do no internal action and no action from the set X. *failures*(P) is the set of P's stable failures in this sense. (This model is relatively new; it is introduced in [6]. The concepts behind it will, however, be familiar to anyone well-versed in CSP. It differs from those of [7] in that it entirely ignores divergence.)
- The *failures/divergences* model [1]: a process *diverges* when it performs an infinite unbroken sequences of internal actions. $divergences(P)$ is the set of traces *after* or *during* which the process can diverge (this set is always suffix closed). In this model a process is represented by $divergences(P)$ and a modified set of failures in which after any divergence the set of failures is extended so that we do not care how the process behaves

$$failures_{\perp}(P) = failures(P) \cup \{(s, X) \mid s \in divergences(P)\}$$

  This is done both because one can argue that a divergent process looks from the outside rather like a deadlocked one (i.e., refusing everything) and because the technical problems of modelling what happens past divergence are not worth the effort.

We will also only deal with the case where the overall alphabet of possible actions is finite, since this makes the model a little more straightforward, and is an obvious prerequisite to model-checking.

All three of these models have the obvious congruence theorem with the standard operational semantics of CSP. In fact FDR works chiefly in the operational world: it computes how a process behaves by applying the rules of the operational semantics to expand it into a transition system. The congruence theorem are thus vital in supporting all its work: it can only claim to prove things about the abstractly-defined semantics of a process because we happen to know that this equals the set of behaviours of the operational process FDR works with.

The congruence theorems are also fundamental in supporting the hierarchical compression which is the main topic of this paper. For we know that, if $C[\cdot]$ is any CSP context, then the value in one of our semantic models of $C[P]$ depends only on the value (in the same model) of P, not on the precise way it is implemented. Therefore, if P is represented as a member of a transition system, and we intend to compute the value of $C[P]$ by expanding it as a transition system also, it may greatly be to our advantage to find another representation of P with fewer states. If, for example, we are combining processes P and Q in parallel and each has 1000 states, but can be compressed to 100, the compressed composition can have no more than 10,000 states while the uncompressed one may have up to 1,000,000.

3 Generalised Transition Systems

A labelled transition system is usually deemed to be a set of (effectively) structureless nodes which have visible or τ transitions to other nodes. From the point of view of compression in the stable failures and failures/divergences models, it is useful to enrich nodes by a set of minimal acceptance sets and a divergence labelling. We will therefore assume that there are functions that map the nodes of a *generalised labelled transition system* (GLTS) as follows:

- $\mathit{minaccs}(P)$ is a (possibly empty) set of incomparable (under subset) subsets of Σ (the set of all events). $X \in \mathit{minaccs}(P)$ if and only if P can stably accept the set X, refusing all other events, and can similarly accept no smaller set. Since one of these nodes is representing more than one 'state' the process can get into, it can have more than one minimal acceptance. It can also have τ actions in addition to minimal acceptances (with the implicit understanding that the τs are not possible when a minimal acceptance is). However if there is no τ action then there must be at least one minimal acceptance, and in any case all minimal acceptances are subsets of the visible transitions the state can perform.

  $\mathit{minaccs}(P)$ represents the stable acceptances P can make *itself*. If it has τ actions then these might bring it into a state where the process can have other acceptances (and the environment has no way of seeing that the τ has happened), but since these are not performed by the node P but by a successor, these minimal acceptances are not included among those of the node P.

- $\mathit{div}(P)$ is either true or false. If it is true it means that P can diverge – possibly as the result of an infinite sequence of implicit τ-actions within P. It is as though P has a τ-action to itself. This allows us to represent divergence in transition systems from which all explicit τ's have been removed.

A node P in a GLTS can have multiple actions with the same label, just as in a standard transition system.

A GLTS combines the features of a standard labelled transition system and those of the normal form transition systems used in FDR 1 to represent specification processes [11]. These have the two sorts of labelling discussed above, but are (apart from the nondeterminism coded in the labellings) deterministic in that there are no τ actions and each node has at most one successor under each $a \in \Sigma$.

The structures of a GLTS allow us to compress the behaviour of all the nodes reachable from a single P under τ actions into one node:

- The new node's visible actions are just the visible transitions (with the same result state) possible for any Q such that $P \xrightarrow{\tau} {}^{*} Q$.
- Its minimal acceptances are the smallest sets of visible actions accepted by any stable Q such that $P \xrightarrow{\tau} {}^{*} Q$.
- It is labelled divergent if, and only if, there is an infinite τ-path (invariably containing a loop in a finite graph) from P.
- The new node has no τ actions.

It is this that makes them useful for our purposes. Two things should be pointed out immediately

1. While the above transformation is valid for all the standard CSP equivalences, it is not for most stronger equivalences such as refusal testing and observational/bisimulation equivalence. To deal with one of these either a richer structure of node, or less compression, would be needed.
2. It is no good simply carrying out the above transformation on each node in a transition system. It *will* result in a τ-free GLTS, but one which probably has as many (and more complex) nodes than the old one. Just because $P \xrightarrow{\tau} {}^{*} Q$ and Q's behaviour has been included in the compressed version of P, this does not mean we can avoid including a compressed version of Q as well: there may well be a visible transition that leads directly to Q. One of the main strategies discussed below – diamond elimination – is designed to analyse which of these Q's can, in fact, be avoided.

FDR2 is designed to be highly flexible about what sort of transition systems it can work on. We will assume, however, that it is always working with GLTS ones which essentially generalise them all. The operational semantics of CSP have to be extended to deal with the labellings on nodes: it is straightforward to construct the rules that allow us to infer the labelling on a combination of nodes (under some CSP construct) from the labellings on the individual ones.

Our concept of a GLTS has been discussed before in [11], and is similar to an "acceptance graph" from [3], though the latter is to all intents the same as the normal form graphs used in FDR1 and discussed in [11, 4].

4 Methods of compression

FDR2 uses at least five different methods of taking one GLTS and attempting to compress it into a smaller one.

1. Strong, node-labelled, bisimulation: the standard notion enriched (as discussed in [11] and the same as Π-bisimulations in [3]) by the minimal acceptance and divergence labelling of the nodes. This is computed by iteration starting from the equivalence induced by equal labelling. This was used in FDR1 for the final stage of normalising specifications.

2. τ-loop elimination: since a process may choose automatically to follow a τ-action, it follows that all the processes on a τ-loop (or, more properly, a strongly connected component under τ-reachability) are equivalent.

3. Diamond elimination: this carries out the node-compression discussed in the last section systematically, so as to include as few nodes as possible in the output graph.

4. Normalisation: discussed extensively elsewhere, this can give significant gains, but it suffers from the disadvantage that by going through powerspace nodes it can be expensive and lead to expansion.

5. Factoring by semantic equivalence: the compositional models of CSP we are using all represent much weaker congruences than bisimulation. Therefore if we can afford to compute the semantic equivalence relation over states it will give better compression than bisimulation to factor by this equivalence relation.

There is no need here to describe either bisimulation, normalisation, or the algorithms used to compute them. Efficient ways of computing the strongly connected components of a directed graph (for τ-loop elimination) can be found in many textbooks on algorithm design (e.g., [9]). Therefore we shall concentrate on the other two methods discussed above, and appropriate ways of combining the five.

Before doing this we will show how to factor a GLTS by an equivalence relation on its nodes (something needed both for τ-loop elimination and for factoring by a semantic equivalence). If $\mathcal{T} = (T, \to, r)$ is a GLTS (r being its root) and $\cong$ is an equivalence relation over it, then the nodes of $\mathcal{T}/\cong$ are the equivalence classes $\overline{n}$ for $n \in T$, with root $\overline{r}$. The actions are as follows:

- If $a \neq \tau$, then $\overline{m} \xrightarrow{a} \overline{n}$ if and only if there are $m' \in \overline{m}$ and $n' \in \overline{n}$ such that $m' \xrightarrow{a} n'$.

- If $\overline{m} \neq \overline{n}$, then $\overline{m} \xrightarrow{\tau} \overline{n}$ if and only if there are $m' \in \overline{m}$ and $n' \in \overline{n}$ such that $m' \xrightarrow{\tau} n'$.

- If $\overline{m} = \overline{n}$, then $\overline{m} \overset{\tau}{\not\longrightarrow} \overline{n}$ but (if we are concerned about divergence) the new node is marked divergent if and only if there is an infinite τ-path amongst the members of $\overline{m}$, or one of the $m' \in \overline{m}$ is already marked divergent.

The minimal acceptance marking of $\overline{m}$ is just the union of those of its members, with non-minimal sets removed.

4.1 Computing semantic equivalence

Two nodes that are identified by strong node-labelled bisimulation are always semantically equivalent in each of our models. The models do, however, represent much weaker equivalences and there may well be advantages in factoring the transition system by the appropriate one. The only disadvantage is that the computation of these weaker equivalences is more expensive: it requires an expensive form of normalisation, so

- there may be systems where it is impractical, or too expensive, to compute semantic equivalence, and

- when computing semantic equivalence, it will probably be to our advantage to reduce the number of states using other compression techniques first – see a later section.

To compute the semantic equivalence relation we require the *entire normal form* of the input GLTS $\mathcal{T}$. This is the normal form that includes a node equivalent to each node of the original system, with a function from the original system which exhibits this equivalence (the map need neither be injective – because it will identify nodes with the same semantic value – nor surjective – because the normal form sometimes contains nodes that are not equivalent to any single node of the original transition system).

Calculating the entire normal form is more time-consuming that ordinary normalisation. The latter begins its normalisation search with a single set (the τ-closure $\tau^*(r)$ of $\mathcal{T}$'s root),but for the entire normal form it has to be seeded with $\{\tau^*(n) \mid n \in T\}$ – usually[1] as many sets as there are nodes in T. As with ordinary normalisation, there are two phases: the first (pre-normalisation) computing the subsets of T that are reachable under any trace (of visible actions) from any of the seed nodes, with a unique-branching transition structure over it. Because of this unique branching structure, the second phase, which is simply a strong node-labelled bisimulation over it, guarantees to compute a normal form where all the nodes have distinct semantic values. We distinguish between the three semantic models as follows:

- For the traces model, neither minimal acceptance nor divergence labelling is used for the bisimulation.
- For the stable failures model, only minimal acceptance labelling is used.
- For the failures/divergences model, both sorts of labelling are used and in the pre-normalisation phase there is no need to search beyond a divergent node.

The map from T to the normal form is then just the composition of that which takes n to the pre-normal form node $\tau^*(n)$ and the final bisimulation.

The equivalence relation is then simply that induced by the map: two nodes are equivalent if and only if they are mapped to the same node in the normal form. The compressed transition system is that produced by factoring out this equivalence using the rules discussed earlier. To prove that the compressed form is equivalent to the original (in the sense that, in the chosen model, every node m is equivalent to $\overline{m}$ in the new one) one can use the following lemma and induction, based on the fact that each equivalence class of nodes under semantic equivalence is trivially τ-convex as required by the lemma.

lemma 1
Suppose $\mathcal{T}$ be any GLTS and let M be any set of nodes in $\mathcal{T}$ with the following two properties

- All members of M are equivalent in one of our three models $\mathcal{C}$.
- M is convex under τ (i.e., if $m, m' \in M$ and m'' are such that $m \xrightarrow{\tau} {}^* m'' \xrightarrow{\tau} {}^* m'$ then $m'' \in M$.

Then let $\mathcal{T}'$ be the GLTS $\mathcal{T}/\equiv$, where $\equiv$ is the equivalence relation which identifies all members of M but no other distinct nodes in $\mathcal{T}$. m is semantically equivalent in the chosen model to $\overline{m}$ (the corresponding node in $\mathcal{T}'$).

proof
It is elementary to show that each behaviour (trace or failure or divergence) is one of $\overline{m}$ (this does not depend on the nature of $\equiv$).

Any behaviour of a node $\overline{m}$ of $\mathcal{T}'$ corresponds to a sequence σ of actions

[1]If and only if there are no τ-loops.

$$\overline{m} = \overline{m_0} \xrightarrow{x_1} \overline{m_1} \xrightarrow{x_2} \overline{m_2} \ldots$$

either going on for ever (with all but finitely many x_i τ's), or terminating and perhaps depending on either a minimal acceptance or divergence marking in the final state. Without loss of generality we can assume that the m_r are chosen so that there is, for each r, m'_{r+1} such that $m_r \xrightarrow{x_r} m'_{r+1}$ and that (if appropriate) the final m_r possesses the divergence or minimal acceptance which the sequence demonstrates. Set $m'_0 = m$, the node which we wish to demonstrate has the same behaviour exemplified by σ.

For any relevant s, define $\sigma \uparrow s$ to be the final part of σ starting at $\overline{m_s}$:

$$\overline{m_s} \xrightarrow{x_{s+1}} \overline{m_{s+1}} \xrightarrow{x_{s+2}} \overline{m_{s+2}} \ldots$$

If M, the only non-trivial equivalence class appears more than once in the final (τ-only) segment of an infinite demonstration of a divergence, then all intermediate classes must be the same (by the τ-convexity of M). But this is impossible since an equivalence class never has a τ action to itself (by the construction of $\mathcal{T}/\equiv$).

Hence, σ can only use this non-trivial class finitely often. If it appears no times then the behaviour we have in $\mathcal{T}'$ is trivially one in $\mathcal{T}$. Otherwise it must appear some last time in σ, as $\overline{m_r}$, say. What we will prove, by induction for s from r down to 0, is that the node m'_s (and hence $m = m'_0$) possesses the same behaviour demonstrated by the sequence $\sigma \uparrow s$ in $\mathcal{T}'$.

If the special node M in T' becomes marked by a divergence or minimal acceptance (where relevant to $\mathcal{C}$) through the factoring then it is trivial that some member of the equivalence class has that behaviour and hence (in the relevant models) *all* the members of M do (though perhaps after some τ actions) since they are equivalent in $\mathcal{C}$. It follows that if $\overline{m_r}$ is the final state in σ, then our inductive claim holds.

Suppose $s \leq r$ is not final in σ and that the inductive claim has been established for all i with $s < i \leq r$. Then the node m_s is easily seen to possess in $\mathcal{T}$ the behaviour of $\sigma \uparrow s$. If the equivalence class of m_s is not M then $m_s = m'_s$ and there is nothing else to prove. If it is M then since m_s and m'_s are equivalent in $\mathcal{C}$ and m'_s has the behaviour, it follows that m'_s does also. This completes the proof of the lemma.

4.2 Diamond elimination

This procedure assumes that the relation of τ-reachability is a partial order on nodes. If the input transition system is known to be divergence free then this is true, otherwise τ-loop elimination is required first (since this procedure guarantees to achieve the desired state).

Under this assumption, diamond reduction can be described as follows, where the input state-machine is $\mathcal{S}$ (in which nodes can be marked with information such as minimal acceptances), and we are creating a new state-machine $\mathcal{T}$ from all nodes explored in the search:

- Begin a search through the nodes of $\mathcal{S}$ starting from its root N_0. At any time there will be a set of unexplored nodes of $\mathcal{S}$; the search is complete when this is empty.
- To explore node N, collect the following information:
  - The set $\tau^*(N)$ of all nodes reachable from N under a (possibly empty) sequence of τ actions.
  - Where relevant (based on the equivalence being used), divergence and minimal acceptance information for N: it is divergent if any member of $\tau^*(N)$ is either marked as divergent or has a τ to itself. The minimal acceptances are the union of those of

the members of $\tau^*(N)$, with non-minimal sets removed. This information is used to mark N in T.

  – The set $V(N)$ of initial visible actions: the union of the set of all non-τ actions possible for any member of $\tau^*(N)$.
  – For each $a \in V(N)$, the set $N_a = N$ *after* a of all nodes reachable under a from any member of $\tau^*(N)$.
  – For each $a \in V(N)$, the set $min(N_a)$ which is the set of all τ-minimal elements of N_a.

- A transition (labelled a) is added to $\mathcal{T}$ from N to each N' in $min(N_a)$, for all $a \in V(N)$. Any nodes not already explored are added to the search.

This creates a transition system where there are no τ-actions but where there can be ambiguous branching under visible actions, and where nodes might be labelled as divergent. The reason why this compresses is that we do not include in the search nodes where there is another node similarly reachable but demonstrably at least as nondeterministic: for if $M \in \tau^*(N)$ then N is always at least as nondeterministic as M. The hope is that the completed search will tend to include only those nodes that are τ-minimal: not reachable under τ from any other. Notice that the behaviours of the nodes not included from N_a are nevertheless taken account of, since their divergences and minimal acceptances are included when some node of $min(N_a)$ is explored.

It seems counter-intuitive that we should work hard *not* to unwind τ's rather than doing so eagerly. The reason why we cannot simply unwind τ's as far as possible (i.e., collecting the τ-maximal points reachable under a given action) is that there will probably be visible actions possible from the unstable nodes we are trying to bypass. It is impossible to guarantee that these actions can be ignored.

The reason we have called this compression *diamond elimination* is because what it does is to (attempt to) remove nodes based on the diamond-shaped transition arrangement where we have four nodes P, P', Q, Q' and $P \xrightarrow{\tau} P'$, $Q \xrightarrow{\tau} Q'$, $P \xrightarrow{a} Q$ and $P' \xrightarrow{a} Q'$. Starting from P, diamond elimination will seek to remove the nodes P' and Q'. The only way in which this might fail is if some further node in the search forces one or both to be considered.

The lemma that shows why diamond reduction works is the following.

lemma 2

Suppose N is any node in $\mathcal{S}$, $s \in \Sigma^*$ and $N_0 \overset{s}{\Longrightarrow} N$ (i.e., there is a sequence of nodes $M_0 = N_0, M_1, ..., M_k = N$ and actions $x_1, ..., x_k$ such that $M_i \xrightarrow{x_i} M_{m+1}$ for all i and $s = \langle x_i \mid i = 1, .., n, x_i \neq \tau \rangle$). Then there is a node N' in $\mathcal{T}$ such that $N_0 \overset{s}{\Longrightarrow} N'$ in $\mathcal{T}$ and $N \in \tau^*(N')$.

proof

This is by induction on the length if s. If s is empty the result is obvious (as $N_0 \in T$ always), so assume it holds of s' and $s = s'\langle a \rangle$, with $N_0 \overset{s}{\Longrightarrow} N$. Then by definition of $\overset{s}{\Longrightarrow}$, there exist nodes N_1 and N_2 of S such that $N_0 \overset{s'}{\Longrightarrow} N_1$, $N_1 \xrightarrow{a} N_2$ and $N \in \tau^*(N_2)$.

By induction there thus exists N_1' in T such that $N_0 \overset{s'}{\Longrightarrow} N_1'$ in $\mathcal{T}$ and $N_1 \in \tau^*(N_1')$. Since $N_1' \in T$ it has been explored in constructing $\mathcal{T}$. Clearly $a \in V(N_1')$ and $N_2 \in (N_1')_a$. Therefore there exists a member N' of $min((N_1')_a)$ (a subset of the nodes of T) such that $N_2 \in \tau^*(N')$. Then, by construction of $\mathcal{T}$ and since $N \in \tau^*(N_2)$ we have $N_0 \overset{s}{\Longrightarrow} N'$ and $N \in \tau^*(N')$ as required, completing the induction.

This lemma shows that every behaviour displayed by a node of $\mathcal{S}$ is (thanks to the way we mark each node of $\mathcal{T}$ with the minimal acceptances and divergence of its τ-closure) displayed by a node of $\mathcal{T}$.

Lemma 2 shows that the following two types of node are certain to be included in $\mathcal{T}$:

- The initial node N_0.
- S_0, the set of all τ-minimal nodes (ones not reachable under τ from any other).

Let us call $S_0 \cup \{N_0\}$ the *core* of S. The obvious criteria for judging whether to try diamond reduction at all, and of how successful it has been once tried, will be based on the core. For since the only nodes we can hope to get rid of are the complement of the core, we might decide not to bother if there are not enough of these as a proportion of the whole. And after carrying out the reduction, we can give a success rating in terms of the percentage of non-core nodes eliminated.

Experimentation over a wide range of example CSP processes has demonstrated that diamond elimination is a highly effective compression technique, with success ratings usually at or close to 100% on most natural systems. To illustrate how diamond elimination works, consider one of the most hackneyed CSP networks: N one-place buffer processes chained together.

$$COPY \gg COPY \gg \ldots COPY \gg COPY$$

Here, $COPY = left?x \longrightarrow right!x \longrightarrow COPY$. If the underlying type has k members then $COPY$ has $k+1$ states and the network has $(k+1)^N$. Since all of the internal communications (the movement of data from one $COPY$ to the next) become τ actions, this is an excellent target for diamond elimination. And in fact we get 100% success: the only nodes retained are those that are not τ-reachable from any other. These are the ones in which all of the data is as far to the left as it can be: there are no empty $COPY$'s to the left of a full one. If $k = 1$ this means there are now $N+1$ nodes rather than 2^N, and if $k = 2$ it gives $2^{N+1} - 1$ rather than 3^N.

4.3 Combining techniques

The objective of compression is to reduce the number of states in the target system as much as possible, with the secondary objectives of keeping the number of transitions and the complexity of any minimal acceptance marking as low as possible.

There are essentially two possibilities for the best compression of a given system: either its normal form or the result of applying some combination of the other techniques. For whatever equivalence-preserving transformation is performed on a transition system, the normal form (from its root node) must be invariant; and all of the other techniques leave any normal form system unchanged. In many cases (such as the chain of $COPY$s above) the two will be the same size (for the diamond elimination immediately finds a system equivalent to the normal form, as does equivalence factoring), but there are certainly cases where each is better.

The relative speeds (and memory use) of the various techniques vary substantially from example to example, but broadly speaking the relative efficiencies are (in decreasing order) τ-loop elimination (except in rare complex cases), bisimulation, diamond elimination, normalisation and equivalence factoring. The last two can, of course, be done together since the entire normal form contains the usual normal form within it. Diamond elimination is an extremely useful strategy to carry out before either sort of normalisation, both because it reduces the size of the system on which the normal form is computed (and the number of seed nodes for the entire normal form) and because it eliminates the need for searching through chains of τ actions which forms a large part of the normalisation process.

One should note that all our compression techniques guarantee to do no worse than leave the number of states unchanged, with the exception of normalisation which in the worst case can expand the number of states exponentially[11, 8]. Cases of expanding normal forms are very rare in practical systems. Only very recently, after nearly four years, have we encountered a class of practically important processes whose normalisation behaviour is pathological. These are the "spy" processes used to seek errors in security protocols [13].

At the time of writing all of the compression techniques discussed have been implemented and many experiments performed using them. Ultimately we expect that FDR2's compression processing will be automated according to a strategy based on a combination of these techniques, with the additional possibility of user intervention.

5 Compression in context

FDR2 will take a complex CSP description and build it up in stages, compressing the resulting process each time. Ultimately we expect these decisions to be at least partly automated, but in early versions the compression directives will be included in the syntax of the target process.

One of the most interesting and challenging things when incorporating these ideas is preserving the debugging functionality of the system. The debugging process becomes hierarchical: at the top level we will find erroneous behaviours of compressed parts of the system; we will then have to debug the pre-compressed forms for the appropriate behaviour, and so on down. On very large systems (such as that discussed in the next section) it will not be practical to complete this process for all parts of the system. Therefore we expect the debugging facility initially to work out subsystem behaviours down as far as the highest level compressed processes, and only to investigate more deeply when directed by the user (through the X Windows debugging facility of FDR).

The way a system is composed together can have an enormous influence on the effectiveness of hierarchical compression. The following principles should generally be followed:

1. Put together processes which communicate with each other together early. For example, in the dining philosophers, you should build up the system out of consecutive fork/philosopher pairs rather than putting the philosophers all together, the forks all together and then putting these two processes together at the highest level.

2. Hide all events at as low a level as is possible. The laws of CSP allow the movement of hiding inside and outside a parallel operator as long as its synchronisations are not interfered with. In general therefore, any event that is to be hidden should be hidden the first time (in building up the process) that it no longer has to be synchronised at a higher level. The reason for this is that the compression techniques all tend to work much more effectively on systems with many τ actions.

3. Hide all events that are irrelevant (in the sense discussed below) to the specification you are trying to prove.

Hiding can introduce divergence, and therefore invalidate many failures/divergences model specifications. However in the traces model it does not alter the sequence of unhidden events, and in the stable failures model does not alter refusals which contain every hidden event. Therefore if only trying to prove a property in one of these models – or if it has already been established by whatever method that one's substantive system is divergence free – the improved compression we get by hiding extra events makes it worthwhile doing so.

We will give two examples of this, one based on the $COPY$ chain example we saw above and one on the dining philosophers. The first is probably typical of the gains we can make with compression and hiding; the second is atypically good.

5.1 Hiding and safety properties

If the underlying datatype T of the *COPY* processes is large, then chaining N of them together will lead to unmanageably large state-spaces whatever sort of compression is applied to the entire system. For it really does have a lot of distinct states: one for each possible contents the resulting N-place buffer might have. Of course there are analytic techniques that can be applied to this simple example that pin down its behaviour, but we will ignore these and illustrate a general technique that can be used to prove simple safety properties of complex networks. Suppose x is one member of the type T; an obviously desirable (and true) property of the *COPY* chain is that the number of x's input on channel *left* is always greater than or equal to the number output on *right*, but no greater than the latter plus N. Since the truth or falsity of this property is unaffected by the system's communications in the rest of its alphabet $\{\mathit{left}.y, \mathit{right}.y \mid y \in \Sigma \setminus \{x\}\}$ we can hide this set and build the network up a process at a time from left to right. At the intermediate stages you have to leave the right-hand communications unhidden (because these still have to be synchronised with processes yet to be built in) but nevertheless, in the traces model, the state space of the intermediate stages grows more slowly with n than without the hiding. In fact, with n *COPY* processes the hidden version compresses to exactly 2^n states whatever the size of T (assuming that this is at least 2).

This is a substantial reduction, but is perhaps not as good as one might ideally hope for. By hiding all inputs other than the chosen one, we are ignoring what the contents of the systems are apart from x, but because we are still going to compose the process with one which will take all of our outputs, these have to remain visible, and the number of states mainly reflects the number of different ways the outputs of objects other than x can be affected by the order of inputting and outputting x. The point is that we do not know (in the method) that the outputs other than x are ultimately going to be irrelevant to the specification, for we are not making any assumptions about the process we will be connected to.

Since the size of system we can compress is always likely to be one or two orders of magnitude smaller than the number of explicit states in the final refinement check, it would actually be advantageous to build this system not in one direction as indicated above, but from both ends and finally compose the two halves together. (The partially-composed system of n right-hand processes also has 2^N states.) Nothing useful would (in this example) be achieved by building up further pieces in the middle, since we only get the simplifying benefit of the hiding from the two ends of the system.

If the (albeit slower) exponential growth of states even after hiding and compressing the actual system is unacceptable, there is one further option: find a network with either less states, or better compression behaviour, that the actual one refines, but which can still be shown to satisfy the specification. In the example above this is easy: simply replace *COPY* with

$$C_x = (\mu\, p.\mathit{left}.x \longrightarrow \mathit{right}.x \longrightarrow p) \mathbin{|||} \mathit{CHAOS}(\Sigma \setminus \{\mathit{left}.x, \mathit{right}.x\})$$

the process which acts like a reliable one-place buffer for the value x, but can input and output as it chooses one other members of T. It is easy to show that *COPY* refines this, and a chain of n C_x's compresses to $n+1$ states (even without hiding irrelevant external communications).

In a sense the C_x processes capture the essential reason why the chain of *COPY*'s satisfy the x-counting specification. By being clever we have managed to automate the proof for much larger networks than following the 'dumb' approach, but of course it is not ideal that we have had to be clever in this way.

The methods discussed in this section could be used to prove properties about the reliability of communications between a given pair of nodes in a complex environment, and similar cases where the full complexity of the operation of a system is irrelevant to why a particular property is true.

5.2 Hiding and deadlock

In the stable failures model, a system can deadlock if and only if $P \setminus \Sigma$ can. In other words, we can hide absolutely all events – and move this hiding as far into the process as possible using the principles already discussed.

Consider the case of the N dining philosophers (in a version, for simplicity, without a Butler process). A natural way of building this system up hierarchically is as progressively longer chains of the form

$$\mathit{PHIL}_0 \| \mathit{FORK}_0 \| \mathit{PHIL}_1 \| \ldots \| \mathit{FORK}_{m-1} \| \mathit{PHIL}_m$$

In analysing the whole system for deadlock, we can hide all those events of a subsystem that do not synchronise with any process outside the subsystem. Thus in this case we can hide all events other than the interactions between PHIL_0 and FORK_{N-1}, and between PHIL_m and FORK_m. The failures normal form of the subsystem will have very few states (exactly 4). Thus we can compute the failures normal form of the whole hidden system, adding a small fixed number of philosopher/fork combinations at a time, in time proportional to N, even though an explicit model-checker would find exponentially many states.

We can, in fact, do even better than this. Imagine doing the following:

- First, build a single philosopher/fork combination hiding all events not in its external interface, and compress it. This will (with standard definitions) have 4 states.
- Next, put 10 copies of this process together in parallel, after suitable renaming to make them into consecutive pairs in a chain of philosophers and forks (the result will have approximately 4000 states) and compress it to its 4 states.
- Now rename this process in 10 different ways so that it looks like 10 adjacent groups of philosophers, compute the results and compress it.
- And repeat this process as often as you like...clearly it will take time linear in the number of times you do it.

By this method we can produce a model of 10^N philosophers and forks in a row in time proportional to N. To make them into a ring, all you would have to do would be to add another row of one or more philosophers and forks in parallel, synchronising the two at both ends. Depending on how it was built (such as whether all the philosophers are allowed to act with a single handedness) you would either find deadlock or prove it absent from a system with doubly exponential number of states.

On the prototype version of FDR2, we have been able to use this technique to demonstrate the deadlock of 10^{1000} philosophers in 15 minutes, and then to use the debugging tool described earlier to tell you the state of any individual one of them (though the depth of the parse tree even of the efficiently constructed system makes this tedious). Viewed through the eyes of explicit model-checking, this system has perhaps $7^{10^{1000}}$ states. Clearly this simply demonstrates the pointlessness of pure state-counting.

This example is, of course, extraordinarily well-suited to our methods. What makes it work are firstly the fact that the networks we build up have a constant-sized external interface (which could only happen in networks that were, like this one, chains or nearly so) and have a behaviour that compresses to a bounded size as the network grows.

On the whole we do not have to prove deadlock freedom of quite such absurdly large systems. We expect that our methods will also bring great improvements to the deadlock checking of more usual size ones that are not necessarily as perfectly suited to them as the example above.

6 Conclusions

We have given details of how FDR2's compression works, and some simple examples of how it can expand the size of problem we can automatically check. At the time of writing we have not had time to carry out many evaluations of this new functionality on realistic-sized examples, but we have no reason to doubt that compression will allow comparable improvements in these.

It is problematic that the successful use of compression apparently takes somewhat more skill than explicit model-checking. Only by studying its use in large-scale case studies can we expect to assess the best ways to deal with this – by automated tactics and transformation, or by design-rule guidance to the user. In any case much work will be required before we can claim to understand fully the capabilities and power of the extended tool.

Acknowledgements

As well as our owing him a tremendous debt for his development of CSP, on which all this work is based, it was a remark by Tony Hoare that led the first author to realise how our methods could check the exponential systems of dining philosophers described in this paper.

We would like to thank the referees for some helpful remarks, in particular for pointing out the need for Lemma 1.

The work of Roscoe and Scattergood was supported in part by a grant from the US Office of Naval Research.

References

[1] S.D. Brookes and A.W. Roscoe, *An improved failures model for communicating processes,* in Proceedings of the Pittsburgh seminar on concurrency, Springer LNCS 197 (1985), 281-305.

[2] J.R. Burch, E.M. Clarke, D.L. Dill and L.J. Hwang, *Symbolic model checking: 10^{20} states and beyond*, Proc. 5th IEEE Annual Symposium on Logic in Computer Science, IEEE Press (1990).

[3] R. Cleaveland and M.C.B. Hennessy, *Testing Equivalence as a Bisimulation Equivalence*, FAC **5** (1993) pp1–20.

[4] Formal Systems (Europe) Ltd., *Failures Divergence Refinement* User Manual and Tutorial, version 1.4 1994.

[5] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall 1985.

[6] L. Jategoankar, A Meyer and A.W. Roscoe, *Separating failures from divergence*, in preparation.

[7] R. Kaivola and A Valmari *The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic* in Proc CONCUR '92 (LNCS 630).

[8] P.C. Kanellakis and S.A. Smolka, *CCS expressions, Finite state processes and three problems of equivalence*, Information and Computation **86**, 43-68 (1990).

[9] K. Melhorn *Graph Algorithms and NP Completeness*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag 1984.

[10] A.W. Roscoe, *Unbounded Nondeterminism in CSP*, in 'Two Papers on CSP', PRG Monograph PRG-67. Also Journal of Logic and Computation **3**, 2 pp131-172 (1993).

[11] A.W. Roscoe, *Model-checking CSP*, in A Classical Mind: Essa ys in Honour of C.A.R. Hoare, A.W. Roscoe (ed.) Prentice-Hall 1994.

[12] A.W. Roscoe, CSP and determinism in security modelling to ap pear in the proceedings of 1995 IEEE Symposium on Security and Privacy.

[13] A.W. Roscoe, *Modelling and verifying key-exchange protocols using CSP and FDR*, to appear in the proceedings of CSFW8 (1995), IEEE Press.

[14] A.W. Roscoe, J.C.P. Woodcock and L. Wulf, *Non-interference t hrough determinism*, Proc. ESORICS 94, Springer LNCS 875, pp 33-53.

[15] J.B. Scattergood, *A basis for CSP tools*, To appear as Oxford University Computing Laboratory technical monograph, 1993.

[16] A. Valmari and M. Tienari *An improved failures equivalence for finite-state systems with a reduction algorithm*, in Protocol Specification, Testing and Verification XI, North-Holland 1991.

A Front-End Generator for Verification Tools *

Rance Cleaveland † Eric Madelaine ‡ Steve Sims †

April 28, 1995

Abstract

This paper describes the Process Algebra Compiler (PAC), a front-end generator for process-algebra-based verification tools. Given descriptions of a process algebra's concrete and abstract syntax and semantics as structural operational rules, the PAC produces syntactic routines and functions for computing the semantics of programs in the algebra. Using this tool greatly simplifies the task of adapting verification tools to the analysis of systems described in different languages; it may therefore be used to achieve source-level compatibility between different verification tools. Although the initial verification tools targeted by the PAC are MAUTO and the Concurrency Workbench, the structure of the PAC caters for the support of other tools as well.

1 Introduction

The past ten years have seen the development of a variety of automatic verification tools for finite-state systems expressed in process algebra; examples include MAUTO [6], the Concurrency Workbench [10], TAV [14], and Aldébaran [11]. In general, these tools support a specific language, such as CCS [19], Meije [1], or Basic Lotos [5], for describing systems and provide users different methods, such as equivalence checking, preorder checking, model checking, random simulation, and abstraction mechanisms, for analyzing their behavior. The utility of these tools has been demonstrated via several case studies [7, 18]. However, the impact on system design practice of such tools has been limited by the fact that the languages they support, while possessing nice theoretical properties, are not widely used by system engineers. In addition, as each tool in general supports a different language, it is difficult to compare the tools and to investigate approaches to using them in collaboration with one another.

This paper presents the Process Algebra Compiler (PAC), a system that substantially simplifies the task of changing the language supported by verification tools. The PAC is a "front-end generator"; given a description of the syntax and semantics of a language, it produces routines for parsing and unparsing programs in the language and for computing user-defined semantic relations. By providing users with high-level notations for defining languages and managing the difficult and

*This work is partially funded by NSF-INRIA collaboration # CCR-9247478, ESPRIT Basic Research Action CONCUR2, NSF/DARPA grant CCR-9014775, ONR Young Investigator Award N00014-92-J-1582, and NSF Young Investigator Award CCR-9257963.

†`rance@csc.ncsu.edu, gr-sts@galois.csc.ncsu.edu`, Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA, (919) 515-7862.

‡`Eric.Madelaine@sophia.inria.fr`, INRIA, B.P. 93, 06902 Sophia Antipolis Cedex, France, (+33) 93 65 78 07.

technically tedious development of syntactic and semantic functions, the PAC provides the research community with a useful tool for expanding the repertoire of languages their tools can support.

The remainder of the paper is organized along the following lines. The next section sharpens the motivation for the PAC by presenting two verification tools, MAUTO and the Concurrency Workbench (CWB) , and the common semantic framework underlying the (different) languages each supports. The section following presents an overview of the architecture of the PAC and describes the specification language used for defining algebras and their semantics, while Section 4 discusses issues in generating semantic functions from their PAC specifications. Section 5 then gives experimental results obtained from PAC-produced front ends for the Concurrency Workbench; somewhat surprisingly, the PAC-generated code significantly outperforms existing hand-produced code built for this tool. The final section contains our conclusions and directions for future work.

2 Verification Tools and Structural Operational Semantics

This section presents an overview of two verification tools, MAUTO and the Concurrency Workbench. Although similar in intent, the tools differ markedly in terms of the analyses they support, and yet at the moment there is no way for a user to use the tools collaboratively. On the other hand, the languages supported by the two tools have a semantics that is given in a very similar style, which we also discuss at the end of this section. These observations provided the impetus for the development of the PAC.

2.1 Verification tools

Both MAUTO and the Concurrency Workbench provide utilities for verifying finite-state systems expressed in process algebra. The specific process algebras supported differ, however, as do the supported analyses. The following provides more detail about the systems.

MAUTO. MAUTO is a system for analyzing networks of finite-state systems. MAUTO builds automata from programs in the Meije process algebra and is capable of reducing and comparing them with respect to various bisimulation-based equivalences. It also provides a novel facility that enables users to define *abstract* transition relations on a given automaton, obtaining a new system, usually smaller and more tractable, that highlights specific behaviors of the original system. Much attention has been devoted to issues of efficiency. In particular, the building of automata from terms is mixed with the reduction of the automata using congruence properties of the semantic equivalences, thereby ensuring that automata are kept as small as possible. Facilities are also provided for explaining the results of analysis and for drawing the resulting automata in a graphical editor [21].

The Concurrency Workbench. The Concurrency Workbench (CWB) is an extensible tool for verifying systems written in the process algebra CCS . In contrast with other process algebra tools, the CWB supports the computation of numerous different semantic equivalences and preorders; it does so in a modular fashion in that generic equivalence- and preorder-checking routines are combined with suitable process transformations (see, e.g., [8]) in order to compute different relations. The CWB also includes a flexible *model-checking* facility for determining when a process satisfies a formula in a very expressive temporal logic, the propositional mu-calculus. Recently the CWB has been extended to deal with a discrete-time version of CCS (TCCS), and with the synchronous algebra SCCS.

2.2 Structural Operational Semantics

MAUTO and the CWB are similar in that they analyze systems by converting them into finite automata and then invoking routines on these automata. However, the languages and forms of analysis they support, and the approaches they take to construct automata from systems, differ markedly. In the last case in particular, MAUTO adopts a "bottom-up" approach, with automata recursively constructed for subsystems and then assembled into a single machine for the entire system. The CWB, on the other hand, uses an "on-the-fly" approach, with transitions of components calculated and then combined appropriately into transitions for the over-all system.

One characteristic shared by MAUTO and the CWB, however, is that the languages they support have operational semantics given in the Structural Operational Semantic (SOS) [20] style; this fact motivates our inclusion in the PAC of capabilities for generating routines from SOS descriptions. A SOS for a language consists of rules for inferring the execution behavior of programs written in the language. Rules have the following general form.

$$\frac{\mathit{premises}}{\mathit{conclusion}}(\mathit{side\ condition})$$

The intuitive reading of the rule is that if one is able to establish the premises, which typically involve statements about the execution behavior of subprograms of the one mentioned in the conclusion, and the side condition holds, then one may infer the conclusion. As an example, the following describes the synchronizations allowed by the parallel composition operation in CCS.

$$\frac{p \xrightarrow{a} p' \quad q \xrightarrow{b} q'}{p|q \xrightarrow{\tau} p'|q'}(a, b\ \mathit{inverses})$$

The rule states that if p can engage in an action a and evolve to p' and q can engage in b and evolve to q', and a and b are inverses (i.e. constitute an input/output pair on the same communication channel), then $p|q$ can execute an internal action, τ, corresponding to the synchronized execution of a and b.

The SOS style has evolved in many ways since the Plotkin's seminal paper [20] and has been applied to many areas of language semantics. The SOS style is very flexible, as numerous languages with widely varying features have been given semantics using this framework. Recent work has focused on the metatheory of SOS [4, 17, 12, 2, 22]; in particular, researchers have shown that when SOS rules conform to different syntactic formats, the resulting languages have nice properties. In the area of process algebras, one important property is congruence of the behavioral equivalences with respect to the operators of the language; this allows one to reason about the system components in a compositional way.

3 Using the PAC

This section provides an overview of the PAC architecture and indicates how users specify process algebras for processing by the PAC.

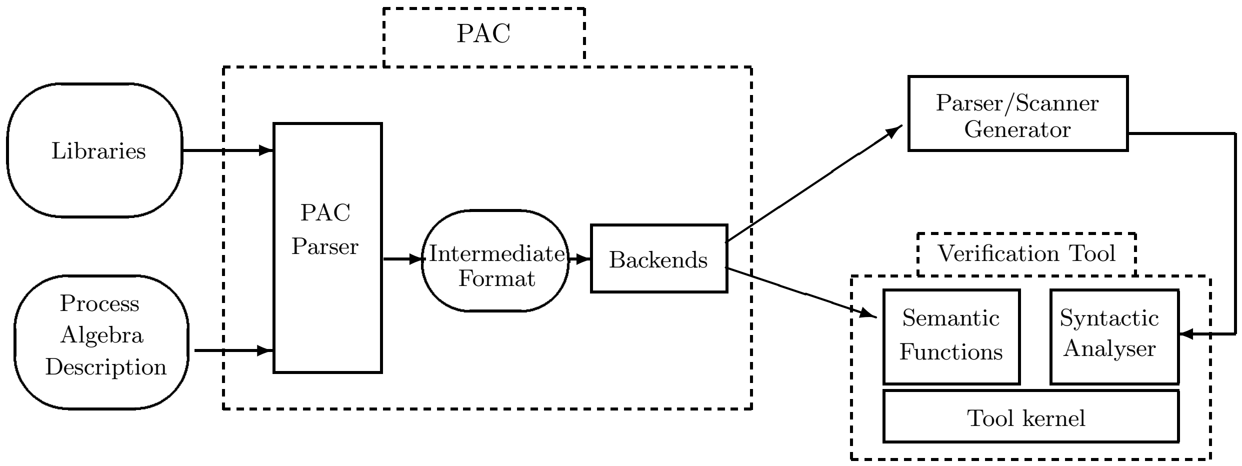


Figure 1: Architecture of the PAC

3.1 PAC Overview

Figure 1 sketches the organization of the PAC. The system takes as input files containing the syntactic and semantic description of a process algebra as well as libraries containing the definitions of any necessary auxiliary functions. It then produces two (sets of) files:

- A YACC/LEX[1] specification of the language's syntax.
- Semantic routines to analyze programs written in the language.

To specialize a target verification tool to the given language, the PAC user must run YACC/LEX on the first set of files to produce a parser and then insert the parser and the semantic routines into the verification engine. Provided that the target tool separates the syntactic analysis of programs from their verification, the language-independent part of the verification tool (its *kernel*) need not change at all. It should be noted that the PAC is in fact a "compiler": it takes a PAC specification of a language as input and produces source code which is compiled along with the kernel of the target verification tool. There is **no** PAC run-time system that becomes part of the target verification tool. The PAC can also be viewed as a "compiler compiler" since the generated code is a "compiler" which accepts a process algebra program as input and produces a labeled transition system as output.

The PAC itself is organized into several components centered around an internal representation of the syntax and semantics of the language being processed. This internal structure is produced by the PAC parser from files provided by the user. From it, *back ends* produce the required routines for the target systems. As target verification tools are typically written in different languages (MAUTO in Lisp, the Concurrency Workbench in SML, Aldébaran in C, for example), there will in general be several back ends in the PAC. The initially targeted systems are MAUTO and the Workbench; accordingly, the existing prototype includes back ends that generate Lisp and SML, respectively.

[1]Using YACC/LEX provides an easy way of guaranteeing the compatibility of parsers generated for a given algebra by different back ends. Other parser-generators may also be used at the discretion of the back-end writer.

The PAC parser. The PAC parser tries to factor out as much of the back-end-independent work as possible from the processing of user-supplied algebra descriptions; in particular, it checks the PAC input for syntactic correctness and performs certain consistency checks. If user files satisfy these criteria, the parser then produces an intermediate representation of the input which contains:

- A representation of the abstract syntax of the process algebra.
- A structured description of the concrete syntax from which specifications for scanners, parsers, and unparsers may be generated.
- A representation of the sets of SOS rules used for defining the semantics of the operators in the algebra.

Back ends. The back ends build the actual routines to be included in the verification tools. They accept as input the intermediate format generated by the PAC parser and generate as output a YACC grammar together with routines that compute the semantics of a system from its abstract syntax. The routines typically differ from one verification tool to another; typical examples include those for computing the single-step transitions of a process, generating the composition of several automata by a given composition operator, computing sufficient syntactic conditions for a process to be finite-state [17], and calculating whether or not a process is divergent.

Implementation. The PAC is implemented in Standard ML (SML). The system, which currently consists of roughly 15,000 lines of code, is batch-oriented; it processes inputs and either generates output files or reports error messages.

3.2 PAC Process Algebra Specifications

A PAC process algebra specification consists of two components. The `ALGEBRA` module contains descriptions of the concrete and abstract syntax of actions, processes and semantic relations. The `RULE_SET` modules contain SOS rules defining the relations used to define the semantics of processes. Users may also provide *library* files containing code that directly implements auxiliary structures (such as sets or environments) and operations (such as set membership or lookup functions) used in defining the semantics and for which users do not wish to provide SOS definitions. Back ends directly insert this code into the files they generate; consequently this code must be written in a language compatible with that of the target tool. The remainder of this section discusses each of these modules using as an example the CWB-6.0 version of Milner's CCS .

3.2.1 `ALGEBRA` Modules

`ALGEBRA` modules consist of several sections.

- In the `sorts` section, users define the syntactic categories for their language.
- The `cons` section defines the term constructors (i.e. the abstract syntax) to be used to build elements in the different syntactic categories.
- The `funs` sections introduces the names and "types" of functions that may be applied to elements of sorts. The implementations of these functions must be supplied by the PAC user.

```
ALGEBRA CCS
sorts
    id, act, id_set, restriction, ('a eqn), agent, ('a frame), ('a env), ...
cons
    Tau         : act
    Input       : id -> act
    Output      : id -> act
    ...
    Res_set     : id_set -> restriction
    Res_var     : id -> restriction
    Eqn         : id * 'a -> ('a eqn)
    ...
    Nil         : agent
    Bottom      : agent
    Ag_var      : id    -> agent
    Prefix      : act * agent -> agent
    Plus        : agent * agent -> agent
    Restriction : agent * restriction -> agent
    Fix         : agent * (agent frame) -> agent
    ...
funs
    id_parse    : string -> id
    id_eq       : id * id -> bool
    ...
    inverses    : act * act -> bool
    ...
    mk_id_set   : (id list) -> id_set
    member      : id * id_set -> bool
    ...
    mk_frame    : (('a eqn) list) -> ('a frame)
    mk_frame_inv: ('a frame) -> (('a eqn) list)
    empty       : 'a env
    push_frame  : ('a frame) * ('a env) -> ('a env)
    ...
rels
    transition  : (agent env) * (id_set env) * agent * act * agent -> bool
    diverges    : (agent env) * (id_set env) * agent -> bool
inputs
    transition is [1,2,3]
    diverges is [1,2,3]
pragmas
    ...
    CWB "parser entries:  act, agent, id_set"
    ...
SYNTAX
    ...
RULE_SYNTAX
    ...
end
```

Figure 2: An `ALGEBRA` module for CCS

- The `rels` section defines the names and "types" of the semantic relations to be defined by SOS rules and for which the PAC will generate an implementation. The `inputs` section then indicates what type the generated functions computing these relations should have (i.e. which positions in the relation should be "inputs" and which should be "outputs").

- The `pragmas` section contains back-end-specific directives, such as locations of library files and names to be assigned to functions generated by back ends.

- The `SYNTAX` and `RULE_SYNTAX` sections contain descriptions of the concrete syntax of both the process algebra and of the relations used to define the algebra's semantics.

To illustrate what appears in these sections, consider the (elided) version of an `ALGEBRA` module for CCS in Figure 2. The `sorts` section declares the kinds of objects that appear in the definition of the algebra, including `act` (actions), `agent` (CCS processes), `'a eqn` (equations), and `'a frame` (frames, or mappings from identifiers to values). Note that sorts may be polymorphic: in the case

```
SYNTAX
tokens
        "nil"     => NIL
        "where"   => WHERE
        "and"     => AND
        "end"     => END
        ...
priorities
        ...
nonterminals
        agent of agent
        ag_eqn_list of (agent eqn) list
        ...
grammar
        agent : NIL                            (Nil())
              | agent WHERE agent_frame END    (Fix(agent, agent_frame))
                ...
        agent_frame : ag_eqn_list              (mk_frame(ag_eqn_list))
lists
        ag_eqn_list is non_empty_list EMPTY_STR COMMA AND EMPTY_STR of ag_eqn
        ...
RULE_SYNTAX
        ...
grammar
        relation : agent_env COMMA id_set_env COLON agent DASHDASH act ARROW agent
                   (transition (agent_env, id_set_env, agent1, act, agent2))
...
```

Figure 3: A SYNTAX and RULE_SYNTAX section for CCS

of 'a frame, for instance, the 'a may be instantiated with any well-formed sort. The PAC also includes three built-in sorts: string for character strings, bool for booleans, and 'a list for polymorphic lists.

The next section of the example introduces the constructors used in CCS and their sorts. For example, Tau is introduced as a constructor taking no arguments and producing a value of sort act; that is, Tau is an action constant. Input and Output take an identifier (intuitively, a channel name) as an argument and produce an action. In the CWB version of CCS, users may bind identifiers to sets of actions and then use these identifiers in place of sets in the restriction operator. To cater for this possibility, the algebra introduces a sort restriction and two constructors, Res_set and Res_var, permitting sets of identifiers (i.e. a label set, in CCS terms) or a single identifier (a variable name bound to a set) to be viewed as "restrictions". Eqn is used to construct equations from identifiers and values, while the remaining constructors are used to build agents. Note that the Fix operator takes an agent and an agent frame as arguments; intuitively, the frame contains bindings for the free variables that may appear in the agent.

The funs section introduces operations that may be applied to elements of given sorts. These operations differ from constructors in that the PAC will generate implementations for the latter but not for the former; users must provide routines for these. This feature permits users to re-use existing code and to program efficient implementations of low-level data structures as appropriate.[2] Thus, to generate a CWB front end on the basis of the example algebra module a user would neet to provide implementations in Standard ML (the language in which the CWB is written) for operations such as id_parse, mk_id_set and mk_frame.

[2]PAC back ends also generate implementations of sorts having constructors declared for them; it relies on users to specify implementations of sorts for which no constructors have been specified. For example, act would have a PAC-supplied implementation, since three constructors have act as their return sort. The sort 'a frame, on the other hand, does not have constructors defined for it; consequently, a user must supply code defining the data structure to be used to represent frames.

The `rels` component of the example introduces two semantic relations: `transition` and `diverges`. In this version of CCS, the transitions and potential for divergence of an agent depend on two environments: one to resolve free agent variables, and one to resolve free variables used in restrictions. Thus each relation includes an `agent environment` and an `id_set environment` argument.

For each relation the `inputs` section indicates the form the PAC-generated function for computing this relation should take. In the case of `transition`, for example, the input specification indicates that the generated function should have three inputs corresponding to the first three positions of the relation (here, two environment arguments and an agent). Given such a triple, the function will return the set of all action-agent pairs which, when combined with the triple, yield a quintuple in the relation. In the case of `diverges`, all places are mentioned in the input list; in this case, the PAC will generate a function taking three arguments and returning a boolean.

The `pragmas` section includes miscellaneous directives for specific back ends. In the above example, the given pragma indicates that the parser produced by the PAC back end for the CWB should have entries for agents, actions and identifier sets. These are needed since the CWB supports commands requiring users to provide information from these sorts. Other pragmas might be used to rename sorts appropriately (some tools might require a type `proc` rather than `agent`, for example) or supply names of library files.

Samples of the syntax sections of the CCS algebra specification appear in Figure 3. The `SYNTAX` component contains information needed to generate the parsers to be used by the target tool; this currently takes the form of a YACC-like grammar whose semantic actions consists of "sort-correct" expressions built using the constructors and functions declared previously. In the example, the syntax of the fixpoint agent operator is defined to be *a* `where` e_1 `and` ... `and` e_n `end`, where each e_i is an equation. Note that PAC grammars extend YACC grammars by permitting list specifications; the nonterminal `ag_eqn_list`, for example, yields lists of `ag_eqn` (agent equations) whose beginning and ending delimiters are the empty string and whose separator is the token AND (`and` in concrete syntax). The `RULE_SYNTAX` section enriches this syntactic specification with information needed to parse the SOS rules that define the semantics of processes; in particular, it includes definitions of the concrete syntax of relations. This example defines the syntax of the `transition` relation to be *ae*`,` *se* `:` *p* `--`*a*`-->` *q*. The PAC fits this information into a general "rule template" in order to produce a grammar which is processed and then used to parse the user-supplied rules.

3.2.2 `RULE_SET` Modules

The second part of a PAC process algebra specification consists of the SOS rules needed to define the semantics of processes. In general, a user must supply a collection of rules for each relation introduced in the `ALGEBRA` module. Each rule in turn consists of four components: a name, a list of *premises*, a *side condition*, and a *conclusion*. In general, premises and conclusions involve relations, while side conditions can be any expression generated using the following grammar,

$$be ::= \texttt{true} \mid \texttt{not}\ be \mid be\ \texttt{and}\ be \mid be\ \texttt{or}\ be \mid P(t_1, ..., t_n)$$

where P is a predicate: any boolean-sorted function declared in the `funs` section or any relation in the `rels` section all of whose positions are input positions. The t_i should be terms in the appropriate sort, based on the definition of P. A fragment of the rules for the `transition` relation

```
RULE_SET transition
vars
    a, b                    : act
    p, p', p1, p1', p2, p2' : agent
    s                       : id_set
    ae                      : (agent env)
    se                      : (id_set env)
    ...
rules
    prefix
                  -------------------------(true)
                   ae, se : a.p -- a --> p
    ...
    parallel_1
                        ae, se : p1 -- a --> p1'
                  ------------------------------------(true)
                   ae, se : p1 | p2 -- a --> p1' | p2
    parallel_2
                        ae, se : p2 -- a --> p2'
                  ------------------------------------(true)
                   ae, se : p1 | p2 -- a --> p1 | p2'
    parallel_3
                   ae, se : p1 -- a --> p1', ae, se : p2 -- b --> p2'
                  --------------------------------------------------(inverses(a,b))
                          ae, se : p1 | p2 -- t --> p1' | p2'
    ...
end
```

Figure 4: A `RULE_SET` module for the CCS `transition` relation

for CCS appears in Figure 4. Note that premises appear above, and conclusions below, a line of hyphens, with the side condition appearing in parentheses after the hyphens. All expressions in the rules are written using the concrete syntax declared in the `ALGEBRA` module; this enables the rules to look very close to what appears in the literature. In addition, functions defined in the `ALGEBRA` module may be used in the rules; for example, `parallel_3` contains a reference to the `inverses` predicate, which intuitively should hold when the given actions represent an input and output on the same channel (note that `t` is the concrete syntax that has been defined for the CCS internal action). Rule sets can refer to relations defined in other rule sets, although no such reference is made in this example.

4 PAC Back Ends

The PAC currently includes back ends dedicated to the CWB and to MAUTO. The former generates code in SML, the language in which the CWB is written, while the latter, which is still under construction, produces LeLisp, the programming language in which MAUTO is implemented. In each case, the produced code contains a parser, some unparsing functions, and a number of semantic functions encoding the SOS rules of the algebra. The parsers (generated using respectively LeLisp-Yacc and SML-Yacc) are fully compatible, meaning that PAC-generated front ends for MAUTO and the CWB handle the same syntax. As Section 2 indicated, however, the analysis functions of the target tools are different, as are the semantic functions. The following discusses what semantic functions the different back ends must produce and how they are generated from SOS specifications.

4.1 The CWB Back End

In addition to various parsing and unparsing routines, the CWB requires that its front end include implementations of types `act` and `agent` and functions `transition`, `diverges` and `sort`. The functions each take an agent and return a set of action-agent pairs, a boolean, and a set of actions, respectively. This section describes how the CWB back end generates code from the SOS definitions of semantic relations. Generally speaking, given a rule set for a particular semantic relation, the technique constructs a function whose inputs correspond to the places in the relation declared as inputs in the `inputs` section of the `ALGEBRA` module. On a given input, the generated routine produces a set of tuples as outputs; the idea is that each output tuple, when combined with the input tuple, yields an element in the relation. As an example, in the case of the `transition` relation defined in Section 3.2, positions 1, 2 and 3 of `transition` are declared as inputs; the procedure that is produced will therefore accept an $\langle \texttt{agent env}, \texttt{id\_set env}, \texttt{agent}\rangle$ triple as input and produces a set of $\langle \texttt{action}, \texttt{agent}\rangle$ pairs as output with the property that if the input is $\langle ae, se, p\rangle$, then pair $\langle a, q\rangle$ is in the set of outputs if and only if $\langle ae, se, p, a, q\rangle$ is in relation `transition`.

In order for the procedure described below to work, the rules used to define semantic relations must obey certain syntactic restrictions. Recall from Sections 2.2 and 3.2.2 that SOS rules have the following general form:

$$\frac{premises}{conclusion}(side\ condition)$$

where *conclusion* is an element of the relation being defined, *premises* is a list of elements of the relation being defined or of other relations declared in the `rels` section, and *side condition* is a boolean expression that may involve predicate expressions of the form $P(t_1, \ldots, t_n)$, with the t_i being terms that may involve variables. For the code produced by the CWB back end to compile, each rule must satisfy the following constraints.

1. All variables appearing in the input positions of the premises must appear in the input positions of the conclusion.

2. All variables appearing in the output positions of the conclusion or in the side condition must appear either in the input positions of the conclusion or in the output positions of a premise.

3. All variables appearing in the input positions of the conclusion or the output positions of a premise are distinct.

These constraints place restrictions on the “flow of data” through a rule: information flows from the inputs of the conclusion to the premises, and the outputs of premises flow (together with inputs of the conclusion) to the side condition and the outputs of the conclusion. Note also that patterns of arbitrary depth can appear in the input or output positions of the conclusion or premises. It should be noted that this rule format subsumes the positive GSOS format of [4] while being incomparable to the tyft/tyxt pattern of [12] and the path scheme of [2]. However, restriction 1 can be relaxed without too much difficulty to allow variables appearing in the output positions of premises to appear in the input positions of other premises; with this generalization, our format would subsume pure and well-founded tyft/tyxt and path. Other formats allow negative premises [4, 22] and are incomparable to ours.

The basic strategy used by the code generated from rules involves pattern matching: given a tuple of inputs, a generated function in essence determines which rules have conclusions whose input positions match the input tuple. Using the premises of these rules, appropriate (recursive) calls are issued, and the results which satisfy the side condition are combined into a set of result tuples using the form of the conclusion. To illustrate this idea, consider the rules given for CCS in Section 3.2.2, and suppose that the generated semantic function is given an input of the form

$\langle \texttt{ae}, \texttt{se}, p|q\rangle$. In this case, three rules are applicable— `parallel_1`, `parallel_2` and `parallel_3`. Each of the rules mentions the transitions of p or q in the premises; consequently, the generated code would include recursive calls to calculate the transitions for $\langle \texttt{ae}, \texttt{se}, p\rangle$ and $\langle \texttt{ae}, \texttt{se}, q\rangle$. On the basis of the first rule, transitions of p would be combined appropriately with q, while the second rule would transform transitions of q by combining them with p. The final rule combines transitions of the form $\langle a, p'\rangle$ and $\langle b, q'\rangle$ into $\langle \tau, p'|q'\rangle$ *provided that* the predicate $\texttt{inverses}(a, b)$ is satisfied. The results of these combinations are collected into one set and returned.

To improve the performance of the generated code, the CWB back end also employs several optimizations. For example, in order to minimize matching overhead, rules with the same input pattern in their conclusions are grouped and processed simultaneously. Also, the generated routines cache results of recursive calls in a hash table; before issuing a recursive call, this table is consulted to determine if the call has been made before. To demonstrate the savings from this technique, consider how the CWB would compile the agent $p|q$ into a labeled transition system. First, a call is made to the generated CCS transition function with $\langle ae, se, p|q\rangle$ as input (ae and se are the current agent and set environments). After making the recursive calls to compute the transitions of p and q, the generated `transition` function saves the results of these calls in a table. From *par_rule1*, it follows that $p|q$ has a transition $\langle a, p'|q\rangle$ for every transition $\langle a, p'\rangle$ of p. The next step in compiling the labeled transition system for $p|q$ will include computing the transitions of each $p'|q$; but, in this case instead of making recursive calls to recompute the transitions of q, `transition` would simply look this up in the transitions table. This strategy leads to significant time savings when computing the finite-state representation of a system; somewhat surprisingly, it can also lead to substantial space savings as well, since sharing becomes possible in the computation of output tuples.

4.2 The MAUTO Back End

MAUTO uses a "compositional" (bottom-up) approach to building automata; language constructs are interpreted as automaton transformations, and thus it is not in general possible to use directly the transition function described above. The same SOS rules that yield the *transition* function for the CWB are interpreted as specifying these automata transformers. Thus, the semantic functions generated by the PAC for MAUTO encode *transition system transducers* in the sense of Larsen and Xinxin [15]. Computing the automaton for $p|q$, for example, involves computing separately the finite automata describing the full behaviors of p and q, eventually reducing each of them according to any congruence at hand, then combining them using the transducer for parallel composition. In practice, the rule format required by this interpretation is more restrictive than the one in the preceding section: it ensures that the transducer generated for any context expression is finitely represented, and that the combination of finite automata always yields a finite global automaton. The structure of the functions produced for MAUTO is also very different from the structure of those produced for the CWB. In general, there are two functions for each operator, one describing the recursive structure of the bottom-up traversal, and one describing how to combine the transitions of a tuple of argument automata.

A static analysis of the structure of the SOS rules of the transition relation allows us to classify the process operators in the algebra. This is used to produce optimized automata-constructing routines, to ensure the finiteness of the produced transducers, and to guarantee *a priori* the termination of automata construction. The classification is a generalization of the notions defined in [17], and distinguishes between:

- *Combinators*, which are typically operators used for parallel composition. The format ensures that they do not generate infinite transition systems from finite arguments.

- *Switches*, which have only one process argument active at a time, and will eventually select one of them (sum, sequence). They are used for defining static conditions for finiteness of recursive definitions.

- *Sieves*, which have exactly one process argument and act as action transformers, keeping their structure unchanged (hiding, restriction, relabeling). Identifying sieves enables various run-time optimizations to be employed that avoid some intermediate automaton constructions.

We have produced code using these ideas that has proved to be very efficient and flexible, and easy to integrate with other compositional approaches.

5 Results

The current prototype of the PAC includes an algebra description parser and a back end for the CWB, with the development of the MAUTO back end in progress. In this section we describe our experience with using the PAC to generate front ends for the CWB. The experiments take two forms. In the first, we compare the efficiency of a PAC-generated front end for CCS with existing hand-coded front ends for CCS, while in the second we investigate the performance of front ends produced by the PAC for other languages. Our initial results suggest that the PAC does indeed ease the task of changing the language supported by the CWB and that the generated interfaces perform well. Our tests used a version of the Concurrency Workbench under development at North Carolina State University and were run on a Sun Sparc 5 with 128 megabytes of RAM. The functionality of this version of the CWB is similar to the Edinburgh CWB [10], but the NCSU version includes more efficient graph-construction and equivalence-checking routines.

Table 1 compares the performance of a PAC-generated front end with two hand-coded front ends for CCS. The first of these is the front end included with Version 6.0 of the CWB, while the second is a hand-tuned version of the first one developed at NCSU. The numbers describe the amount of processor time in seconds (time needed for system activities and for garbage collection have been omitted) needed by the NCSU CWB to build finite-state automata from different CCS sample programs using the transitions function supplied by the given interface. The example programs we used to test the interfaces included:

- Two communications protocols: an implementation of the Alternating Bit Protocol (ABP) and an implementation of part of the data link control layer of IEEE 802.2 (802-2).

- Two solutions of the two-process critical-section problem: an implementation of Dekker's algorithm (Dekker-2) and an implementation using semaphores (Semaphore-2).

- Milner's Jobshop example [19] (Jobshop).

- A specification of the Edinburgh mail system (Mail-system).

In addition, we tried some examples consisting of the parallel composition of these examples in order to assess the performance of the front ends on systems with large state spaces. In the table these examples have the form System$_1$| System$_2$. As the table indicates, the PAC-generated

Example	*Number of states*	*Interface*		
		CWB 6.0	NCSU CWB	PAC-generated
ABP	57	0.12	0.13	0.14
Jobshop	77	0.14	0.14	0.12
Dekker-2	127	0.38	0.35	0.39
802-2	331	1.67	1.33	1.83
Semaphore-2	468	2.66	2.44	2.25
Mail-system	1616	9.12	8.68	7.59
ABP | Jobshop	4389	18.82	13.76	10.73
Dekker-2 | Semaphore-2	59436	522.82	288.19	101.88
ABP | Mail-system	92112	Ran out of memory	Ran out of memory	340.90
Dekker-2 | Mail-system	205232	Ran out of memory	Ran out of memory	779.03

Table 1: This table shows the program time in seconds required by the different interfaces to construct automata for various CCS examples.

CCS interface actually performs substantially better than existing CCS interfaces while using less memory; the main reason for this lies in the caching of recursive calls outlined in Section 4.1.

We have also used the PAC to generate CWB front ends for several other languages as well. Examples have included a simple language of regular expressions and a version of CCS in which actions take priority [9]; the latter is noteworthy in that its semantic account requires the use of auxiliary semantic relations. In general, the amount of effort required has been much less than what would be required to generate interfaces by hand, although more experience with the tool is necessary to substantiate this claim. However, the fact that the notations the tool provides for expressing semantic and syntactic specifications of languages are more abstract than those provided by standard programming languages lead us to believe that the PAC will greatly simplify the production of front ends for verification tools.

Our most involved example has been the generation of a CWB front end for Basic Lotos, which is more complex, both syntactically and semantically, than the others we have tried. We have analyzed a number of Basic Lotos examples with the generated interface. Since no Basic Lotos interface existed previously for the CWB it is harder to evaluate the efficiency of the generated code than it was in the case of CCS. One crude measure, however involves comparing the states generated per unit time from LOTOS programs against a similar figure for the CCS front ends described previously. The states-per-second measures for the CCS front ends were computed from the first eight examples in the table above (the ones that all interfaces were able to handle), while the figure for the Basic Lotos interface was calculated based on timing results from the compilation of 8 examples ranging in size from 20 states to 45,000 states. The results are shown in Table 2, which shows that the front end generated for Basic Lotos is roughly 8 times slower than the one generated for CCS. This difference is not necessarily due to the inadequacy of the code-generating scheme used by the PAC, but rather arises from the fact that Basic Lotos is syntactically and semantically more complex than CCS.

	Interface			
	CWB 6.0	NCSU CWB	PAC-generated CCS	PAC-generated Basic Lotos
States per second	119.66	211.10	532.31	65.47

Table 2: Average number of states generated per second for four different interfaces.

6 Conclusions

In this paper we have presented the Process Algebra Compiler, a tool for generating front ends for verification tools. The PAC allows users to specify the syntax and semantics of a language they wish their verification tool to support; the system then produces the syntactic and semantic routines needed to specialize the given tool for the language. Experimental results indicate that PAC-generated routines exhibit performance that can in fact improve on that of hand-coded routines.

Regarding future work, our most immediate goal is to complete the MAUTO back end so that it and the CWB may become source-level compatible. Back ends for other verification tools could also be built and our experiences in building the CWB and MAUTO back ends would certainly ease this task. We anticipate that this will be possible for most tools based on transition system semantics, although some reorganization of the target tools may be necessary.

We would also like to investigate the addition of features in the PAC specification language. In particular, the lexical specifications supported by the PAC can be made more flexible, and providing some facility for modularity in the algebra section would be desirable. We have experimented with the latter; defining concrete syntax in a modular way, however, appears to be very difficult. We have also experimented with a less flexible, but much easier to use, format for expressing concrete syntax and plan to study this issue more. We have also worked on and would like to further investigate routines in the PAC for analyzing a rule set and reporting to the user whether it satisfies a given rule format, such as those mentioned in Section 2.2.

We also would like to explore the possibility of using the PAC for activities other than generating front ends for verification tools. Given the widespread use of SOS rules for defining the semantics of languages, it might be possible to use the PAC to automatically generate interpreters and compilers. We are also examining the feasibility of using the PAC as an implementation engine for generating on-the-fly verification routines, as these may often be formulated using SOS-style rules [3]. Obviously, these uses are greatly different from the PAC's initial purpose, and it remains to be seen if they are indeed practical.

Related Work. Other verification tools have also aimed at providing some parametricity with respect to the language analyzed. The ECRINS system [16] permitted users to specify the SOS semantics of their algebra, and to prove algebraic laws of their operators. MAUTO allows users to extend the syntax of the language it supports, although semantic routines must be altered by hand. As a compiler for syntactic and semantic specifications, the PAC is closely related to the CENTAUR system, and in particular to its semantic component TYPOL [13]. TYPOL provides a general framework for defining languages, interpreters, and compilers, using SOS rules. The more restrictive PAC rule format allows for the generation of simpler and more efficient code.

References

[1] D. Austry and G. Boudol. Algèbre de processus et synchronisation. *Theoretical Computer Science*, 30:91–131, 1984.

[2] J.C.M. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. Technical Report 93/05, Eindhoven University of Technology, 1994.

[3] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *Tenth Annual Symposium on Logic in Computer Science (LICS '95)*, San Diego, July 1995. IEEE Computer Society Press.

[4] B. Bloom, S. Istrail, and A. Meyer. Bisimulation can't be traced. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages (PoPL '88)*, pages 229–239, San Diego, January 1988. IEEE Computer Society Press.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P.H.J.van Eijk, C.A.Vissers, and M.Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–76. North-Holland, 1989.

[6] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process calculi, from theory to practice: Verification tools. Rapport de Recherche RR1098, INRIA, October 1989.

[7] R. Cleaveland. Analyzing concurrent systems using the Concurrency Workbench. In P.E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 129–144. Springer-Verlag, 1993.

[8] R. Cleaveland and M.C.B. Hennessy. Testing equivalence as a bisimulation equivalence. In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, pages 11–23. Springer-Verlag, 1989.

[9] R. Cleaveland and M.C.B. Hennessy. Priorities in process algebra. *Information and Computation*, 87(1/2):58–77, July/August 1990.

[10] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[11] J.C. Fernandez. Aldébaran: A tool for verification of communicating processes. Technical Report Spectre-c 14, LGI-IMAG, Grenoble, 1989.

[12] J.F. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 2(100):202–260, 1992.

[13] G. Kahn. Natural semantics. Technical Report RR601, INRIA, 1987.

[14] K.G. Larsen, J.C. Godskesen, and M. Zeeberg. TAV, tools for automatic verification, user manual. Technical Report R 89-19, Dep^t of Mathematics and Computer Science, Ålborg university, 1989.

[15] K.G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. In M.S. Paterson, editor, *Automata, Languages and Programming (ICALP '90)*, volume 443 of *Lecture Notes in Computer Science*, pages 526–539, Warwick, England, July 1990. Springer-Verlag.

[16] E. Madelaine, R. de Simone, and D. Vergamini. *ECRINS*, user manual, 1988. Technical Documentation.

[17] E. Madelaine and D. Vergamini. Finiteness conditions and structural construction of automata for all process algebras. In R. Kurshan, editor, *proceedings of Workshop on Computer Aided Verification*, New-Brunswick, June 1990. AMS-DIMACS.

[18] E. Madelaine and D. Vergamini. Specification and verification of a sliding window protocol in LOTOS. In K. R. Parker and G. A. Rose, editors, *Formal Description Techniques, IV*, volume C-2 of *IFIP Transactions*, Sydney, December 1991. North-Holland.

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

[20] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, September 1981.

[21] V. Roy and R. de Simone. Auto and autograph. In R. Kurshan, editor, *proceedings of Workshop on Computer Aided Verification*, New-Brunswick, June 1990. AMS-DIMACS.

[22] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. In B. Jonsson and J. Parrow, editors, *Proceedings CONCUR 94,* Uppsala, Sweden, volume 836 of *Lectures Notes in Computer Science*, pages 433–448. Springer-Verlag, 1994.

CAVEAT : technique and tool for Computer Aided VErification And Transformation

E. Pascal Gribomont and Didier Rossetto
Institut Montefiore, Université de Liège, Sart-Tilman B28,
B-4000 Liège (Belgium)
gribomon,rossetto@montefiore.ulg.ac.be

accepted for presentation to CAV'95 (LNCS)

Abstract. We describe caveat, a technique and a tool (under development) for the stepwise design and verification of *nearly finite-state concurrent systems (NFCS).* A concurrent system is nearly finite-state when most of its variables have a finite range (Booleans, bounded integers). The heart of caveat is a tool for verifying invariants, i.e., inductive safety properties. The underlying method is classical : formula I is an invariant for system $\mathcal{S}$ if and only if some formula $\Phi_I =_{def} \{I\}\mathcal{S}\{I\}$ is valid. If $\mathcal{S}$ is an NFCS, the formula Φ_I contains only a small set of non-boolean variables. caveat uses the *connection method* to extract from Φ_I a (small) set Ψ of *paths* (some kind of assertions) about the non-boolean variables; Φ_I is valid if and only if all paths contain *connections*, i.e., are inconsistent. For typical NFCS given with a correct invariant, the formula Φ_I is rather large (more than 100 lines) but Ψ is quite small (a dozen one-line formulas). The second part of caveat (not implemented yet) supports an incremental development method that is fairly systematic, but has proved to be flexible enough in practice.

1 Introduction

From the theoretical point of view, formal methods are a rather satisfactory answer to the problem of unreliable software. However, from the practical point of view, these methods are nearly useless without appropriate tools.

It is well-known that fully automatic tools for general program design and/or verification can not exist, so we have to be satisfied with semi-automatic tools and/or restricted classes of programs.

The most classical approach to non-automatic program verification is the invariant method. Its principle is to reduce the correctness problem ("Is this program correct w.r.t. this specification ?") to the validity problem ("Is this formula a valid formula of classical first order logic ?"). Even when automation is not considered, the invariant method has two drawbacks : it is restricted to safety properties, and it is "creative" in the sense that the validation of a safety property implies the (non-trivial) design of an adequate invariant, that is, a stronger safety property which can be proved by induction. The first problem has been satisfactorily solved by the introduction of temporal logic; the second problem is dealt with in more or less satisfactory ways, and for more or less general classes of programs. The pragmatic view (and caveat is / will be a pragmatic tool) is that formal methods become interesting when, first, testing methods *really* prove disappointing, second, reliability is *really* required, and third, programs are subtle and tricky even when they are not long. This is often the case for concurrent, distributed, reactive systems, and the problem of invariant construction seems especially important for such systems. caveat is an attempt to automate an invariant-based stepwise design/verification method introduced in [13, 14, 16].

Most earlier approaches to (semi-)automatic program verification have been based on (semi-)automatic theorem proving, for classical logic and sometimes for temporal logic. A pragmatic drawback of theorem provers is that they are mostly interactive. Even if the prover really performs the biggest part of the verification task, the user has to oversee the whole verification process, and from time to time needs to interact with it. The problem lies with the rather poor ability of proving systems to extract from a large set of mostly elementary verification steps the small subset which is outside the scope of purely automatic tools. The success of the semi-automatic theorem proving approach depends on the skill of the user [11].

A more recent approach is restricted to finite-state systems. The principle is that both the finite-state system and the specification can be modelled by a formula of propositional temporal logic, or by some kind of automaton. As a result, system verification is decidable, for instance by model checking algorithms [8, 29]. Recent improvements in the performances of computer systems, and also in the search algorithms, have led to rather powerful tools. This induced attempts to extend these techniques to some classes of infinite-state systems, but only moderately successful results have been obtained until now [20, 30]. On the contrary, some severe theoretical restrictions to this approach have been obtained [1]. Besides, when a tested finite-state program is incorrect, the verification system gives little high-level insight about how the program should be corrected; similarly, the validation of a correct program gives little insight about how the program works and why it is correct.

Another promising track comes from recent improvements in tautology checking, especially the connection method (see [5, 31]) and the concept of (ordered) binary decision diagram (see [6, 26]). It is rather natural to wonder whether these techniques remain *practically* usable outside pure propositional logic. caveat has evolved from some successful experiments in this area.

Section 2 introduces caveat with a very elementary example and discusses the main choices we have made in the strategy of invariant verification. Section 3 accounts for a more significant experiment and demonstrates the usefulness of the approach in a restricted but important class of applications. It also presents an introduction to incremental design and verification. Section 4 is a brief comparison with related works.

2 The heart of caveat : tautological reduction

2.1 Position of the problem

A formula I is an invariant of a concurrent system $\mathcal{S}$ if, in all computations, successors of states satisfying I also satisfy I. Hoare's axiom, or the liberal version of Dijkstra's weakest precondition calculus, reduces the problem of invariant verification to the purely logical problem of validity checking. With familiar notation (illustrated below), the formula to validate is $\Phi_I =_{def} (I \Rightarrow wlp[\mathcal{S}; I])$. The construction of Φ_I, when $\mathcal{S}$ and I are given, is (usually) straightforward. The validation, however, is not, since Φ_I is typically a rather large formula.

A formula J expresses a safety property of $\mathcal{S}$ with initial condition A if every state of every computation satisfies J. This holds if and only if an invariant I of $\mathcal{S}$ exists such that $(A \Rightarrow I)$ and $(I \Rightarrow J)$. The standard verification problem is to determine whether some system $\mathcal{S}$ with initial condition A satisfies the safety property (expressed by) J.

If $\mathcal{S}$ is a non-parametric finite-state system, the formulas A, J, I and Φ_I are propositional and full automation is possible. However, the construction of the invariant I is not a trivial task. Model checking is usually more effective here, since an explicit form of the invariant I is not needed; the model checker simply verifies that all accessible states satisfy J. (The set of accessible states determines the strongest invariant implied by A, often denoted $sin[A; \mathcal{S}]$.)

Pure model-checking does not apply if $\mathcal{S}$ is an infinite-state system. In this case, A, J, I and Φ_I are formulas of some first-order language (for instance, the language of number theory) and the verification problem becomes theoretically unsolvable even for a rather restricted class of programs. The invariant method still works, but is not easily turned into a reasonably efficient semi-automatic method.

There is, however, a large and interesting class of "borderline" cases, for which Φ_I is a large formula with only few occurrences of non-boolean variables. The method illustrated in the sequel seems very promising for this class. For the sake of simplicity, it is first introduced with the help of a purely finite-state example, even though it does not show its full potential in this case.

2.2 The connection method

The connection method can be viewed as an efficient implementation of the classical tableau method, used to determine whether a formula or a set of formulas has a model. The principle of the method is to reduce the initial formula into sets of literals, in such a way that the initial formula has no model if and only if each of the sets of literals contains a *connection*, i.e., a tuple of contradictory elements. In a purely propositional framework, only pairs like $\{p, \neg p\}$ are considered. In our framework, a connection is a bit more general; typical instances are $\{x > y,\ x = y,\ x < y\}$ and $\{at\ \ell_0,\ at\ \ell_1\}$, where ℓ_0 and ℓ_1 are distinct locations of the same process.

The connection method can be a powerful technique [31]; it is illustrated in the sequel of this section, first with a simplistic example.

The example is a two-process naive mutual exclusion algorithm, that has to be checked for mutual exclusion. The set of processes is $\{P, Q\}$. Each process contains three locations, identified by subscripts 0 (idle state), w (waiting state) and c (critical state), so $P = \{p_0, p_w, p_c\}$ and $Q = \{q_0, q_w, q_c\}$. There are two

Boolean variables inP and inQ. The set of transitions is

$$\begin{aligned}\mathcal{T} \ = \ \{&(p_0,\ inP := true\ ,\ p_w)\,,\quad (q_0,\ inQ := true\ ,\ q_w)\,,\\ &(p_w,\ \neg inQ \longrightarrow skip\ ,\ p_c)\,(q_w,\ \neg inP \longrightarrow skip\ ,\ q_c)\,,\\ &(p_c,\ inP := false\ ,\ p_0)\,,\quad (q_c,\ inQ := false\ ,\ q_0)\ \}\,.\end{aligned}$$

Comment. The formal notation used here and in caveat to write programs has been introduced in [14]. It is similar in spirit to many other notations based on states and transitions, e.g. the language of Action Systems introduced in [3].

There are two Boolean variables and each process has three control locations, leading to a state space of 36 possible states. The main safety property of interest is mutual exclusion, formalized as

$$J \ =_{def} \ \neg(at\ p_c \ \wedge \ at\ q_c)\,.$$

An acceptable initial condition is:

$$A \ =_{def} \ (at\ p_0 \ \wedge \ at\ q_0 \ \wedge \ \neg inP \ \wedge \ \neg inQ)\,.$$

An appropriate invariant I is

$$\begin{aligned}&(at\ p_c \ \Rightarrow \ (\neg inQ \ \vee \ at\ q_w)) \ \wedge \ (at\ q_c \ \Rightarrow \ (\neg inP \ \vee \ at\ p_w)) \ \wedge\\ &(at\ p_0 \ \equiv \ \neg inP) \ \wedge \ (at\ q_0 \ \equiv \ \neg inQ)\,.\end{aligned}$$

One sees easily that both $A \Rightarrow I$ and $I \Rightarrow J$ holds,[1] and caveat is used to check that I is really an invariant. It is sufficient to show that formula $(I \Rightarrow wlp[\tau; I])$ holds for each transition τ, and we consider here $\tau = (p_w,\ \neg inQ \longrightarrow skip,\ p_c)$. The corresponding verification formula, say Φ, is obtained by wlp-calculus:

$$\begin{aligned}&(\neg\neg inP \ \wedge \ (at\ q_0 \Rightarrow \neg inQ) \ \wedge \ (\neg inQ \Rightarrow at\ q_0)) \ \ \Rightarrow\\ &(\neg inQ \ \Rightarrow \ [(\neg inQ \ \vee \ at\ q_w) \ \wedge \ (at\ q_c \Rightarrow \neg inP) \ \wedge \ \neg\neg inP \ \ \wedge\\ &\qquad (at\ q_0 \Rightarrow \neg inQ) \ \wedge \ (\neg inQ \Rightarrow at\ q_0)])\,.\end{aligned}$$

With standard elementary techniques, Φ is reduced into two formulas, i.e.,

$$((at\ q_0 \Rightarrow \neg inQ) \ \wedge \ (\neg inQ \Rightarrow at\ q_0)) \ \ \Rightarrow \ \ (\neg inQ \ \Rightarrow \ (\neg inQ \ \vee \ at\ q_w)) \qquad (1)$$

and

$$(\neg\neg inP \ \wedge \ (at\ q_0 \Rightarrow \neg inQ) \ \wedge \ (\neg inQ \Rightarrow at\ q_0)) \ \Rightarrow \ (\neg inQ \Rightarrow (at\ q_c \Rightarrow \neg inP))\,.$$

Comment. The first one should have been

$$(\neg\neg inP \wedge (at\ q_0 \Rightarrow \neg inQ) \wedge (\neg inQ \Rightarrow at\ q_0)) \Rightarrow (\neg inQ \Rightarrow (\neg inQ \vee at\ q_w))$$

but inP occurred only once, and has therefore been replaced by its polarity **T**, leading to formula (1). This transformation and some similar ones are automated in caveat.[2]

The *subformula tableau* for formula (1) is given in Figure 1.

Each line of the subformula tableau corresponds to a node of the syntactic tree of the formula (the tree is traversed depthfirst). Let us consider formula (1). The *polarity* of the formula itself (root line a_1) is **F**, meaning that our "goal" (hopefully unreachable) is to falsify the (hopefully valid) formula a_1. This formula is an implication; it will be false if and only if its antecedent a_2 is true and

[1] Recall that each process is at exactly one location at a time; this location rule is "wired in" caveat.

[2] These simplifications are classical in resolution-based theorem provers; see e.g. [23].

	Polarity	Formula	P-type	S-type
a_1	**F**	$((at\ q_0 \Rightarrow \neg inQ) \wedge (\neg inQ \Rightarrow at\ q_0)) \Rightarrow (\neg inQ \Rightarrow (\neg inQ \vee at\ q_w))$	α	$-$
a_2	**T**	$(at\ q_0 \Rightarrow \neg inQ) \wedge (\neg inQ \Rightarrow at\ q_0)$	α	α
a_3	**T**	$at\ q_0 \Rightarrow \neg inQ$	β	α
a_4	**F**	$at\ q_0$	$-$	β
a_5	**T**	$\neg inQ$	α	β
a_6	**F**	inQ	$-$	α
a_7	**T**	$\neg inQ \Rightarrow at\ q_0$	β	α
a_8	**F**	$\neg inQ$	α	β
a_9	**T**	inQ	$-$	α
a_{10}	**T**	$at\ q_0$	$-$	β
a_{11}	**F**	$\neg inQ \Rightarrow (\neg inQ \vee at\ q_w)$	α	α
a_{12}	**T**	$\neg inQ$	α	α
a_{13}	**F**	inQ	$-$	α
a_{14}	**F**	$(\neg inQ \vee at\ q_w)$	α	α
a_{15}	**F**	$\neg inQ$	α	α
a_{16}	**T**	inQ	$-$	α
a_{17}	**F**	$at\ q_w$	$-$	α

Fig. 1. Subformula tableau for formula (1)

its consequent a_{11} is false. As a result, the polarities of a_2 and a_{11} respectively are **T** and **F**. Besides, the P-type (*primary type*) of a_1 is α, meaning that a_1 is a conjunctive line.[3] The S-type (*secondary type*) of a line if the P-type of its father, so the root line has no S-type and the atomic lines (corresponding to atomic subformulas) have no P-type.

The subformula tableau is used to construct the *verification acyclic graph*, or *VAG*. The VAG corresponding to formula (1) is given in Figure 2 (left).

Each node in a VAG is a sequence of subformula indices; we distinguish *atomic* and *non-atomic* indices, corresponding respectively to atomic and non-atomic subformulas. The index i of an atomic subformula a_i is printed in bold-face. The VAG is constructed from root to leaves according to the following rule. A node has successor(s) if it contains at least one non-atomic index i. If a_i has P-type α, there is only one successor, obtained by replacing i by j, k, where a_j and a_k are the immediate subcomponents of a_i (if a_i is a negation, there is only one subcomponent a_j). If a_i has P-type β, there are two successors, obtained

[3] Conjunctive lines are denied implications, denied disjunctions, asserted conjunctions; disjunctive lines (P-type is β) are asserted implications, asserted disjunctions and denied conjunctions. P-type is not really relevant for unary connectives, but we attribute P-type α to negations.

$$\begin{tikzcd}
 & \underline{1} \arrow[d] & \\
 & \underline{2},11 \arrow[d] & \\
 & 3,7,\underline{11} \arrow[d] & \\
 & 3,7,\underline{12},14 \arrow[d] & \\
 & 3,7,\mathbf{13},\underline{14} \arrow[d] & \\
 & 3,7,\underline{15},\mathbf{17} \arrow[d] & \\
 & \underline{3},7,\mathbf{16} \arrow[dl] \arrow[dr] & \\
\mathbf{4},\underline{7} \arrow[d] \arrow[ddrr] & & \underline{5},7 \arrow[d] \\
\underline{8} \arrow[d] & & \mathbf{6},\underline{7} \arrow[ll] \arrow[d] \\
\mathbf{9} & & \mathbf{10}
\end{tikzcd}$$

1	4, 9, 13, 16, 17	9, 13
2	4, 10, 13, 16, 17	4, 10
3	6, 9, 13, 16, 17	6, 9
4	6, 10, 13, 16, 17	6, 16

Fig. 2. Verification acyclic graph and path list for formula (1)

by replacing i by j and k, respectively. To save place, atomic indices of a node are not inherited by its successor(s). Leaves contain atomic indices only. The construction is non-deterministic, since a node can contain several non-atomic indices (in Figure 2, selected indices are underlined). The strategy of considering indices of P-type α first leads to a smaller VAG and is therefore adopted.

An example about formula (1). Three non-atomic indices 3, 7, 15 occur in node $(3, 7, 15, \mathbf{17})$; only index 15 is of P-type α, so it is selected. There is only one successor, obtained by replacing 15 by 16; besides, atomic index 17 is omitted.

It is now clear why the elementary claims introduced in paragraph 2.2 are called paths : each claim corresponds to a (maximal) path in the VAG. The last step of the connection method is to explore the VAGs and to list their paths. Each path connects the root of the VAG to a leaf and is identified in the list by the atomic indices occurring in the labels of its nodes. For instance, the first path of the VAG corresponding to formula (1) is

$$1 \to 2, 11 \to \ldots \to 3, 7, \mathbf{13}, 14 \to 3, 7, 15, \mathbf{17} \to 3, 7, \mathbf{16} \to \mathbf{4}, 7 \to 8 \to \mathbf{9}\,,$$

so this path will be identified as $\mathbf{4}, \mathbf{9}, \mathbf{13}, \mathbf{16}, \mathbf{17}$.

A formula is valid if each path of the corresponding path list contains a connection. The path list for formula (1) is given in Figure 2 (right), with a connection for each path. As a consequence, formula (1) is valid.

2.3 Concurrent construction and exploration of the VAG

An elementary but useful optimization consists in closing a path as soon as a connection is detected in it. This gives rise to shortened VAGs and path lists;

those for formula (1) are given in Figure 3.

$$\begin{tikzcd}
\underline{1} \arrow[d] \\
\underline{2}, 11 \arrow[d] \\
3, 7, \underline{11} \arrow[d] \\
3, 7, \underline{12}, 14 \arrow[d] \\
3, 7, \mathbf{13}, \underline{14} \arrow[d] \\
3, 7, \underline{15}, \mathbf{17} \arrow[d] \\
3, 7, \mathbf{16}
\end{tikzcd}$$

1	3, 7, 13, 16, 17	13, 16

Fig. 3. Optimized VAG and shortened path list for formula (1)

2.4 The non-propositional case

The connection method reduces the problem of checking the validity of formula

$$\Phi =_{def} (a \wedge b \wedge c) \Rightarrow \neg(b \Rightarrow \neg a)$$

to the problem of finding a connection within each element of the path list

$$\Psi =_{def} (\{\mathbf{T} : a\,;\ \mathbf{T} : b;\ \mathbf{T} : c\,;\ \mathbf{F} : b\}\,,\ \{\mathbf{T} : a\,;\ \mathbf{T} : b;\ \mathbf{T} : c\,;\ \mathbf{F} : a\})\,.$$

The second problem is trivial, but the reduction process itself is not. Now, let us consider a rather similar case. The formula

$$\Phi' =_{def} (x > y \wedge b \wedge c) \Rightarrow \neg(b \Rightarrow x < y)$$

is valid if each path of the list

$$\Psi' =_{def} (\{\mathbf{T}\!: x > y\,;\ \mathbf{T}\!: b\,;\ \mathbf{T}\!: c;\ \mathbf{F}\!: b\}\,,\ \{\mathbf{T}\!: x > y\,;\ \mathbf{T}\!: b\,;\ \mathbf{T}\!: c;\ \mathbf{T}\!: x < y\})$$

is connected. The first path contains a trivial, propositional connection (atom b appears with both polarities) but the connection contained in the second path is $\{\mathbf{T} : x > y\,;\ \mathbf{T} : x < y\}$, which is non-propositional.[4]

The interesting point is that half the verification work (in this example) remains within the propositional framework and hence fully automatic. Our working hypothesis is that, for many "nearly finite-state systems", most of the invariant verification work will reduce to tautology checking and nearly all paths (say 99.9 %) will be closed (a connection will be found) in an automatic way. The main advantage we seek from our approach with respect to the more classical theorem-proving approaches, is the inherent ability of the connection method to "extract" the tiny fraction of the verification work which falls outside the propositional framework. This fraction is then isolated from the rest, and dealt

[4] More precisely, analysis of the "atoms" $x > y$ and $x < y$ is needed to detect that they are contradictory.

with in a classical way; either we use a knowledge base and a theorem prover that would tell us that $x > y$ and $x < y$ are not simultaneously satisfiable, or we simply report to the user the sublist of unconnected paths. The example presented in the next section illustrates what can be achieved even without ATP (automatic theorem proving).

3 Ricart and Agrawala's algorithm

It is now usual to verify some fine-grained version of a concurrent system by first considering some coarser-grained version(s). This approach was already used in [9] and [21] and is turned into a systematic method in [14] and [16]. We extract from the latter [16, p. 43] an intermediate, medium-grained version of Ricart and Agrawala's N-process mutual exclusion algorithm, introduced in [27].

3.1 The algorithm and its invariant

The basic idea, as introduced in [27], is as follows. A node attempting to invoke mutual exclusion sends a request to all other nodes. On receipt of the request, the other nodes send an immediate reply or defer it. When all replies have been received, the access to the critical section is granted. A deferred reply is delayed until the replying node has completed its own access to the critical section.

Some notation is introduced, mostly in accordance with [27].

rcs_p :	p needs access to the resource;
$[\,]$:	internal activity, able to alter rcs only;
RCS_p :	p requests access to the resource (Request Critical Section);
OR_p^q :	p waits for a reply from q (Outstanding Reply);
RD_p^q :	p defers a reply to q (Reply Deferred);
SN_p :	p has requested access at that time (Sequence Number) (time-stamp, implementing Lamport's "Bakery algorithm");
$SN_p < SN_q$:	p takes precedence over q;
$time$:	monotonically increasing integer, models real time;
$\mathcal{P}$:	the set of nodes; $\mathcal{P} = \{p, q, r, \ldots\}$;
N :	the number of nodes; $\lvert\mathcal{P}\rvert = N$;
$\mathcal{P}_p$:	short for $\mathcal{P}\setminus\{p\}$; $\lvert\mathcal{P}_p\rvert = N - 1$;
X_p :	variable ranging over subsets of $\mathcal{P}_p$.

The transitions of $\mathcal{S}$ are given in Figure 4 (for all distinct stations p and q).

In order to switch, say, from control location p_2 to p_3, node p has to send a request to each member of $\mathcal{P}_p$. In system $\mathcal{S}$, the $N-1$ corresponding messages are already modelled by $N - 1$ distinct transitions, but each of them remains rather abstract and the communication itself is modelled by switching the Boolean variable OR_p^q from false (i.e. 0) to true (i.e. 1), just as if communications always were reliable and timeless; system $\mathcal{S}$ is not very coarse-grained, but not really fine-grained either. Also note that the $N - 1$ communications are performed in arbitrary order; $q \in X_p$ means, "node q has been issued the request from p". Intuitively, the predicate $at^q\, p_3$ means: "from the point of view of station q, the

$(p_0,\ \neg rcs_p \ \longrightarrow\ [\,],\ p_0)\,,$
$(p_0,\ rcs_p \ \longrightarrow\ (RCS_p, SN_p, time) := (1, time, time+1)\,,\ p_2)\,,$
$(p_2,\ q \in \mathcal{P}_p\backslash X_p \ \longrightarrow\ (OR_p^q, X_p) := (1, X_p\cup\{q\})\,,\ p_2)\,,$
$(p_2,\ X_p = \mathcal{P}_p \ \longrightarrow\ X_p := \emptyset\,,\ p_3)\,,$
$(p_3,\ q \in \mathcal{P}_p\backslash X_p \wedge RCS_q \wedge SN_q < SN_p \ \longrightarrow\ (RD_q^p, X_p) := (1, X_p\cup\{q\})\,,\ p_3)\,,$
$(p_3,\ q \in \mathcal{P}_p\backslash X_p \wedge [\neg RCS_q \vee SN_p < SN_q] \ \longrightarrow\ (OR_p^q, X_p) := (0, X_p\cup\{q\})\,,\ p_3)\,,$
$(p_3,\ X_p = \mathcal{P}_p \ \longrightarrow\ X_p := \emptyset\,,\ p_4)\,,$
$(p_4,\ q \in \mathcal{P}_p\backslash X_p \wedge \neg OR_p^q \ \longrightarrow\ X_p := X_p\cup\{q\}\,,\ p_4)\,,$
$(p_4,\ X_p = \mathcal{P}_p \ \longrightarrow\ X_p := \emptyset\,,\ p_5)\,,$
$(p_5,\ rcs_p \ \longrightarrow\ [\,],\ p_5)\,,$
$(p_5,\ \neg rcs_p \ \longrightarrow\ RCS_p := 0\,,\ p_6)\,,$
$(p_6,\ q \in \mathcal{P}_p\backslash X_p \wedge RD_p^q \ \longrightarrow\ (RD_p^q, OR_q^p, X_p) := (0, 0, X_p\cup\{q\})\,,\ p_6)\,,$
$(p_6,\ q \in \mathcal{P}_p\backslash X_p \wedge \neg RD_p^q \ \longrightarrow\ X_p := X_p\cup\{q\}\,,\ p_6)\,,$
$(p_6,\ X_p = \mathcal{P}_p \ \longrightarrow\ X_p := \emptyset\,,\ p_0)\,.$

Fig. 4. Abstract code of an intermediate version of R-A algorithm

place predicate $at\ p_3$ is true". Below is the formal definition of the latter and other similar predicates.

$$\begin{aligned}
at^q\, p_0 &=_{def} ((at\ p_6 \wedge q \in X_p) \vee at\ p_0)\,,\\
at^q\, p_2 &=_{def} (at\ p_2 \wedge q \in \mathcal{P}_p\backslash X_p)\,,\\
at^q\, p_3 &=_{def} ((at\ p_2 \wedge q \in X_p) \vee (at\ p_3 \wedge q \in \mathcal{P}_p\backslash X_p))\,,\\
at^q\, p_4 &=_{def} ((at\ p_3 \wedge q \in X_p) \vee (at\ p_4 \wedge q \in \mathcal{P}_p\backslash X_p))\,,\\
at^q\, p_5 &=_{def} ((at\ p_4 \wedge q \in X_p) \vee at\ p_5)\,,\\
at^q\, p_6 &=_{def} (at\ p_6 \wedge q \in \mathcal{P}_p\backslash X_p)\,.
\end{aligned}$$

The invariant I is the conjunction, for all distinct p and q, of the assertions

$$\begin{aligned}
1_p &: [X_p \subset \mathcal{P}_p \wedge (at\ p_{05} \Rightarrow X_p = \emptyset)]\,,\\
2_p &: [at\ p_{06} \equiv \neg RCS_p]\,,\\
3_{pq} &: [SN_p \neq SN_q \wedge SN_p < time \wedge ((at^q\, p_5 \wedge RCS_q) \Rightarrow SN_p < SN_q)]\,, \qquad (2)\\
4_{pq} &: [(RD_q^p \Rightarrow OR_p^q) \wedge (\neg at^q\, p_3 \equiv (OR_p^q \Rightarrow RD_q^p))]\,,\\
5_{pq} &: [(at^q\, p_6 \wedge at^p\, q_4) \vee (RD_p^q \equiv (at^p\, q_4 \wedge RCS_p \wedge SN_p < SN_q))]\,.
\end{aligned}$$

Acceptable initial conditions are, for all distinct stations p and q,

$$at\ p_0 \wedge X_p = \emptyset \wedge \neg RCS_p \wedge \neg OR_p^q \wedge \neg OR_q^p \wedge \neg RD_p^q \wedge \neg RD_q^p \wedge SN_p \neq SN_q\,.$$

3.2 What can be obtained in an automatic way ?

Our task is to use caveat in order to determine whether I really is an invariant of system $\mathcal{S}$. This system is typically "nearly propositional". Most of the variables are Boolean; SN_p and X_p are not, but $SN_p < SN_q$ and $q \in X_p$ are. Another worrying point is the parameter N (number of nodes in the network). Clearly enough, there exists a formula $I(p,q)$ — in fact, the conjunction of formulas (2) — such that the invariant really is

$$I =_{def} \forall p \forall q \neq p\, I(p,q)\,.$$

Due to symmetry, we can now *fix* two specific distinct stations p and q and decide that only the transitions explicitly written in Figure 4 ought to be checked against invariant I. Now, we have 14 transitions to consider instead of $O(N^2)$, but the size of the invariant is still $O(N^2)$. We cannot similarly reduce the triple $\{I\}\tau\{I\}$ to the triple $\{I(p,q)\}\tau\{I(p,q)\}$. However, we can reduce the triple $\{I\}\tau\{I\}$ to the family of $N*(N-1)$ triples $\{I\}\tau\{I(p',q')\}$. We can further observe that, what matters about p',q' are whether they belong to $\{p,q\}$ or not. Let us now assume that p,q,r,s are four distinct stations (and therefore, that $N \geq 4$). We can reduce the aforementioned family of triples to only seven triples,[5] listed below:

1. $\{I\}\,\tau\,\{I(p,q)\}$,
2. $\{I\}\,\tau\,\{I(q,p)\}$,
3. $\{I\}\,\tau\,\{I(p,s)\}$,
4. $\{I\}\,\tau\,\{I(r,q)\}$,
5. $\{I\}\,\tau\,\{I(s,p)\}$,
6. $\{I\}\,\tau\,\{I(q,r)\}$,
7. $\{I\}\,\tau\,\{I(r,s)\}$.

Triple 3, for instance, serves as a pattern for $N-2$ triples of the family since s stands for any node distinct from p and q.[6]

A similar reduction can be operated on the precondition. For instance, triple 1 can be replaced by

$$\{I(p,q) \wedge I(q,p) \wedge I(p,s) \wedge I(r,q) \wedge I(s,p) \wedge I(q,r) \wedge I(r,s)\}\ \tau\ \{I(p,q)\}$$

Such a triple, when fully developed, is a finite piece of text. The corresponding formula Φ is truly propositional, provided that predicates like $SN_p < SN_q$ and $q \in X_p$ are considered as atoms (we call *pseudo-atoms* these predicates; true atoms are location predicates and Boolean variables). The connection method will work, but only connections involving atoms will be detected with certainty (they contain the same atom with both polarities). Connections involving pseudo-atoms can remain undetected. For instance,

$$\{\mathbf{T} : SN_p < SN_q,\ \mathbf{F} : SN_p < SN_q\}$$

will be detected, but

$$\{\mathbf{T} : SN_p < SN_q,\ \mathbf{T} : SN_p = SN_q\}$$

will not. The simple default strategy followed by `caveat` is to suppose that when a path can not be closed using atoms only, pseudo-atoms form a connection. Such paths are collected, and the suspected connections are put into a table to be validated by the user. Even if, say, 1000 paths contain connections involving pseudo-atoms, it is possible that only a dozen distinct connections exist. So `caveat` should sort the paths according to suspected connections, in order to minimize the work performed by the user.

[5] Only four triples are needed if $N = 3$, and two triples if $N = 2$.

[6] Taking symmetry into account may allow to reduce the number of assertions and the number of transitions. The favourable case (as for this version of Ricart and Agrawala's algorithm) occurs when these numbers become true constants (independent from the size N of the network, or from any other parameter). An example of the unfavourable case is reported in [14].

3.3 What is obtained using caveat ?

The main data file for caveat contains the program to be verified. The declarations are rather standard and omitted here. The languages for transitions and assertions are slightly adapted from those used in Figure 4 and Formula (2).

Two differences exist between the real code and the abstract code in Figure 4. First, [ ] becomes `skip`, since it does not interfere with the invariant. Second, the set X_p is implemented as a Boolean array `XP`, with `XP[q]` meaning $q \in X_p$ (we suppose that `XP[p]` is true, although that does not really matter). The constant `XPempty` is such that `XPempty[q]` holds only if `q=p`; the variable `XPCard` records the number of true elements in the array `XP`. The transformation induced in the real code is straightforward.

The main data file also contains the invariant to be verified.

caveat generates and explores the VAG for each of the 14 transitions; if the invariant to be checked is $I(p,q) \wedge I(q,p)$, this takes ten minutes (SUN Sparc 10) since, in spite of the simplifications introduced above, the path list remains long.[7] However, most of the paths are closed by the system, and the set of "suspected connections" submitted to the user is short: 11 small sets, all of which being inconsistent. Here they are, in abstract notation:

1. $\{\mathbf{F} : SN_q = SN_p,\ \mathbf{F} : SN_q < SN_p,\ \mathbf{F} : SN_p < SN_q\}$,
2. $\{\mathbf{T} : SN_p < SN_q,\ \mathbf{T} : SN_q < SN_p\}$,
3. $\{\mathbf{F} : q \in X_p,\ \mathbf{T} : |X_p| = N\}$,
4. $\{\mathbf{T} : q \in \{p\}\}$,
5. $\{\mathbf{T} : SN_q < \mathit{time},\ \mathbf{T} : \mathit{time} = SN_q\}$,
6. $\{\mathbf{T} : |X_q| = 1,\ \mathbf{T} : p \in X_q\}$,
7. $\{\mathbf{F} : \mathit{time} < \mathit{time} + 1\}$,
8. $\{\mathbf{T} : |X_p| = 1,\ \mathbf{T} : q \in X_p\}$,
9. $\{\mathbf{T} : SN_q < \mathit{time},\ \mathbf{T} : \mathit{time} < SN_q\}$,
10. $\{\mathbf{T} : SN_q < \mathit{time},\ \mathbf{T} : SN_q = \mathit{time}\}$,
11. $\{\mathbf{T} : SN_q < \mathit{time},\ \mathbf{F} : SN_q < \mathit{time} + 1\}$.

In this favourable case, caveat succeeds in isolating exactly the (tiny) non-propositional part of the verification work; in order to understand the program, the user has to know that all the eleven small lists of formulas given above are inconsistent.[8]

3.4 Limitations of caveat

Within the restricted, but important subclass of programs caveat is intended to validate, two worrying limitations have been found. First, the gap of running time between the short version of the invariant, i.e.

$$I(p,q) \wedge I(q,p)$$

and the full version, i.e.

[7] Nearly twenty hours are needed for the full version of the invariant, i.e.
$$I(p,q) \wedge I(q,p) \wedge I(p,s) \wedge I(r,q) \wedge I(s,p) \wedge I(q,r) \wedge I(r,s)\,.$$

[8] Observe that, although the invariant is symmetric w.r.t. p and q, the code is not, and neither is the connection set.

$$I(p,q) \wedge I(q,p) \wedge I(p,s) \wedge I(r,q) \wedge I(s,p) \wedge I(q,r) \wedge I(r,s)\,,$$

is clearly not acceptable (ratio is worse than 1 to 100), especially since the last five assertion groups of the full version are mostly trivial. This limitation prevents us for now to consider larger, more realistic systems. Techniques for decomposing invariants are currently investigated.

Second, caveat is not efficient for parametric systems whose parameter is not the number of processes. An example is Stenning's "sliding window" protocol, where the parameter is the size of the window. The problem is that quantification elimination is more difficult in this case, and leads to longer propositional formulas.

3.5 A necessary extension

caveat is inspired by the classical idea that the best way to validate (the safety part of) the specifications of a concurrent system is to provide an appropriate invariant. However, as many designers are already reluctant to write specifications in a formal way, they are even less likely to be willing to also provide the invariant. Indeed, although the invariant is usually not more complex than the program code, it is more complex than the specification and not obvious to derive. The conclusion is that the construction of the invariant itself should be automated as much as possible.

The point of view adopted in [14, 16] is to view the system under study, say $\mathcal{S}_n$, and its invariant I_n, as the last pair of a sequence $((\mathcal{S}_k, I_k) : k \leq n)$ of "specified systems". Small transformation steps lead quite systematically from one version to the next, and the initial system $\mathcal{S}_0$ is very abstract, so the construction of its invariant I_0 is usually easy. As can be seen in [16], the design/verification process is quite lengthy but *much more time was devoted in verifying the "candidate-invariants" (by hand) than in their actual construction*; this construction will be integrated in the next version of caveat. Note however that the construction process is not always amenable to automation. The exercise considered in [14] is probably a worst case in this respect. The extension to liveness and other temporal properties may be possible, using e.g. the technique reported in [15].

4 Related work

Several successful experiments have been made in combining model checking and theorem proving. In [19], an 8.2^m-bit multiplier is verified in this way. The principle is to verify the basic component of the multiplier, i.e., the 8-bit multiplier, by model-checking. Theorem proving (in temporal logic) is used to validate the recursive way in which four N-bit multipliers are combined to form a $2N$-bit multiplier. This approach takes full benefits of the now powerful implementations of model-checking algorithms, but applies to a more restricted class of programs than ours. The reason is that many parametric systems (including Ricart and Agrawala's algorithm) cannot be decomposed into non-parametric ones.

In this paper, we avoid this decomposition problem and consider the system to be verified as a whole. Simplification is performed on the verification conditions. As a result, we do not use model-checking, but tautology-checking.

Our approach is more similar to the approach reported in [25]. The system STeP uses model-checking whenever possible, and reverts to (temporal) theorem proving when model-checking fails. STeP does not rely on our incremental approach for obtaining invariants, but attempts to synthesize invariants directly from the program code. It also integrates various simplification methods, including two decision procedures for Presburger arithmetic (the first one is efficient, the second one is complete). STeP does not appear to achieve a full separation between the automatic part and the ATP-supported part, which is one of our main objectives. Indeed, in our opinion, this separation allows to reduce the ATP-part to short and elementary formulas, for which complicated ATP techniques are not really needed.

The incremental approach that is currently integrated in caveat is not the only way to transform concurrent systems, from higher-level to lower-level versions. Other approaches might be amenable to partial automation, for instance those refining atomicity with a reduction principle [2, 22], those using refinements and hierarchical design [24, 18] or phase decomposition [10, 28], and those based on property preserving abstractions [4, 12].

Symbolic model-checking and tautology-checking can be improved by using (ordered) binary decision diagrams [6]. This approach is followed in [7], and successfully applied to the verification of a simple synchronous pipeline. Besides, using Boolean automata can be more effective than using Boolean formulas, and this kind of approach is not restricted to investigating concurrent systems [17]. First experiments with (O)BDD in caveat have not been encouraging, however, since we lack an effective procedure for ordering atoms and pseudo-atoms. No experiment has been made yet in the area of digital circuits.

References

1. K.R. Apt and D.C. Kozen, Limits for Automatic Program Verification, *Inform. Process. Letters* **22** (1986) 307-309.
2. R.J. Back, A Method for Refining Atomicity in Parallel Algorithms, PARLE'89, *Lect. Notes in Comput. Sci.* **366** (1989) 199-216.
3. R.J. Back and R. Kurki-Suonio, Distributed co-operation with action systems, *ACM Trans. Programming Languages Syst.* **10** (1988) 513-554.
4. S. Bensalem et al., Property Preserving Abstractions for the Verification of Concurrent Systems, to appear in *Formal Methods in System Design* (1994).
5. W. Bibel, *Deduction – Automated Logic*, Academic Press, 1993.
6. R.E. Bryant, Graph-based algorithms for Boolean function manipulation, *IEEE Trans. on Computers* **C-35** (1986) 677-691.
7. J.R. Burch et al., Symbolic Model Checking: 10^{20} States and Beyond, *Proc. 5th. Symp. on Logic in Computer Science* (1990) 428-439.
8. E. Clarke, E. Emerson and A. Sistla, Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, *ACM Trans. Programming Languages Syst.* **8** (1986) 244-263.

9. E.W. Dijkstra and al., On-the-Fly Garbage Collection: An Exercise in Cooperation, *Comm. ACM* **21** (1978) 966-975.
10. T. Elrad and N. Francez, Decomposition of Distributed Programs into Communication-closed Layers, *Sci. Comput. Programming* **2** (1982) 155-173.
11. D.M. Goldschlag, Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover, *IEEE Trans. on Software Eng.* **16** (1990) 1005-1023.
12. S. Graf, Verification of a distributed Cache memory by using abstractions, *Lect. Notes in Comput. Sci.* **818** (1994) 207-219.
13. E.P. Gribomont, Synthesis of parallel programs invariants, TAPSOFT'85, *Lect. Notes in Comput. Sci.* **186** (1985) 325-338.
14. E.P. Gribomont, Stepwise refinement and concurrency : the finite-state case, *Sci. Comput. Programming* **14** (1990) 185-228.
15. E.P. Gribomont, Design, verification and documentation of concurrent systems, in *Proc. 4th. Refinement workshop*, J.M. Morris and R.C. Shaw (Eds), pp. 360-377, Springer-Verlag, 1991.
16. E.P. Gribomont, Concurrency without toil : a systematic method for parallel program design". *Sci. Comput. Programming* **21** (1993) 1-56.
17. N. Halbwachs and F. Maraninchi, On the symbolic analysis of combinational loops in circuits and synchronous programs, REACT Report, 1994.
18. B. Jonsson, Compositional Specification and Verification of Distributed System, *ACM Trans. Programming Languages Syst.* **16** (1994) 259-303.
19. R.P. Kurshan and L. Lamport, Verification of a Multiplier : 64 Bits and Beyond, CAV'93, *Lect. Notes in Comput. Sci.* **697** (1993) 166-179.
20. R.P. Kurshan and M. McMillan, A structural induction theorem for processes, *Proc. 8th ACM Symp. on Principles of Distributed Computing*, Edmonton (1989).
21. L. Lamport, An Assertional Correctness Proof of a Distributed Algorithm, *Sci. Comput. Programming* **2** (1983) 175-206.
22. L. Lamport and F.B. Schneider, Pretending Atomicity, DEC SRC Rep. 44, May 1989.
23. R. Letz, J. Schumann, S. Bayerl and W. Bibel, SETHEO : A High-Performance Theorem Prover, *Jl. of Automated Reasoning* **8** (1992) 183-212.
24. N.A. Lynch and M.R. Tuttle, Hierarchical Correctness Proofs for Distributed Algorithms, *Proc. 6th ACM Symp. on Principles of Distributed Computing*, New-York (1987) 137-151.
25. Z. Manna et al., STeP : the Stanford Temporal Prover (Draft), June 1994.
26. J.S. Moore, Introduction to the OBDD algorithm for the ATP Community, *Jl. of Automated Reasoning* **12** (1994) 33-45.
27. G. Ricart and A.K. Agrawala, An optimal algorithm for mutual exclusion, *Comm. ACM* **24** (1981) 9-17 (corrigendum : *Comm. ACM* **24** (1981) 578).
28. F. Stomp and W.P. de Roever, A principle for sequential phased reasoning about distributed systems, *Formal Aspects of Computing* **6** (1994) 716-737.
29. M.Y. Vardi, P. Wolper, An Automata-Theoretic Approach To Automatic Program Verification, *Proc. Symp. on Logic in Comput. Sci.*, Cambridge (1986) 322-331.
30. P. Wolper and V. Lovinfosse, Verifying Properties of large Sets of Processes with Network Invariants, CAV'89, *Lect. Notes in Comput. Sci.* **407** (1990) 68-80.
31. L. Wallen, *Automated Deduction in Nonclassical Logics*, MIT Press, 1990.

What if Model Checking Must Be Truly Symbolic*

Hardi Hungar
OFFIS, Oldenburg
hungar@informatik.uni-oldenburg.de

Orna Grumberg
The Technion, Haifa
orna@cs.technion.ac.il

Werner Damm
University Oldenburg
damm@informatik.uni-oldenburg.de

Abstract

There are many methodologies whose main concern it is to reduce the complexity of a verification problem to be ultimately able to apply model checking. Here we propose to use a model checking like procedure which operates on a small, truly symbolic description of the model. We do so by exploiting systematically the separation between the (small) control part and the (large) data part of systems which often occurs in practice. By expanding the control part, we get an intermediate description of the system which already allows our symbolic model checking procedure to produce meaningful results but which is still small enough to allow model checking to be performed.

1 Introduction

This paper is about a close marriage of two well known verification paradigms: that of *model checking* and *generation of verification conditions*. There is no need for reiterating the success story of model checking in the verification of reactive systems originating with the seminal paper by Clarke, Emerson and Sistla on CTL model checking [7]; indeed it is safe to say that the combination of (so-called) symbolic techniques [6], abstraction [8] and compositional reasoning [15, 18] have rendered this technology to a state where industrial usage is feasible.

But beyond doubt even those combined approaches are inadequate for a complete verification of the majority of designs. In particular, applications with large or complicated data parts will escape them. We will bring in the generation of verification conditions to overcome some of the limitations.

The story of generation of verification conditions dates back to Floyd's seminal paper [13] from 1967. A large body of research has been conducted over the years on sequential program verification for increasingly more complex programming language constructs [1]. More recently, parallel programming languages [2] have also been extensively investigated. However, the inherent complexity of the task and less stringent commercial need for formally verified software systems has impeded industrial applications of this technology. A few exceptions mainly come from the area of secure systems.

The arguments impeding industrial applications of software verfication do not hold if we look at systems closer to the hardware level. For such systems, the incentive to avoid errors is higher. Moreover, many of them combine data and control in a way that enables simplifying or even automating large parts of the verification.

In this paper we will show a method that avoids some of the difficulties with verification condition generation. We will demonstrate how model checking techniques may be used to reduce automatically first-order temporal logic specifications to simpler verification conditions. These conditions concern either purely sequential behavior of subsystems or first-order data properties. Our procedure is very different from what is usually called "symbolic" model checking, which operates on *codes* for the state sets of the system. Here, we represent data and data operations by first-order formulas and substitutions, similar to their respective representations in the specification logic and the system description language. We called this "truly symbolic" in contrast to the coding approach of "symbolic" model checking.

*This research has been funded in part by the MWK under No. 210.3 - 70631 - 99 - 14/93. The views and conclusions contained in this document are those of the authors.

The class of applications we aim at include processors where the data path is simply too wide to be reasonably considered finite state, or embedded control applications, where complex interfacing logic is combined with sometimes nontrivial computations on sampled data (e.g. solving differential equations numerically). These applications have in common, that there is a clear separation between the handling of *control* and *data*. I.e.:

- The pipelined execution of a RISC instruction is solely determined by the instruction type, the pipeline stage, and other state information collected in the controller, which together constitute the *control* part of the design; register contents as well as address fields etc. form the *data* part and are evaluated separately and do influence control only sparsely.
- In embedded control applications it is the control part which governs the interaction between the controller and the controlled system (determining e.g. the sampling rate, strobes, etc.); whenever sampled data are latched into the controller, it initiates the data part of the computation, causing a possibly complex but terminating evaluation.

We find the perfect match for our approach when the data part does not affect control at all. In this case, we show that specifications can be *tested* by conventional model checking on the control part of the system. If the test result is negative, not only the control part of the specification, but also the complete specification involving data is not satisfied by the system. A positive result, on the other hand, tells that the control part of the specification is true in the complete system.

Specifications (and systems) which survive this test phase may then be analyzed more thoroughly. For that, we propose a method that generates first-order verification conditions. This phase does not require a complete separation of control from data. The restriction on their interdependence is more relaxed. Therefore, this phase is applicable also to systems for which the test phase is not.

The procedure we apply is based on a first-order extension of local model checking in the style of [22], using the control information present in the system description to investigate only those first-order aspects of the model consistent with the required behavior of its control part. The first-order verification conditions to be generated appear as success conditions of the model checking procedure. A *sufficient criterion* for the generation to be performed completely automatically is that the control part only allows a bounded number of computations on the data. This criterion subsumes e.g. Wolper's data independence property [23], which forbids any computation on data. Sometimes it is possible to transform a system description which does not meet our criterion to one which does. A loop which computes on data may be replaced by a finite (first-order) representation of its effects. This generates a sequential verification condition which can be treated separately.

Our approach differs from others addressing the verification of first-order temporal logic specifications mainly by exploiting the above separation between control and data to achieve a high degree of automation of the verification process. Also, its scope of application certainly goes beyond what can be done in others.

Approaches based on abstraction [8, 14] and, to some extent, [11] try to reduce the state space to a small resp. finite one, where the proof engineer is required to find suitable abstractions for program variables. In our approach, the verifyer's main involvement is in deciding which variables to consider as control. Remaining are of course first-order and sequential verification conditions. But even these may often be discharged automatically, e.g. if each single data loop can be handled by BDD-techniques after it is extracted from the context of the rest of the system.

More similar results involving data/control separation can be found in [17] where another generalization of Wolper's data independence is pursued. Due to the different system description format used there, separation has a different meaning and thus the results are complementary to ours. However, [17] does not even attempt to cope with data computations, and does not include techniques for first-order verification condition generation.

Verification techniques in the style of [20] which underly e.g. procedures of the STEP system [19] are closer to our approach. Indeed, one could certainly integrate a variant of our generation method as one subprocedure of STEP, suited to deal with a specific class of problems.

Although our techniques and results are rather independent from the overall framework, we chose one particular for their demonstration.

Our specification logic is FO-ACTL, a first-order version of ACTL (which resembles CTL, but allows only universal path quantifiers). The programming language might be thought of as being VHDL, stripped to its semantical essence: a flat parallel composition of sequential processes, which are essentially while-programs extended by one communication construct inspired from VHDL's **wait** statement called **step**. A **step** can only be executed jointly by all processes and thus serves as a *synchronization barrier*; whenever the processes synchronize in a **step**, they exchange information through typed in- resp. outports. All local computations (between steps) work only on local variables.

A program is given as a transition system in which the transitions are annotated by the actions performed between states. Such a program stands for a (possibly infinite-state) Kripke structure, whose states represent the current position in the program and the current variable valuation. Halfway to this large Kripke structure, we have the *control-expanded program*, where only control valuations are explicitly coded into the states and operations on the data variables still annotate the transition symbolically, in the same way as in the original program. This is the structure on which our verification procedures operate.

The test whether a specification is consistent with control of the system is performed by *stripping* the control-expanded program from its data annotations (e.g. turning branches governed by data dependent predicates into nondeterministic choice). This process may introduce nonterminating loops which, if data were considered, would always terminate. In the stripped program, these loops get annotated by fairness constraints ensuring their eventual termination. The validity of a similarly stripped formula will then be evaluated using standard (i.e. propositional) model checking. The data/control separation we require in the original program guarantees that this evaluation approximates validity of the specification in the desired way.

The verification condition generation essentially collects data operations on those paths through the control-expanded model which justify the specification. Besides the sufficient criterion mentioned above which guarantees fully automatic verification condition generation, the procedure works in several other cases as well (which do not seem to have a nice characterization).

The paper is organized as follows. Having developed the programming language and its semantics including the control-expanded program and its stripped version in Section 2, Section 3 defines the logic as well as a stripping operator on formulas, reducing them to their control aspects. Section 4 develops the theory to provide the quick test of validity of a FO-ACTL formula, while the generation of verification conditions is described in Section 5.

A fully formal development of our method would require numerous definitions and constructions, which would be impossible to fit into the available space. So we appeal to the reader's intuition whenever a concept is introduced not rigorously but informally or by example.

2 Semantical Foundation

This section introduces the programming language and its semantics. We treat a toy language vaguely similar to VHDL; any other parallel programming language would serve the purpose of this paper. The main novel notion introduced is that of a *control-expanded* program, which makes the distinction between data and control aspects of a program explicit, thus providing the semantic basis of the subsequent sections.

Programs in our toy language consist of a flat parallel composition $P_1 \| \ldots \| P_r$ of sequential processes. We retain from VHDL that processes communicate over *ports*, which in our toy language almost reduce to read-only variables modelling *in*ports resp. write-only variables modelling *out*ports. In contrast to variables, updates of ports are possible only when executing a *step*-statement discussed below.

Process definitions are of the form

process <*process-declarative-part*> **begin** <*sequential-statement*> **end.**

The process declarative part of a process P defines in particular the sets of its *in*- resp. *outports* I_P resp. O_P, and V_P of *P's local variables*. We require ports and variables to be initialized and omit the index P whenever it is understood from the context. Its body is given by a so-called *sequential statement*, which is executed continuously as if enclosed in a *do forever* loop. We allow, like VHDL, standard statements such as *variable assignments*, *if-, case-,* and *while-* statements, and *sequential composition*. Given an assignment

v:=e, we will call v the *sink* of the assignment. In our toy language we have collapsed *signal assignment*s and *wait statement*s from VHDL in the *step statement* taking the form

$$\textbf{step(in } v_1, \ldots, v_m\texttt{;}\ \textbf{out } e_1, \ldots, e_n\texttt{)}\ .$$

A step statement is executed iff all processes are willing to do a step; in this case, P's inports $I_P = \{i_1, \ldots, i_m\}$ are copied into the local variables $v_1, \ldots, v_m$, while its outports $O_P = \{o_1, \ldots, o_n\}$ take values determined by expressions $e_1, \ldots, e_n$. For simplicity, we assume that "wiring" of ports is given by equality of port names, hence the collection of all ports are variables shared between all processes, which are updated only in the disciplined style provided by the *step* statement; in VHDL jargon, this restriction would correspond to using only signal assignments with *delta delay.* We also require that for each port p there is at most one process assigning a value to p.

Our language is strongly typed; for the purpose of this paper we simply assume a collection of types with typical element τ. Example types are **bool, bit, integer, real, bitvector, array**, and enumeration types. At latest at verification time we assume, that types are classified in two *mode*s, *data* and *control*, with the obvious restriction that the domain D_τ of expressions of type τ is *finite* whenever τ is of *mode control*. This classification of types induces a classification of ports and variables.

As a simple example, consider the program from Fig. 1. Depending on the value of the boolean input `op`, until the next **step** the program either computes `res:=arg*2` or - by executing a terminating loop - `res:=arg^2`. A typical choice of modes is to consider the inport `op` and the corresponding local variable `c` to be of mode *control*.

```
process small
  in op:  bool := f, arg:  nat := 0
  out res:  nat := 0
  var x,y,z:  nat := 0, c:  bool := f
  begin
    step(in c, x; out z);
    if c
      then z:= x+x
      else y:= x; z:= 0;
        while y>0
        do
          y:= y-1; z:= z+x
        od
    fi
  end
```

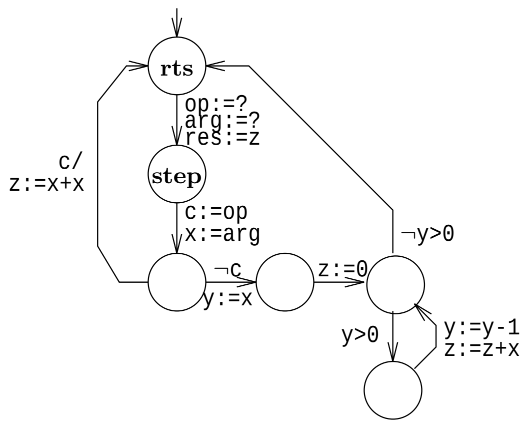


Figure 1: Example program and its flowchart

We use a variant of labeled transition systems as intermediate models for the semantics of our toy language. As a first step, a program is translated into a *flowchart* which represents the flow of control in a graphical format, see again Fig. 1 for an example. States in the flowchart correspond to positions in the program. They are labeled by **rts**, **step** or **none** to indicate whether in that position, the program is willing to engage in a step action, performing a step, or doing neither. To get the second intermediate model, the values of variables and ports of mode *control* get *expanded*: Their values will then be represented explicitly in the states. This results in a structure we call the *control-expanded program*, denoted *CEP*. It is this control-expanded program on which the verification condition generation will operate. Removing the transition labels yields the *stripped CEP* or *SCEP*, which will allow tbe propositional test of specifications. If, instead of removing the transition labels, we expand *all* variables, we get the *fully expanded program*, or *FEP*. The *FEP* is a Kripke structure. Its states include a valuation of all variables and ports, and its transitions are not labeled any more. This Kripke structure is the reference structure for defining the satisfaction relation between first-order temporal logic formulas and processes of our toy language.

For the more formal development, we fix a set of inports I, outports O, and variables V, and abbreviate $V \cup I \cup O$ by Var.

A *labeled symbolic transition system over I, O, V* assumes a classification of each element of *Var* as either being *expanded* (Var_{exp}) or *symbolic* (Var_{symb}). It is an (ordinary) labeled transition system whose state space consists of *pairs* of so called *control points* from a finite set S and valuations of the expanded variables Var_{exp} collected in the set Γ. Its transitions are labeled by an *enabling condition* on the symbolic variables and a set of assignments to symbolic variables. We use s (resp. γ) as meta variables for control points (resp. valutations of expanded variables). The *initial value* of expanded variables is given by a designated valuation γ_0, while the initial valuation of symbolic variables is given by a set of *initial assignments* A_{init}. The initial control point is designated s_0. The (standard) labeling function of states L assigns to any control point atoms of our logic in the set $\{\mathbf{rts}, \mathbf{step}, \mathbf{none}\}$. Assignments are of the form $v{:=}e$ s.t. v and all variables occurring in e are *symbolic*. All sinks of assignments occuring in one transition label must be mutually distinct. Moreover we require, that sinks of assignment are local variables, except for transitions originating from control points labeled **rts**, where also assignments to outports are allowed.

Collecting all items into a structure yields an eight-tuple $(S, \Gamma, L, R, Var_{exp}, s_0, \gamma_0, A_{init})$ as constituents of a labeled symbolic transition system M. Flowcharts, *CEP*s and *FEP*s are all instances of symbolic labeled transition systems. So the flowchart in Fig. 1 constitutes an example with $Var_{exp} = \emptyset$ (only that the initial assigmnments "`op := c := f, arg := out := x := z := y := 0`" have been omitted in the picture). In a *CEP*, the expanded variables are those of mode *control*, while in the *FEP*, the set Var_{exp} consists of all variables (i.e. it equals *Var* and Var_{symb} is empty.

We translate processes of our toy language into flowcharts by induction on the structure of processes. With each statement, we associate a canonically derived flowchart with a unique entry- and exit control-point, which are used in the inductive definition as gluing points. Since the definition is otherwise routine, we only discuss the semantics of the step statement in detail.

The flowchart of **step**(**in** $v_1, \ldots, v_m$; **out** $e_1, \ldots, e_n$) has three control points s_0, s, s_e labeled **rts**, **step**, **none**, respectively. In s_0 the process is willing to synchronize with its brother processes. If and only if this happens - as modeled in the definition of the product of the transition systems at the end of this section - it will pass to the designated control point s representing the *passage* of the synchronization barrier. The transition from s_0 to s is labeled by *random assignments* for all inports, which *guess* the value produced by some brother process during this synchronization step, as well as a collection of assignemts to its outports with the expressions occuring in the step statement. More formally,

$$tt \;/\; i_1 \;:=\; ?, \ldots, \; i_m \;:=\; ?, \; o_1 \;:=\; e_1, \ldots, \; o_n \;:=\; e_n$$

labels the transition connecting s_0 and s. The subsequent postlude transition copies the values received through inports into the local variables specified in the step statement:

$$tt \;/\; v_1 \;:=\; i_1, \ldots, \; v_m \;:=\; i_m$$

Compound statements are handled trivially by appropriate gluing and possibly introduction of fresh control points, e.g. using fresh s_0, s_e in the semantics of **if** b **then** π_0 **else** π_1 **fi** to relate s_0 with the entry point of π_0 using a transition labeled with b and the entry point of π_1 labeled $\neg b$. The exit points of π_i are linked with the new exit point.

The *flowchart semantics of process P* , $FC[\![P]\!]$, is obtained from the semantics of its *body* by relating its exit point with its entry point and adding as set of initial assignments those canonically induced from P's *process declarative part.*

The semantics of *programs* is given by defining a parallel composition operator on labeled symbolic transition systems capturing VHDL's communication and synchronization semantics. Since synchronization is only required at steps, all transitions except for those relating control-points labeled **rts** with **step** control points can be taken in *any* order, e.g. in an interleaved fashion. Transitions handling the step are taken in *lock step*, replacing random values assigned to inports by those expressions provided by the processes running in parallel. Due to space restrictions, we do not discuss this in detail; the reader might refer to [10] for a full definition of the comparable operator of VHDL.

Let us now turn to the process of *expanding* a labeled symbolic transition system M. Fig. 2 shows an expansion of our example flowgraph.

To the right, the result of expanding the variable `c` and the port `op` in the flowgraph from Fig. 1 is shown. This is the *CEP* belonging to the example program. The picture omits initial assignments and does not contain unreachable states.

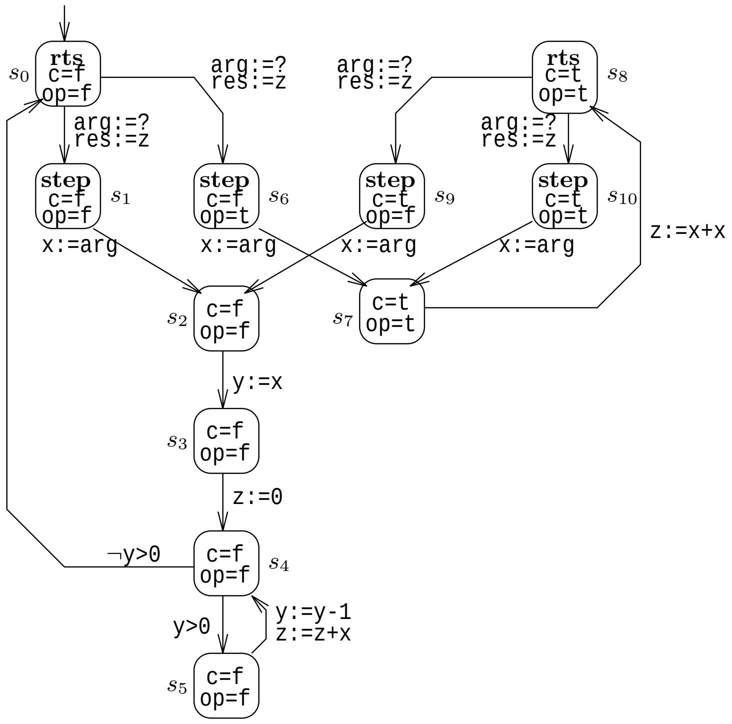


Figure 2: Example *CEP*

Each symbolic variable $v \in Var_{symb}$ can be expanded separately. The expansion of M w.r.t. v is obtained by essentially substituting each occurrence of v in transition labels by its value now represented in the valuation component of states. The only situation deserving special attention arises, whenever v occurs as the sink of an assignment $v := e$ in a transition label. In this case, the assignment is deleted from the transition label. But the query $d = e$ is added to the condition part of the transition leading to a state where v is evaluated to a value d. Expanding the variables and ports of mode *control* in the flowchart $FC[\![P]\!]$ of a program yields its *control-expanded* version, $CEP[\![P]\!]$. Expanding all variables gives the *fully expanded program* $FEP[\![P]\!]$.

By abstracting from data annotations, any labeled symbolic transition structure turns into a classical Kripke structure allowing safe model checking of properties related only to expanded variables and the synchronization atoms, provided the expanded structure is finite. The next section shows that this abstraction, called the *stripped* transition system, enriched by suitable fairness constraints, may in fact be a *precise* abstraction for such formulas under some additional assumptions. The definition of stripping is trivial: for a labeled symbolic transition system M we simply delete all transition labels, thus replacing conditional selection by nondeterminism. We are only interested in the stripped version of the control-expanded program, and will denote this structure by $SCEP[\![P]\!]$.

When the program P is understood from the context, the parameter $[\![P]\!]$ will be omitted and we will simply write *FEP* or *CEP*. For ease of exposition we will assume in the following that all control variables are of type **bool** (instead of an arbitrary finite type).

3 The Logic

The logic FO-ACTL (first-order ACTL) is a *branching-time first-order* temporal logic. It is similar to the propositional temporal logic ACTL (universal CTL) except that it is defined over first-order atomic formulas. Following Emerson [12], a formula in the logic is interpreted over a Kripke structure and an interpretation which is fixed for all states of the Kripke structure.

Similarly to propositional ACTL, FO-ACTL provides only universal path quantifiers. To avoid the invocation of existential path quantifiers via negations, the logic is given in a *positive normal form* in which negations are applied only to atomic formulas. Since only universal path quantifiers are allowed, path quantifiers are left implicit in the syntax. Thus, $\phi\,\mathbf{U}\,\psi$ represents the ACTL formula $\mathbf{A}(\phi\,\mathbf{U}\,\psi)$ and similarly for any other temporal operator.

Definition 1 (FO-ACTL) *Let $\mathcal{L}$ be a first-order language over some signature and let Var be a set of (typed) variables. A formula in our logic is defined inductively as follows:*

1. *Every first-order formula of $\mathcal{L}$ over Var is an atomic formula.*
2. **rts**, **step**, **none** *are atomic formulas.*
3. *If p is an atomic formula then $\neg p$ is a formula.*
4. *If ϕ and ψ are formulas and $x \in \mathit{Var}$ then $\phi \vee \psi$, $\phi \wedge \psi$, $\exists x.\phi$, $\forall x.\phi$ are formulas.*
5. *If ϕ and ψ are formulas then $\mathbf{X}\,\phi$, $\phi\,\mathbf{U}\,\psi$ and $\phi\,\mathbf{W}\,\psi$ are formulas.*

The operator **U** is the usual *until*. I.e. $\phi\,\mathbf{U}\,\psi$ requires to eventually reach a state satisfying ψ and not violate ϕ before that event. **W** is *weak until* and allows the formula to the left to hold forever.

We use the following abbreviations:

$$\mathbf{F}\,\phi = tt\,\mathbf{U}\,\phi \quad \text{and} \quad \mathbf{G}\,\phi = \phi\,\mathbf{W}\,ff.$$

Let Int be an interpretation for $\mathcal{L}$ over domains D_τ for occurring type τ. The semantics of FO-ACTL formulas is defined with respect to an interpretation Int and a Kripke structure K. For simplicity we denote $T = S \times \Gamma$ and omit the empty set of assignments A_{init} in K. For $t = (s, \gamma)$, with a slight abuse of notation, we use $L(t)$ and $t(v)$ instead of $L(s)$ and $\gamma(v)$. A Kripke structure has now the form $K = (T, L, R, \mathit{Var}, t_0)$. A path in a Kripke structure K is a sequence, $\pi = w_0, w_1, \ldots$, such that for every i, $(w_i, w_{i+1}) \in R$.

$K, \mathit{Int}, t \models \phi$ denotes that the formula ϕ is true in state t of structure K under interpretation Int. If clear from the context, Int is omitted.

We sometimes want to restrict our attention to *fair paths* only, based on some given fairness criterion F that characterizes fair paths. We use $K, t \models_F \phi$ to denote that ϕ holds at t in K with respect to the fair paths only. In particular, the relation $\models_F$ for the temporal operators **X** , **U** , and **W** is defined with respect to every *fair* path rather than with respect to every path.

In the sequel, we will only consider specifications that do not contain the next-time operator. This operator will be used, however, in the tableau construction in Section 5.

Stripped formulas Given a specification written in FO-ACTL, we extract its propositional part by applying the *strip* operator. The strip operator eliminates all first-order components of the formula, thus results in a propositional ACTL formula. Data-dependent parts of the formula are replaced by *tt*, so the stripped formula will be more often true.

Definition 2 (stripped formula) *Let $\mathit{Var}_c \subseteq \mathit{Var}$ be a set of boolean (control) variables and let ϕ be a FO-ACTL formula, strip(ϕ) with respect to Var_c is defined as follows.*

1. $strip(p(v_1,\ldots,v_k)) = p(v_1,\ldots,v_k)$*;* $strip(\neg(p(v_1,\ldots,v_k))) = \neg(p(v_1,\ldots,v_k))$ *if* $v_1,\ldots,v_k \in \mathit{Var}_c$.
2. $strip(p(v_1,\ldots,v_k)) = strip(\neg p(v_1,\ldots,v_k)) = tt$ *if some variable* $v_i \notin \mathit{Var}_c$.
3. $strip(l) = l$*;* $strip(\neg l) = \neg l$ *for* $l \in \{\mathbf{rts}, \mathbf{step}, \mathbf{none}\}$.
4. $strip(\phi \vee \psi) = strip(\phi) \vee strip(\psi)$.
5. $strip(\phi \wedge \psi) = strip(\phi) \wedge strip(\psi)$.
6. $strip(\exists x.\phi) = strip(\phi[tt/x]) \vee strip(\phi[ff/x])$ *for* $x \in \mathit{Var}_c$.
7. $strip(\forall x.\phi) = strip(\phi[tt/x]) \wedge strip(\phi[ff/x])$ *for* $x \in \mathit{Var}_c$.

8. *$strip(\exists x.\phi) = strip(\forall x.\phi) = strip(\phi)$, for $x \notin Var_c$.*

9. *$strip(\phi \, \mathbf{U} \, \psi) = strip(\phi) \, \mathbf{U} \, strip(\psi)$.*

10. *$strip(\phi \, \mathbf{W} \, \psi) = strip(\phi) \, \mathbf{W} \, strip(\psi)$.*

Lemma 3 *If ϕ is a FO-ACTL formula then $strip(\phi)$ with respect to Var_c is a propositional ACTL formula over Var_c.*

Example: Consider two specifications for the example in Figure 1, where `op` is a control variable and `arg`, `res` and `x` are data variables. Let $\phi_1 = (\mathbf{F}\,\mathbf{rts})\,\mathbf{W}\,(\mathbf{step} \wedge \neg \texttt{op})$, then $strip(\phi_1) = \phi_1$.
Consider now the formula $\phi_2 = \forall x.\mathbf{G}\,((\mathbf{step} \wedge \texttt{arg} = x \wedge \texttt{op}) \rightarrow \mathbf{F}\,(\mathbf{step} \wedge \texttt{res} = x * 2))$. Then, $strip(\phi_2) = \mathbf{G}\,((\mathbf{step} \wedge tt \wedge \texttt{op}) \rightarrow \mathbf{F}\,(\mathbf{step} \wedge tt))$ which is equivalent to $\mathbf{G}\,((\mathbf{step} \wedge \texttt{op}) \rightarrow \mathbf{F}\,\mathbf{step})$.

4 The Propositional Verification Methodology

In this section, we restrict our concern to programs for which there is a clear separation between data and control. In particular, data cannot influence control variables. For such programs, their verification with respect to first-order temporal specification can take advantage of a preliminary phase in which propositional temporal specifications are proved for the control part of the program.

More precisely, let a *data-dependent condition* be a boolean condition that contains (also) data variables. A program has the *separation property* if no control variable gets assigned a value depending on data, and neither assignments to control variables nor step statements occur in the scope of a data-dependent condition.

The separation property ensures that data do not directly influence control values. But there is a more subtle way in which the validity of a temporal formula not referring to data may be affected: by the termination behavior of data-controlled loops it might be determined whether observable changes to control might happen or not. This influence we eliminate by the assuming - which at least in a hardware context is not unreasonable - that data-controlled loops always terminate. Formally, the assumption enters in the form of fairness constraints.

In more detail, the situation is as follows. Let P have the separation property. Since the transition labels in $CEP[\![P]\!]$ contain no control variables, stripping the *CEP* from its transition labels eliminates data-dependent conditions only. But the separation property implies that also no control variable changes its value along a transition if the condition labeling it is different from tt. Thus, the stripping does not introduce changes of control which did not happen before. And if the stripping results in an infinite loop that did not occur before, then this must be a *data loop* in which only data variables may change their value. For all these loops, we assume termination and check the stripped formula in the stripped *CEP* based on this assumption (To complete the verification, we must of course later show that in the fully expanded Kripke structure $FEP[\![P]\!]$ all data loops are indeed terminating). As a result, control properties are not affected by stripping the *CEP*.

For the verification of formulas which also depend on data, we can conclude the following. If the check of the stripped formula in *SCEP* (the stripped *CEP*) returns tt, then we can conclude that the stripped formula is true of $FEP[\![P]\!]$. But if the check returns $f\!f$, then we know that the *original* formula is $f\!f$ on $FEP[\![P]\!]$. As mentioned before, we consider the latter as a significant contribution that enables model checking together with termination proofs to debug any first-order temporal specification.

Our methodology is summarized in the following theorem, where F denotes termination of all data loops. We refer to the well-known notion of a *generalized* Kripke structure [12] to explain the meaning of validity of a temporal logic formula under fairness assumptions.

Theorem 4 *If $FEP[\![P]\!] \models F$ then*

1. *$SCEP[\![P]\!] \models_F strip(\phi) \implies FEP[\![P]\!] \models strip(\phi)$, and*

2. *$SCEP[\![P]\!] \not\models_F strip(\phi) \implies FEP[\![P]\!] \not\models \phi$.*

The proof of the theorem could not be included in this paper due to space limitations. The main technical result in the proof states that, if all data loops terminate, then $\mathit{SCEP}[\![P]\!]$ and $\mathit{FEP}[\![P]\!]$ are fair stuttering bisimilar and therefore agree on all propositional ACTL formulas (with no next-time operator).

Example: Consider again the example of Figure 1. Once we verify that the while loop always terminates, we can use $\mathit{SCEP}[\![P]\!]$ to verify propositional ACTL formulas and to refute FO-ACTL formulas. $\mathit{SCEP}[\![P]\!]$ is obtained from the *CEP* of Figure 2 in the appendix by eliminating all transition labels.

For instance, since $\mathit{SCEP}[\![P]\!]$ satisfies $\phi_1 = (\mathbf{F}\,\mathbf{rts})\,\mathbf{W}\,(\mathbf{step} \wedge \neg \mathtt{op})$ under loop termination assumption, we can conclude that this formula is true also in $\mathit{FEP}[\![P]\!]$ (recall that $\mathit{strip}(\phi_1) = \phi_1$).
Consider the FO-ACTL formula $\phi_2 = \forall x.\mathbf{G}\,((\mathbf{step} \wedge \mathtt{arg} = x \wedge \mathtt{op}) \rightarrow \mathbf{F}\,(\mathbf{step} \wedge \mathtt{res} = x * 2))$. Since the formula $\mathit{strip}(\phi_2) = \mathbf{G}\,((\mathbf{step} \wedge \mathtt{op}) \rightarrow \mathbf{F}\,\mathbf{step})$ is true of $\mathit{SCEP}[\![P]\!]$ we can conclude that $\mathit{strip}(\phi_2)$ is true also in $\mathit{FEP}[\![P]\!]$. Note that we cannot conclude that ϕ_2 is true in $\mathit{FEP}[\![P]\!]$. For that we must use the method developed in Section 5.
Consider also the FO-ACTL formula $\phi_3 = \mathbf{G}\,\mathbf{F}\,(\mathbf{step} \wedge \neg \mathtt{op} \wedge y = 0 \wedge z = \mathtt{arg}^2)$. Then $\mathit{strip}(\phi_3) = \mathbf{G}\,\mathbf{F}\,(\mathbf{step} \wedge \neg \mathtt{op} \wedge \mathit{tt})$. Recall that our formulas have an implicit universal path quantifier accompanied with any temporal operator. Thus, $\mathit{strip}(\phi_3)$ means that for every path, $(\mathbf{step} \wedge \neg \mathtt{op})$ is true for infinitely many states on that path. This does not hold, for instance, on the path $s_0, s_6, s_7, s_8, s_{10}, s_7, \ldots$ in Fig. 2. Hence, $\mathit{strip}(\phi_3)$ is false in $\mathit{SCEP}[\![P]\!]$ and as a result we can conclude that ϕ_3 is false in $\mathit{FEP}[\![P]\!]$.

5 Verification Condition Generation

To handle specifications including data, we propose to verify the temporal aspects relative to first-order verification conditions. As we did before, we start by expanding control variables to get the *CEP*. The key idea then is to use an approach close to what is usually called *local model checking*. Local model checking searches for a sufficient reason for the specification to be satisfied. It has the advantage over iterative model checking that it may turn out that some parts of the program behavior are irrelevant to the specification considered. Here, it may be the case that control information alone can tell that a loop, which can not be handled in general, does not affect validity of the specification. In such and further cases, local model checking will be successful without expanding every data domain. This is essential if some of the data domains are infinite or too large or complex to be completely expanded.

A tableau system for FO-ACTL Local model checking consists in constructing a tableau proving the validity of the formula in question for the start state - or, in the negative case showing the nonexistence of such a tableau. A tableau is essentially a proof tree. Ignoring data for the moment, the root of the tableau is the sequent $s_0 \vdash \phi$, where s_0 is the initial state of the system and ϕ is the formula in question. The successors of each node provide sufficient reason for the validity of that node. Rules are available for each form of node which fix possible successor sets. If the expansion of a tableau is stopped at some point, a success criterion tells whether the tableau constitutes a complete proof for the sequent at its root.

Our tableau system serves as the basic formalism to derive first-order temporal properties involving data, by providing a well-defined method to generate pure first-order conditions from the system and a specification. Below we present our rules for tableaux construction. They differ in two respects from the usual rules for CTL. One is notational: Usually, the different possibilities for proving a sequent (i.e. the different possibilities for successor sets of one vertex) are given in different rules which could be applied alternatively. Our format comprises them in one schema, the alternatives being separated by " | ". Elements in one successor set are separated by " , ". But the rules also reflect that we deal with a first-order model: The state component of a sequent consists of a control state (an element of S) *and* a condition on a variable valuation, given in the form of a first-order formula.

Or Rule

$$\frac{s, p \vdash \phi \vee \psi}{s, p \vdash \phi \mid s, p \vdash \psi}$$

And Rule

$$\frac{s, p \vdash \phi \wedge \psi}{s, p \vdash \phi \,,\, s, p \vdash \psi}$$

Exists Rule

$$\frac{s, p \vdash \exists x.\phi}{s, p \vdash \phi[y/x]}\ ,\ y \notin \mathit{free}(p)$$

Forall Rule

$$\frac{s, p \vdash \forall x.\phi}{s, p \vdash \phi[y/x]}\ ,\ y \notin \mathit{free}(p)$$

Until Rule

$$\frac{s, p \vdash \phi\,\mathbf{U}\,\psi}{s, p \vdash \psi \vee (\phi \wedge \mathbf{X}\,(\phi\,\mathbf{U}\,\psi))}$$

Unless Rule

$$\frac{s, p \vdash \phi\,\mathbf{W}\,\psi}{s, p \vdash \psi \vee (\phi \wedge \mathbf{X}\,(\phi\,\mathbf{W}\,\psi))}$$

Next Rule

$$\frac{s, p \vdash \mathbf{X}\,\phi}{s_1, p_1 \vdash \phi\ ,\ \ldots,\ s_n, p_n \vdash \phi}\ ,\ s \longrightarrow \{s_1, \ldots, s_n\}$$

where $p \wedge c_i \rightarrow \mathrm{subst}(p_i, A_i)$ for $s \overset{c_i/A_i}{\longrightarrow} s_i,\ i = 1, \ldots, n$.

Case Split Rule

$$\frac{s, p \vdash \phi}{s, p_1 \vdash \phi\ ,\ s, p_2 \vdash \phi}\ ,\ p \rightarrow p_1 \vee p_2$$

$\mathrm{subst}(p_i, A_i)$ in the rule dealing with "$\mathbf{X}$ " means the parallel substitution of e for v in p_i for each assignment $v := e \in A_i$. The rules above are chosen to be as simple rules as possible. For convenient application, usually several of them would be combined. For instance, a more useful rule to deal with "$\vee$" like

$$\frac{s, p \vdash \phi \vee \psi}{s, p_1 \vdash \phi \mid s, p_2 \vdash \psi}\ ,\ p \rightarrow p_1 \vee p_2$$

is derived from our rule set by combining the Case Split and the Or Rule.

The reader may have noted that $\exists$ and $\forall$ as well as $\mathbf{U}$ and $\mathbf{W}$ are treated in the same way by the rules. The difference between the operators is captured by (global) success conditions, see below.

A *tableau* is a finite tree of sequents $s, p \vdash \phi$ where the set of successors of each internal node are instances of one of the alternative successor sets according to the rules. The nodes on the path from the root of the tableau to a given node are called its predecessors.

To each tableau we associate a first-order formula which specifies whether the tableau is *successful*. This *success formula* is computed bottom-up. The success formulas of leaves are as follows.

- $p \rightarrow \bigvee_i p_i$ for leaves $s, p \vdash \phi\,\mathbf{W}\,\psi$, where $p_i,\ i = 1, \ldots, n$ are the first-order conditions in predecessors of the form $s, p' \vdash \phi\,\mathbf{W}\,\psi$,
- $p \rightarrow r_s$ for leaves $s, p \vdash r$ with a first-order formula r (see below for the computation of r_s), and
- $p \rightarrow \mathit{ff}$ for other leaves $s, p \vdash \phi$.

For a first-order formula r and a state s, replace the atoms **rts** and **step** as well as control variables in r by their truth values in the state s to obtain the formula r_s.

At inner nodes, the success formula is computed by conjuncting the success formulas of the subtableaux following it. If case split is applied, the appropriate implication is added. At quantifier steps, the respective quantifier is applied.

A tableau is *successful* in a data domain, if its success formula is valid in the domain. A sequent is *provable* if it has a successful tableau. A formula ϕ is provable if $s_0, \mathit{tt} \vdash \phi$ is a provable sequent.

Theorem 5 (Soundness) *The tableau system is sound. I.e., if a sequent $s, p \vdash \phi$ is provable, then all copies of s in the full model where the data variable valuation satisfies p have property ϕ. If a formula is provable, it is valid in the system.*

The tableau system does not provide us with a decision method, though. One reason is that of course the validity of success formulas can not be decided in general. Another one concerns the treatment of the

$\mathbf{U}$ operator. To achieve a stronger form of completeness than we do, we would have to allow a successful recurrence of $\mathbf{U}$-formulas in the style of the recurrence condition for minimal fixpoints of [4, 3]. This, however, would introduce a new dimension of undecidability, because successful $\mathbf{U}$-recurrence would have to involve a well-foundedness condition. We do not strive for completeness in general, though. We do achieve completeness and even decidability relative to first-order questions for a certain class of interesting cases, as indicated by the results below.

The construction of a generic tableau Roughly spoken, systematic tableau construction will provide a proof or a refutation (up to first-order verification conditions) if all "nontrivial cycles" are "broken by control". This is a property of system and formula combined. A "nontrivial cycle" occurs when a data variable value at one position in the program may result by applying a function other than identity to the value the same variable had at that same location at an earlier stage of the execution of a program.[1] Such cycles may cause unbounded expansion of the tableau during construction. A cycle like that one is "broken by control", if one can tell from control information that there is a bound on the number of iterations through this cycle which are necessary to decide the validity of the formula. As an extreme case, the path through the program which introduces the cyclic dependency might not be executable at all without violating an essential control condition in the formula, giving zero as a bound.

A formalization of this informal concept will take several steps. First of these is the construction of a *generic tableau* which comprises in some sense all tableaux which can be constructed for a given formula ϕ. It represents, essentially, the control part of each first-order tableau. Thus, it can later be used to detect cycles broken by control. The rules for the generic tableau are derived from the above rules essentially by removing all first-order aspects.

$$\frac{s \vdash \phi \vee \psi}{s \vdash \phi \mid s \vdash \psi} \qquad \frac{s \vdash \phi \wedge \psi}{s \vdash \phi \,,\, s \vdash \psi}$$

$$\frac{s \vdash \exists x.\phi}{s \vdash \phi[\mathit{ff}/x] \mid s \vdash \phi[\mathit{tt}/x]}\ ,\ x \in V_c \qquad \frac{s \vdash \forall x.\phi}{s \vdash \phi[\mathit{ff}/x] \,,\, s \vdash \phi[\mathit{tt}/x]}\ ,\ x \in V_c \qquad \frac{s \vdash \exists/\forall x.\phi}{s \vdash \phi}\ ,\ x \notin V_c$$

$$\frac{s \vdash \phi\,\mathbf{U}\,/\,\mathbf{W}\,\psi}{s \vdash \psi \mid s \vdash \phi \,,\, s \vdash \mathbf{X}\,(\phi\,\mathbf{U}\,/\,\mathbf{W}\,\psi)} \qquad \frac{s \vdash \mathbf{X}\,\phi}{s_1 \vdash \phi \,,\ldots,\, s_n \vdash \phi}\ ,\ s \longrightarrow \{s_1,\ldots,s_n\}$$

With these rules, we construct the generic tableau for a given *CEP* and a temporal formula by the following deterministic procedure. Starting with $s_0 \vdash \phi$, the appropriate rule gets applied. But different from the first-order tableau, no choice is made between alternative successors. Instead, all alternatives are pursued. The expansion of the generic tableau stops if the temporal formula is reduced to a pure first-order formula (*first-order leaf*) or if a node recurs, i.e. at a node which has a predecessor labeled by the same sequent (*recurring leaf* resp. *recurrence node*). Since there is a finite number of states and subformulas, the process is bound to terminate.

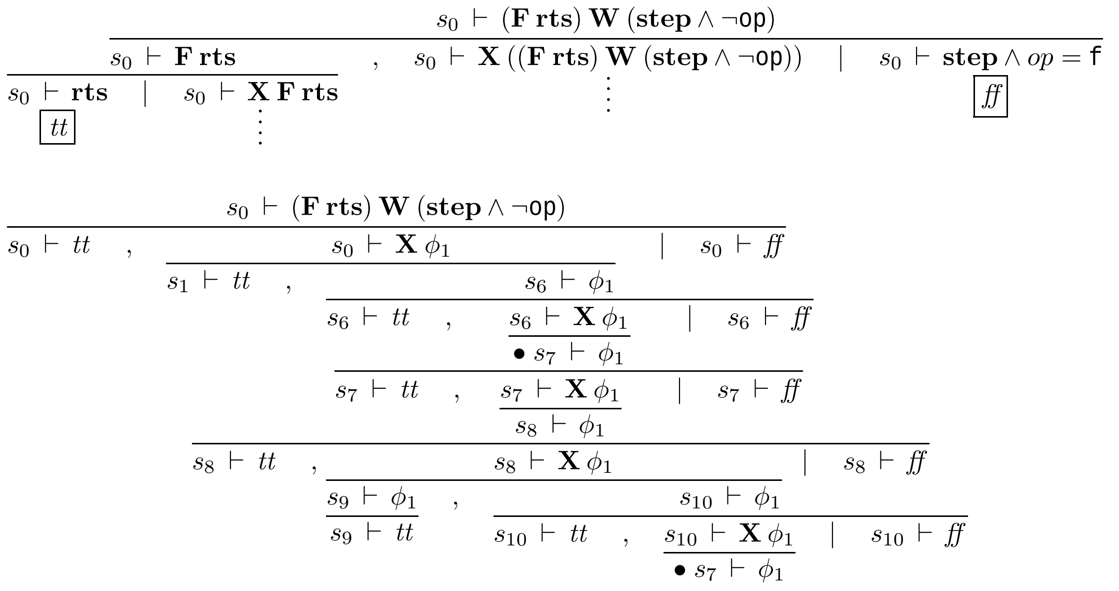
Next, irrelevant branches are removed. This starts at non-recurring leaves. $\mathbf{X}$-leaves can be replaced by $s \vdash \mathit{tt}$. Also, some of the first-order leaves $s \vdash p$ can be evaluated. To do this, first the formula p_s is constructed. Then, the control information present in p_s is used to determine whether by propositional reasoning and trivial first-order identities like $(\exists x.\mathit{ff}) \rightarrow \mathit{ff}$ the formula can be reduced to tt or ff.

Then, tt and ff are propagated upwards in the tableau. A successor set gets replaced by ff (resp. tt) if one (resp. all) of its components becomes ff (resp. tt). If one of the alternative successor sets of a node becomes tt, the node itself is replaced by tt, and if all alternatives become ff, it is replaced by ff. The resulting, reduced structure is called the *generic tableau* for ϕ.

[1]More general, the value need not be computed from the previous value alone, but also other variables might influence the result.

Observation 6 *For every system and formula, there is one (unique) generic tableau.*

Let us return to our example program from Fig. 1, and take $\phi_1 = (\mathbf{F}\,\mathbf{rts})\,\mathbf{W}\,(\mathbf{step} \wedge \neg\texttt{op})$ as a specification. Fig. 3 shows the first steps of the construction of the generic tableau (indicating the evaluation of first-order leaves in boxes) and the final result, after removing irrelevant branches. The generic tableau contains one pair of a recurring leaf and recurrence node. These are marked with "•". Note that other recurrences (e.g. of sequences involving **F rts**) occurring during its construction have been eliminated by the reduction process.



$s_0 \vdash (\mathbf{F}\,\mathbf{rts})\,\mathbf{W}\,(\mathbf{step} \wedge \neg\texttt{op})$
$s_0 \vdash \mathbf{F}\,\mathbf{rts}$, $s_0 \vdash \mathbf{X}\,((\mathbf{F}\,\mathbf{rts})\,\mathbf{W}\,(\mathbf{step} \wedge \neg\texttt{op}))$ | $s_0 \vdash \mathbf{step} \wedge op = \texttt{f}$
$s_0 \vdash \mathbf{rts}$ | $s_0 \vdash \mathbf{X}\,\mathbf{F}\,\mathbf{rts}$
$\boxed{tt}$
$\boxed{ff}$
$s_0 \vdash (\mathbf{F}\,\mathbf{rts})\,\mathbf{W}\,(\mathbf{step} \wedge \neg\texttt{op})$
$s_0 \vdash tt$, $s_0 \vdash \mathbf{X}\,\phi_1$ | $s_0 \vdash ff$
$s_1 \vdash tt$, $s_6 \vdash \phi_1$
$s_6 \vdash tt$, $s_6 \vdash \mathbf{X}\,\phi_1$ | $s_6 \vdash ff$
$\bullet\, s_7 \vdash \phi_1$
$s_7 \vdash tt$, $s_7 \vdash \mathbf{X}\,\phi_1$ | $s_7 \vdash ff$
$s_8 \vdash \phi_1$
$s_8 \vdash tt$, $s_8 \vdash \mathbf{X}\,\phi_1$ | $s_8 \vdash ff$
$s_9 \vdash \phi_1$, $s_{10} \vdash \phi_1$
$s_9 \vdash tt$
$s_{10} \vdash tt$, $s_{10} \vdash \mathbf{X}\,\phi_1$ | $s_{10} \vdash ff$
$\bullet\, s_7 \vdash \phi_1$

Figure 3: Constructing the generic tableau

If the program has the separation property, the construction of the generic tableau can profit from the results of the test computation according to Theorem 4. They enable early detection of irrelevant or always successful branches.

Instantiating the generic tableau The relevance of the generic tableau construction relies on the fact that every successful tableau can be put in a form that it is an *instance* of the generic one. Instances are built by adding first-order formulas to the state components of sequents and perhaps by unfolding the generic tableau at its recurring leaves.

To be more precise, a first-order tableau T with root $s, p \vdash \phi$ is an instance of a subtableau (to get an inductive condition) of the generic tableau starting at node n if:

- n has the form $s \vdash \phi$, and
- if n is not a leaf, the rule applied to n is matched by an appropriate rule combination in T, and subtableaux starting at end nodes of the rule combination are instances of the corresponding end nodes of the generic rule ("Matching" requires choosing among the alternatives present in the generic tableau, and we allow the matching combination to contain applications of Case Split). And
- if n is a recurring leaf and T is not a leaf itself, it is an instance of the subtableau starting at the recurrence node. And
- if $n = s \vdash tt$ where this is the result of a reduction, T is an instance of the subtableau reduced to n.

The restrictions imposed on a tableau to be an instance of the generic tableau are rather modest. They require complete case distinction for control values, and that branches which are always successful (and have been reduced to tt in the generic tableau construction) have to be chosen. So we have:

Observation 7 *If a formula is provable at all, it is also proved by an instance of its generic tableau.*

Now we give a procedure which tries systematically to construct an instance of the generic tableau. It will not terminate in general. The procedure operates on the generic tableau. It computes a first-order formula, called *instantiating formula*, for each node of the generic tableau. These formulas can subsequently be used to generate an instance.

First-order leaves $s \vdash p$ are instantiated with p_s. Recurring **U**-leaves are initialized with *ff* and recurring **W**-leaves with *tt*. For inner nodes, the instantiating formulas are computed from those for their successor nodes. Disjunction is used for $\vee$, conjunction for $\wedge$, existential quantification for $\exists$, and universal quantification for $\forall$. For a **X**-node with successor formulas $p_1, \ldots, p_n$, the conjunction over $c_i \rightarrow \mathrm{subst}(p_i, A_i)$ is taken. Inner **U**- and **W**-nodes get instantiated with their successor formulas. But if such a node is a recurrence node, the process of computing the instantiating formula is iterated after instantiating the corresponding recurring leaves with the formula computed for the recurrence node. The iteration stops if a fixpoint is reached for a recurrence node. Propositional and control reasoning is applied to detect a fixpoint.

Although this process does not literally generate an instance of the generic tableau, it performs all necessary computations. Due to lack of space we can not show the formal construction of the instance. One point to note is that the iteration steps at **U**-nodes during the computation process correspond to unfoldings in the construction. Most importantly, we can prove that the result of a terminating instantiation provides us with a first-order characterization of the correctness of the program.

Theorem 8 *If the instantiation process terminates for a specification ϕ, the success formula of the generated instance characterizes validity of ϕ. I.e. the success formula is valid in a data domain iff under this interpretation the specification ϕ is valid (for the system).*

In our example in Fig. 3, data do not matter at all. A successful tableau can be derived directly from the generic tableau. One only has to restore branches which have been reduced to the form $s \vdash tt$. As an example for a nontrivial, but still terminating instantiation process the reader may consider the specification $\forall x.\mathbf{G}\,((\mathbf{step} \wedge \texttt{arg} = x \wedge \texttt{op} = \texttt{t}) \rightarrow \mathbf{F}\,(\mathbf{step} \wedge \texttt{res} = x * 2))$. We have to leave the development of this example to the reader.

The formulas computed for recurrence nodes form chains of monotonically weaker (**U**) resp. stronger (**W**) approximations of the strongest resp. weakest fixpoint formula. For infinite data domains, this process need not come to an end, or the end, if reached, need not be detected. Below we will formalize the notion of "cycles broken by control" by a criterion sufficient for the termination of the instantiation.

Termination of the instantiation The termination criterion is based on an annotation of the generic tableau with variable sets. Basically, one just takes the sets of free variables of the instantiating formulas which would be computed by the process sketched above. But it is not necessary to compute the formulas themselves. Instead, one can operate on the finite domain of sets of variables involved (namely, the data variables of the program and the bound variables of the formula) where termination is guaranteed.

The case of next nodes may serve as an example of how these sets are computed. If x annotates the ith successor node of a next node in the CEP, and $x := e \in A_i$, all variables in e annotate the next node. Additionally we take the variables from c_i.

On the completed annotation sets, we draw edges indicating for each variable which other annotations caused its introduction. E.g. if x annotates the ith successor node of a next node, and e gets assigned to x along the edge, all variables in e have an edge pointing to x. Edges always go from inner node annotations to their successor annotations and from recurring leaves to recurrence nodes. Edges originating at next nodes which arise from some $x := e$ where e contains a function application get marked. Let us call the generic tableau *cycle-free* if there is no cycle in the resulting graph contains a marked edge.

Theorem 9 *The instantiation of the generic tableau of a formula terminates if the tableau is cycle-free.*

Critical points for termination of the instantiation are the fixpoint computations at recurrence nodes. During a fixpint computation, only substitutions and boolean operations are applied. If the generic tableau is cycle-free, only a finite number of terms will occur in those computations. Since only finitely many propositionally nonequivalent formulas can be constructed with finitely many terms, fixpoints will be reached and detected.

The condition on the annotations of the generic tableau can be viewed as describing a set of specifications having a finite reason in every data domain. It gives rise to a proof procedure which subsumes properly everything which can be gained by *data independence* reasoning [23]. A program is said to be data independent if, intuitively, its behavior does not depend on the identity of input values (changes to input values lead to similar changes of output values).

Any program which meets appropriate syntactic criteria on its data ports[2] will have only cycle-free generic tableaux, regardless of the formula. On the other hand, there are programs with cycle-free tableaux which perform a control-bounded number of computations and also tests on their data and which are thus not data independent.

This becomes clear if we draw a *value flow graph* of the CEP, similar to the graph on the annotations of the generic tableau. I.e. we annotate each state with the full set of data variables and draw edges and marked edges between variables annotating successive nodes according to the value flow. Transferring the notion of cycle-freeness to value flow graphs, we get a class of programs which will have only cycle-free tableaux.

Proposition 10 *If the value flow graph of a program is cycle-free, then each generic tableau built on its CEP is cycle-free.*

The proposition is implied by the observation that cycles in the generic tableau come from cycles in the *CEP*. This criterion is not necessary, but close to. It should be kept in mind, though, that the automatic instantiation process works in far more cases than just for programs having cycle-free value flow graphs. To decide specific properties, it is not necessary that *each* generic tableau is cycle-free.

Elaborations of the method The basic proof procedure described above, which is already quite powerful and has the advantage of being completely automatic, can be improved in several ways. For instance, it may be adapted to make use of the first-order theory of the data domain. Also, the user might be allowed to propose invariants or other guidance.

6 Conclusion

We envision the techniques described in this paper to be integrated into current design verification environments, providing interfaces to standard design languages. Given a system in one of those languages, the designer would provide formal specifications in FO-ACTL. Based on design knowledge and the properties to be checked, the designer would then debug the system by model checking stripped versions of the specifications in stripped control-expanded versions of the system. Note that the selection of the expanded set of variables will typically depend on the formula to be verified. In this phase, the full range of techniques for "classical" symbolic model checking will come into play. Only after surviving this debugging phase, *truly symbolic model checking* enters the stage.

Truly symbolic modelchecking will unfold the *CEP* in the verification process; data loops touched in this unfolding process have to be contracted using guidance on the source language level by the designer to a single transition labeled by the effect of the loop on the data variables and a condition guaranteeing termination. The verification of the purely sequential loop against such a total correctness formula is a classical task handled by a dedicated prover component, which will also have to handle termination proofs for loops claimed to be terminating by the introduction of fairness assumptions during the debugging phase. Given the contraction of loops, the techniques described in Section 5 will automatically generate verification conditions reducing the correctness of the FO-ACTL formula to be checked to a pure first order formula.

The scenario described above will be realized on the basis of the FORMAT verification tools [9], using *symbolic timing diagrams* [21] as graphical representations of FO-ACTL specifications, within a new industrial project aiming at safety critical embedded control applications.

[2]These are: No computations on data variables, no tests depending on them.

References

[1] Apt, K.R. *Ten years of Hoare's logic: A survey – part I*, TOPLAS **3** (1981), 431–483.

[2] Apt, K.R. and Olderog, E.-R. *Verification of sequential and concurrent programs*, Springer, New York (1991).

[3] Bradfield, J.C. *Verifying temporal properties of systems*, Birkhäuser, Boston (1992).

[4] Bradfield, J.C. and Stirling, C.P. *Verifying temporal properties of processes*, CONCUR '90, LNCS 458 (1990), 115-125.

[5] Brown,M.C., Clarke, E.M. and Grumberg, O. *Characterizing finite Kripke structures in propositional temporal logic*, TCS **59** (1988), 115–131.

[6] Burch, J.R., Clarke, E.M., McMillan, K.L. and Dill D.L. *Sequential circuit verification using symbolic model checking* DAC '90, 46–51.

[7] Clarke, E.M., Emerson, E.A. and Sistla, A.P. *Automatic verification of finite state concurrent systems using temporal logics*, POPL '83, 117–126.

[8] Clarke, E.M., Grumberg, O. and Long, D.E. *Model checking and abstraction*, POPL '92, 343–354.

[9] Damm, W., Döhmen, G., Helbig, J., Herrmann, R., Josko, B., Kelb, P., Korf, F. and Schlör, R. *Correct system level design with VHDL*, Tech. Rep., Oldenburg (1994), 54p.

[10] Damm, W., Josko, B. and Schlör, R. *Specification and verification of VHDL-based system-level hardware designs*, in Börger (ed.) Specification and Validation Methods, Oxford Univ. Press, 331–410 (to appear).

[11] Dingel, J. and Filkorn, T. *Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving*, CAV '95, to appear.

[12] Emerson, E.A. *Temporal and modal logic*, in: Handbook of Theor. Comp. Sc., B, North Holland (1990), 997–1072.

[13] Floyd, R.W. *Assigning meanings to programs*, Proc. AMS Symp. Applied Math. 19 (1967), 19–31.

[14] Graf, S. *Verification of a distributed cache memory by using abstractions*, CAV '94, LNCS 818 (1994), 207–219.

[15] Grumberg, O. and Long, D.E. *Model checking and modular verification*, TOPLAS **16** (1994), 843–871.

[16] Herrmann, R. and Pargmann, H. *Compiling VHDL data types into BDDs*, EURO-VHDL '94, 578–583.

[17] Hojati, R. and Brayton, R.K. *Automatic datapath abstraction in hardware systems*, CAV '95, to appear.

[18] Josko, B. *Verifying the correctness of AADL modules using model checking,* in: Stepwise refinement of distributed systems: models, formalisms, correctness, LNCS 430 (1990), 386–400.

[19] Manna, Z. *Beyong model checking*, CAV '94, LNCS 818 (1994), 220–221.

[20] Manna, Z. and Pnueli, A. *The temporal logics of reactive and concurrent systems. Specification.* Springer, New York 1992.

[21] Schlör, R. and Damm, W. *Specification and verification of system-level hardware designs using timing diagrams*, EDAC '93, 518–524.

[22] Stirling, C. and Walker, D. *Local model checking in the modal mu-calculus*, TAPSOFT '89, LNCS 351, 369–383.

[23] Wolper, P. *Expressing interesting properties of programs in propositional temporal logic*, POPL '86, 184–193.

Analytic and locally approximate solutions to properties of probabilistic processes (Extended Abstract)

C. Tofts*
Department of Computer Science,
The University,
Manchester,
M13 9PL,
email: cmnt@cs.man.ac.uk

April 24, 1995

Abstract

Recent extensions to process algebra can be used to describe performance or error rate properties of systems. We examine how properties of systems expressed in these algebras can be elicited. Particular attention is given to the ability to describe the behaviour of system components parametrically. We present how analytic formulae for performance properties can be derived from probabilistic process algebraic descriptions. Demonstrating how local approximate solutions can be derived for the properties when their exact solutions would be too computationally expensive to evaluate. As an example we derive the performance of an Alternating Bit protocol with respect to its error and retry rates.

1 Introduction

Process algebra [Mil80, Mil83, BK84, Hoa85, BBK86,M il90] is a methodology for formally calculating the behaviour of system in terms of the behaviours of its components. Recent extensions have added: timing properties [RR86, Tof89, MT90, Yi90, CAM90]; probabilistic properties [GSST90, Tof90, SS90, Tof94]; priority properties [BBK86, Cam89, Tof90, SS90, Tof94] and combinations of the above [Tof90, Han92, Tof94]. Process algebras with these extensions can be exploited to formally analyse the performance (in terms of either success or failure) of the design of systems [VW92,Tof93]. The analysis of a design can be greatly facilitated if an analytic solution to the performance of the system can be generated from an abstract description of the performance of its components. A possibly more important question is how tolerant to error (in the precise value of component parameters) are system level predictions.

Within the process algebra community the standard approach, to a verification problem, is to describe and compose the sytem components and then verify by comparing the systems behaviour with another process[Chr90,JS90] or a logical predicate[Han94,HJ94]. Whilst in many cases, where for instance design criterion are known in advance, this can be an appropriate methodology, it is however limited for the analysis of choices of system design. Of great importance is the ability to 'track' the effect of a single component upon system performance. To achieve this we need two things, firstly a syntactic presentation of system components, secondly an abstract method of calculating the components contribution to the systems performance.

Within a system subject to failure system requirements are often expressed in terms like, *the probability of error is less than 0.05.* It is hard to see how to interpret such a requirement in terms of the behaviour at a particular state. Indeed such requirements would often be re-expressed as *the probability of failure at* **any**

*This work is supported by an SERC Advanced Fellowship.

state *is less than 0.05.* Whilst this condition is certainly sufficient to ensure the conformance of a system to the requirement, is it reasonable? Consider the following WSCCS [Tof90,Tof94] process:

$$\begin{aligned} P_1 &\overset{def}{=} 9.\surd : P_1 + 1.\surd : P_2 \\ P_2 &\overset{def}{=} 1.error : P_1 + 9.\surd : P_1 \end{aligned}$$

The process P_1 certainly does not obey the condition that the probability of error in all states is less than 0.05 as this probability is 0.1 in state P_2. However, the process will only spend 10% of its time in state P_2 hence the probability of error is only 0.01, which does indeed meet our performance requirement. In order to calculate the error probability of this system we need to know the probability of the system being in any particular state. These probabilities can only be evaluated with respect to the complete system, and hence any logic suitable to express these properties will need to express probabilities of being in a particular state, and thus will not be an abstraction on any underlying transition description.

A frequently used method to formally derive the compliance of a probabilisitic system with some requirements is to express those requirements in the form of a 'standard' process [Chr90,JS90,Tof90]. Then demonstrating an equality between the intended implementation and the standard. If we attempt to describe our requirement on errors in this fashion we might write the following process:

$$Q \overset{def}{=} 95.\surd : Q + 5.error : Q$$

A process which certainly does not produce errors at a greater rate than 0.05. There's appears to be no sensible formal equivalence between the process Q and our previous example P_1. Again the reason for this incompatibility is that we compare processes on a state by state basis.

A possibly more realistic question would be the following. Given the process:

$$\begin{aligned} R_1 &\overset{def}{=} p.\surd : R_1 + 1.\surd : R_2 \\ R_2 &\overset{def}{=} 1.error : P_1 + q.\surd : P_1 \end{aligned}$$

what values of the expressions p and q will ensure that the process does not produce error actions at a greater rate than 0.05?

In many cases systemic requirements are expressed in terms of average performance. That is to say the average time before an error is seen will be greater than some amount, or alternatively the average time to see a 'good' outcome will be less than some amount. In order that such performance parameters can be derived we need to know not only the probability of reaching a particular state, but also how long it will take system to do so.

In Section 2 we present an extension to WSCCS to permit reasoning over weight expressions containing variables, and demonstrate how a Markov chain [Kei,Kle75,GS82] can be derived from a WSCCS process. In Section 3 we discuss how the properties of terminating processes can be derived. In Section 4 we discuss the solution of finite processes, in particular we examine how approximate analytic solutions can be derived when the system is too computationally expensive to solve exactly.

2 WSCCS

Our language WSCCS is an extension of Milner's SCCS [Mil83] a language for describing synchronous concurrent systems. To define our language we presuppose a free abelian group *Act* over a set of atomic action symbols with identity $\surd$, the inverse of a being $\overline{a}$, and action product denoted by $\#$. As in SCCS, the complementary actions a (conventionally input) and $\overline{a}$ (output) form the basis of communication. Within our group we define that $\overline{a} = a^{-1}$.

2.1 Expressions

We define a set of expressions.

Definition 2.1 *A* relative frequency expression (RFE) *is formed from the following syntax, with x ranging over a set of variable names VRF, and c ranging over a fixed field (such as $\mathcal{N}$ or $\mathcal{R}$):*

$$e ::= x|c|e+e|e*e$$

Further we assume that the following equations hold for relative frequency expressions:

$$\begin{array}{lcl} e+f & = & f+e \\ (e+f)+g & = & e+(f+g) \\ e*f & = & f*e \\ (e*f)*g & = & e*(f*g) \\ e*(f+g) & = & e*f+e*g \end{array}$$

alternatively, we have commutative and associative addition and multiplication, with multiplication distributing over addition. We shall assume that two expressions are equivalent if they can be shown so by the above equations.

In the sequel we shall omit the $*$ in expressions, denoting expression multiplication by juxtaposition. It should be noted that unlike other calculi with expressions [Mil90, Hen91] the value of our expressions can have **no effect** on the structure of the transition graph of our system. Hence we should not expect that adding this extra structure to our probabilistic process algebra will cause any new technical difficulties.

2.2 Weights

We also take a set of weights $\mathcal{W}$, denoted by w_i, which are of the form[1] $e\omega^k$ with e from the relative frequency expressions and the ω^k (with $k \geq 0$) a set of infinite objects, with the following multiplication and addition rules (assuming $k \geq k'$), we consider the objects e used as weights to be abbreviations for $e\omega^0$:

$$\begin{array}{ll} e\omega^k + f\omega^{k'} = e\omega^k = f\omega^{k'} + e\omega^k & e\omega^k + f\omega^k = (e+f)\omega^k = e\omega^k + f\omega^k \\ e\omega^k * f\omega^{k'} = (ef)\omega^{k+k'} = f\omega^{k'} * e\omega^k & \end{array}$$

2.3 The Calculus

The collection of WSCCS expressions ranged over by E is defined by the following BNF expression, where $a \in \mathit{Act}$, $X \in Var$, $w_i \in \mathcal{W}$, S ranging over renaming functions, those $S : \mathit{Act} \longrightarrow \mathit{Act}$ such that $S(\surd) = \surd$ and $\overline{S(a)} = S(\overline{a})$, action sets $A \subseteq \mathit{Act}$, with $\surd \in A$, and arbitrary *finite* indexing sets I:

$$E ::= X \mid a.E \mid \sum\{w_i E_i | i \in I\} \mid E \times E \mid E\lceil A \mid \Theta(E) \mid E[S] \mid \mu_i \tilde{x}\tilde{E}.$$

We let Pr denote the set of closed expressions, and add $\mathbf{0}$ to our syntax, which is defined by $\mathbf{0} \overset{def}{=} \sum\{w_i E_i | i \in \emptyset\}$.

The informal interpretation of our operators is as follows:

- $\mathbf{0}$ a process which cannot proceed;
- X the process bound to the variable X;
- $a : E$ a process which can perform the action a whereby becoming the process described by E;
- $\sum\{w_i.E_i | i \in I\}$ the *weighted* choice between the processes E_i, the weight of the outcome E_i being determined by w_i. We think in terms of repeated experiments on this process and we expect to see over a large number of experiments the process E_i being chosen with a relative frequency of $\frac{w_i}{\sum_{i \in I} w_i}$.
- $E \times F$ the synchronous parallel composition of the two processes E and F. At each step each process must perform an action, the composition performing the composition (in *Act*) of the individual actions;
- $E\lceil A$ represents a process where we only permit actions in the set A. This operator is used to enforce communication and bound the scope of actions;
- $\Theta(E)$ represents taking the prioritised parts of the process E only.

[1] Here e is the relative frequency with which this choice should be taken and k is the priority level of this choice. The choice of notation is based in [Tof90] arising from the observation that priority is similar to infinite weight.

- $E[S]$ represents the process E relabelled by the function S;
- $\mu_i\tilde{x}\tilde{E}$ represents the solution x_i taken from solutions to the mutually recursive equations $\tilde{x} = \tilde{E}$.

Often we shall omit the dot when applying prefix operators; also we drop trailing $\mathbf{0}$, and will use a binary plus instead of the two (or more) element indexed sum, thus writing $\sum\{1_1.a : \mathbf{0},\ \ 2_2 : b.\mathbf{0} | i \in \{1,2\}\}$ as $1.a\ +\ 2.b$. Finally we allow ourselves to specify processes definitionally, by providing recursive definitions of processes. For example, we write $A \overset{def}{=} a.A$ rather than $\mu x.ax$. The weight n is an abbreviation for the weight $n\omega^0$, and the weight w^k is an abbreviation for the weight $1\omega^k$.

The semantics, congruences and equational theory of this (minor) extension of WSCCS are essentially identical to that of [Tof95] up to arithmetic on weight expressions.

The congruences of WSCCS[Tof90,Tof94] are important as they permit us to algebraically manipulate our processes. However, in many instances these equivalences are too fine, consider the following pair of processes:

$$2.(2.P + 4.Q) \qquad\qquad 4.P + 8.Q$$

in many instances we should like to be able to consider these processes as equivalent. Hence, we would like a notion of equivalence that permits us to disregard the structure of the choices and just look at the total chance of reaching any particular state. whilst this notion of equivalence is useful it is known *not* to produce a congruence [SST89] for the complete language. However, such problems do not arise if we restrict our process syntax to only allow a single depth of summation, in which case $\overset{a}{\sim}$ coincides with the original probability preserving congruence [Tof94].

Definition 2.2 *We define an abstract notion of evolution as follows;*

$$P \xrightarrow{a[w]} P' \text{ iff } P \overset{w_1}{\longmapsto} \ldots \overset{w_n}{\longmapsto} \xrightarrow{a} P' \text{ with } w = \prod w_i.$$

As an example, $5.(3.(2.a : Q + 4.b : P) + 1.c : R) + 7.d : S \xrightarrow{a[30]} P$.

In order to define an equivalence which uses such transitions we need a notion of accumulation.

Definition 2.3 *Let S be a set of processes then:*

$$P \xrightarrow{a[w]} S \text{ iff } w = \sum\{w_i | P \xrightarrow{a[w_i]} Q \text{ for some } Q \in S\};\ ^2$$

We can now define an equivalence that ignores the choice structure but not the choice values.

Definition 2.4 *We say an equivalence relation $R \subseteq Pr \times Pr$ is an* abstract bisimulation *if $(P, Q) \in R$ implies that:*

> *there are $e, f \in RFE$ such that for all $S \in Pr/R$ and for all $w, v \in \mathcal{W}$, $P \xrightarrow{a[w]} S$ iff $Q \xrightarrow{a[v]} S$ and $ew = fv$.*

Two processes are abstract bisimulation equivalent*, written $P \overset{a}{\sim} Q$ if there exists a abstract bisimulation R between them.*

In particular this description of a WSCCS process gives us (essentially) a **probability transition graph**[Paz71].

Definition 2.5 *A probabilistic transition graph is a quintuple (V, T, s_0, A, RFE) where V is a set of states, T a set of transitions $\subseteq V \times (a \times p) \times V$, $s_0 \in V$ is an initial state, A ranged over by a an alphabet, and RFE ranged over by p the set of relative frequency expressions.*

[2]Remembering this is a multi-relation so some of the Q and w_i may be the same process and value. We take all occurences of processes in S and add together all the weight arrows leading to them.

3 Terminating Systems

As an example consider the following simple game. Two identical (possibly) biased coins are tossed repeatedly. If the coins both show heads then the game is won, if the coins both show tails then the game is lost, otherwise the coins are tossed again. What is the probability of winning the game? And how many tosses will be needed on average to see an outcome?

$$\begin{array}{lcl} Coin & \overset{def}{=} & p.\overline{head} : Coin + q.\overline{tail} : Coin \\ GR & \overset{def}{=} & 1.head^2\#\overline{win} : 0 \\ & & 1.head\#tail : GR \\ & & 1.tail^2\#\overline{lose} : 0 \\ Game & \overset{def}{=} & (Coin \times Coin \times GR)\lceil\{win, lose\} \end{array}$$

The probability of winning a game can be computed by solving the following simultaneous equation:

$$P(win) \quad = \quad \frac{2pq}{(p+q)^2}P(win) + \frac{p^2}{(p+q)^2}.1$$

and the average number of coin tosses equired to reach an outcome:

$$\begin{array}{lcl} E(Game) & = & \frac{2pq}{(p+q)^2}(E(Game)+1) + \frac{p^2}{(p+q)^2}(E(0)+1) + \frac{q^2}{(p+q)^2}(E(0)+1) \\ E(0) = 0 & & \end{array}$$

In the above we can rearrange the first equation to obtain the following:

$$E(Game) \quad = \quad \frac{2pq}{(p+q)^2}E(Game) + \frac{p^2}{(p+q)^2}E(0) + \frac{q^2}{(p+q)^2}E(0) + 1$$

Definition 3.1 *The total output of a state* $T(s) = \sum\{p|s \overset{a[p]}{\to} s'\}$.

Definition 3.2 *Let* $Win \subseteq A$ *be a set of winning actions, and* $P(s_0, Win)$ *be the probability of observing an action in the set* Win *starting from state* s_0*, and the average number of ticks before an action in* Win *is observed* $D(s_0, win)$.

$P(Win, s_0)$ is the solution of the following set of simultaneous equations, for all $s \in V$

$$\begin{array}{lcl} P(s, Win) & \overset{def}{=} & \sum\{\frac{p_i}{T(s)}P(s', Win)|s \overset{a[p_i]}{\to} s', a \notin Win\} \\ & & +\sum\{\frac{p_j}{T(s)}|s \xrightarrow{a} a[p_j], a \in Win\} \end{array}$$

Similarly we can define $D(Win, s_0)$ to be the a solution of the following set of simulataneous equations:

$$\begin{array}{lcl} D(s, Win) & \overset{def}{=} & \infty \text{ if } s \overset{a[p]}{\not\to} \\ D(s, Win) & \overset{def}{=} & \sum\{\frac{p_i}{T(s)}D(s', Win)|s \overset{a[p_i]}{\to} s', a \notin Win\} \\ & & +1 \end{array}$$

Hence given a probabilistic transition graph with n states we can produce a set of n simultaneous equations which describe the probabilities and averages we are interested in. Generating the equations from the graph is straightforward and the equations can subsequently be solved by any symbolic mathematics package.

4 Finite State Non-terminating Systems

Consider the following process:

$$\begin{array}{lcl} W1 & \overset{def}{=} & 6.sunny : W1 + 4.cloudy : W2 \\ W2 & \overset{def}{=} & 5.cloudy : W2 + 5.sunny : W1 \end{array}$$

If we assume that the environment (of the process) is unbiased with respect to the *sunny* and *cloudy* actions then the above system can be represented by the following Markov [Kei, Paz71, Kle75, GS82] transition matrix:

$$\begin{pmatrix} 0.6 & 0.5 \\ 0.4 & 0.5 \end{pmatrix}$$

A question that is asked about the above system is with what probability is the action *sunny* seen. This question can be answered by knowing with what probability the system is likely to be in state $W1$ or $W2$ at an arbitrary time. In Markov chain theory this is known as the **stable distribution**[Kei,GS82] of the chain. For a chain whose transition matrix is $\underline{\underline{A}}$ then a stable distribution $\underline{v}$ is one which satifies the following equation:

$$\underline{\underline{A}}\underline{v} = \underline{v}$$

and $|\underline{v}|$ is equal to 1.

In the case of the transition system above the stable distribution [Kei,Kle75,GS82] is given by the vector:

$$\begin{pmatrix} 5/9 \\ 4/9 \end{pmatrix}$$

and hence the probability of observing a *sunny* action is:

$$\begin{aligned} P(sunny) &= \tfrac{5}{9}\tfrac{6}{10} + \tfrac{4}{9}\tfrac{5}{10} \\ &= \tfrac{5}{9} \end{aligned}$$

Whilst in principle it is possible to convert a probability graph into its associated Markov chain and then solve for the stable distribution (by exploiting eigen theory) this is a highly inefficeint method of solving the problem. Given an n state probability transition graph the associated Markov chain matrix will be of size n^2. Since the number of states tends to be exponential in the number of components this would be highly prohibitive in its use of memory. An alternative manner of presenting the system is as folows. Remembering the original equation:

$$\underline{\underline{A}}\underline{\pi} = \underline{\pi}$$

by defining $r_i(\underline{\underline{A}})$ as the ith row of the matrix $\underline{\underline{A}}$ this is equivalent to solving the set of equations:

$$r_i(\underline{\underline{A}}).\underline{\pi} = \underline{\pi}_i$$

Whilst this would appear to still require $O(n^2)$ memory to represent the problem, in practice this is not the case. The probabilistic transition graphs that in practice result from process algebraic descriptions tend to be very sparse. On the whole very few of the states of a system are reachable from any particular state, in fact there is generally a (small) bound (k) on the number of permitted transitions from any particular state and therefore in this representation an amount $O(kn)$ of memory will be necessary to represent the solution.

We can define the necessary set of simultaneous[3] equations directly in terms of the original graph as follows:

$$\underline{\pi}_s \quad = \quad \{\sum \tfrac{p_j \underline{\pi}_j}{T(s_j)} | s_j \overset{a[p_j]}{\rightarrow} s\}$$

together with the condition that $\sum\{\underline{\pi}_s\} = 1$. the solution vector $\underline{\pi}$ being a stable distribution[4] for the transition system.

In this case the unstructured sparseness of the equation set makes the use of standard symbolic mathemtical equation packages very inefficient. A sparse equation solver was written to directly solve sets of equations generated by the above. By solving equations in inverse order of their fan out a considerable speed up can be achieved. The system generates a back substitution list which can be evaluated using a symbolic mathematics package.

To calculate the mean occurence of an action, the probability of that action occuring at a particular state is multiplied by the probability of being in that state.

[3]The set of equations derived for Markov transition matrix will *not* be independent [GS82] and hence an extra condition is neede to ensure a unique solution. This condition is derived from the definition that a probability distribution must sum to 1

[4]Care should be excersised when using stable distributions as their uniqueness is only guaranteed under restricted circumstances [Kei,GS82]

Example 4.1 *Consider two processes competing for the same resource. Each process issues a request with probability $1/p$ after the last time it had the resource, and then will release the resource with probability $1/q$ at each instant. We present the processes in the syntax of our analysis tool see Appendix A.*

```
*Simple competition example

bs U1 p.t:U1G + 1-p.t:U1
bs U1G 1@1.get^-1:U1Got + 1.baulk:U1G
bs U1Got q.put^-1:U1 + 1-q.t:U1Got

bs Res 1.get:RG + 1.t:Res
bs RG 1.put:Res + 1.t:RG

basi C baulk

btr Sys U1|U1|Res/C
```

Having solved the equations the above system generates we obtian the average number of baulk actions seen at each tick is given by the following eqaution:

```
     2       3              -15           2         3
(8. p  - 4. p  - 4.4408910    p q + 4. p  q - 8. p  q +

       4                    2  2  2
    4. p  q + 1. (2. - 2. p)  p  q ) /

      2       3                       -15  2         3
  (8. p  - 4. p  + 8. p q + 4.44089 10    p  q - 8. p  q +

       4         2       2  2       3  2      4  2
    4. p  q + 4. q  - 4. p  q  - 4. p  q  + 4. p  q )
```

As a result of using a real representation for numerical values in the toolset there are rounding errors[5] *in the above and if these are corrected we can obtain the following formula:*

$$\frac{p^2(4-2p+2q-4pq+4p^2q+(1-p)^2q^2)}{4p^2-2p^3+4p^3q-4p^3q+2p^4q+2q^2-2p^2q^2-2p^3q^2+2p^4q^2}$$

Unfortunately, the solution of symbolic simultaneous equations requires NP-space in the number of equation to represent the solutions. In practice it appears that symbolic solutions are unfeasible for systems of more than about 30 states.

An alternative definition[GS82] of the stable distribution of a markov system is presented in the following fashion:

$$\underline{\pi_{i+1}} = \underline{\underline{A}}\underline{\pi_i}$$

with $\underline{\pi} = lim_{i \leadsto \infty}\underline{\pi_i}$ if there is a unique stable distribution.

Using the above definition we can define an iterative calculation over the probability transition graph in the following fashion:

$$\underline{\pi_{i+1}}_s \quad = \quad \{\sum \frac{p_j \underline{\pi_i}_j}{T(s_j)} | s_j \overset{a[p_j]}{\rightarrow} s\}$$

[5]Caused by the use of real numbers in the analysis system, easily identifiable as the terms are insignificant. The exponential growth of terms exhibited by products of processes forced the use of (truncated) real aritmetic in the tool, rather than the prefered (exact) integer arithmetic

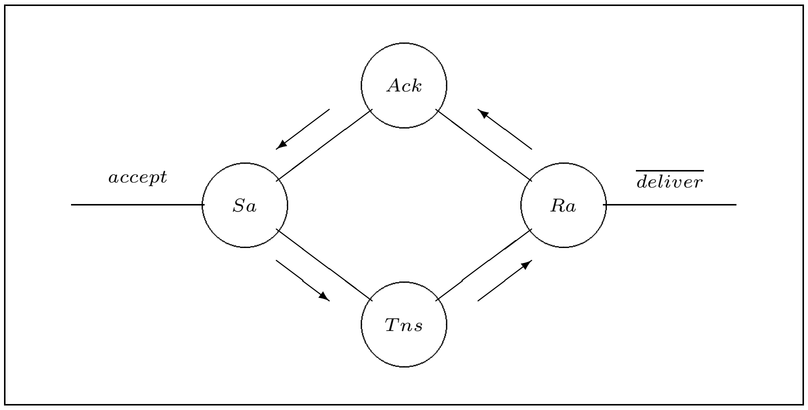


Figure 1: Alternating Bit Protocol

If an attempt is made to calculate an exact solution to the above iteration procedure then the same representation problem is encountered. However, it is possible to exploit the above method to provide an approximate solution to the stable vector problem. By truncating the $\underline{\pi_i}$ to a particular accuracy after each iteration of the calculation. If the terms (in a variable x say) are maintained to order k then, standard numerical solution of eigensystems theory, [Wil65] shows that the solution will have an absolute error of $O(x^k)$.

Hence the following procedure can be exploited to give an approximate solution to the distribution problem, choose any non-zero length 1 $\underline{\pi_0}$[6]:

1. Compute $\underline{\pi_{i+1}}$ from $\underline{\pi_i}$;
2. truncate $\underline{\pi_{i+1}}$ to required accuracy k
3. repeat from 1 until stability is achieved

In practice one can compute a central approximation and then compute the further terms by increasing the approximation level steadily until the desired level is reached.

4.1 The alternating bit protocol

In [Mil90] Milner presents an implementation of the alternating bit protocol in CCS, and demonstrates that the protocol is correct. Perforce this implementation ignores the temporal and probabilistic properties of the system and its components.

Our alternating bit protocol realisation is depicted in Figure 1.

The process Sa will work in the following manner. After accepting a message, it sends it with bit b along the channel Tns and waits. Subsequently there are three possibilities:

- it times out and retransmits the message;
- it gets an acknowledgement b from the Ack line (signifying a correct transmission), so that it can now accept another message;
- it gets an acknowledgement $\neg b$ from the Ack line (signifying a superfluous extra acknowledgement of earlier message) which it ignores.

The replier Ra works in a dual manner. After a message is delivered it sends an acknowledgement with bit b along the Ack line. Subsequently there are again three possibilities:

- it times out, and retransmits the acknowledgement;
- it gets a new message with bit $\neg b$ from the Tns line, which it delivers and acknowledges with bit $\neg b$;

[6]In practice we use the vector $\underline{\pi_0}_i = \frac{1}{n}$ when the system has n states.

- it gets a repetition of the old message with bit b which it ignores.

We assume that messages are lost by the medium with probability err on each transmission. The sender and replier processes will retry with probability rt at for each tick whilst they are waiting for an acknowledgement or the message bit to change. In order that we can apply our peturbation theory to the variables err and rt, we assume perturbations of ere and rte upon their basic values.

```
*Probabilistic alternating bit protocol
*
*Chris Tofts 9/8/94 after CWB versions
*

*The basic sender process, note do everything asynchronously
*this should be Sa 1.send:Sa1 + 1.t:Sa

bs Sa 1.t:Sa1

*send out a signal as soon as possible

bs Sa1 1@1.s0^-1:Sa2 + 1.t:Sa1

*wait for acknowledgement to come through

bs Sa2 1.rack0:S1s + 1.rack1:Sa1 + rt-rte.t:Sa1 + 1-rt-rte.t:Sa2

*tell the world that it got through OK and invert sending bit

bs S1s 1.succ:S1

*the dual of the above system for sending with bit set to 1
*this should be bs S1 1.send:S11+ 1.t:S1

bs S1 1.t:S11
bs S11 1@1.s1^-1:S12 + 1.t:S11
bs S12 1.rack1:Sas + 1.rack0:S11 + rt-rte.t:S11 + 1-rt-rte.t:S12
bs Sas 1.succ:Sa

*the receiver for all of our endeavours....

bs Ra  1.r0:Rar1+1.r1:Ra2 + rt-rte.t:Ra2 + 1-rt-rte.t:Ra
bs Rar1 1.receive:Ra1

*try to send the data as quickly as possible

bs Ra1 1@1.sack0^-1:R1 + 1.t:Ra1 + 1.r0:R11 + 1.r1:Ra12
bs Ra2 1@1.sack1^-1:Ra + 1.t:Ra2 + 1.r1:R12 + 1.r0:Rar1
bs R1 1.r1:Ra12 + 1.r0:R11 + rt-rte.t:R11 + 1-rt-rte.t:R1
bs Ra12 1.receive:R12
bs R11 1@1.sack0^-1:R1 + 1.t:R11 + 1.r0:R11 + 1.r1:Ra12
bs R12 1@1.sack1^-1:Ra + 1.t:R12 + 1.r1:R12 + 1.r0:Rar1

*the lower channel for sending data on
*we send out data as soon as possible after the transmission
*time if it is not lost to error...

bs Ml1 1.s0:Ml1a0 + 1.s1:Ml110 + 1.t:Ml1

*decide if the data was transmitted OK, or was subject to error

bs Ml1a0 1-err-ere.t:Ml1a + err-ere.t:Ml1
bs Ml1a 1@1.r0^-1:Ml1 + 1.t:Ml1a
bs Ml110 1-err-ere.t:Ml11 + err-ere.t:Ml1
bs Ml11 1@1.r1^-1:Ml1 + 1.t:Ml11

*this is the transmitting medium for the return of the data

bs Ml2 1.sack0:Ml2a1 + 1.sack1:Ml211 + 1.t:Ml2
bs Ml2a1 1-err-ere.t:Ml2a + err-ere.t:Ml2
bs Ml2a 1@1.rack0^-1:Ml2 + 1.t:Ml2a
bs Ml211 1-err-ere.t:Ml21 + err-ere.t:Ml2
bs Ml21 1@1.rack1^-1:Ml2 + 1.t:Ml21

*That's all the sequential bits done so we can now have a go at putting
*it all together...

basi Allow send, receive, succ

*this is the complete system

btr ABP Ra|Ml1|Ml2|Sa/Allow
```

For a value of $rt = 0.5$ and $rte = 0.0$ and err in the range 0.05 to 0.35, we obtain the following piecewise approximation in the form of an SML function:

```
fun ABP(vl)
=if 0.05<=vl andalso vl<=0.15
 then let val ere = (0.05+0.15)/2.0 - vl in
(26.2801394665775-285.522119242462ere^3 -94.9741260228557ere^2 +26.6274947013716ere )/
(270.683502505543+1.32871491587139E~12ere^3 -
7.47846229387505E~13ere -3.12638803734444E~13ere^2 )
end
else if 0.15<=vl andalso vl<=0.25
 then let val ere = (0.15+0.25)/2.0 - vl in
(11.7544078755176-83.86909484614ere^3 -16.6668138922172ere^2 +19.7005844688574ere )/
(139.000000000001-3.19744231092045E~13ere^3 -
4.01456645704457E~13ere -3.92574861507455E~13ere^2 )
end
else if 0.25<=vl andalso vl<=0.35
 then let val ere = (0.25+0.35)/2.0 - vl in
(9.68600554074613-41.7301838198085ere^3 +1.29897621781181ere^2 +20.937477181878ere )/
(139.0-1.59872115546023E~14ere^3 -
3.5438318946035E~13ere -2.46025422256935E~13ere^2 )
end
else 0.0;
```

The total time to construct the process graph and the above approximations was about 1/2 hour on a Sun II.

5 Conclusions

Whilst it is possible to verify the behaviour of a system by checking the process that describes it against another process[Mil80,Mil90,Chr90,JS90,Tof90,SS90] or a predicate[Mil90,Han94,HJ94] this is often not the best approach. In many cases the intention of the design analysis is to determine how well a system can function which is why simulation[BDMN79,Bir79,Kre86,BFS87] is often resorted to. It is important in such circumstances to be able to identify the contribution of the underlying components to the overall system performance. The verify strategy works well when system requirements are known in advance but in many cases the design problem is one of: what is the best way of using these components to solve a particular problem? In this case the components are fixed, and we need to be able to derive the resulting systemic behaviour.

It might seem that we are not exploiting the algebraic properties of the process algebraic description in deriving our systemic properties. This is not the case. In order to minimize computation time care must be taken to keep the probability transition graph as small as possible. Thus we exploit algebraic properties of WSCCS to maintain as small a description of our systems as is necessary.

Whilst it is true that for the majority of problems a symbolic approach to process representation cannot give an analytic solution this approach still has major advantages. Since we can describe performance aspects of the system components symbolically and construct its probability transition graph in terms of those symbols (a costly operation in time even if all of the transition probabilities are constants) and then instantiate the graph with particular values of interest. We can study the systemic behaviour quickly under a wide range of conditions.

The generation of local approximations to the solutions of systems is of great importance. It has long been known that the behaviour of complex systems can critically dependent on the precise values of their parameters. In any real implementation of a system the true values of its components performances are liable to vary slightly from the exact values in our models. Local approximations allow us to assess the effect that these small variations may have on the systems true behaviour. For instance if performance could be heavily compromised by a small variance in one components performance it may be a good idea to redesign the system to be more tolerant or replace that component.

A sublanguage of WSCCS and the algorithms in this paper have been implemented as a set of SML functions (Probabilistic Algebra Tools set) which can be obtained from cmnt@cs.man.ac.uk. In terms of scale the exact solution generator can cope with systems of about 30 states, and will execute upon systems of this scale in 2 hours on a Sun 2. The approximation method can cope with systems of 1000's of states and can take 24 hours to execute on such systems. Automatic scanning functions have been written to generate the piecewise approximations. For numerical problems the system can successfully manage systems of 10000s of states.

6 Bibliography.

[BBK86] J. Baeten, J. Bergstra and J. Klop, Syntax and defining equations for an interrupt mechanism in process algebra, Fundamenta Informatica IX, pp 127-168, 1986.

[BDMN79] G. Birtwistle, O-J Dahl, B. Myhrhaug and K. Nygaard, Simula Begin, 2nd Edition, Studentliteratur, Lund, Sweden, 1979.

[BFS87] P. Bratley, B. Fox and L. Schrage, A guide to simulation, second edition, 1987.

[Bir79] G. Birtwistle, DEMOS — discrete event modelling on Simula. Macmillen, 1979.

[BK84] J.A. Bergstra, J.W. Klop, The algebra of recursively defined processes and the algebra of regular processes, in Proc 11th ICALP, Springer LNCS 172, pp 82-85, 1984.

[CAM90] L. Chen, S. Anderson and F. Moller, A Timed Calculus of Communicating Systems, LFCS-report number 127

[Cam89] J. Cammilleri. Introducing a Priority Operator to CCS, Computer Laboratory Technical Report, Cambridge University, 1989.

[Chr90] I. Christoff, Testing Equivalences and Fully Abstract Models for Probabilistic Processes, Proceedings Concur '90, LNCS 458, 1990.

[DLSB82] V.A. Dyck, J.D. Lawson, J.D. Smith and R.J. Beach, Computing: An Introduction to Structured Problem Solving Using Pascal: Reston, Reston, 1982.

[GS82] G.R. Grimmet and D.R. Stirzaker, Probability and Random Processes, Oxford Science Publications, 1982.

[GSST90] R. van Glabbek, S. A. Smolka, B. Steffen and C.Tofts, Reactive, Generative and Stratified Models of Probabilistic Processes, proceedings LICS 1990.

[Han94] M.R. Hansen, Model checking discrete duration calculus, FACS 6A:826-845, 1994.

[Hen91] M. Hennessy, A proof system for CCS with value passing, FACS 3: 346-366.

[HJ94] H. Hansson and B. Jonsson, A Logic for Reasoning about Time and Reliability, FACS (6):512-535, 1994.

[Hoa85] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall 1985.

[HR90] M. Hennessey and T. Regan, A Temporal Process Algebra, Technical Report, Department of Cognitive Science, Sussex University, 1990.

[Jon90] C. C. M. Jones, Probabilistic Non-determinism, PhD Thesis University of Edinburgh 1990.

[Kei] J. Keilson, Markov Chain Models - Rarity and exponentiality, Applied Mathematical Sciences 28, Springer Verlag.

[Kin69] J.F.C. Kingman, Markov Population Processes, Journal of Applied Probability, 6:1-18, 1969.

[Kle75] L. Kleinrock, Queueing Systems, Volumes I and II, John Wiley, 1975.

[Kre86] W. Kreutzer, System Simulation, Addison Wesley, 1986.

[JS90] C. Jou and S. Smolka, Equivalences, Congruences and Complete Axiomatizations for Probabilistic Processes, Proceedings Concur '90, LNCS 458, 1990.

[LS89] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. proceedings POPL 1989.

[Mil80] R. Milner, Calculus of Communicating System, LNCS92, 1980.

[Mil83] R. Milner, Calculi for Synchrony and Asynchrony, Theoretical Computer Science 25(3), pp 267-310, 1983.

[Mil90] R. Milner, Communication and Concurrency, Prentice Hall, 1990.

[MT90] F. Moller and C. Tofts, A Temporal Calculus of Communicating Systems, Proceedings Concur '90, LNCS 458, 1990.

[OW78] G. F. Oster and E. O. Wilson, Caste and Ecology in Social Insects, Princeton University Press, 1978.

[Paz71] A. Paz, Introduction to probabilistic automata, Academic Press, 1971.

[Plo81] G. D. Plotkin, A structured approach to operational semantics. Technical report Daimi Fn-19, Computer Science Department, Aarhus University. 1981.

[RR86] G. Reed and W. Roscoe, A Timed Model for CSP, Proceedings ICALP '86, LNCS 226, 1986.

[SS90] S. Smolka and B. Steffen, Priority as Extremal Probability, Proceedings Concur '90, LNCS 458, 1990.

[SST89] S. Smolka, B. Steffen and C. Tofts, unpublished notes. Working title, Probability + Restriction $\Rightarrow$ priority.

[THF92] C. Tofts, M.J.Hatcher, N. Franks, Autosynchronisation in Leptothorax Acervorum; Theory, Testability and Experiment, Journal of Theoretical Biology 157: 71-82.

[TF92] C. Tofts, N. Franks, Doing the Right Thing: Ants, Bees and Naked Mole Rats, Trends in Evolution and Ecology 7: 346-349.

[Tof89] C. Tofts, Timing Concurrent Processes, LFCS-report number 103, 1989.

[Tof90] C. Tofts, A Synchronous Calculus of Relative Frequency, CONCUR '90, Springer Verlag, LNCS 458.

[Tof93] C. Tofts, Exact Solutions to Finite State Simulation Problems, Research Report, Department of Computer Science, University of Calgary, 1993.

[Tof94] C. Tofts, Using Process Algebra to Describe Social Insect Behaviour, Transactions on Simulation, 1993.

[Tof94] C. Tofts, Processes with Probabilities, Priorities and Time, FACS 6(5): 536-564, 1994.

[Yi90] Yi W., Real-Time Behaviour of Asynchronous Agents, Proceedings Concur '90 LNCS 458, pp 502-520, 1990.

[VW92] S. F. M. van Vlijmen, A. van Waveren, An Algebraic Specification of a Model Factory, Research report, University of Amsterdam Programming research Group, 1992.

[Wil65] J. Wilkinson, The Numerical Eigenvalue Problem, Oxford University press 1965.

A PRobabilistic Algebra Toolset (PRAT)

The basic process definition mechanism is to present a file in Edinburgh concurrency workbench like syntax. The system then generates an extended probabilistic transition graph (it takes account of priorities) and provides a set of analysis functions which can be applied to the system.

A.1 Weights

Weights are defined by the following syntax, where n is an integer and *string* an ascii character string:

$$e ::= n|string|e - e^7$$
$$w ::= e@n$$

So the following are weights: `5, 1-p, 5@2, 1-p@3`. the last two being weights at priority level 2 and 3 repsectively. Note that we do not allow symbolic priorities, as this would actually affect the computational structure.

A.2 Actions

Actions are defined as products of powers of strings;

$$A ::= string[\hat{}n]|A\#A$$

again we do not allow symbolic action powers.

So the following are actions: `a, a^-1,a#b^-2, c^4#a#b^3`.

To form a permission free group we provide a binding operator for sets of actions:

```
bs Set a,b,c
```

binds the name `Set` to the actions `a,b,c`.

A.3 Processes

We define the following constructions on processes which we present by example, it should be noted that we only allow one depth of operator application, this permits automatic absorbtion of equivalent state in parallel compositions:

Sequential	`bs Coin p.head:C1 + 1-p.tail:C2`
Parallel	`bpa Sys S1|S2`
Permission	`bperm S1 Sys/Set`
Priority	`bpi Sn S`
Pri(Perm(Par))	`btr Sys S1|S2|S3/Set`
Perm(Par)	`bpc Sys S1|S2/Set`
Comment	`*this is a comment`

As the sytem constructs processes it prints out the number of states it has allocated so far as an indication of the work left to do.

[7] It should be noted that the current parser works LR so that $1 - p - q$ is actually $1 - p + q$

A.4 Analysis

The following functions are presented to allow the maintenance, exploration and analysis of systems:

- `cle()` clear the current process environment.
- `rf(filename)` read a process definition from `filename`.
- `dupo(filename)` duplicate all output to the file.
- `co()` close duplicate output file.
- `sim(Pname)` simulate the process state called `Pname`. The simulator presents a menu of actions. Typing the number of the action causes the system to continue from the labelled state. Hitting return takes option 0 and hitting q exits the simulator.
- `fd(Pname)` find deadlocks in the process `Pname`. If a deadlock is found then the shortest transition to that state is printed.
- `ll(Pname)` find livelocks in the process `Pname`.
- `do_prob(Pname,Win,Lose)` generate a set of equations describing the probability of seeing the `Win` action, ignoring other actions but terminating on the `Lose` action. Action syntax as above.
- `do_mean(Pname,Win,Lose)` generate a set of equations describing the mean number of ticks to see a `Win` or `Lose` action, ignore all other actions.
- `ibv(vn,real)` bind the weight variable `vn` to the real value `real`.
- `sc(Pname)` generate a set of back substitutions for the stable distribution of process `Pname`.
- `solmn(Pname,action,file)` produce an expression for the mean number of `action` in process `Pname` output the results to `file`. This is separate from the above to allow reuse of the stable solution information.
- `genfun(Pname,Action,vn,low,hi,step,inR,gR,aL)` generate a piecewise approximation to the mean number of actions `Action` the parameters are as follows:

`Pname`	the process;
`Action`	the action;
`vn`	the variable name to range over;
`low`	lower limit of solution generation;
`hi`	upper limit of solution generation;
`step`	step between solutions;
`inR`	initial number of iterations at approx level 0, to generate coarse approximation;
`gR`	number of iterations at final approx level;
`aL`	required accuracy of the final answer.

- `genml(evrn,aprx,file)` generate an SML function to evaluate the piecewise approximation generated by above function, the error variable is given by `evrn` and `file` is the name of a file to copy the result to.

The system supplies two iterative solution packages, one for numeric solutions the other for local approximations, we describe their use below:

	Numerical	Analytical
Initialise	`start_iterate(Pname)`	`SI(Pname)`
Iterate	`itn(n)`	`APN(n)`
Print Sol	`pt()`	`CAP()`
Mean	`ma(action)`	`AM(action)`
Mean (Cyclic)	`mcyc(action,n)`	`APC(acs,n)`

In the above the `n` in the cyclic means is the number of states in the cycle. Often good approximations to nearly cyclic systems can be obtained cheaply by exploiting this function. A further function `sal(n)` is supplied to set the approximation level in the second set of functions.

Model Checking of non-finite state processes by Finite Approximations

N. De Francesco ♣, A. Fantechi ♣ ◇, S. Gnesi ◇ and P. Inverardi ◇ ♡

♣ Dipartimento di Ingegneria dell'Informazione, Univ. di Pisa, Italy, e-mail: nico@iet.unipi.it, fantechi@iet.unipi.it

◇ Istituto di Elaborazione dell'Informazione, C.N.R. Pisa, Italy, e-mail: gnesi@iei.pi.cnr.it

♡ Dip. di Matematica Applicata, Univ. dell'Aquila, Italy, e-mail:inverard@iei.pi.cnr.it

Abstract

In this paper we present a verification methodology, using an action-based logic, able to check properties for full CCS terms, allowing also verification on infinite state systems. Obviously, for some properties we are only able to give a semidecision procedure. The idea is to use (a sequence of) finite state transition systems which approximate the, possibly infinite state, transition system corresponding to a term. To this end we define a particular notion of approximation, which is stronger than simulation, suitable to define and prove liveness and safety properties of the process terms.

1 Introduction

Many verification environments are presently available which can be used to automatically verify properties of reactive systems specified by means of process algebras, with respect to behavioural relations and logical properties. Most of these environments [7, 12, 13, 19] are based on the hypothesis that the system can be modelled as a finite state Labelled Transition Systems (LTS) and that the logic properties are regular properties. That is, no means are provided to deal with non-finite state LTS's. Usually, in these environments, to avoid the nontermination of the generation phase a term must satisfy some finiteness syntactic conditions: in the case of CCS, for example, terms where a process variable x occurs in a parallel composition belonging to the definition of x are not handled [22].

We are interested here to deal with non finite-state systems; approaches have been proposed to this aim, which are not based on LTS's [1, 4, 14, 15, 16]; we consider instead LTS based verification. The idea is to use, for proving a logical property, a sequence of finite state LTSs approximating the, possibly infinite state, LTS corresponding to a term by the standard CCS semantics.

In this paper we present a verification methodology to check properties expressed in ACTL, an action based logic [11], on full CCS terms (with no syntactic restriction), thus allowing complete generality of the class of reactive systems to be specified. We are able to carry on the verification even though the "usual" LTS generation fails. Obviously, for some of the properties, we are able to give only a semidecision procedure. This procedure is based on a notion of approximation and on the study of the ACTL properties *preserved* by the approximation. In this way, we can infer the satisfaction of a property by the whole system from the satisfaction of the property by a chain of approximations. In particular, we define an approximation chain, denoted as $\{N_i\}$, which is very expressive with respect to liveness properties.

In order to reason on the properties that we are able to prove with approximation chains, we start giving a syntactic characterization of different kinds of properties. Moreover, we define a

$$\textbf{Act}\ \frac{}{\mu.p \xrightarrow{\mu} p} \qquad \textbf{Con}\ \frac{p \xrightarrow{\mu} p',\ x \overset{def}{=} p}{x \xrightarrow{\mu} p'}$$

$$\textbf{Sum}\ \frac{p \xrightarrow{\mu} p'}{p+q \xrightarrow{\mu} p' \text{ and } q+p \xrightarrow{\mu} p'} \qquad \textbf{Par}\ \frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q \text{ and } q|p \xrightarrow{\mu} q|p'}$$

$$\textbf{Com}\ \frac{p \xrightarrow{\alpha} p',\ q \xrightarrow{\overline{\alpha}} q'}{p|q \xrightarrow{\tau} p'|q'} \qquad \textbf{Res}\ \frac{p \xrightarrow{\mu} p',\ \mu, \overline{\mu} \notin A}{p\backslash A \xrightarrow{\mu} p'\backslash A}$$

$$\textbf{Rel}\ \frac{p \xrightarrow{\mu} p'}{p[f] \xrightarrow{f(\mu)} p'[f]}$$

Figure 1: The SOS rules

criterion to compare the suitability of approximation chains to prove properties. Following this notion, we formalize the fact that a chain is "better" than another one, if its set of provable properties is greater. Our work differs from the abstract interpretation approaches for model checking of transition systems [2, 6, 8] since we do not build an abstract (with respect to values) model on which the properties are proved, but a suitable chain of finite labelled transition systems based on the operational semantics: when dealing with infinite systems, this allows us to choose the approximation level case by case.

2 Background

2.1 CCS

We summarize the most relevant definitions regarding CCS, and refer to [21] for more details. The CCS syntax is the following:
$p ::= \mu.p \mid nil \mid p+p \mid p|p \mid p\backslash A \mid x \mid p[f]$

Terms generated by p ($Terms$) are called *process terms* (called also *processes* or *terms*); x ranges over a set $\{X, Y, ..\}$, of process variables. A process variable is defined by a process definition $x \overset{def}{=} p$, (p is called the expansion of x). As usual, there is a set of visible actions $Vis = \{a, \overline{a}, b, \overline{b}, ...\}$ over which α ranges, while μ, ν range over $Act = Vis \cup \{\tau\}$, where τ denotes the so-called *internal action*. We denote by $\overline{\alpha}$ the action complement: if $\alpha = a$, then $\overline{\alpha} = \overline{a}$, while if $\alpha = \overline{a}$, then $\overline{\alpha} = a$. By *nil* we denote the empty process. The operators to build process terms are prefixing ($\mu.p$), summation ($p+p$), parallel composition ($p|p$), restriction ($p\backslash A$) and relabelling ($p[f]$), where $A \subseteq Vis$ and $f : Vis \to Vis$. Given a term p, an occurrence of a process variable x is *guarded in* p if it is within some sub-term of the form $\mu.q$. We assume that (i) Vis is finite; (ii) for each definition $x \overset{def}{=} p$, each occurrence of each process variable is guarded in p; (iii) all terms are closed, i.e. all variables occurring in a term are defined.

An operational semantics OP is a set of inference rules defining a relation $D \subseteq Terms \times Act \times Terms$. The relation is the least relation satisfying the rules. If $(p, \mu, q) \in D$, we write $p \xrightarrow{\mu}_{OP} q$. The rules defining the semantics of CCS [21], from now on referred to as *SOS*, are recalled in Figure 1.

A *labelled transition system* (or simply *transition system*) TS is a quadruple (S, T, D, s_0), where

S is a set of states, T is a set of transition labels, $s_0 \in S$ is the initial state, and $D \subseteq S \times T \times S$. A transition system is finite if D is finite.

A finite computation of a transition system is a sequence $\mu_1\mu_2..\mu_n$ of labels such that: $s_0 \xrightarrow{\mu_1}_{OP} .. \xrightarrow{\mu_n}_{OP} s_n$.

Given a term p (and a set of process variable definitions), and an operational semantics OP, $OP(p)$ is the transition system $(Terms, Act, D, p)$, where D is the relation defined by OP.

Let $TS_1 = (S_1, T_1, D_1, s_{0_1})$ and $TS_2 = (S_2, T_2, D_2, s_{0_2})$ be transition systems and let $s_1 \in S_1$ and $s_2 \in S_2$. s_1 and s_2 are *strongly equivalent* (or simply *equivalent*) ($s_1 \sim s_2$) if there exists a *strong bisimulation* that relates s_1 and s_2. $\mathcal{B} \subseteq S_1 \times S_2$ is a strong bisimulation if $\forall(s_1, s_2) \in \mathcal{B}$ (where $\mu \in T_1 \cup T_2$),

- $s_1 \xrightarrow{\mu}_1 s_1'$ implies $\exists s_2' : s_2 \xrightarrow{\mu}_2 s_2'$ and $(s_1', s_2') \in \mathcal{B}$; $s_2 \xrightarrow{\mu}_2 s_2'$ implies $s_1 \xrightarrow{\mu}_1 s_1'$ and $(s_1', s_2') \in \mathcal{B}$

s_2 *simulates* s_1 if there exists a *strong simulation* that relates s_1 and s_2. $\mathcal{R} \subseteq S_1 \times S_2$ is a strong simulation if $\forall(s_1, s_2) \in \mathcal{R}$ (where $\mu \in T_1 \cup T_2$): $s_1 \xrightarrow{\mu}_1 s_1'$ implies $\exists s_2' : s_2 \xrightarrow{\mu}_2 s_2'$ and $(s_1', s_2') \in \mathcal{R}$.

TS_1 and TS_2 are said to be *equivalent* ($TS_1 \sim TS_2$) if a strong bisimulation exists for s_{0_1} and s_{0_2}. Two CCS terms p and q are *equivalent* ($p \sim q$) if $SOS(p) \sim SOS(q)$.

TS_2 simulates TS_1 if a strong simulation $\mathcal{R}$ exists such that $(s_{01}, s_{02}) \in \mathcal{R}$.

Given a state s of a transition system $TS = (S, T, D, s_0)$, we say that $s \not\rightarrow$ if no $s' \in S$ and $\mu \in T$ exist such that $(s, \mu, s') \in D$.

CCS can be used to define a wide class of systems, that ranges from Turing machines to finite systems [22]; therefore, in general, CCS terms cannot be represented as finite state systems.

2.2 ACTL

We introduce now the action based branching temporal logic ACTL defined in [11]. This logic is suitable to express properties of reactive systems defined by means of TS's. ACTL is in agreement with the notion of bisimulation defined above. Before defining syntax and semantics of ACTL operators, let us introduce some notions and definitions which will be used in the sequel.

For $A \subseteq Act$, we let $D_A(s)$ denote the set $\{s'$: there exists $\alpha \in A$ such that $(s, \alpha, s') \in D\}$. We will also use the action name, instead of the corresponding singleton denotation, as subscript. Moreover, we let $D(s)$ denote in short $D_{Act}(s)$ and $D_{A_\tau}(s)$ denote $D_{A\cup\{\tau\}}(s)$.
For $A, B \subseteq Act$, we let A/B denote the set $A - (A \cap B)$.
Given a LTS TS=(S,T,D,s_0), we define:

- σ is a path from $r_0 \in S$ if either $\sigma = r_0$ (the empty path from r_0) or σ is a (possibly infinite) sequence $(r_0, \alpha_1, r_1)(r_1, \alpha_2, r_2) \ldots$ such that $(r_i, \alpha_{i+1}, r_{i+1}) \in D$ for each $i \geq 0$.

- A path σ is called maximal if either it is infinite or it is finite and its last state r has no successor states ($D(r) = \emptyset$). The set of maximal paths from r_0 will be denoted by $\Pi(r_0)$.

- If σ is infinite, then $|\sigma| = \omega$.
  If $\sigma = r_0$, then $|\sigma| = 0$.
  If $\sigma = (r_0, \alpha_1, r_1)(r_1, \alpha_2, r_2) \ldots (r_n, \alpha_{n+1}, r_{n+1})$, $n \geq 0$, then $|\sigma| = n + 1$. Moreover, we will denote the i^{th} state in the sequence, i.e. r_i, by $\sigma(i)$. □

To define the logic ACTL [11], an auxiliary logic of actions is introduced. The collection $\mathcal{AF}$ of *action formulae* over Vis is defined by the following grammar where χ, χ', range over action formulae, and $\alpha \in Vis$:

$\chi ::= \alpha | \neg\chi | \chi \wedge \chi$

We write ff for $\alpha_0 \wedge \neg\alpha_0$, where a_0 is some chosen action, and tt stands for $\neg\mathit{ff}$. Moreover, we will write $\chi \vee \chi'$ for $\neg(\neg\chi \wedge \chi')$. An action formula permits the expression of constraints on the actions that can be observed (along a path or after next step); for instance, $\alpha \vee \beta$ says that the only possible observations are α or β, while tt stands for "all actions are allowed" and ff for "no actions can be observed", that is only silent actions can be performed.

The satisfaction of an action formula χ by an action α, $\alpha \models \chi$, is defined inductively by:

•$\alpha \models \beta$ iff $\alpha = \beta$; •$\alpha \models \neg\chi$ iff not $\alpha \models \chi$; •$\alpha \models \chi \wedge \chi'$ iff $\alpha \models \chi$ and $\alpha \models \chi'$

Given an action formula χ, the set of the actions satisfying χ can be given by the function $\kappa : \mathcal{AF}(Vis) \rightarrow 2^{Vis}$ as follows:

•$\kappa(\mathit{tt}) = Vis$; •$\kappa(\alpha) = \{\alpha\}$; •$\kappa(\neg\chi) = Vis/\kappa(\chi)$; •$\kappa(\chi \vee \chi') = \kappa(\chi) \cup \kappa(\chi')$.

The syntax of ACTL is defined by the state formulae generated by the following grammar:
$\phi ::= \mathit{tt} \mid \phi \wedge \phi \mid \neg\phi \mid E\gamma \mid A\gamma$
$\gamma ::= X_\chi\phi \mid X_\tau\phi \mid \phi\ {}_\chi U\ \phi \mid \phi\ {}_\chi U_{\chi'}\ \phi$

where χ, χ' range over action formulae, E and A are path quantifiers, X and U are *next* and *until* operators respectively.

Let $TS = (S, Act, D, s_0)$ be a LTS. *Satisfaction* of a state formula ϕ (path formula γ) by a state s (path σ), notation $s \models_{TS} \phi$ ($\sigma \models_{TS} \gamma$) is given inductively by :

$s \models_{TS} \mathit{tt}$		always;
$s \models_{TS} \phi \wedge \phi'$	iff	$s \models_{TS} \phi$ and $s \models_{TS} \phi'$;
$s \models_{TS} \neg\phi$	iff	not $s \models_{TS} \phi$;
$s \models_{TS} E\gamma$	iff	there exists a path $\sigma \in \Pi(s)$ such that $\sigma \models_{TS} \gamma$;
$s \models_{TS} A\gamma$	iff	for all maximal paths $\sigma \in \Pi(s)$, $\sigma \models_{TS} \gamma$;
$\sigma \models_{TS} X_\chi\phi$	iff	$\lvert\sigma\rvert \geq 1$ and $\sigma(2) \in D_{\kappa(\chi)}(\sigma(1))$ and $\sigma(2) \models_{TS} \phi$;
$\sigma \models_{TS} X_\tau\phi$	iff	$\lvert\sigma\rvert \geq 1$ and $\sigma(2) \in D_{\{\tau\}}(\sigma(1))$ and $\sigma(2) \models_{TS} \phi$;
$\sigma \models_{TS} \phi\ {}_\chi U \phi'$	iff	there exists $i \geq 1$ such that $\sigma(i) \models_{TS} \phi'$, and for all $1 \leq j \leq i-1$: $\sigma(j) \models_{TS} \phi$ and $\sigma(j+1) \in D_{\kappa(\chi)_\tau}(\sigma(j))$;
$\sigma \models_{TS} \phi\ {}_\chi U_{\chi'} \phi'$	iff	there exists $i \geq 2$ such that $\sigma(i) \models_{TS} \phi'$ and $\sigma(i) \in D_{\kappa(\chi')}(\sigma(i-1))$, and for all $1 \leq j \leq i-1$: $\sigma(j) \models_{TS} \phi$ and $\sigma(j) \in D_{\kappa(\chi)_\tau}(\sigma(j-1))$.

Several useful modalities can be defined, starting from the basic ones. In particular, we will write:

- $EF\phi$ for $E(\mathit{tt}\ {}_{\mathit{tt}}U\ \phi)$, and $AF\phi$ for $A(\mathit{tt}\ {}_{\mathit{tt}}U\ \phi)$; these are called the *eventually* operators.
- $EG\phi$ for $\neg AF\neg\phi$, and $AG\phi$ for $\neg EF\neg\phi$; these are called the *always* operators.

ACTL can be used to define *liveness* (something good eventually happen) and *safety* (nothing bad can happen) properties of reactive systems. In a branching time logic both liveness and safety properties could be divided into two classes: *universal* liveness (safety) properties and *existential* liveness (safety) properties. The former state that a condition holds at some (all) states of *all* computation paths. The latter state that a condition holds at some (all) states of *one* computation path. Moreover liveness properties can be better classified as in the following [17, 20]:

Termination properties: "a good thing happens at some states of a (all) computation(s)".
Recurrence properties: "a good thing happens at infinitely many states of a (all) computation(s)".
Persistence property: "a good thing happens at all but finitely many states of a (all) computation(s)".
We can also talk of *finite properties*, that state some condition on the finite initial part of the behaviour of the system.

2.3 Infinite state systems and logical properties

We know that all ACTL formulae are decidable on finite state transition systems and the linear time ACTL model checker [10] can be used to do this job. Hence, when we have a CCS description of a system and we want to prove on it ACTL properties, the labeled transition system associated to it needs to be built. This will be the model on which the satisfiability of the formulae will be checked. Problems, obviously, arise when the system to be modelled has an infinite state representation, due for example to the interplay between parallel composition and recursion operators.

As an example, let us consider the CCS definition of a bag containing two kinds of elements:

$$X = p1.(g1.nil|X) + p2.(g2.nil|X)$$

where p_1 and p_2 represent insertions and g_1 and g_2 deletions of the two kinds of elements, respectively. It is known that X is neither finite state nor context-free. Some typical properties of a bag could be requested to be checked on this specification, in order to validate it:
1) The bag is not a set, therefore it is possible to put twice the same value in the bag consecutively: $AFAX_{p_1}EX_{p_1}tt$.
2) It is possible, on all (but finitely many) states to do a *put* action immediately followed by a *get* action: $EFEG(EX_{p_1}EX_{g_1}tt)$.
3) There exists a computation path on which it is possible to do infinitely often *put* actions: $EGAF(EX_{p_1 \vee p_2}tt)$.
4) It is always possible to perform a put action: $AGEX_{p_1 \vee p_2}tt$.

3 Verification by approximations

Let us first present a syntactic characterization, as ACTL formulae, of the logical properties we will deal with. We then introduce the general notion of chain of finite approximations of the transition system of a term p. Finally, we introduce a notion of approximation suitable to prove liveness properties.

3.1 Temporal properties

Definition 3.1 (Positive formula) *We say that π' is a positive formula if it is an ACTL formula without negations.*

Definition 3.2 (Liveness property) *We say that ψ is a liveness property if one of the following holds, where π' is a positive formula:*

- $\psi = AF\pi'$ *or* $\psi = EF\pi'$ *(*termination property*)*
- $\psi = AFAG\pi'$, $\psi = EFAG\pi'$, $\psi = AFEG\pi'$ *or* $\psi = EFEG\pi'$ *(*persistence property*)*
- $\psi = AGAF\pi'$, $\psi = EGAF\pi'$, $\psi = AGEF\pi'$ *or* $\psi = EGEF\pi'$ *(*recurrence property*)*

Definition 3.3 (Finite property) *We say that σ is a finite property if it can be expressed by an ACTL formula defined by the following grammar:* $\sigma ::= tt \mid \sigma \wedge \sigma \mid \sigma \vee \sigma \mid \neg\sigma \mid E\gamma \mid A\gamma$
$\gamma ::= X_\chi\sigma \mid X_\tau\sigma$

Definition 3.4 (Positive finite property) *We say that π is a positive finite property if it is a finite property without negations.*

Definition 3.5 (Safety property) *We say that θ is a safety property if $\theta = AG\pi$ or $\theta = EG\pi$ and π is a positive finite property.*

The given syntactical presentation of liveness and safety properties does not obviously cover all the liveness and safety properties expressible by means of all the ACTL operators as the negation operator. Indeed, negation makes the syntactic classification of formulae difficult. Following this classification, we have that properties 1) to 3) of the bag example are liveness properties, while 4) is a safety one.

Finite, liveness and safety properties are decidable on a finite state LTS. In general, while finite properties are provable, liveness (including termination, persistence and recurrence) and safety properties can be undecidable for a non-finite state term p.

3.2 Approximation chains

Given a CCS term p, we define chains of finite LTSs which more and more accurately simulate the behaviour of SOS(p). Since each LTS in a chain is finite proof checking methodologies for finite LTSs can be used. First we define in the most general way the concept of approximation chain. In the following we denote, with $\mathcal{T}$ and T, the set of all LTSs and a generic LTS, respectively.

Definition 3.6 (Approximation chain) *Let $\preceq$ a preorder over $\mathcal{T}$. We say that T_1 approximates by $\preceq$ ($\preceq$-approximates) T_2 iff $T_1 \preceq T_2$. Given a term p, a chain $\{T_i(p) | i \geq 0\}$ on $(\mathcal{T}, \preceq)$ is called* approximation chain for p by $\preceq$ *($\preceq$-approximation chain) iff:*

- *for each i, $T_i(p)$ is finite;*
- *for each i, $T_i(p) \preceq T_{i+1}(p)$;*
- *$\sqcup\{T_i(p)\} \sim SOS(p)$.*

Note that, if we have a finite approximation chain $\{T_i(p) | r \geq i \geq 0\}$, then $T_r(p) \sim SOS(p)$.

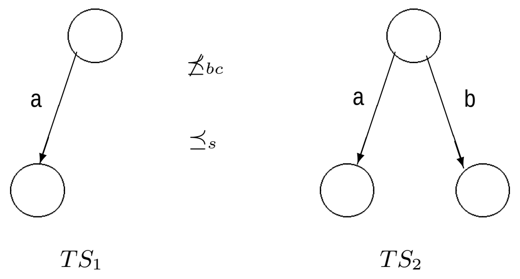


Figure 2: Simulation vs. BC-simulation.

Definition 3.7 (Properties preserved by $\preceq$) *A preorder $\preceq$ preserves a property ϕ if whenever T_1 verifies ϕ and $T_1 \preceq T_2$ then T_2 verifies ϕ.*

The above definitions allow us to define a procedure for proving the validity of a property on an infinite state-system, by checking the property on the elements of an approximation chain, starting from the first one, until we find that the property is verified. The procedure is sound if the chain preserves the property, i.e. it must happen that, if we are able to prove ϕ on an element of the chain, we can assert the validity of ϕ on $SOS(p)$. This means that the property must be monotonic on the preorder. The first result we show is that simulation, from now on denoted by $\preceq_s$, is not suitable to prove all liveness properties.

Proposition 3.1 *$\preceq_s$ does not preserve all liveness properties.*
Proof *Let us consider the following liveness property:*
Each path contains a state from which all the outcoming arcs are labelled by a, *expressed by $(AFAX_a tt)$ and the transition systems TS_1 and TS_2 in Figure 2.*
We have that $TS_1 \preceq_s TS_2$, but TS_1 verifies the property and TS_2 does not.

In order to manage all liveness properties, we now introduce a stronger notion of simulation between transition systems. This notion, in contrast to simulation, permits the definition of approximation chains that *preserve* the branching structure, that is, for each approximation, if a node has been exploded, all its branches have been developed.

Definition 3.8 (Branching Complete Simulation) *Let $TS_1 = (S_1, T_1, D_1, s_{0_1})$ and $TS_2 = (S_2, T_2, D_2, s_{0_2})$ be transition systems and let $s_1 \in S_1$ and $s_2 \in S_2$.*
s_2 BC-simulates s_1 *if there exists a* strong BC-simulation *that relates s_1 and s_2. $\mathcal{R} \subseteq S_1 \times S_2$ is a strong BC-simulation if $\forall (s_1, s_2) \in \mathcal{R}$, $\mu \in T_1 \cup T_2$,*

- *$s_1 \xrightarrow{\mu}_1 s_1'$ implies $\exists s_2' : s_2 \xrightarrow{\mu}_2 s_2'$ and $(s_1', s_2') \in \mathcal{R}$.*
- *$s_2 \xrightarrow{\mu}_2 s_2'$ implies either $s_1 \not\rightarrow_1$ or $s_1 \xrightarrow{\mu}_1 s_1'$ and $(s_1', s_2') \in \mathcal{R}$.*

TS_2 BC-simulates TS_1 *($TS_1 \preceq_{bc} TS_2$) if a branching complete simulation $\mathcal{R}$ exists such that $(s_{0_1}, s_{0_2}) \in \mathcal{R}$.*

It is easy to see that $TS_1 \preceq_{bc} TS_2$ implies $TS_1 \preceq_s TS_2$, but the converse is not true in general. For example, TS_2 does not BC-simulate TS_1 in Figure 2.
The following proposition holds.

Proposition 3.2 *$\preceq_{bc}$ is a preorder.*
Proof Sketch. *It is reflexive and transitive.*

The notion of approximation chain based on BC-simulation preserves the branching structure of the transition systems all along the chain. This allow us to prove properties not provable on a chain based on simulation. One of the main results of the paper is the following:

Proposition 3.3 *$\preceq_{bc}$ preserves liveness properties.*
Proof sketch *By structural induction on the structure of the liveness formulae and taking into account that the liveness properties are defined on a positive fragment of ACTL and that the BC-simulation forces the simulating transition system to exactly mantain all the (bisimilar) branches of the simulated one, if any.*

It is now easy to relate approximation chains, based on BC-simulation, with liveness properties. The following proposition is the basis of our verification method.

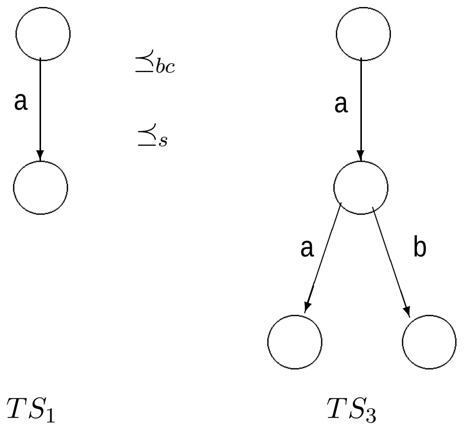


Figure 3: Simulation and BC-simulation.

Proposition 3.4 *Let p be a term and $\{T_i(p)\}$ a $\preceq_{bc}$-approximation chain for p. If ϕ is a liveness property, it holds that: if $s_0 \models_{T_i(p)} \phi$ for some i, then $s_0 \models_{SOS(p)} \phi$.*
Proof. *It follows by proposition 3.3.*

For safety properties, the following proposition holds :

Proposition 3.5 *$\preceq_s$ and $\preceq_{bc}$ do not preserve safety properties.*
Proof *Let us consider the safety property* "It is always possible to perform an action a " *, expressed by $AGAX_a tt$, and the transition systems TS_1 and TS_3 in Figure 3. Then $TS_1 \preceq_{bc} TS_3$ and $TS_1 \preceq_s TS_3$, but TS_1 verifies the property and TS_3 does not.*

This means that we cannot define a semidecision procedure based on approximation chains for the satisfaction of safety properties; on the converse, the following proposition gives a method to prove that a safety property does not hold for $SOS(p)$.

Proposition 3.6 *Let p be a term and $\{T_i(p)\}$ a $\preceq_{bc}$-approximation chain for p. If ψ is a safety property, it holds that: if $s_0 \not\models_{T_i(p)} \psi$ for some i, then $s_0 \not\models_{SOS(p)} \psi$.*
Proof sketch *For duality from Prop. 3.4*

Prop. 3.6 corresponds, according to the definition of safety properties in [20, 17], to say that if a finite approximation of a term p violates the property, then p itself violates the property.

Let us now consider finite properties. The following holds:

Proposition 3.7 *$\preceq_s$ does not preserve all finite properties.*
Proof sketch *Consider the finite property:* Each path starts with an action *a ($AX_a tt$), with TS_1 and TS_2 of Figure 2. We have that $TS_1 \preceq_s TS_2$, but TS_1 verifies the property and TS_2 does not.*

Since finite properties represent a particular class of liveness properties we have a semidecision procedure for testing the validity of these properties by using approximation chains based on $\preceq_{bc}$. We can do more, as one should have expected, and provide a decision procedure for finite properties. To this end, we furtherly constrain our chains. Let us consider, for example, the following finite property for $SOS(p)$ for some p:
All paths start with the action b and contain at least an action a as a second action ($AX_b EX_a tt$).

Approximation chains based on $\preceq_{bc}$ are not suitable to give a positive or negative answer if $SOS(p)$ is infinite: in fact a new path of length 2 may appear in whatever element of the chain. The property is decidable if, instead, each transition system $T_i(p)$ of the chain grows on all possible paths with respect to $T_{i-1}(p)$. This suggests the following notion:

Definition 3.9 (Transition system path-approximation) *Let TS_1 and TS_2 be transition systems. We say that TS_1 is an* n-path-approximation *of TS_2 ($TS_1 \preceq_n TS_2$) if*

- *$TS_1 \preceq_{bc} TS_2$;*
- *either $TS_1 \sim TS_2$ or the paths of length $\leq n$ of TS_1 and TS_2 coincide.*

We can now state the following:

Proposition 3.8 *Let π be a finite property of depth n, that is with only n nested next operators, and $\{T_i(p)\}$ a $\preceq_{bc}$-approximation chain for a term p such that $T_i(p) \preceq_i SOS(p)$ for each i. Then $s_0 \models_{SOS(p)} \pi$ iff $s_0 \models_{T_n(p)} \pi$.*
Proof sketch *We have that $T_n(p)$ has all the paths of length n of $SOS(p)$.*

4 How to build approximations

In this section, we present some ways of constructing approximation chains. In order to obtain correct approximations for a term p, the idea is to derive p using the operational semantics until some stopping condition, thus obtaining a partial transition system, which is furtherly expanded to obtain the successive elements of the chain. The first chain we present, described in the following sub-secton, is based on the standard SOS semantics. In order to obtain better approximations, we then introduce a second chain, which is based on a different semantics, able to produce "more expressive" transition systems.

4.1 SOS approximations

Definition 4.1 ($\{M_i(p)\}$) *Given a term p, the chain $\{M_i(p) = (S_{M_i}, Act, D_{M_i}, s_0)\}$ is inductively defined as follows:*

- *$M_0(p) = (\{p\}, Act, \{\}, p)$*
- *$M_{i+1}(p) = (S_{M_{i+1}}, Act, D_{M_{i+1}}, p)$ where*
  - *$S_{M_{i+1}} = S_{M_i} \cup \{q | p \in S_{M_i}$ and $\mu \in Act$ exist such that $p \xrightarrow{\mu}_{SOS} q\}$;*
  - *$D_{M_{i+1}} = D_{M_i} \cup \{(p, \mu, q) | p \in S_{M_i}$ and $\mu \in Act$ exist such that $p \xrightarrow{\mu}_{SOS} q\}$.*

Informally, $M_0(p)$ has the only state p without transitions and $M_{i+1}(p)$, $i \geq 0$, is obtained from $M_i(p)$, by adding to the states (and the related transitions) of $M_i(p)$ all those states reachable from them with only one action. The following proposition holds:

Proposition 4.1 *Given a term p, the chain $\{M_i(p)\}$ is a $\preceq_{bc}$-approximation chain for p.*
Proof sketch. *By induction on the length of the chain and by definig suitable BC-simulations.*

Actually, the chain $\{M_i(p)\}$ is the simplest chain derivable from SOS(p) which is a $\preceq_{bc}$-approximation chain. In fact the simpler approximation chain which at any step adds a single new transition to the previous element of the chain, is not a $\preceq_{bc}$-approximation chain.

Example 4.1 Let us now reconsider the bag example of section 2.3, and try to prove the properties on the chain $\{M_i(X)\}$. Since $\{M_i(X)\}$ is a $\preceq_{bc}$-approximation chain, it preserves all properties from 1) to 3) and does not preserve the safety property 4). Thus, if we find that an approximation $M_i(X)$ verifies a property among 1) and 3), we prove that the property holds for the bag (i.e. $SOS(X)$). $M_0(X)$ is given by a transition system with only one state, i.e. X itself, while $M_1(X)$ and $M_2(X)$ are represented in Figures 4 and 5 respectively.

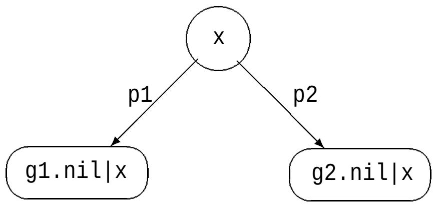


Figure 4: $M_1(X)$

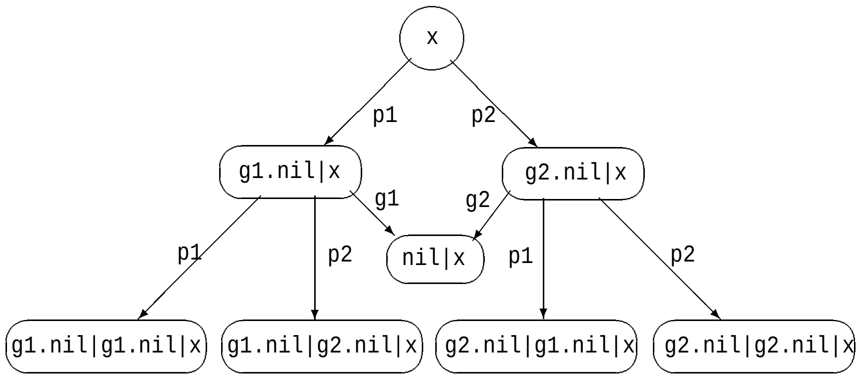


Figure 5: $M_2(X)$

We have that property 1) is not satisfied by $M_1(X)$, it is satisfied by $M_2(X)$ and thus it is true for the bag. Moreover, property 4) is verified by $M_1(X)$, but this does not mean that it is true for the bag, since it is a safety property. Properties 2) and 3) are not verified by $M_1(X)$ neither by $M_2(X)$. It is easy to see that these properties are not verified by any $M_i(X)$, for each i. In fact their satisfiability implies detecting a cycle in the transition system: this cycle will never appear in the chain $\{M_i(X)\}$.

Thus, if we use this chain to approximate $SOS(X)$, these properties are not provable, while they hold for $SOS(X)$. Nothing can instead be asserted about property 4), following proposition 3.6. The following proposition states that each M_i is a $\preceq_i$-approximation of $SOS(p)$, i.e. the size of the transition system grows.

Proposition 4.2 *Given a term p, for each $i \geq 0$, $M_i(p) \preceq_i SOS(p)$.*
Proof. *By proposition 4.1 it holds that $M_i(p) \preceq_{bc} SOS(p)$. Moreover, by induction on the length of $\{M_i(p)\}$, we have by definition that $M_i(p)$ has all the paths of length less or equal to i.*

As a consequence, using $\{M_i(p)\}$ we can decide any finite property of depth n of a term p: it sufficies to check the property on $M_n(p)$.

4.2 SS Approximations

In this section, we present a way of approximating $SOS(p)$ based on a different operational semantics, which allows us to prove a greater set of properties than those proved by $\{M_i(p)\}$. In [9] the semantics SS was defined, which is more abstract than SOS, since the SS rules have built in some behavioural equivalence axioms, i.e. they accomplish some simplifications on the terms during the derivations, with the purpose of obtaining, if possible, a finite-state transition system for p. The rules of SS are such that SS(p) is strongly equivalent to SOS(p). The definition of SS, whose rules are shown in figure 6, is based on the following considerations. Given the CCS syntax, those operators that, in presence of recursion, would give rise to the derivation of growing terms (and therefore to an infinite number of derivations) are parallel composition, restriction and relabelling. For restriction and relabelling, in a language with finite action set, the unlimited growth of terms can be prevented by using suitable inference rules. In fact, successive, possibly intermixed, occurrences of restriction and relabelling can be reduced to only one restriction, followed by only one relabelling. Moreover, the parallel operator can be deleted as soon as one of the two arguments terminates, i.e. is equivalent to *nil*. The SS inference rules accomplish these strong equivalence preserving simplifications during the derivation. The following notation is used in the rules:

$p\backslash\backslash A =$	$p\backslash A$, if $p \neq q\backslash B, p \neq q[f]$	$q\backslash A \cup B$, if $p = q\backslash B$
	$q\backslash f^1(A)[f]$, if $p = q[f], q \neq r\backslash B$	$q\backslash f^1(A) \cup B[f]$, if $p = q\backslash B[f]$
$p[[f]]=$	$p[f]$, if $p \neq q[g]$	$q[f \circ g]$, if $p = q[g]$

S-Act=Act S-Con=Con S-Sum=Sum

$$\textbf{S-Par}_1 \; \frac{p \xrightarrow{\mu} p', \mathit{not}\ p' \not\to}{p|q \xrightarrow{\mu} p'|q \text{ and } q|p \xrightarrow{\mu} q|p'} \qquad \textbf{S-Par}_2 \; \frac{p \xrightarrow{\mu} p',\ p' \not\to}{q|p \xrightarrow{\mu} q \text{ and } q|p \xrightarrow{\mu} q}$$

$$\textbf{S-Com}_1 \; \frac{p \xrightarrow{\alpha} p',\ q \xrightarrow{\overline{\alpha}} q',\ \mathit{not}\ p' \not\to\ \mathit{and\ not}\ q' \not\to}{p|q \xrightarrow{\tau} p'|q'} \qquad \textbf{S-Com}_2 \; \frac{p \xrightarrow{\alpha} p',\ q \xrightarrow{\overline{\alpha}} q',\ p' \not\to}{p|q \xrightarrow{\tau} q' \text{ and } q|p \xrightarrow{\tau} q'}$$

$$\textbf{S-Res} \; \frac{p \xrightarrow{\mu} p', \mu, \overline{\mu} \notin A}{p\backslash A \xrightarrow{\mu} q\backslash\backslash A} \qquad \textbf{S-Rel} \; \frac{p \xrightarrow{\mu} q}{p[f] \xrightarrow{f(\mu)} q[[f]]}$$

Figure 6: The SS rules

The chain $\{N_i(p)\}$ is defined in a similar way to $\{M_i(p)\}$, but using the above rules:

Definition 4.2 (**$\{N_i(p)\}$**) *Given a term p, the chain $\{N_i(p) = (S_{N_i}, Act, D_{N_i}, s_0)\}$ is inductively defined as follows:*

- $N_0(p) = (\{p\}, Act, \{\}, p)$

- $S_{N_{i+1}}(p) = (S_{N_{i+1}}, Act, D_{N_{i+1}}, p)$ *where*
  - $S_{N_{i+1}} = S_{N_i} \cup \{q | p \in S_{N_i}$ *and* $\mu \in Act$ *exist such that* $p \xrightarrow{\mu}_{SS} q\}$*;*
  - $D_{N_{i+1}} = D_{N_i} \cup \{(p, \mu, q) | p \in S_{N_i}$ *and* $\mu \in Act$ *exist such that* $p \xrightarrow{\mu}_{SS} q\}$*.*

If we reconsider the bag example, Figures 7, 8 show $N_1(X)$ and $N_2(X)$, respectively.

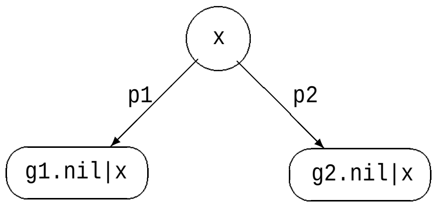


Figure 7: $N_1(X)$

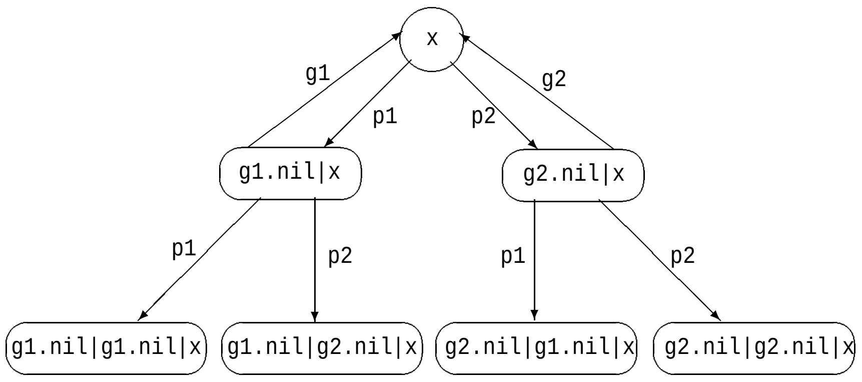


Figure 8: $N_2(X)$

The following proposition holds:

Proposition 4.3 *Given a term p,*

- *the chain* $\{N_i(p)\}$ *is a* $\preceq_{bc}$*-approximation chain for p;*
- *for each* $i \geq 0$, $N_i(p) \preceq_i SOS(p)$

Proof sketch *Analogous to the proof of proposition 4.1 and 4.2 and since* $SOS(p) \sim SS(p)$*.*

If we check the properties 1) ... 4) on the chain $\{N_i(p)\}$, we have the same results as with $\{M_i(p)\}$ for 1) and 4), but $N_2(X)$ satisfies properties 2) and 3), which are then true for the bag, while their validity is not provable on the chain $\{M_i(X)\}$.

The following proposition relates the two chains we have introduced.

Proposition 4.4 *Given a CCS term p, for each* $i \geq 0$, $\exists j$ *such that* $M_i(p) \preceq_{bc} N_j(p)$*.*
Proof. *The finite paths are equal in* $M_i(p)$ *and* $N_i(p)$*, since they are both* $\preceq_i SOS(p)$*. Moreover,*

it holds that: $\forall s \in S_{M_i}, \exists s' \in S_{N_i}$ such that $s \sim s'$ and $length(s') \leq lengh(s)$, where $length(t)$ denotes the number of operators occurring in the term t. This holds since terms generated by SS are "shorter" than terms generated by SOS. Consider an infinite path in $M_i(p)$, i.e. a path leading from a state $s \in S_{M_i}$ to itself and take n equal to the number of terms t equivalent to s and such that $length(t) \leq lengh(s)$. Take $j = i + n$.

Note that the converse of the above proposition is not true: if we consider the bag example, no $M_i(X)$ exists which is $\preceq_{bc} N_2(X)$.

5 Suitability of approximation chains

Let us consider a liveness property ϕ and a $\preceq_{bc}$-approximation chain $\{T_i(p)\}$ for a term p. Proposition 3.4 above ensures that, if we are able to prove ϕ on an element of the chain, we can assert the validity of ϕ on $SOS(p)$. Thus an algorithm to check the validity of a liveness property is that of checking it on the elements of the chain, starting from the first one, until we find that the property is verified. But the converse of proposition 3.4 is not true in general: if a liveness property ϕ is verified on $SOS(p)$, this does not imply that it is true for some $\{T_i(p)\}$. Thus, given an approximation chain, the above algorithm (which checks a liveness property on the elements of the chain) is not in general a semidecision procedure for the validity of a formula. This is the case of the chain $\{M_i(p)\}$ and the properties 2) and 3) of our example above. Moreover, different approximation chains for the same term can be used to check different sets of properties, in the sense that, given a property ϕ, it is possible that the above algorithm is a semidecision procedure for ϕ if using a chain, while it cannot be used to semidecide the validity ϕ with another chain. This suggests a comparison criterion on the suitability of approximation chains for proving liveness properties.

Definition 5.1 (Checkable properties) *Let be given a term p and a $\preceq_{bc}$ approximation chain $\{T_i(p)\}$. We say that a liveness property ϕ is checkable by $\{T_i(p)\}$ if*

- *either ϕ is not verified by $SOS(p)$ or*
- *$(T_r(p) \in \{T_i(p)\})$ exists such that $s_0 \models_{T_r(p)} \phi$.*

The set of checkable properties of p by $\{T_i(p)\}$ is denoted as $\mathcal{P}_{T_i}(p)$.

Thus $\mathcal{P}_{T_i}(p)$ includes the properties for whose validity there is a semidecision procedure using $\{T_i(p)\}$.

Definition 5.2 (Suitability of approximation chains) *Let be given a term p and two $\preceq_{bc}$ approximations chains $\{T_i(p)\}$ and $\{S_i(p)\}$. We say that $\{T_i(p)\}$ is* more suitable or equal for p *than $\{S_i(p)\}$ if $\mathcal{P}_{S_i}(p) \subseteq \mathcal{P}_{T_i}(p)$. Moreover, $\{T_i(p)\}$ is* strictly more suitable for p *than $\{S_i(p)\}$ if $\mathcal{P}_{S_i}(p) \subset \mathcal{P}_{T_i}(p)$.*

Note that the notion of suitability of approximation chains is different from a notion considering the "growing rate" of the chains. Given, for example, an approximation chain $\{T_i(p)\}$, let us consider the chain containing a subset of the elements of $\{T_i(p)\}$, for example the elements of even position, i.e. $\{S_i(p)\} = \{T_0(p), T_2(p), T_4(p), \cdots\}$. We have that $\{S_i(p)\}$ grows faster than $\{T_i(p)\}$, but it is not more suitable. As a consequence of the above definitions and propositions 4.4, we can state the following propositions:

Proposition 5.1 *For each term p, $\mathcal{P}_{M_i}(p) \subseteq \mathcal{P}_{N_i}(p)$.*
Proof sketch. *By proposition 4.4.*

The following proposition states that the converse of proposition 5.1 is not true in general.

Proposition 5.2 *Given a term p, $\mathcal{P}_{M_i}(p) \subset \mathcal{P}_{N_i}(p)$, i.e. $\{N_i(p)\}$ is strictly more suitable than $\{M_i(p)\}$.*
Proof sketch *Properties 2) and 3) in the bag example are checkable by $\{N_i(p)\}$ but not by $\{M_i(p)\}$.*

6 Implementation in the JACK environment

The JACK system [3] is a verification environment for process algebra description languages. It is able to cover a large extent of the formal software development process, such as rewriting techniques, behavioural equivalence proofs, graph transformations, and (ACTL) logic verification. In JACK a particular description format is used to represent TSs, the so called *format commun fc2*, that has been proposed as standard format for automata [18]. The ACTL model checker was built on the basis of an algorithm similar to that of the EMC model checker [5], so it guarantees model checking of an ACTL formula on a TS in a linear time complexity [10].

The JACK environment has been extended with a tool to build the chain $\{N_i(p)\}$. We now describe the methodology for proving properties. Let be given a CCS term p and a list of ACTL formulae to be checked on it. A verification session has the following steps:

1. The term is input to JACK. If the term satisfies the finiteness condition of the transition system generator inside JACK, a corresponding transition system TS is built and the list of ACTL formulae is checked on it. The session terminates.

2. If the syntactic finiteness conditions are not satisfied, then we call the chain generator of JACK. Once obtained the first approximation $N_1(p)$, we put $TS := N_1(p)$.

3. The list of ACTL formulae is input to the model checker which checks them on TS. If $N_{i+1}(p) = TS$, the session terminates, since $TS \sim SOS(p)$. Otherwise, the results of the model checker are analyzed according to propositions 3.4, 3.6 and 3.8. This means that, possibly, a new approximation is built, i.e. $TS := N_{i+1}(p)$ and we repeat step 3.

Acknowledgement

We wish to acknowledge Luigi Polverini for its work on the implementation of the NSS approximation generator, Rocco De Nicola and Gioia Ristori for interesting discussion about the topics of this paper.

References

[1] J. C. M. Baeten, J. A. Bergstra, J. W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. Journal of ACM 40,3,1993, pp. 653-682.

[2] G. Bruns. A practical technique for process abstraction. CONCUR'93, LNCS 715, pp. 37-49.

[3] A. Bouali, S. Gnesi, S. Larosa. The integration Project for the JACK Environment. Bulletin of the EATCS, n.54, October 1994, pp.207-223.

[4] O. Burkart, B. Steffen. Pushdown processes: Parallel Composition and Model Checking. Proceedings, CONCUR 94, LNCS 836, 1994, pp.98-113.

[5] E.M. Clarke, E.A. Emerson, A.P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. ACM Toplas, 8 (2), 1986, pp. 244-263.

[6] E.M.Clarke, O.Grumberg, D.E.Long. Model Checking and Abstraction. ACM Toplas, 16 (5), 1994, pp.1512-1542.

[7] R. Cleaveland, J. Parrow, B. Steffen. The Concurrency Workbench. Proceedings of Automatic Verification Methods for Finite State Systems. Lecture Notes in Computer Science 407, Springer-Verlag, 1990, pp. 24-37.

[8] D.Dams, O.Grumberg, R.Gerth. Automatic Verification of Abstract Interpretation of Reactive Systems: Abstractions Preserving ∀CTL*, ∃CTL*, CTL*. IFIP working conference on Programming Concepts, Methods and Calculi (PROCOMET'94), 1994.

[9] N. De Francesco, P. Inverardi. Proving Finiteness of CCS Processes by Non-standard Semantics. Acta Informatica, 31 (1), 1994, pp. 55-80.

[10] R. De Nicola, A. Fantechi, S. Gnesi, G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. Computer Network and ISDN systems, Vol. 25, No.7, 1993, pp 761-778.

[11] R. De Nicola, F. W. Vaandrager. Action versus State based Logics for Transition Systems. Proceedings Ecole de Printemps on Semantics of Concurrency. LNCS 469, 1990, pp. 407-419.

[12] J.C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, J. Sifakis. A Toolbox for the Verification of LOTOS Programs. 14th ICSE, Melbourne, 1992, pp. 246-261.

[13] J. C. Godskesen, K. G. Larsen, M. Zeeberg. TAV Users Manual. Internal Report, Aalborg University Center, Denmark, 1989.

[14] H. Hungar, B. Steffen. Local Model Checking for Context-Free Processes. Proceedings, ICALP 93, LNCS 700, 1993, pp.593-605.

[15] H. Hungar. Local Model Checking for Parallel Composition of Context-Free Processes. Proceedings, CONCUR 94, LNCS 836, 1994, pp.114-128.

[16] H. Huttel, C Stirling. Actions speak louder than words: Proving Bisimilarity for Context Free Processes. LICS 91, IEEE Computer Society Press, 1991, pp. 376-386.

[17] E. Kindler. Safety and Liveness Properties: A Survey. Bulletin of the EATCS, 53, 1994, pp.268-272.

[18] E. Madelaine. Verification Tools from the Concur Project. Bulletin of EATCS 47, 1992, pp. 110-120.

[19] E. Madelaine, D. Vergamini. AUTO: A Verification Tool for Distributed Systems Using Reduction of Finite Automata Networks. FORTE '89, North-Holland, 1990, pp. 61-66.

[20] Z. Manna, A. Pnueli. The Anchored Version of the Temporal Framework, Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. Lecture Notes in Computer Science 354, Springer-Verlag, 1989, pp. 201-284.

[21] R. Milner. Communication and Concurrency. Prentice Hall, 1989.

[22] D. Taubner. Finite Representations of CCS and TCSP Programs by Automata and Petri Nets. LNCS 369, 1989.

On Automatic and Interactive Design of Communicating Systems[*]

Jürgen Bohn Stephan Rössig

FB Informatik, C.v.O. Universität Oldenburg
Postfach 2503, 26111 Oldenburg, Germany[†]

Abstract

This paper presents a transformational approach to the design of distributed systems where environment and concurrently running components communicate via synchronous message passing along directed channels. System specifications that combine trace-based with state-based reasoning are gradually modified by application of transfromation rules until occam-like programs are achieved finally. We consider interactive and automatic aspects of such a design process and illustrate our approach by sketching the development of a shared register implementation.

1 Introduction

The design of provable correct software requires formal methods whose usage should be assisted by suitable tools. Following a transformational approach the design needs interactive user help when important design decisions have to be made. Nevertheless simple parts should be automated as far as possible. Ideally the user only guides the design process by indicating the design ideas which are then carried out automatically. Typically sequential implementations are more appropriate for automation while parallelization needs interaction to determine the intended program architecture.

Our approach deals with the transformational development of communicating systems in the mixed term language MIX which encompasses specification and programming notation. A formal refinement notion guarantees that starting from a specification of a desired system only correct implementations can be reached. As part of the ESPRIT Basic Research Action ProCoS a refinement calculus for communicating systems was developed in order to provide a constructive and mathematically sound way for bridging the gap between specifications and programs [Old91, Rös94]. We consider communicating systems as an approach to distributed computing that integrates the state transformation aspect of iterative programs in the sense of UNITY [CM88] and action systems [Bac90] with the CSP paradigm of synchronous message passing along communication channels. When designing such systems several different aspects like concurrency, communication, nondeterminism, deadlock, termination, divergence and assignment to variables have to be considered. A state-trace-readiness semantics in a specification-oriented fashion provides the necessary power to express such properties and concepts. Additionally it induces immediately a refinement relation which is used to define correctness of system transformations.

[*]This research was partially supported by the CEC with the ESPRIT Basic Research Project No. 7071 ProCoS II and by the German Ministry of Research and Technologie (BMFT) as part of the project KORSO (Korrekte Software) under grant No. 01 IS 203 N.

[†]{bohn,roessig}@informatik.uni-oldenburg.de

The rest of this paper is structured as follows. Section 2 introduces our specification language SL and explains how SL constructs can be applied in order to specify a regular register with concurrent access. Section 3 considers basic aspects of a transformational approach to system design. Section 4 sketches major steps within the development process of a parallel architecture of sequential components implementing the regular register. Section 5 treats the derivations of sequential implementations by systematic exploitation of specifications. Section 6 deals with the automation of such systematic proceeding in order to decrease the degree of user interaction within the whole design process. A final section concludes this paper with a short discussion of the achieved results.

2 Specification Language SL

The specification language SL develops further the ProCoS specification language SL_0 [JROR90] that was designed to describe continuously running embedded systems communicating with their environment via synchronous message passing along directed channels. A communication along a channel takes place if both, system and environment, are ready for communication on that channel. A system is in a deadlock whenever it does not become ready for communication on at least one channel.

An SL specification provides several parts to describe such communicating systems in a constraint-oriented style. Syntactically a specification is a list of so-called basic items enclosed by `spec` – `end` brackets. The following sketches the basic ideas of these constructs using the general specification pattern given in figure 1. Afterwards a few more details

`spec`

$dir_c\ c$ [`of` ty_c] $mode_x\ x$ [`of` ty_x]	Δ
$ta =$ `trace` α_{ta} `in` re_{ta}	TA
$ca =$ `com` na_{ca} `write` $\overline{w}_{ca}$ `read` $\overline{r}_{ca}$ `when` wh_{ca} `then` th_{ca}	CA
`var` v `of` ty_v	lV
`chan` c [`of` ty_c]	lC
$ini =$ `initial` p_{ini} $sta =$ `stable` p_{sta} `for` $\overline{c}_{sta}$ $alw =$ `always` p_{alw} $est =$ `establish` p_{est} by $\overline{c}_{est}$	SR^{ini} SR^{sta} SR^{alw} SR^{est}

`end`

Figure 1: Specification format.

are discussed in the context of an example specification (cf. figure 3).

The *interface* Δ stresses a static view of the intended system by listing all entities which may be used for interaction with the environment. It consists of optionally typed declarations of external channels with associated direction indication (`input` or `output`) and of global variables with assoicated access mode (`write` or `read`-only).

Essentially the description of the intended dynamic behaviour is split into two parts in SL. The *trace part TA* specifies in which order communications may take place on the various channels. A trace assertion $ta \in TA$ describes a sequencing constraint for the channels of alphabet α_{ta} by giving a regular expression[1] re_{ta} over these channels. Several ordering aspects can be specified in a modular fashion by stating different trace assertions. Technically the so-called *trace language* $\mathcal{L}[\![\Delta, TA]\!]$ of the specification is that regular language over all channels which obeys all sequencing constraints simultaneously. The trace part prevents any communication trace of which the channel order does not belong to $\mathcal{L}[\![\Delta, TA]\!]$.

The *state part CA* relates single communications with the current system state. A communication assertion $ca \in CA$ consists of a channel name na_{ca}, two disjoint lists $\overline{w}_{ca}$ and $\overline{r}_{ca}$ of write and read-only variables, respectively, and two predicates. The when-predicate wh_{ca} over free variables $\overline{w}_{ca}, \overline{r}_{ca}$ disables channel na_{ca} for communication whenever wh_{ca} does not hold in the current state. The value of a communication refered to by $@na_{ca}$ as well as its effect on the system state are specified by the then-predicate th_{ca} over free variables $\overline{w}_{ca}, \overline{r}_{ca}, \overline{w}'_{ca}, @na_{ca}$. In the style of TLA [Lam94] and Z [Spi89] the unprimed variables refer to the values in the before state while the primed ones to those in the after state in which read-only variables $\overline{r}_{ca}$ must not change their values. Giving empty lists as well as predicate `true` is optional. Several communcation assertions for the same channel must be obeyed all together.

The use of a more operational formalization approach to the behaviour specification is supported by declarations of local variables lV and local channels lC. The various state restrictions SR provide a good basis for the integrated reasoning with state-based arguments as invariance and stable properties and with control flow arguments as initial state and establish properties. Technically these latter constraints could be replaced by certain more or less complex combinations of other basic items of which intuitive understanding is then often lost. The same holds for the always possible replacement of the trace part by additional local variables and communications assertions.

In [LG89] a good overview can be found about the various kinds of shared registers treated in the literature on distributed algorithms. According to the classification in [Lam86] we use as running example in this paper a regular register with a single reader and a single writer. In general a register stores values of a type V and the most recently written value shall be returned to the reader if its access does not overlap with a write. In the case of overlapping phases the regular behaviour guarantees that a read phase will return a value that was hold before or after one of write accesses. Figure 2 presents our

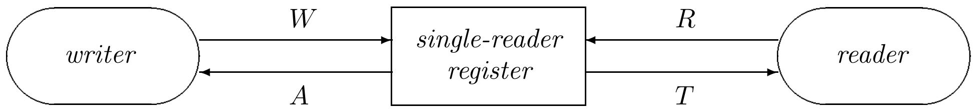


Figure 2: Register as communicating system

view of single-writer, single-reader register as communicating system. The writer initiates a writing phase by sending the new value along the input channel W. This phase ends when a corresponding acknowledgment signal is output on channel A. Conversely, the reader initiates a reading phase by sending a signal along the input channel R. This phase ends when a value is returned along the output channel T.

Figure 3 shows a complete SL specification of a regular register which is explained here shortly.[2] Here the interface consisting of the declarations of channels W, A, R, T together

[1] of an extended format additionally using `pref` as prefix closure operator

[2] Similar SL specifications of various registers are presented in [OR93, Rös94] together with a very detailed motivation of the single components.

```
RegisterSpec = spec input $W$ of $V$
                    output $A$ of signal
                    input $R$ of signal
                    output $T$ of $V$
          $ta_1$:   trace $W, A$ in pref$(W.A)^*$
          $ta_2$:   trace $R, T$ in pref$(R.T)^*$
                    var $new, old$ of $V$
                    var $C$ of $V$−set
          $ca_W$:   com $W$ write $new, C$ then $new' = @W \wedge C' = C \cup \{@W\}$
          $ca_A$:   com $A$ write $old$ read $new$ then $old' = new$
          $ca_R$:   com $R$ write $C$ read $new, old$ then $C' = \{new, old\}$
          $ca_T$:   com $T$ read $C$ then $@T \in C$
               end
```

Figure 3: Specification of a regular register.

with the trace part consisting of trace assertions ta_1, ta_2 formalize the value independent aspects. Communications along channels of type `signal` are used for synchronization purposes only but do not pass any message value. The trace assertions guarantee that initiating and ending communications of write as well as read phases always occur in alternating order starting with channels W and R, respectively.

To specify the values that may be returned we use local variables to store certain pieces of information. Variables *old* and *new* shall hold the before and the after value when a write access is active and otherwise that unique value which is stored in the register. Therefore a communication on W updates *new* with the newly received value what is formalized by conjunct $new' = @W$ in the then-predicate of ca_W. Analogously $old' = new$ expresses that *old* gets the value of *new* whenever an A signal ends up a write phase. The idea of the set-valued variable C is to collect all possible return values for a read access. Thus the value $@T$ to be passed by an ending T communication can be easily chosen from C. The initialization of variable C is performed by the initiating R signal of a read phase. Note that the values of *new* and *old* are equal outside of write phases and hence then-predicate $C' = \{new, old\}$ of ca_R leads to the only return value. Any write phase starting during a read phase overlaps this and thus every newly written value becomes a possible return value. Therefore each W communication enriches C by its communication value $@W$.

3 Transformational Implementation Design

To implement communicating systems we use an `occam`-like programming language PL [INM88]. Programs are terms constructed from the 0-ary operators `STOP`, `SKIP`, multiple assignments, input and output on channels, the unary operators `WHILE`, `var` and `chan` for describing loops and declaration of local variables and channels, and the operators `SEQ`, `IF`, `ALT` and `PAR` for sequential, conditional, alternative and parallel composition of lists of n arguments. Figure 4 shows a PL program which implements the register specification of figure 3. Analogously to specifications a program declares its interface to the environment explicitly. The `system` – `end` brackets emphasize that programs represent implementations of communicating systems.

Semantically a communicating system is viewed as pair $\Delta : P$ where the interface Δ declares the communication channels and global variables. The predicate P characterizes the dynamic behaviour of the system as the set of possible observations in a state-trace-readiness model. This model integrates a purely event-based readiness approach [OH86] and a standard input/output semantics into a specification-oriented semantics of which

```
    system input W of V
           output A of signal
           input R of signal
           output T of V
           chan u, d of V
           chan r of signal
           PAR[ var new of V
                  WHILE true do SEQ[ W?new, u!new, A! ] od,
                var x of V
                  WHILE true do ALT[ u?x-->SKIP, r?-->d!x ] od,
                var y of V
                  WHILE true do SEQ[ R?, r!, d?y, T!y ] od            ]
      end
```

Figure 4: Register Implementation.

details are presented in [Old91, Rös94]. A major reason for this semantics construction is the immediate presence of a refinement notion for communicating systems. A system $\Delta_1 : P_1$ *refines* a system $\Delta_2 : P_2$ if both ones have the same interface and if behaviour P_1 implies behaviour P_2:

$$\Delta_1 : P_1 \Rrightarrow \Delta_2 : P_2 \quad \text{iff} \quad \Delta_1 = \Delta_2 \text{ and } \models P_1 \;\Rightarrow\; P_2.$$

This definition encompasses a correctness notion *Prog* $\Rrightarrow$ *Spec* since specifications and programs are special representations of communicating systems.

Figure 5 shows a design sequence of a transformational implementation approach. Starting from an SL specification *Spec*, a PL implementation *Prog* is derived in a top-

$$\begin{tikzcd}
\mathit{Spec} \equiv & S_1 & \\
& \vdots \arrow[u, Rrightarrow] & \\
& S_n \arrow[u, Rrightarrow] & \equiv \mathit{Prog}
\end{tikzcd}$$

Figure 5: Implementation design sequence.

down fashion by iterated application of transformation rules such that the specification notation is gradually replaced by programming language constructs. The intermediate system expressions S_i are so-called *mixed terms* of the language MIX. This language encompasses specifications and programs as disjoint subsets and extends the application of every programming operator to arbitrary mixed terms. Moreover, there exist additional MIX specific operators in order to express intermediate stages of a system design much more conveniently. E.g. the treatment of the semantically complex PL operator `PAR` can be reduced within MIX to a combination of the simpler operators `SYN` and `HIDE` dealing separately with the aspects of multiple synchronization and of divergence raised by infinite internal communication.

Typically a transition step from mixed term S_i to S_{i+1} is performed by replacing some specification expression S in S_i by a mixed term T where the refinement $T \Rrightarrow S$ is guaranteed by a transformation rule. Then the overall implementation correctness follows from the transitivity of $\Rrightarrow$ and the monotonicity of all operators.

In easy cases a transformation step will replace a specification by a basic PL statement as e.g. an input or output communication or an assignment. Figure 13 below shows appropriate equivalences of specification and programming constructs. But more often more complex specifications have to be decomposed into mixed terms applying some composition operator to several simpler arguments. As typical example supporting this later kind of refinements, figure 6 shows a transformation rule which introduces the synchronization

$$\texttt{spec}\ \Delta\ TA\ CA\ lV\ \texttt{end}$$

$$\Uparrow$$

$$\begin{array}{l}\texttt{SYN[}\ \texttt{spec}\ \Delta_1\ TA_1\ CA_1\ lV_1\ \texttt{end}, \ldots, \\ \qquad \texttt{spec}\ \Delta_n\ TA_n\ CA_n\ lV_n\ \texttt{end} \quad \texttt{]}\end{array}$$

provided $\Delta = \bigcup_{||\,i=1}^{n}\Delta_i$, $TA = \bigcup_{i=1}^{n} TA_i$, $CA = \bigcup_{i=1}^{n} CA_i$, $lV = \biguplus_{i=1}^{n} lV_i$ and ...

Figure 6: Transformation rule SYN decomposition.

operator SYN. Generally a side condition "provided ..." restricts the applicability of the transformation rule and describes how the new mixed term is derived by syntactic modifications from the given one. In the example it is expressed that essentially the basic items of the given specification have to be shared out between the new argument specifications $\texttt{spec}\ \Delta_i\ TA_i\ CA_i\ lV_i\ \texttt{end}$ obeying some static semantic constraints.

For practical implementation designs a user needs guidance how to realize intuitive implementation ideas by application of such transformation rules. Here so-called design *strategies* provide recipes how to combine several rules in order to derive implementations in certain situations systematically or even mechanically. Data refinement, parallelization concepts or the development of specific sequential implementations are implementation concepts that can be supported by such strategies. As example we will later consider the automated synthesis of sequential programs based on the syntax directed transformation strategy SDT.

Tool support

An interesting consequence of basing all semantic reasoning on a uniform predicate language is that this reasoning comes close to what can be mechanically supported higher order logic theorem provers. In the German national research project KORSO one of the goals was to provide tool support for formal methods in software design. As part of this work a computer assisted validation of our semantical model was performed within the theorem prover Lambda [BR95]. To this end first the model was implemented in the higher order logic of Lambda [FM91, FFHM93] and various basic propositions about the model have been verified in the Lambda framework interactively. On the one hand this validation gives great confidence in soundness of the model as well as of its formalization in Lambda. On the other hand a basic transformation environment for communicating systems emerges from the verification of transformation rules since Lambda provides mechanisms for the representation of syntactic objects and supports their modification by rule applications. Particularly a transformational design processes is assisted by saving the design history, backtracking mechanisms, generation of proof obligations and a rule browser. Furthermore the tactics concept provides a possibility to perform algorithmic rule applications and automatic condition checking.

4 Parallel Register Architecture

Frequently specifications require that sometimes a system should be ready for communication on several channels. As in occam, the restriction to so-called input guards as arguments of the alternative operator ALT forces parallel implementations in such cases where an output channel must be together ready with at least one other channel.

In the regular register such a situation is present e.g. when a first communication took place. Initially the regular register must be ready for input channels W and R. Independently on the channel along which a communication is performed in the next situation common readiness is required for an input and an output channel. Hence an occam-like implementation of this register has to use concurrently running subcomponents which interact via internal communication. Obviously we shall choose one write manager component *WM* dealing with write access and a read manager *RM* serving the reader. Both these components require access to the value stored in the register. But PL does not provide shared variables and therefore a third component *SV* will play this role. Figure 7 indicates how these components are connected via local channels u, r, d of which usage is

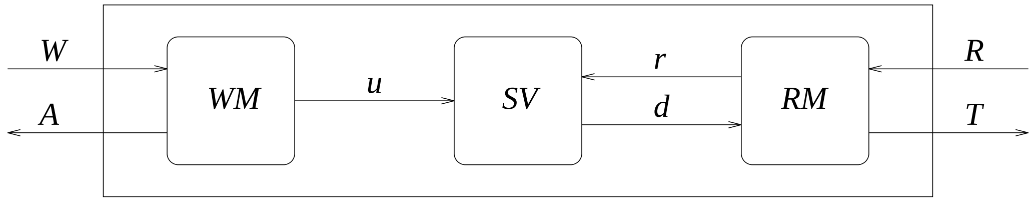


Figure 7: Intended process architecture.

as follows. After having received a new value along W the *WM* component updates the current register value by sending the new value along channel u to *SV* before the external acknowledgment on A is offered. *RM* serves a read request along R by sending an internal request along r to *SV*. The shared variable process immediately answers by delivering its actual value along channel d to *RM* which then transmits this value along T to the reader.[3][4] The specifications *WMspec*, *SVspec* and *RMspec* presented in figure 8 express this intuitive description of *WM*, *SV* and *RM* formally. They are designed by systematic transformation from the original specification shown in figure 3. Due to space limitations we list the major transformation steps towards the parallel decomposition in the following only.[5] Essentially these steps are motivated by the intended architecture which reflects the overall design ideas.

1. The local channels u, r, d are declared and their global communication behaviour is restricted according to the indicated communication order by modification of the trace assertions ta_1, ta_2 to

$$\texttt{trace}\ W, A, u\ \texttt{in}\ \texttt{pref}(W.u.A)^*$$
$$\texttt{trace}\ R, T, r, d\ \texttt{in}\ \texttt{pref}(R.r.d.T)^*$$
$$\texttt{trace}\ u, r, d\ \texttt{in}\ \texttt{pref}(u + r.d)^*\ .$$

2. To store the register value in *SV* and to hold the return value in *RM*, respectively, the state space is extended by variable declaration `var` x,y `of` V.

[3]The different treatment of write and read accesses to the shared variable process are necessary in order to allow a sequential implementation of *SV* because otherwise the problem of output channels in non singleton readysets would only be delayed.

[4]Note that this register implementation refines the regular specification properly because of a more deterministically chosen return value in the case of overlapping write and read accesses. Essentially this implementation realizes the stronger behaviour of an atomic register.

[5]In [OR93, Rös94] detailed explanations are given on the execution of such steps.

WMspec =

```
spec
  input W of V
  output A of signal
  output u of V
  trace W, A, u
        in pref(W.u.A)*
  var new of V
  com W write new
         then new' = @W
  com A
  com u read new
        then @u = new
 end
```

SVspec =

```
spec
  input u of V
  input r of signal
  output d of V
  trace u, r, d
        in pref(u + r.d)*
  var x of V
  com u write x
        then x' = @u
  com d read x
        then @d = x
 end
```

RMspec =

```
spec
  input R of signal
  output T of V
  output r of signal
  input d of V
  trace R, T, r, d
        in pref(R.r.d.T)*
  var y of V
  com T read y
        then @T = y
  com d write y
        then y' = @d
 end
```

Figure 8: Component specifications.

3. The original variables *old* and C are removed. To this end they are made auxiliary variables by introduction of appropriate state restrictions and strengthening of communication effects.
4. The local channel declarations are moved in front of the specification and thus they become global ones for the body. Since the trace part prevents infinite communication on local channels u, r, d only, their hiding from the specification does not introduce divergence.
5. The communication assertions of channels u and d are split in order to enable the intended distribution of local variables new, x, y onto the components *WM*, *SV*, *RM*.
6. Now the synchronous decomposition rule shown in figure 6 is applied and we end up with the mixed term

   ```
   chan u, d of V
   chan r of signal
   HIDE u, r, d in SYN[ WMspec, SVspec, RMspec ]
   ```

   with the component specifications of figure 8. Finally the operators HIDE and SYN are replaced by PAR because exactly all channels linking two argument systems of SYN are hidden.

The steps 2. and 3. perform a data refinement on the internal state space thereby proceeding quite systematically. A partial automation of this strategy would be very useful and seems to be possible. Generally executing the above steps and especially those performing the parallel decomposition requires a high degree of user interaction because the underlying rules allow various instantiations of their parameters leading to quite different refinements.

In contrast implementations of the three component specifications *WMspec*, *SVspec* and *RMspec* can be achieved by automatic synthesis of sequential programs. The conceptual basis of this automation and its implementation within `Lambda` are dealt with in the rest of this paper.

5 Designing Sequential Implementations

A notion of termination is essential when dealing with sequential implementations. In this section we present a suitable extension of SL to enable the description of termination. This new notion provides the basis for a transformational design of sequential implementations.

In order to refine a specification into a sequential composition of several specifications of reduced complexity, the circumstances have to be expressed, under which the control flow passes from one system to the next one. Therefore so-called *T-specifications* are introduced in SL. These are syntactically distinguished by `system` – `end` brackets instead of `spec` – `end` bracketed so-called S-specifications. Dependent on the trace part T-specifications may terminate in certain situations where the corresponding S-specifications would reach a deadlock. For a detailed comparison of S- and T-specifications see [Rös94]. A consequence of this differentiation is that an empty T-specification `system end` immediately terminates what is equivalent to the `SKIP` statement at the programming level. In contrast the empty S-specification `spec end` denotes an immediate deadlock which is represented in PL by `STOP`.

The following presents two transformation rules which relate S- and T-specifications. The first one in figure 9 allows in particular to switch from an S- to a T-specification which at most differs in the trace part. The trace language $\mathcal{L}[\![\Delta, \mathit{TA}]\!]$ of the S-specification must be equal to $pc\mathcal{L}[\![\Delta, \mathit{TA}_1]\!]$ which denotes the prefix closure of the trace language of the T-specification. When the refined system reaches the STOP it starves in a deadlock.

$$\texttt{spec}\ \Delta\ \mathit{TA}\ \mathit{CA}\ \texttt{end}$$

$$\Uparrow$$

$$\texttt{SEQ[system}\ \Delta\ \mathit{TA}_1\ \mathit{CA}\ \texttt{end, STOP]}$$

provided $\mathcal{L}[\![\Delta, \mathit{TA}]\!] = pc\mathcal{L}[\![\Delta, \mathit{TA}_1]\!]$.

Figure 9: Linking S- and T-specifications

Figure 10 shows a more general rule for the sequential decomposition of S-specifications. The first condition of this rule links the trace languages of the different specifications. The

$$\texttt{spec}\ \Delta\ \mathit{TA}\ \mathit{CA}\ \texttt{end}$$

$$\Uparrow$$

$$\texttt{SEQ[system}\ \Delta\ \mathit{TA}_1\ \mathit{CA}\ \texttt{end, spec}\ \Delta\ \mathit{TA}_2\ \mathit{CA}\ \texttt{end]}$$

provided	$\mathcal{L}[\![\Delta, \mathit{TA}]\!] = pc\mathcal{L}[\![\Delta, \mathit{TA}_1]\!] \cup \mathcal{L}[\![\Delta, \mathit{TA}_1]\!].\mathcal{L}[\![\Delta, \mathit{TA}_2]\!]$
and	$\mathcal{L}[\![\Delta, \mathit{TA}_1]\!]$ is prefix free.

Figure 10: Transformation rule sequential decomposition.

other condition "$\mathcal{L}[\![\Delta, TA_1]\!]$ is prefix free" guarantees a unique transition of the control flow from the first to the second argument in the mixed term.

In the following we concentrate on the implemention of T-specifications. The introduction of while-loops within the implementation design process simplifies T-specifications of which trace languages are iterations of prefix-free base languages. The body of an achieved while-loop is built up from the given specification by reducing the trace language to this base language as shown in the conditions of the while rule in figure 11. The termination condition $(\bigvee_{c \in \mathit{first}(\Delta, \mathit{TA}_1)} wh_c)$ is constructed from the when-predicates of those channels which are initially enabled by the trace language.

The decomposition of S-specifications into while-loops can be performed by an preparatory application of the rule in figure 9 and afterwards introducing a while-loop for the T-specification part. In case of a never terminating loop as first argument the sequential composition with `STOP` as second argument can be simplified using the rewriting rule:

$$\texttt{SEQ[WHILE true do}\ P\ \texttt{od},\ Q\ \texttt{]} \quad \rightarrow \quad \texttt{WHILE true do}\ P\ \texttt{od}\ .$$

$$\texttt{system}\ \Delta\ \mathit{TA}\ \mathit{CA}\ \texttt{end}$$

$$\Uparrow$$

$$\texttt{WHILE}\ \bigvee_{c \in \mathit{first}(\Delta,\, \mathit{TA}_1)} wh_c\ \texttt{do}\ \texttt{system}\ \Delta\ \mathit{TA}_1\ \mathit{CA}\ \texttt{end}\ \texttt{od}$$

provided $\mathcal{L}[\![\Delta,\, \mathit{TA}]\!] = \mathcal{L}[\![\Delta,\, \mathit{TA}_1]\!]^*$ and $\mathcal{L}[\![\Delta,\, \mathit{TA}_1]\!]$ is prefix free.

Figure 11: Transformation rule loop decomposition.

Another way of decomposing a specification into several ones with simpler trace languages are disjunctive decompositions thereby introducing an ALT or IF operator. Figure 12 shows a transformation rule for alternative decomposition which splits a T-specification into k subspecifications, where k is the number of that interface channels that occur as first element in at least one word of the trace language. Immediate termination is impossible due to the first rule condition. Each subspecification contains an additional trace assertion that marks one channel to precede each communication trace of that subsystem.

$$\texttt{system}\ \Delta\ \mathit{TA}\ \mathit{CA}\ \texttt{end}$$

$$\Uparrow$$

$$\begin{array}{l} \texttt{ALT[}\ \texttt{system}\ \Delta\ \mathit{TA}\ \texttt{trace}\ \overline{c}\ \texttt{in}\ d_1.(c_1 + \cdots + c_n)^*\quad \mathit{CA}\ \texttt{end}, \\ \qquad \ldots, \\ \qquad \texttt{system}\ \Delta\ \mathit{TA}\ \texttt{trace}\ \overline{c}\ \texttt{in}\ d_k.(c_1 + \cdots + c_n)^*\quad \mathit{CA}\ \texttt{end}\ \ \texttt{]} \end{array}$$

provided $\varepsilon \notin \mathcal{L}[\![\Delta,\, \mathit{TA}]\!]$ and $\mathit{first}(\Delta,\, \mathit{TA}) = \{d_1, \ldots, d_k\} \neq \emptyset$ and $\overline{c} = \mathit{Chans}(\Delta)$.

Figure 12: Transformation rule alternative decomposition.

Using these decomposition rules and similiar ones a specification can be systematically refined into a mixed term where the trace languages of all occuring specifications are very simple. Here the languages consists of the empty word or of a single channel name. If furthermore the state part is also of a simple pattern then such specifications can be directly replaced by PL statements. Figure 13 shows that certain T-specifications are

```
$c?v$  ≡  system input $c$  write $v$                 $c?$  ≡  system input $c$ of signal
                 trace $c$ in $c$                                  trace $c$ in $c$
                 com $c$ write $v$ then $v' = @c$             end
          end

$c!e$  ≡  system output $c$  read $\mathit{free}(e)$      $c!$  ≡  system output $c$ of signal
                 trace $c$ in $c$                                  trace $c$ in $c$
                 com $c$ read $\mathit{free}(e)$ then $@c = e$       end
          end
```

Figure 13: Meaning of input and output communication statements in PL.

equivalent to input and output communications in PL. Other simple specifications can be transformed into these patterns and are therefore automatically implementable, as described in the next chapter.

Tool support for application of single rules

A transformational design step based on one rule application can be supported by a tool with the generation of the modified system and the check of the side conditions.

A single application of one transformation rule in a theorem prover like Lambda on the one hand modifies the current MIX term and on the other hand generates proof obligations from the rule conditions. To reduce the necessary interaction with the tool the proof programming language of tactics can be used. Tactics are based on possibly guided single rule applications and equational rewriting which are combined by tactical composition constructs like sequences, if-then-else statements and repetitions to proof searching algorithms.

Since most application conditions of our transformation rules are decidable their verification can be automated. For example all conditions concerning regular expressions are decidable. Many other conditions are provable by simple set operations. The tool only needs user guidance when a transformation rule modifies a MIX term in a way that cannot be generated from the context. For example the user should describe the desired subspecifications when applying the parallel decomposition of figure 10.

6 Automatic Program Synthesis

A transformational software design requires even with tool assistance user support to realize creative design decisions. Nevertheless, if the designer has made some decision a tool should perform all necessary transformation steps and check their correct execution. Thus we have started to implement design strategies thereby exploiting the Lambda implementations of the transformation rules which arose from a formal validation of our approach [BR95].

There are two ways how to integrate strategies inside Lambda. The first one is provided by tactics. Strategies can be realized by sequential combinations of tactics for single transformation rules. This method allows a flexible combination of previously defined tactics. But reasoning about the strategies is impossible in Lambda itself because tactics are expressed in a meta language. E.g. termination of tactic applications cannot be proven in Lambda.

The second way overcomes this disadvantage. Here strategies are formalized within Lambda as functions which implement algorithms that describe the design ideas. This integrated treatment allows us to prove properties of strategies as termination and applicability in certain situations in Lambda. While the correctness of tactical strategies follows immediately from the correctness of their underlying rules the correctness of strategy functions has to be proved itself, although these proofs are also reducible to easier rules or simple statements. The correctness of a function *strat* realizing a certain strategy is easily

$$\begin{array}{c} S \\ \Uparrow \\ strat(S) \end{array}$$

provided ...

Figure 14: Strategy as function.

expressed as transformation rule (cf. figure 14) where "..." characterize all side conditions of the strategy. The automated application of such a strategy in Lambda is then reduced to a call of a simple tactic which applies the corresponding rule and afterwards expands the definition of *strat*.

A tactical combination of several rules requires the explicit condition check for each rule application. Often in the context of a strategy similar conditions have to be checked for the various rules applications. Such overlapping checks can be avoided in the case of functional strategy implementation. Here all these checks are collected in the single strategy condition thereby removing redundant checks.

SCS: implementing specifications of single communications

In a last step of any transformation process simple specifications of communications and their effects to the systems state have to be implemented. Therefore the equivalences of input and output communications in figure 13 are extended to specifications with less restricted communication assertions. Figure 15 shows the implementation of a so-called SCS (*Single Communication System*) for an input channel. The new variable v_c is introduced to

$$\begin{array}{l}
\texttt{system input } c \texttt{ of } ty \;\; \texttt{write } w \;\; \texttt{read } r \\
\qquad \texttt{trace } c \texttt{ in } c \\
\qquad \texttt{com } c \texttt{ write } w \texttt{ read } r \texttt{ when } wh_c \texttt{ then } th_c \\
\quad \texttt{end} \\
\\
\hfill \Uparrow \hfill \\
\\
\texttt{IF[} wh_c \rightarrow \texttt{var } v_c \texttt{ of } ty \;\; \texttt{SEQ[} c?v_c, impl(th_c[v_c/@c]) \texttt{]]}
\end{array}$$

provided v_c is a fresh variable

Figure 15: SCS transformation for input channel.

pass the received value from the input to the effect computation. An analogous rule with the sequence $\texttt{SEQ[} impl(th_c[v_c'/@c]), c\,!\,v_c \texttt{]}$ holds in the case of an output channel. Here the communication value has to be computed before it can be offered to the environment.

The mixed term derived from an SCS rule applications is transformed further by replacing $impl(th_c[v_c/@c])$ and $impl(th_c[v_c'/@c])$, respectively. For a transition predicate p we use $impl(p)$ to denote any program that computes this state transition and afterwards terminates. Not every transition predicate is implementable, e.g. $false$. Thus the design process should yield then-predicates which can be treated by rules of the following kind:

$$\begin{array}{c}
impl(x' = e) \\
\Uparrow \\
x := e
\end{array}
\qquad\qquad
\begin{array}{c}
impl(p \wedge q) \\
\Uparrow \\
\texttt{SEQ[} impl(p),\; impl(q[Writes(p)/Writes(p)']) \texttt{]}
\end{array}$$

provided $Writes(p) \cap Reads(q) = \emptyset$

Figure 16: Implementing transition predicates.

Applying SCS and $impl()$ rules recursively yields a little basic strategy which implements specifications of which the trace part cannot be further decomposed. Automating this SCS strategy as tactic would first apply the SCS rules and then repeatedly $impl()$-rules. A formalization as function in Lambda recursively walks through the structure of a mixed term and replaces SCS suitable systems by PL implementations as follows:

$$\begin{array}{lcl}
\texttt{SCS(SEQ[} P, Q \texttt{])} & = & \texttt{SEQ[SCS(} P \texttt{), SCS(} Q \texttt{)]} \\
\texttt{SCS(ALT[} P, Q \texttt{])} & = & \texttt{ALT[SCS(} P \texttt{), SCS(} Q \texttt{)]} \\
\texttt{SCS(WHILE } b \texttt{ do } P \texttt{ od)} & = & \texttt{WHILE } b \texttt{ do SCS(} P \texttt{) od} \\
\ldots & & \\
\multicolumn{3}{l}{\texttt{SCS(system ouput } c \texttt{ of } ty_c \ldots \texttt{ trace } c \texttt{ in } c \;\; \texttt{com } c \ldots \texttt{ end)}} \\
 & = & \texttt{IF[} wh_c \rightarrow \texttt{var } v_c \texttt{ of } ty_c \texttt{ SEQ[} impl(th_c[v_c'/@c]), c\,!\,v_c \texttt{]]}
\end{array}$$

$$\texttt{SCS(system input } c \texttt{ of } ty_c \ldots \texttt{ trace } c \texttt{ in } c \texttt{ com } c \ldots \texttt{ end)}$$
$$= \texttt{IF[} wh_c \rightarrow \texttt{var } v_c \texttt{ of } ty_c \texttt{ SEQ[} c?v_c, impl(th_c[v_c/@c]) \texttt{]]}$$

All other mixed terms remain unchanged by SCS. The corresponding strategy rule is presented in figure 17.

$$\begin{array}{c} S \\ \Uparrow \\ SCS(S) \end{array}$$

provided no local variable v_c occurs free in th_c

Figure 17: SCS strategy rule

In the following SDT strategy we will use this SCS implementation as basic strategy.

SDT: generating sequential implementations

For restricted classes of specifications it is possible to generate a program structure from the trace part automatically. The idea of the *Syntax Directed Transformation* strategy (SDT) is to drive the transformation process by the structure of the regular expression of the only trace assertion of a specification. A tactical automation of this strategy would recursively apply the decomposition rules presented in chapter 5. This tactic would perform many similar checks of application conditions which are avoided by the following functional implementation.

The function PCS formalizes in Lambda the inductive construction of a *Program Control Structure* from the operators of one regular expression and calls the SCS strategy to generate communication statements for channel names in the regular expression.

$$\begin{array}{lcl}
\texttt{PCS(} \Delta, re1 + re2, CA \texttt{)} & = & \texttt{ALT[SEQ[PCS(} \Delta, re1, CA \texttt{), PCS(} \Delta, re2, CA \texttt{)]]} \\
\texttt{PCS(} \Delta, re1.re2, CA \texttt{)} & = & \texttt{SEQ[PCS(} \Delta, re1, CA \texttt{), PCS(} \Delta, re2, CA \texttt{)]} \\
\texttt{PCS(} \Delta, re*, CA \texttt{)} & = & \texttt{WHILE} \ldots \texttt{do} \ldots \texttt{od} \\
\texttt{PCS(} \Delta, c, CA \texttt{)} & = & \texttt{SCS(system } \Delta|_c, \texttt{trace } c \texttt{ in } c \texttt{ , } CA|_c \texttt{ end)}
\end{array}$$

The interface Δ and communication assertions CA are used for calls of the SCS strategy where $\Delta|_c$ denotes the restriction of Δ and $CA|_c$ gives the communication assertion of channel c. Figure 18 shows the corresponding PCS rule which generates sequential programs for certain T-specifications.

$$\begin{array}{c} \texttt{system } \Delta \texttt{ trace } \overline{c} \texttt{ in } re \quad CA \texttt{ end} \\ \Uparrow \\ \texttt{system } \Delta \texttt{ PCS(} \Delta, re, CA \texttt{) end} \end{array}$$

provided re is SDT suitable and $impl(th_c)$ is defined for all $c \in \overline{c} = Chans(\Delta)$.

Figure 18: PCS implementation of system specifications.

Basically PCS uses the rules presented in chapter 5 and the SCS function. The conditions of the PCS rule guarantee that all application conditions corresponding to the intermediate transformation steps are satisfied. SDT suitable regular expressions contain no nested iterations (stars). Further more alternative regular expressions are restricted to input channels as first letters.

Now the *SDT strategy* is defined as follows: An S-specification is transformed by the rule in figure 9 into a T-specification with a following STOP. Then PCS and SCS are applied to this T-specification. Based on algebraic laws, the so far generated program is finally simplified by rewriting rules like those in figure 19.

SEQ[WHILE do $true$ odP, Q]	$\rightarrow$	WHILE $true$ do P od
ALT[ALT[...]]	$\rightarrow$	ALT[...]
IF[$true, P$]	$\rightarrow$	P
SEQ[$c?v, x$:=v]	$\rightarrow$	SEQ[$c?x, v$:=x]
SEQ[v:=$e, c!v$]	$\rightarrow$	SEQ[v:=$e, c?e$]
var v of ty P	$\rightarrow$	P, if v is an auxiliary variable in P

Figure 19: Rewriting Rules for the SDT strategy

The SDT strategy can be applied to each of the component specifications *WMspec*, *RMspec* and *SVspec* (see figure 8) of the register example. The combined application of PCS, SCS, *impl*() and simplifying rewriting rules yield the implementations which are shown as the three arguments of the PAR operator in figure 4.

The three specifications *WMspec*, *SVspec* and *RMspec*satisfy the application conditions of the SDT strategy. Its application yields the following implementations of *WM*, *SV* and *RM*:

```
WM  =  var new of V
         WHILE true do SEQ[ W?new, w!new, A! ] od
SV  =  var x of V
         WHILE true do ALT[ w?x → SKIP, r? → t!x ] od
RM  =  var y of V
         WHILE true do SEQ[ R?, r!, t?y, T!y ] od
```

Figure 20: Implementations of *WMspec*, *RMspec* and *SVspec*.

7 Discussion

We reported on a mixed term language MIX for the transformational design of communicating systems. Using the example of a register specification we demonstrated how to realize certain implementation ideas in a transformational design approach.

In the theorem prover Lambda the mixed terms and transformation rules have been formalized in order to validate the whole approach and prove the rules mechanically. At a first stage this embedding provides a simple tool for interactive execution of transformation steps.

In a transformational setting strategies systematically combine several rules in order to direct large transformation steps. To decrease the degree of user interaction in a design process the execution of such strategies has been automated in Lambda. Aspects of different realizations are discussed on the examples SCS and PCS. These strategies are used to generate implementations for the sequential components of the previously parallel decomposed register specification. A formal treatment of strategies inside Lambda allows to prove properties like correctness, termination and applicability to certain mixed terms.

Ideas for further strategies reveals in the context of parallel implementations concerning the systematic treatment of shared variables and methods of data refinement. Building up these strategies together with their integration in a design tool yields improved support of important design tasks.

References

[Bac90] R.J.R. Back. Refinement calculus, Part II: Parallel and Reactive Programs. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness*, LNCS 430, pages 67–93. Springer-Verlag, 1990.

[BR95] J. Bohn and S. Rössig. Towards a design assistant for communicating systems. Technical Report ProCoS Doc. Id. OLD JB 2/1, Univ. Oldenburg – FB10, 1995.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design - A Foundation.* Addison-Wesley, 1988.

[FFHM93] M. Francis, S. Finn, R.B. Hughes, and E. Mayger. *LAMBDA Version 4.3, Documentation Set.* Abstract Hardware Limited, London, 1993.

[FM91] M. Fourman and E. Mayger. Integration of formal methods with system design. In *Proc. VLSI'91, Edingburgh*, 1991.

[INM88] INMOS Ltd. *occam 2 Reference Manual.* Prentice Hall, 1988.

[JROR90] K.M. Jensen, H. Rischel, E.-R. Olderog, and S. Rössig. Syntax and informal semantics of the ProCoS specification language level 0. Technical Report ESPRIT Basic Research Action ProCoS, Doc. Id. ID/DTH KMJ 4/2, Technical University of Denmark, Lyngby, Dept. Comput. Sci., 1990.

[Lam86] L. Lamport. On interprocess communications Part II. *Distributed Comp.*, 1:86–101, 1986.

[Lam94] L. Lamport. The temporal logic of actions. *TOPLAS*, 16(3):872–923, 1994.

[LG89] N.A. Lynch and K.J. Goldman. Distributed algorithms. Technical Report MIT/LCS/RSS 5 6.852 Fall 1988, MIT, 1989.

[OH86] E.-R. Olderog and C.A.R. Hoare. Specification-oriented semantics for communicating processes. *Acta Inform.*, 23:9–66, 1986.

[Old91] E.-R. Olderog. Towards a Design Calculus for Communicating Programs. In J.C.M. Baeten and J.F. Groote, editors, *Proc. CONCUR '91*, LNCS 527, pages 61–77. Springer-Verlag, 1991. invited paper.

[OR93] E.-R. Olderog and S. Rössig. A case study in transformational design of concurrent systems. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, LNCS 668, pages 90–104. Springer-Verlag, 1993.

[Rös94] S. Rössig. *A Transformational Approach to the Design of Communicating Systems.* PhD thesis, Univ. Oldenburg, 1994. Tech. report 4-94, Fachbereich Informatik, Univ. Oldenburg.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, London, 1989.

Composition and Refinement Mapping based Construction of Distributed Applications

Arnulf Mester, Heiko Krumm
Universität Dortmund, Fachbereich Informatik, D-44221 Dortmund, Germany
Phone +49 231 755-4662, Fax -4730, {mester|krumm}@ls4.informatik.uni-dortmund.de

Abstract

Major steps of the design of distributed applications correspond to the integration of predefined patterns. To support such design steps, a concept for refinement by pattern composition is introduced which applies formal process composition and provides functions for the tool assisted construction and modification of specifications. The approach is based on L. Lamport's Temporal Logic of Actions TLA and the related theory of refinement mappings and system composition.

1 Introduction

The increasing complexity of distributed applications can be faced by enhanced design support, by implementation-near tools and building blocks as well as by more abstract design guides, building blocks and methods. On an implementation near level, syntax and static semantics based tools - as they are integrated into current development environments (e.g., [18]) - support the construction of design documents and implementation modules. Libraries provide for reusable solutions of specific application fields and implementation platforms (e.g., [7, 15, 19]). On a more abstract informal level, the design process can be guided by design patterns [9]. Formal description techniques (e.g., Lotos [12], TLA [14]) model the dynamic semantics and support the specification and verification of system behaviour. Correctness preserving transformation methods (e.g., [3, 4, 5, 8, 13]) view the design as a series of stepwise refinements which can be guided by transformation rules.

Like correctness preserving transformation methods, our approach supports the stepwise refinement and verification of formal description technique based specifications. A new type of transformation 'pattern integration' provides for the efficient reuse of specification modules. It corresponds to syntactical transformations which can be performed by a tool. So, even very specific design patterns can be utilized efficiently (However, unlike CPTs, some transformations have to be accompanied by explicit verification steps).

The approach concentrates on the design of control structures of distributed applications. They can be viewed as a composition of patterns which provide abstract distributed algorithms for the distribution of functionality, user models of standardized communication services, and detailed interface mechanisms to application program interfaces of application platforms.

The approach is based on L. Lamport's Temporal Logic of Actions (TLA [14]) and refers to the concepts of refinement mappings [2] and composition by logical conjunction [1]. We apply a compositional TLA style which supports the specification of process systems and corresponding decompositional verifications [10].

The paper introduces pattern integration and an additional operation for process splitting in principle and with respect to tool functions. (A similar process splitting operation is proposed in [6] for algebraic process definitions.) The concepts are exemplified by views of an example (presented in more detail in [17]). The example is 'academic'. However, it is designed to sketch practical aspects of the application of pattern integration in the complete design process of computer network applications. So, the paper concentrates on the application, the formal background is outlined only.

2 TLA and Refinement Mappings

TLA is a linear time temporal logic modelling systems as state transition systems [14]. The state space is defined by a set of variables. The next state relation is structured into so-called actions. A closed system can be described by a TLA formula in canonical form.

```
variables id, od, s ;
 Init    ≜ s=idle ;
 Start(p)≜ s=idle ∧ id'=p  ∧ s'=pre ;
 Step    ≜ s=pre  ∧ id'=id ∧ od'=f(id) ∧ s'=post ;
 F       ≜    Init                                          ! initial states
           ∧ □ (  Step ∨ ∃ p: Start(p)                      ! next state: actions
                ∨ (id'=id ∧ od'=od ∧ s'=s) )                ! stuttering steps
           ∧ ∀ p: WF_id,od,s(Start(p)) ∧ WF_id,od,s(Step);  ! liveness
```

The canonical formula F above describes a system with three variables *id*, *od*, and *s*. Each system execution starts with a state fulfilling the initial predicate *Init*. A pair of successive states $< \sigma, \sigma' >$ always fulfills one of the actions *Start* or *Step* or is a stuttering step (i.e., $\sigma = \sigma'$). In the definitions of the actions, the primed variables refer to the successor state. The fairness assumptions in the last line of *F* express the liveness of the system: both actions are assumed to be scheduled weakly fairly. An action is called weak fair in a behaviour, if it is infinitely often executed unless infinitely often disabled.

Besides of the weak fairness operator WF, TLA provides for a strong fairness operator SF (not used in this paper). The always operator is denoted by $\Box$. $\mathsf{Enabled}(\alpha)$ is true in a state σ, if there can exist a value combination σ', such that $< \sigma, \sigma' >$ fulfills the action α.

Verification is performed by proofs of implications. Assume the formulas *F* and *G* to describe two systems, *F* intended to be a refinement / implementation of *G*. The two systems shall correspond to each other with respect to so called visible variables which occur in both formulas. Additionally *F* and *G* may contain internal variables x_F and x_G. *F* is a correct refinement of *G* iff the formula $\exists x_F : F \Rightarrow \exists x_G : G$ can be proved. Proofs are supported by the TLA inference rules. Refinement proofs can profit from refinement mappings [2]. The existence of a refinement mapping f_R from *F* to *G* is sufficient for the proof. Later we utilize an appropriate specification construction, where a refinement mapping gets obvious. A refinement mapping f_R is a function from the state components of the fine-system to the state components of the coarser system. f_R has to map states of *F* to states of *G* mapping visible variables by identity. Initial states of *F* have to be mapped to initial states of *G*. The images of state pairs fulfilling the next state relation of *F* have to fulfill the next state relation of *G*. Finally each image of a state sequence of *F* has to fulfil the fairness assumptions of *G*.

3 Processes in a compositional TLA style

A compositional TLA style – supporting both safety and liveness properties – is introduced in [10] and applied here in an enhanced version. It facilitates the description of process systems. The style provides for process abstractions[1]. As in Lotos [12], a process in principle is an open subsystem which can perform joint actions with its environment, and data parameters of actions support the communication of values. Moreover, a process specification can be interpreted for itself. It reflects a closed system consisting of the process and an environment which is universal in the sense that it does not constrain the process.

```
variables  e_Start, e_Step,                             ! environment readiness variables
            id, od, s ;                                 ! private state variables
 Init     ≜ s=idle ;
 Start(p) ≜ s=idle ∧ id'=p  ∧ s'=pre ;
 Step     ≜ s=pre  ∧ id'=id ∧ od'=f(id) ∧ s'=post ;
 cStart(p)≜ Start(p) ∧ p ∈ e_Start ;                    ! conditional Start action
 cStep    ≜ Step ∧ e_Step ≠ ∅;                          ! conditional Step action
 Appl1    ≜    Init                                     ! initial states
         ∧ □ (  Step ∨ ∃ p: Start(p)                    ! next state: actions
              ∨ (id'=id ∧ od'=od ∧ s'=s) )              ! stuttering steps
         ∧ ∀ p: WF_id,od,s(cStart(p))
         ∧ WF_id,od,s(cStep) ;                          ! liveness
```

[1]TLA and it's accompanying notations are intended as broad-spectrum specification tools. The compositional TLA style aroused from the experiences of our 'Tools for TLA based specifications' project [16], where the need of developers for more guidance in specification has been identified.

The canonical TLA formula *Appl1* above describes a process. The formula $Appl1 \wedge \Box(e_{Start} = V_p \wedge e_{Step} \neq \emptyset)$ describes a corresponding separated process and is equivalent to the formula F of Sect. 2. V_p denotes the set of values of the data type of the parameter p of the action *Start*.

The liveness properties are described by fairness assumptions on conditional actions which are conjunctions of actions and environment conditions.

The environment readiness variables (introduced for each fair process action) are assumed to be set by the environment of the process. By writing these variables, the environment can define, which possible system action execution it tolerates – either combined as joint action or with a stuttering step of the environment. For our example above, e.g. if $p \in e_{Start}$, the environment can tolerate the action *Start(p)* in the next step, i.e. either stutters or makes a joint action with it.

By style convention, the actions only affect private variables. Additional conventions concern those actions which are supplied with a fairness assumption. Fair actions of a process have to be disjoint to all other actions of the process. Also, they have to be disjoint to stuttering steps.

```
process Appl1 ;
  variables id, od : array [0..5] of integer ; s : (idle, pre, post) ;
  Init ≜ s=idle ;
  actions
    Start(p : array [0..5] of integer) ≜ s=idle ∧ id′=p ∧ s′=pre ;
    Step ≜ s=pre ∧ id′=id
      ∧ od′=[2*id[0],2*id[1],..,2*id[5] ] ∧ s′=post ;
  WF : Start, Step ;
end ;
```

Fig. 1 *Process* Appl1

For the description of processes, we use a PASCAL like notation in the sequel. The notation only specifies the non-canonical parts. So, Fig. 1 represents the formula *Appl1* from above. Because it represents tool input syntax, the variables are now typed and the unknown symbol *f* is replaced by scalar multiplication. We will refer to this form as flat system formula or flat process definition synonymously. 'Flat' characterizes a definition with all specification operations (cf. Sect. 5) performed. The formal semantics of this notation is given by the correspondence to TLA formulas outlined above.

4 Process patterns

Any productive use of formal design must rely on building blocks. Our building blocks for specification are processes and process patterns. They are able to define various abstractions in mixed bottum-up, top-down and 'middle-out' design.

The process *Appl1* of Fig. 1 corresponds to a more general pattern. It is a sequential and terminating process which is defined over three control states and computes result data by the application of a function to input data.

```
process Prepost (tdata : datatype ; f : function) ;
  variables d : tdata ; s : (idle, pre, post) ;
  Init ≜ s=idle ;
  actions
    Start(p : tdata) ≜ s=idle ∧ d′=p ∧ s′=pre ;
    Step ≜ s=pre ∧ d′=f(d) ∧ s′=post ;
  WF : Start, Step ;
end ;
process Auxvar (tdata : datatype) ;
  variables v : tdata ;
  Init ≜ true ;
  actions  Write(p : tdata) ≜ v′=p ;
end ;
```

Fig. 2 *Process patterns* Prepost *and* Auxvar

The pattern is called *Prepost* and shown in Fig. 2. *tdata* and *f* denote generic parameters of the pattern. The second pattern *Auxvar* can be used to introduce auxiliary variables, i.e. variables used for verification purposes only and not to appear in implementations.

It's obvious, that a library of process patterns modelling common abstractions[2] in distributed applications design significantly increases the productivity of the designer. Appropriate concepts and tool

[2]More representative process patterns appear in Sect. 6.

support for using process patterns and processes as building blocks in design are described in the next section.

5 Operations on specifications

During design, design decisions are normally introduced stepwise into the artifact. In formal design, the formal specification of each maturity 'level' captures the sum of design decisions done and introduced already. We are capturing design decisions by recording the sequence of specification operations and their process/pattern parameters. Specification operations are formally relating their resulting process definitions with their input process definitions.

Features, formal background and motivation for the operations are presented. The syntax for the operations is exemplified in the next section.

5.1 Process composition

The compose operation takes as input a number of processes and a coupling description and delivers a flat process definition for the resulting system.

Formally, a system S composed of processes $P_1, P_2, .., P_m$ is described by a compositional system formula $S \triangleq P_1 \wedge P_2 \wedge .. \wedge P_m \wedge CC$. The different P_j stand for the process formulas. CC is an invariant and describes the coupling of the system. CC defines the system actions which are conjunctions of process actions and process stuttering steps. To each system action, each process contributes by exactly one action respectively stuttering step.

With respect to fair process actions, CC defines the environment readiness variables of each process to be functions of the 'real' state variables of the other processes. Let α be a fair action of a process P_i which occurs in a system action β together with the actions γ_j of other processes. The readiness variable e_α of process P_i has always to be equal to the intersection of the system readiness variable e_β and the sets $\{p : \mathsf{Enabled}(\gamma_j(p))\}$.

The occurence of fair process actions in system actions is constrained by additional style conventions. Each fair action of a process has to occur exactly once in the set of system actions. If the fair process action $\alpha(p_1, .., p_n)$ occurs in a system action $\beta(q_1, .., q_m)$, the actual parameters $p_1, .., p_n$ of α have to be identical to the formal parameters $q_1, .., q_m$ of β. If α is supplied with the strongest fairness assumption of the process actions of β, the fairness assumption of β is analogous to the fairness assumption of α.

Due to the conventions, the compositional system formula can be transformed syntactically into an equivalent flat formula (as defined in Sect. 3) which meets the syntax of a process description.

5.2 Pattern integration

Input to the pattern integration operation are processes P_i which are instances of patterns. We intend R to be a combination of the P_i for which implications $R \Rightarrow P_i$ hold (dependent on the role of P_i, for some P_i, $R \Rightarrow \exists e_\alpha, \ldots : P_i$ may be sufficient). The benefit of this is very important: The resulting process is formally related by refinement mappings with all its integrated processes, i.e. their behaviour and theorems.

The style conventions of composition (cf. last Subsect.) are weakened. There may be fair process actions which occur in more than one system action. The actual parametrization of fair process actions not necessarily has to be identical to the formal parameters of system actions.

We split the task of pattern integration into a syntactical operation and accompanying proof. The syntactical operation is prepared mechanically and followed by creative modification. The preparing operation results in an intermediate system IR which is constructed by firstly computing the flat form of the composition and then applying variable substitutions which replace component variables by system variables. The substitutions can serve as proposals of refinement mappings for proofs of $R \Rightarrow P_i$.

5.3 Composed patterns

Composition and pattern integration can support the application design. Additionally it can facilitate the definition of patterns if a pattern regards different aspects in combination. So, a pattern may be a composition of resource oriented processes which reflect a predefined distribution of functionality. By means of a composition of constraint oriented processes, a pattern can reflect a logical decomposition of system properties.

5.4 Process splitting

Design by pattern integration would result in one large flat specification of a distributed application. Nevertheless, distributed applications consist of several components which are located at different sites. Therefore, we propose the operation of process splitting. Like pattern integration, it is based on composition and performed interactively.

Process splitting starts from a flat specification F and transforms it into a compositional specification C which is composed from a set of processes P_i and which is a refinement of F. We assume that the actions of F (as well as the *Init* predicate) are defined as conjunctions of predicates.

The process splitting is performed in a sequence of steps. First, the designer proposes a set of process names $P_1, \ldots, P_n$. Then he has to define the distribution of variables. For each variable of F he can define a non-empty set of those processes, in which a copy of the variable will occur. Thereafter a tool can check, if the process splitting can be prepared mechanically. It checks the occurrence of variables in predicates, and decides if all variables of a predicate are assigned to the same set of processes. If this is the case, the predicate is associated with this set of processes. If all predicates can be associated with process sets, a proposal IC for C can be computed mechanically. Finally IC can be modified interactively to form C.

IC is a composition of processes $P_1, \ldots, P_n$. IC has the notation of a composition which is equivalent to F. Additionally, the specifications of the processes P_i are parts of IC. The variables of P_i are copies of all the variables of F which are assigned to P_i. The *Init* condition and the actions are conjunctions of copies of predicates. For each action β of F, a process P_i will contain a corresponding action α. α is defined by the conjunction of these predicates of β the process set of which contains P_i (if β does not contain such predicates, α equals to *true*).

An arbitrary splitting – not only motivated by physical variable distribution – is possible, e.g. for preparing simplification and verification tool employment (cf. Sect. 7).

6 A distributed ISO/OSI-conformant producer/consumer example

We sketch parts of the stepwise design of a distributed application, modelling a producer and a consumer agent targeted at a realistic use of an ISO/OSI-conformant transfer service and appropriate procedure style application program interface. The full example is available as [17]. The first three subsections exemplify the specification operations described above, using the syntax of our manipulation tool. The last two subsections illustrate the benefits of the manipulation tool when using larger process patterns and specifications as building blocks for specifications.

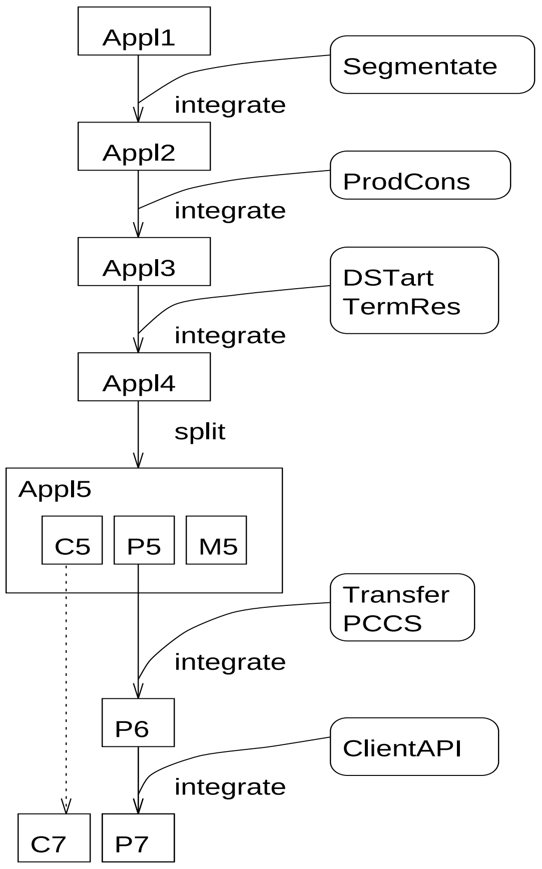


Fig. 3 gives an outline of the example design. An abstract process *Appl1* performs a terminating two-step operation. The final result is a system of two remote processes *P7* and *C7*. *P7* produces segments of data, *C7* consumes and processes segments. *P7* and *C7* communicate via a 'standard' communication service. The application programming interface to the service is provided by a special application platform. In order to refine *Appl1* into this producer consumer system, the design references the predefined patterns *Segmentate*,*ProdCons*,*DStartTermRes*, *TransferPCCS*, and *ClientAPI*.

Fig. 3 *Overview of the example design*

6.1 Process composition

```
type  aoi = array [0..5] of integer ;
process CAppl1 ;
  processes  P : Prepost(aoi, [x₀,..,x₅]→[2*x₀,..,2*x₅]);
             A : Auxvar(aoi) ;
  actions
    Start(p : aoi) ≜ P.Start(p) ∧ A.Write(p) ;
    Step ≜ P.Step ∧ A.Stutter ;
end ;
process FAppl1 ;
  variables P.d : aoi ; P.s : (idle, pre, post) ; A.v : aoi ;
  Init ≜ P.s=idle ∧ true ;
  actions
    Start(p : aoi) ≜ P.s=idle ∧ P.d′=p ∧ P.s′=pre ∧ A.v′=p ;
    Step ≜     P.s=pre ∧ P.d′=[2*P.d[0],..,2*P.d[5] ]
            ∧ P.s′=post ∧ A.v′=A.v ;
  WF : Start, Step ;
end ;
```

Fig. 4 *System composed from* Prepost *and* Auxvar *instances*

Fig. 4 shows the processes *CAppl1* and *FAppl1*. *CAppl1* is composed from instances *P* and *A* of the patterns *Prepost* and *Auxvar* (cf. Fig. 2). Furthermore it is an example for the syntax of a compositional system. The coupling constraint *CC* follows from the actions part of *CAppl1*. It defines the system actions to be joint actions of *P.Start* and *A.Write*, or to be *P.Step* actions accompanied by stuttering steps of *A*. Implicitly, it defines the environment readiness variables of the component *P* to be functions: $\Box(P.e_{Start} = \{p \in aoi : \mathsf{Enabled}(A.Write(p))\} \cap e_{Start} \wedge P.e_{Step} = e_{Step})$. *CAppl1* is equivalent to *FAppl1* which represents the flat system formula (which is – under appropriate variable renaming – equivalent to *Appl1* of Fig. 1).

6.2 Pattern integration

```
process IAppl1 ;
  variables id, od : aoi ; s : (idle, pre, post) ;
  integrate
       A : Auxvar(aoi) substitute A.v by id ;
       P : Prepost(aoi,[x_0,..,x_5]→[2*x_0,..,2*x_5])
           substitute P.s by s ; P.d by if s=post then od else id ;
  actions
    Start(p : aoi) ≜ A.Write(p) ∧ P.Start(p) ;
    Step ≜ A.Stutter ∧ P.Step ;
 end ;

process IAppl1 ;
  variables id, od : aoi ; s : (idle, pre, post) ;
  Init ≜ true ∧ s=idle ;
  actions
    Start(p : aoi) ≜
     id′=p ∧ s=idle ∧ (if s=post then od else id)′=p ∧ s′=pre ;
    Step ≜
     id′=id ∧ s=pre ∧ s′=post
     ∧ (if s=post then od else id)′=
          [2*(if s=post then od else id)[0],..,
           2*(if s=post then od else id)[5] ] ;
  WF : Start, Step ;
 end ;
```

Fig. 5 *Construction of* Appl1 *by pattern integration*

For exemplification, we alternatively construct *Appl1* of Fig. 1 by pattern integration. Fig. 5 shows definition and result of the mechanical operation. The variable *id* of *IAppl1* is shared between an *Auxvar* and a *Prepost* process. By simplifications *IAppl1* can be modified interactively and *Appl1* of Fig. 1 can be derived.

6.3 Design by pattern integration

Pattern integration steps can substantially facilitate the construction of the detailed specification of a distributed application. We assume that a suitable library of patterns exists. The designer starts from an abstract specification of the application and successively develops more detailed specifications by the integration of pattern instances.

```
 process Segmentate (n : cardinal ; t1, t2 : datatype ; fs : function) ;
  variables  id : array[0..n-1] of t1 ; od : array[0..n-1] of t2 ;
             sc : 0..n ;  s : (idle, ready, term) ; stc : cardinal ;
  Init ≜ s=idle ∧ stc>0 ;
  actions
    Start(p : array[0..n-1] of t1) ≜
      s=idle ∧ id′=p ∧ sc′=0 ∧ s′=ready ∧ stc′=stc ;
    Reset ≜
      s=ready ∧ stc>0 ∧ id′=id ∧ sc′=0 ∧ s′=ready ∧ stc′<stc;
    DoSeg ≜
      s=ready ∧ sc<n ∧ id′=id ∧ od′=od besides od[sc]=fs(id[sc])
      ∧ sc′=sc+1 ∧ s′=ready ∧ stc′=stc ;
    Terminate ≜ s=ready ∧ sc=n ∧ id′=id ∧ od′=od
      ∧ sc′=sc ∧ s′=term ∧ stc′=stc ;
  WF : Start, DoSeg, Terminate ;
 end ;
```

Fig. 6 *Pattern* Segmentate

Now, we outline the refinement of *Appl1* of Fig. 1 into the distributed application, which consists of two application processes, a producer of segments and a remote consumer. The first design step is the integration of the pattern *Segmentate* (Fig. 6) in order to introduce the segmentwise processing of data. *Segmentate* successively applies the function *fs* to the elements of array *id*, the results stored in *od*. To prepare reactions on exceptions of the distributed system, the process may be reset and may start the processing again. Since the number of resets is limited (by *stc*), the process will eventually terminate due to the fairness assumptions.

```
process IAppl2 ;
  variables  id, od, aod : array[0..5] of integer ; sc : 0..6 ;
             s : (idle, ready, term) ; stc : cardinal ;
  integrate
    A : Appl1 substitute
          A.id by id, A.od by if s=term then od else aod,
         A.s by  if s=idle then idle elsif s=ready then pre else post;
   S : Segmentate(6,aoi,aoi,x→2*x) substitute
         S.id by id, S.od by od, S.sc by sc, S.s by s, S.stc by stc;
   actions
         Start(p : aoi) ≜ A.Start(p) ∧  S.Start(p) ;
         Reset ≜          A.Stutter  ∧  S.Reset ;
         DoSeg ≜          A.Stutter  ∧  S.DoSeg ;
         Terminate ≜      A.Step     ∧  S.Terminate ;
end ;
```

Fig. 7 *Integration of* Segmentate *into* Appl1

The integration of *Segmentate* into *Appl1* is shown in Fig. 7. We want *IAppl2* to be a refinement of a composition of an instance of *Segmentate* with an instance of *Appl1*. Note, that the third array variable *aod* is an auxiliary variable as it is only written. It does not influence the execution of the system and may be omitted in an implementation.

```
process IAppl2 ;
  variables
    id, od, aod : array[0.. 5] of integer ;
    sc : 0.. 6 ; s : (idle, ready, term) ; stc : cardinal ;
  Init ≜
    (if s=idle then idle elsif s=ready then pre else post)=idle
    ∧  s=idle ∧  stc>0 ;
  actions
    Start (p : aoi) ≜
          (if s=idle then idle elsif s=ready then pre else post)=idle
      ∧   id′=p
      ∧   (if s=idle then idle elsif s=ready then pre else post)′=pre
      ∧   s=idle ∧  sc′=0 ∧  s′=ready ∧  stc′=stc ;
    Reset ≜
          id′=id
      ∧   (if s=term then od else aod)′=(if s=term then od else aod)
      ∧   (if s=idle then idle elsif s=ready then pre else post)′ =
           (if s=idle then idle elsif s=ready then pre else post)
      ∧   s=ready ∧  stc>0 ∧  sc′=0 ∧  s′=ready ∧  stc′<stc ;
    DoSeg ≜
          id′=id
      ∧   (if s=term then od else aod)′=(if s=term then od else aod)
      ∧   (if s=idle then idle elsif s=ready then pre else post)′ =
            (if s=idle then idle elsif s=ready then pre else post)
      ∧   s=ready ∧  sc<6
      ∧   od′=od besides od[sc]=(x→2*x)(id[sc])
      ∧   sc′=sc+1  ∧  s′=ready ∧  stc′=stc ;
    Terminate ≜
          (if s=idle then idle elsif s=ready then pre else post)=pre
      ∧   id′=id
      ∧   (if s=term then od else aod)′ =
          [2*id[0], 2*id[1],.., 2*id[5] ]
      ∧   (if s=idle then idle elsif s=ready then pre else post)′=post
      ∧   s=ready ∧  sc=6 ∧  od′=od ∧  sc′=sc ∧  s′=term
      ∧   stc′=stc ;
  WF : Start, DoSeg, Terminate ;
end ;
```

Fig. 8 *Integration result* IAppl2

```
process Appl2 ;
  variables
         id, od : array[0.. 5] of integer ;
         sc : 0.. 6 ; s : (idle, ready, term) ; stc : cardinal ;
  Init ≜ s=idle ∧  stc>0 ;
  actions
    Start (p : aoi) ≜
          id′=p ∧  s=idle ∧  sc′=0 ∧  s′=ready ∧  stc′=stc ;
    Reset ≜
          id′=id ∧  s=ready ∧  stc>0  ∧  sc′=0
      ∧   s′=ready ∧  stc′<stc ;
    DoSeg ≜
          id′=id ∧  s=ready ∧  sc<6
      ∧   od′=(od besides od[sc]=2*id[sc])
      ∧   sc′=sc+1  ∧  s′=ready ∧  stc′=stc ;
    Terminate ≜
          id′=id  ∧  s=ready  ∧  sc=6 ∧  od′=od  ∧  sc′=sc
           ∧  s′=term ∧  stc′=stc ;
  WF : Start, DoSeg, Terminate ;
end ;
```

Fig. 9 *Modified integration result* Appl2

Fig. 8 shows *IAppl2*, the result of the mechanical integration. It can be modified interactively by simplifications to form *Appl2* of Fig. 9. Moreover we removed the auxiliary variable *aod*. With respect to the action *Terminate*, we see that the primed value of *od* is constrained twice in Fig. 7 (equations $od' = od$ and $(if\ s = term\ then\ od\ else\ aod)' = [2*id[0], 2*id[1], .., 2*id[5]\,]$). One can prove, that both equations are not in contradiction in this system. The equations are equivalent, the proof is an essential part of the safety proof of *Appl2* $\Rightarrow$*Appl1*. The proof can be facilitated if the pattern *Segmentate* is accompanied in the library by a theorem $\Box(s = term \Rightarrow od = [fs(id[0]), .., fs(id[n-1])])$. A second theorem of the pattern *Segmentate* can state that *Terminate* eventually occurs and facilitates the liveness proof.

```
process ProdCons (tdata : datatype);
  variables psq, bsq, csq : queue of tdata;
          s: (idle, ready, term) ; stc: cardinal;
  Init ≜ s=idle ∧ stc>0 ;
  actions
    Start(p: queue of tdata) ≜
      s=idle ∧ psq′=p ∧ bsq′=empty ∧ csq′=empty
      ∧ s′=ready ∧ stc′=stc ;
    Reset(p: queue of tdata) ≜
      s=ready ∧ stc>0 ∧ psq′=p ∧ bsq′=empty ∧ csq′=empty
      ∧ s′=ready ∧ stc′<stc ;
    Produce ≜
      s=ready ∧ psq≠empty ∧ psq′=tail(psq) ∧ stc′=stc
      ∧ bsq′=insert(bsq,front(psq)) ∧ csq′=csq ∧ s′=ready ;
    Consume ≜
      s=ready ∧ bsq≠empty ∧ psq′=psq ∧ bsq′=tail(bsq)
      ∧ csq′=insert(csq,front(bsq)) ∧ s′=ready ∧ stc′=stc ;
    Terminate ≜
      s=ready ∧ psq=empty ∧ bsq=empty ∧ s′=term ;
  WF : Start, Produce, Consume, Terminate ;
end ;
```

Fig. 10 *Pattern* ProdCons

At next, we integrate the pattern *ProdCons* (Fig. 10) into *Appl2*. It models a producer/consumer interaction. Initially the producer gets a queue of data *psq*, which is the source for producing data items and forwarding them into a buffer *bsq*. Data items are removed from *bsq* and stored in the queue *csq* by the consumer.

```
process Appl3 ;
  variables
    buffer : queue of integer ; s : (idle, ready, term) ;
    stc : cardinal ; id, od : aoi ; ic, oc : 0.. 6 ;
  Init ≜ s=idle ∧ stc>0 ;
  actions
    Start (p : aoi) ≜ id′=p ∧ s=idle ∧ s′=ready ∧ stc′=stc
      ∧ ic′=0 ∧ buffer′=empty ∧ oc′ =0 ;
    Reset ≜ id′=id ∧ s=ready ∧ stc>0 ∧ s′=ready ∧ stc′<stc
      ∧ ic′=0 ∧ buffer′=empty ∧ oc′=0 ;
    Produce (p : integer) ≜
        id′=id ∧ od′=od ∧ oc′=oc ∧ s′=s ∧ stc′=stc ∧ s=ready
      ∧ ic<6 ∧ ic′=ic+1 ∧ p=id[ic] ∧ buffer′=insert(buffer, p);
    Consume (p : integer) ≜
        id′=id ∧ s=ready ∧ oc<6 ∧ p=front(buffer)
      ∧ od′=(od besides od[oc]=2*p) ∧ oc′=oc+1
      ∧ s′=ready ∧ stc′=stc ∧ buffer ≠ empty
      ∧ ic′=ic ∧ buffer′=tail(buffer) ;
    Terminate ≜
        id′=id ∧ s=ready ∧ oc=6 ∧ od′=od ∧ oc′=oc ∧ s′=term
      ∧ stc′=stc ∧ ic=6 ∧ buffer=empty ;
  WF : Start, Produce, Consume, Terminate ;
end ;
```

Fig. 11 *Modified integration result* Appl3

The result of the integration of *ProdCons* has been simplified resulting in *Appl3* of Fig. 11. Additionally, parameters were introduced into the actions *Produce* and *Consume* (this will be explained on page 11).

6.4 Composed patterns

We exemplify composed patterns by considering the pattern *DStartTermRes*. The example application *Appl3* (Fig. 11) yet contains non-distributed actions *Start*, *Terminate*, and *Reset*. *DStartTermRes* provides for the distribution of these actions. It is composed from three resource-processes, the producer (modelled by *ProdState*), the consumer (modelled by *ConsState*), and the medium. The medium is modelled by a set of constraint processes.

The distributed start of an application is performed by a sequence of two actions, either *StartP1* followed by *StartC2* or *StartC1* followed by *StartP2* will happen. Similarly termination and reset are performed by distributed action sequences.

```
process  ProdState ( pdtyp, ptyp : datatype ; p0 : ptyp ) ;
  variables
    pd : pdtyp ;                        ! problem data structure
    pp : ptyp ;                         ! problem state component
    ps : (idle, ready, compl, term) ; ! control state
  Init ≜  ps=idle ;
  actions
    StartP ( nd : pdtyp ) ≜
              ps=idle ∧ ps′=ready ∧ pp′=p0 ∧ pd′=nd ;
    Produce ( np : ptyp ) ≜
              ps=ready ∧ ps′ in {ready,compl} ∧ pp′=np ∧ pd′=pd ;
    TermP1 ≜ ps=compl ∧ ps′=ps ∧ pp′=pp ∧ pd′=pd ;
    TermP2 ≜ ps=compl ∧ ps′=term ∧ pp′=pp ∧ pd′=pd ;
    ResetP ≜ ps in {ready,compl} ∧ ps′=ready ∧ pp′=p0 ∧ pd′=pd ;
 end ;
```

Fig. 12 *Process* ProdState *of composed pattern* DStartTermRes

Fig. 12 shows the process *ProdState* of *DStartTermRes*. It models a ‘producer’ with respect to its state components and actions. The action *StartP* of *ProdState* contributes to *StartP1* as well as to *StartP2*. The actions *StartC1* and *StartC2* of *DStartTermRes* are joined by stuttering steps of *ProdState*.

Since the application *Appl3* not yet has a marked process structure, we used the flat form of *DStart-TermRes* for the ongoing refinement of the example application while the compositionality of the specification style would support also that subpatterns of a pattern are integrated into processes of an application. The result of the integration of *DStartTermRes* into *Appl3* has been modified by simplification resulting in *Appl4* (Fig. 13).

```
process Appl4 ;
  variables
    id : aoi ; ic : 0 .. 6 ; ps : (idle, ready, term) ; od : aoi ;
    oc : 0 .. 6 ; cs : (idle, ready) ;
    buffer : queue of integer ; sts : set of (statp, statC) ;
    .....
  Init≜
    stc>0 ∧ ps=idle ∧ cs=idle ∧ sts={} ∧ rs={} ∧ ms=idle ;
  actions
    StartP1 (p : aoi)≜ buffer′=buffer ∧ stc′=stc ∧ od′=od
      ∧ ps=idle ∧ ps′=ready ∧ cs′=cs ∧ ic′=0 ∧ id′=p
      ∧ oc′=oc ∧ sts={} ∧ sts′={statP} ∧ rs′=rs ∧ ms′=ms ;
    StartP2 (p : aoi)≜ stc′=stc ∧ buffer′=empty ∧ ps=idle
      ∧ ps′=ready ∧ ic′=0 ∧ id′=p ∧ oc′=oc ∧ cs′=cs
      ∧ buffer′=empty ∧ sts={statC} ∧ sts′={statP, statC}
      ∧ rs={} ∧ rs′=rs ∧ ms=idle ∧ ms′=ready;
    Produce (p : integer)≜
      od′=od ∧ stc′=stc ∧ ic<6 ∧ p=id[ic]
      ∧ buffer′=insert(buffer,p) ∧ ps=ready ∧ ps′=ready
      ∧ ic′=ic+1 ∧ id′=id ∧ oc′=oc ∧ cs′=cs ∧ sts′=sts
      ∧ rs≠{resatP} ∧ rs′=rs ∧ ms=ready ∧ ms′=ms ;
    .....
  WF : StartP1, StartP2, StartC1, StartC2, Produce, Consume,
       ResetP2, ResetC2, TermP1, TermC1, TermC2, TermP2 ;
 end ;
```

Fig. 13 *System* Appl4

6.5 Process splitting

```
split Appl4 ;
  processes  P has ps, id, ic ;
             C has cs, od, oc ;
             M has stc, buffer, rs, ms, sts ;
 end ;

process P ;
  variables  id : aoi ; ic : 0 .. 6 ; ps : (idle, ready, term) ;
  Init≜ps=idle ;
  actions
```

```
    StartP1 ( p : aoi) ≜ 1
      ps=idle ∧ ps'=ready ∧ ic'=0 ∧ id'=p ;
    StartP2 ( p : aoi) ≜
      ps=idle ∧ ps'=ready ∧ ic'=0 ∧ id'=p ;
    StartC1 ( p : aoi) ≜
      id'=id ∧ ic'=ic ∧ ps'=ps ;
    StartC2 ( p : aoi) ≜
      id'=id ∧ ic'=ic ∧ ps'=ps ;
    Produce ( p : integer) ≜ ic<6 ∧ p=id[ic]
      ∧ ps=ready ∧ ps'=ready ∧ ic'=ic+1 ∧ id'=id ;
    .....
  WF : StartP1, StartP2, StartC1, StartC2, Produce, ..... ;
end ;
process C ;  .....  process M ;  .....
process SAppl5 ;
  processes P : P ; C : C ; M : M ;
  actions
    StartP1(p : aoi) ≜ P.StartP1(p) ∧ C.StartP1(p) ∧ M.StartP1(p);
    StartP2(p : aoi) ≜ P.StartP2(p) ∧ C.StartP2(p) ∧ M.StartP2(p);
    Produce(p : integer) ≜ P.Produce(p) ∧ C.Produce(p) ∧ M.Produce(p);
    .....
end ;
```

Fig. 14 SAppl5 - *Splitting of* Appl4 *and result*

To exemplify process splitting, we split *Appl4* (cf. Fig. 13) into three components: a producer process *P*, a consumer process *C*, and a medium process *M*. Fig. 14 shows the corresponding split statement and result. The integration of *DStartTermRes* (cf. page 9) introduced the structuring of the system into the three processes. It also introduces a corresponding allocation of variables. The variables *id*, *ic*, and *ps* represent the state of *P*, the variables *od*, *oc*, and *cs* the state of *C*. All other variables represent the state of the medium *M*.

Note, that we introduced parameters *p* for the actions *Produce*, and *Consume* in *Appl3* (cf. Fig. 11). The parameter introduction supported the process splitting. Originally, the action *Consume* of *IAppl3* contained the predicate $od' = od$ besides $od[oc] = 2 * front(buffer)$ which references private variables of *C* as well as private variables of *M*. By means of the parameter *p* the predicate has been splitted equivalently into the two predicates $od' = od\ besides\ od[oc] = 2{*}p$ and $p = front(buffer)$ which only reference private variables of one process.

With respect to the distribution of actions, the splitting seems to fail. Each process contributes to each system action by a component action while the process system intended shall provide for actions which are local at one site (i.e., if *P* contributes to a system action, *C* has to perform a stuttering step). We look more carefully to *SAppl5*. In fact, the actions *StartC1*, *StartC2*, *Consume*, *ResetC1*, *ResetC2*, *TermC1*, and *TermC2* of *P* are equivalent to stuttering steps of *P*. They can be replaced by *P.Stutter* and removed in the definition of *P*.

The last problem concerns the two-phase execution of start and reset. Globally there is a sequence of two actions, one action at producer's site and one at consumer's site. Locally, an application process is not able to know, if it performs its local action at first or at second while the definition of *P* makes a distinction between these two cases (e.g., two actions *StartP1* and *StartP2* are defined). We recognize that the two actions are pairwise equivalent. Each pair can be merged to one action (e.g., *StartP1* and *StartP2* can be replaced by one action *StartP*).

6.6 Communication service integration

The next two subsections illustrate the benefit of tool support when manipulating larger process patterns. They also serve as an example for a productive pattern library utilization.

At next, the application shall use a special communication service. The service *Transfer* does not directly comply with standards. However, it refers to the ISO/OSI-concepts of session service connections and of unconfirmed respectively confirmed service elements. Service elements are represented in terms of two respectively four service interface events "request→indication" respectively "request→indication, confirmation←response" (cf. [11]).

A pattern *TransferPCCS* is used which describes a system consisting of a service provider *Med*, a data packet producing client *Cli*, and a consuming server *Ser*. The integration of *TransferPCCS* into the simplified result of process splitting (cf. Fig. 14) is performed by separately but consistently integrating

Med to *M*, *Ser* to *C*, and *Cli* to *P*. Since we assume an existing implementation of *Transfer*, the design can concentrate on the two application processes. The resulting producer process *P6*, is shown in Fig. 15. It is a producer process of our application as well as it is a user process of *TransferPCCS*.

```
process P6 ;
  variables
    s : (idle, nocon, incon, conn, inrel, term) ; id : aoi ; ic : 0..6;
  Init ≜ s=idle ;
  actions
    Start (p : aoi) ≜ id′=p ∧ s=idle ∧ s′=nocon ∧ ic′=0 ;
    ConReq ≜ id′=id ∧ ic′=ic ∧ s=nocon ∧ s′=incon ;
    ConCon ≜ id′=id ∧ ic′=ic ∧ s=incon ∧ s′=conn ;
    DatReq (p : usd) ≜
      id′=id ∧ ic<6 ∧ p=id[ic] ∧ ic′=ic+1 ∧ s=conn ∧ s′=conn;
    RelReq ≜ id′=id ∧ ic′=ic ∧ s=conn ∧ ic=6 ∧ s′=inrel ;
    RelCon ≜ id′=id ∧ ic′=ic ∧ s=inrel ∧ s′=term ;
    AboReq ≜
      id′=id ∧ ic′=0 ∧ s in {nocon,incon,conn,inrel} ∧ s′=nocon;
    AboInd ≜
      id′=id ∧ ic′=0 ∧ s in {nocon,incon,conn,inrel} ∧ s′=nocon;
  WF : Start, ConReq, ConCon, DatReq, RelReq, RelCon, AboInd ;
end ;
```

Fig. 15 *Service using producer* P6

We integrated a process pattern modelling the abstract communication pattern (abstract service) of an implementation building block to be integrated in the next step. As the abstract service and the implementation building block are formally related by a refinement, we are assuring that the correct abstract behaviour neccessary is modelled as result of this step.

6.7 Application program interface integration

At last, the producer process of our example shall be refined into a process which applies a special application program interface (API) to an implementation of the service *Transfer*. The application program interface used has a procedure interface, the exchange of service events is represented by invocations and returns of interface procedures. Events which are passed from the user process to the service implementation correspond to procedure calls, events which are passed in the opposite direction correspond to returns of interface procedures, e.g., a call of the procedure *Connect* corresponds to a *ConnectRequest* event, a return of *Connect* corresponds to *ConnectConfirmation* or if the connection is aborted during establishment to *AbortIndication*. The procedure calls have to be parametrized by a data structure *cb* which is used internally by the service implementation. Before any events can be exchanged, the user process has to install *cb* by the invocation of the procedure *Initiate*.

The pattern *ClientAPI* is a refinement of the subpattern *Cli* of *TransferPCCS*. It describes the possible temporal orderings and parametrizations of API procedure calls and has been integrated to *P6* (Fig. 15) resulting in *P7* (Fig. 16). The new variables stem from *ClientAPI*: *pins* and *pouts* represent the actual parameter values of procedure invocations, *cproc* and *prsta* document which procedure is called respectively has returned.

P6 integrates the required properties of a producer of our example application as well as it assures the correct integration of the local API. Since the API handling is described in detail, it is a suitable starting point for the implementation.

```
process P7 (d : aoi ; apicbtype, adrtype : datatype ;
         ownadr, peeradr : adrtype ; length : function) ;
  variables
    id : aoi ; ic : 0 .. 6 ;
    s : (idle,wini,ready,wconn,conn,wsend,wrel,term,wtest,wabo) ;
    cb : apicbtype ; pins, pouts : array cardinal of any ;
    cproc : literal ; prsta : (called, returned) ;
  Init ≜ s=idle ∧ cproc=none ∧ prsta=returned ;
  actions
    InitiateCall ≜ s=idle ∧ s′=wini ∧ cb′=cb
      ∧ cproc′=INITIATE ∧ prsta′=called
      ∧ pins[0]′=cb ∧ pins[2]′=ownadr ∧ id′=d ∧ ic′=0 ;
    InitiateReturn ≜ s=wini ∧ s′=ready ∧ cproc′=cproc
      ∧ prsta′=returned ∧ cb′=pouts[0] ∧ id′=id ∧ ic′=ic ;
    ConnectCall ≜ s=ready ∧ s′=wconn ∧ cb′=cb
```

```
      ∧ cproc′=CONNECT ∧ prsta′=called
      ∧ pins[0]′=cb ∧ pins[1]′=peeradr ∧ id′=id ∧ ic′=ic ;
    ConnectReturnOk ≜
      s=wconn ∧ s′=conn ∧ cproc′=cproc ∧ prsta′=returned
      ∧ cb′=pouts[0] ∧ pouts[1]=connected ∧ id′=id ∧ ic′=ic ;
    ConnectReturnAbo ≜
      s=wconn ∧ s′=ready ∧ cproc′=cproc ∧ prsta′=returned
      ∧ cb′=pouts[0] ∧ pouts[1]=aborted ∧ id′=id ∧ ic′=0 ;
    .....
end ;
```

Fig. 16 *Integration result* P7

Due to integration of a process pattern modelling the correct driving of a special platform, we benefited to be sure about the correct usage of this API in our resulting application.

7 Conclusion

We proposed operations for pattern integration and process splitting which support the computer assisted construction and writing of specifications of distributed applications as well as the understanding and verification of design steps. As the experience of the development of practical computer network applications shows, the dominating design task is not the design of 'new' algorithms, but the design of consistent combinations of known algorithms. Moreover, the introduction of the detailed interface mechanisms to application platforms is important. Both types of design steps correspond well to the integration of process patterns. Furthermore tool support has been realized which mechanically produces major parts of specification texts.

Until now, we studied the application of the approach by means of few 'academic' and two more complex examples. With respect to assistance by tools, we resorted to a set of general TLA tools [16] (editor, browser, interpreter/animator, refinement mapping searcher, refinement mapping checker, model checker, frontend for a general automated predicate logic theorem prover). Additionally, we used prototypical tools which perform the text manipulation operations (combine, integrate, compose, split and refine).

Present work concentrates on the one hand on the extension of the libraries and the examples. While large productive distributed applications are in the scope of our work (e.g., applications of computer integrated manufacturing, administration, and commerce), we firstly collect experiences by means of smaller applications. So, agents for the Internet applications FTP and SMTP are under development.

On the other hand, we want to enhance the tool support. During the construction of a specification, information can be recorded which will be utilized to prepare the application of our TLA verification tools, esp. the refinement mapping checker. Additionally, a tool is under development which performs simplifications of finite state processes automatically in order to enhance the support of pattern integration. The tool utilizes process composition and process splitting. It firstly splits a system into a finite state process and a 'remainder' process. The finite state process is analysed and simplified by reachability graph and term rewriting methods. At last, a composition joins the processes.

Acknowledgements. Thanks to the anonymous referees for their helpful comments.

References

[1] Martin Abadi, Leslie Lamport. Conjoining Specifications. Research Report 118, Digital Equipment Corporation, Systems Research Center, December 1993.

[2] Martin Abadi, Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[3] Friedrich Ludwig Bauer, Bernhard Möller, Helmut Partsch, Peter Pepper. Formal Program Construction by Transformations — Computer Aided, Intuition-Guided Programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, February 1989.

[4] T. Bolognesi (Ed.). Catalogue of LOTOS Correctness Preserving Transformations. Final report. Deliverable Lo/WP1/T1.2/N0045/V03. ESPRIT Project 2304 (Lotosphere). January, 1992.

[5] T. Bolognesi, et al. Correctness Preserving Transformations for the early phases of software development. to appear in: LOTOSphere – software development using LOTOS, Kluwer Academic.

[6] Ed Brinksma, Rom Langerak, Peter Broekroelofs. Functionality Decomposition by Compositional Correctness Preserving Transformation. in: Costas Courcoubetis (Ed.), Proc. of the 5th Int. Conf. *Computer Aided Verification*, CAV '93, Elounda, Greece. 371–384, Springer, 1993.

[7] Citibank Distributed Processing Technology. Objtran Programmer's Guide. Technical Document, December, 1993.

[8] Claus Dendorfer, R. Weber. From Service Specification to Protocol Entity Implementation. in: R.J. Linn,Jr. and M.Ü. Uyar (Eds.) *Protocol Specification, Testing, and Verification XII.* 163–178, Elsevier, 1992.

[9] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. in: O.M. Nierstrasz (Ed.) *7th European conference on Object oriented programming*, Kaiserslautern, Germany Springer, 1993.

[10] Peter Herrmann, Heiko Krumm. Compositional Specification and Verification of High-Speed Transfer Protocols. to appear in: S.T. Vuong and S.T. Chanson (Eds.) *Protocol Specification, Testing, and Verification XIV.* Chapman & Hall, 1994.

[11] ISO. International Standard: Basic Reference Model for Open Systems Interconnection. ISO: IS7498.

[12] ISO. *LOTOS: Language for the temporal ordering specification of observational behaviour.* International Standard ISO/IS 8807, 1987.

[13] R. Kurki-Suonio, H.-M. Järvinen. Action system approach to the specification and design of distributed systems. *ACM Transactions on Programming Languages and Systems*, 4(10):510–54, October 1988.

[14] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[15] William Leddy, Arjun Khanna. DCE++: A C++ API for DCE. Technical Document 030-00209, Hal Computer Systems, Inc. March, 1993.

[16] Arnulf Mester, Peter Herrmann. Tools for TLA-based Specifications. Universität Dortmund, Fachbereich Informatik, Lehrstuhl IV. Technical Report RvS-TLA-94/35, 1994.

[17] A. Mester, H. Krumm. Composition and Refinement Mapping Based Construction of Distributed Applications. Universität Dortmund, Fachbereich Informatik. Research Report/Forschungsbericht No. 548, 1994. `WWW: http://ls4-www.informatik.uni-dortmund.de/RVS/Exports/FB_548.ps.Z`, `FTP: ftp.informatik.uni-dortmund.de` in `/pub/ls04-info/research-reports`

[18] Max Mühlhäuser, W. Gerteis, L. Heuser. DOCASE: A Methodic Approach to Distributed Programming. *Communications of the ACM*, 36(9):127–138, Sept. 1993.

[19] Douglas C. Schmidt. The ADAPTIVE Communication Environment. Technical Report, Dept. ICS, University of California, Irvine, 1994. also: ASX: an Object-Oriented Framework for Developing Distributed Applications. in: *USENIX C++ Technical Conference* April, 1994.

Layers as Knowledge Transitions in the Design of Distributed Systems [*]

(extended abstract)

Wil Janssen [†]
University of Oldenburg [‡]

Abstract

Knowledge based logics allow to give generic specifications of classes of network protocols. This genericity is combined with methods to derive sequentially structured or *layered* implementations of distributed algorithms. Knowledge based logic is used to specify layers in such algorithms as *knowledge transitions.* The resulting layered implementations are transformed to *distributed* algorithms by means a transformation rule based on the principle of *communication closed layers.*
In this way a class of solutions to a problem for different architectures can be derived along the simultaneously. This design technique for distributed algorithms is applied to a class of Two-Phase Commit protocols.

1 Introduction

The design and analysis of distributed systems is a complicated task. Many different processes can be active simultaneously and communicate in a seemingly unstructured way, communication protocols are intertwined with the basic program, and different system architectures can result in completely different algorithms. In the last few years there have been a number of attempts to solve problems concerning the specification and design of distributed systems. One of the possible approaches is to remove all architectural decisions from the specification language, in order to be able to concentrate on the algorithmic aspects. This approach has been taken in, for example, *action systems* or *IO-automata* [5, 16, 3, 19, 18].
A second approach is the use of *knowledge-based* or *epistemic* logics and language constructs [10, 11, 20, 9]. The use of knowledge-based logics allows to express properties of systems and actions in a more global way, abstracting away from communication structures and architectural decisions. Moreover, it has been claimed that programs with knowledge based programming constructs lead to more efficient programs and can still be interpreted [9, 21].
Finally, there has been a considerable amount of attention to the use of *layered methods* in the design of distributed systems [8, 26, 27, 6, 17, 14, 30, 12]. It has been observed that in many protocols in distributed systems the logical structure of the system is basically a sequential one, whereas the actual structure is distributed and depends very much on the details of the implementation architecture. By viewing the algorithm as a sequentially structured system, analysis becomes much simpler and is more or less the same for larger

[*]This work has been supported by Esprit/BRA Project 6021 (REACT).

[†]Most of the work reported on was done when the author was working at the University of Twente.

[‡]Fachbereich Informatik, Postfach 2503, D-26111 Oldenburg, Germany. Phone: + 49 441 7982362; E-mail: Wil.Janssen@informatik.uni-oldenburg.de.

classes of protocols, instead of being applicable to a single algorithm only.
In this paper we combine the above observations. We use the fact that many systems can be designed and analyzed in a layered fashion, plus the fact that knowledge-based logics allow for a specification of such layers at an appropriate level of abstraction, that is, as *knowledge transitions.*

Knowledge concerns facts that we associate a *location* or *distribution* with. Facts can be known to a certain process or set of processes. Knowledge can exist in different ways: *distributed knowledge* is knowledge of the group of processes as a whole. It concerns facts that would be known is all processes would combine all of their information.
The strongest level of knowledge is *common knowledge,* which informally corresponds to facts that are "publicly known." For example, in systems with reliable communication, it is common knowledge that no messages are lost. States of knowledge are expressed using a set of modalities, K, D, S, E, and C. Let G be a group of processes, or *agents* as they are usually called in this context. The expression $K_i\varphi$ states that process i *knows* the proposition φ, that is, it can derive φ from its local state. $S_G\varphi$ states that *somebody* in the group G knows φ, and $E_G\varphi$ gives that *everybody* in G knows φ. Finally, $D_G\varphi$ states that it is *distributed* knowledge in G that φ holds, which means that if we combine the knowledge of every process $i \in G$ we can derive φ, and $C_G\varphi$ that it is *common* knowledge in G that φ holds. In this paper common knowledge will not play an important role and is not discussed further.

Protocols, distributed algorithms, and conceptual layers in them can often be described as transitions from one state of knowledge to another. A transition

$$\Phi \rightsquigarrow \Psi$$

states that if we start in a state satisfying Φ, on termination we will be in a state satisfying Ψ. Therefore, knowledge transitions can be viewed as a generalization of Hoare style preconditions and postconditions to knowledge based assertions. For example, broadcasting protocols can be specified as a transition from a state of knowledge where one process i (the broadcaster) knows a fact φ to a state where all processes in the set G of participating processes know the same fact. So it is a transition of the form

$$K_i\varphi \rightsquigarrow E_G\varphi.$$

If we do not know the identity of the broadcaster this would result in

$$S_G\varphi \rightsquigarrow E_G\varphi.$$

(In fact, this is a simplification. There must also be some common knowledge in the system for this to hold [10], but this is beyond the scope of this paper.) Often parts of protocols are used to gather information of all processes to a single coordinating process. This means that from a state where every process i knows some fact φ_i, the system evolves to a state where a single process c knows all these facts:

$$\wedge_{i\in G} K_i\varphi_i \quad \rightsquigarrow \quad K_c(\wedge_{i\in G}\varphi_i),$$

or stated differently:

$$D_G(\wedge_{i\in G}\varphi_i) \quad \rightsquigarrow \quad K_c(\wedge_{i\in G}\varphi_i).$$

Larger protocols can often also be specified in such a manner. Take for example *atomic commit* protocols for distributed databases (see Bernstein, Hadzilacos and Goodman [4] for an overview of this field). Informally speaking, the protocol has to make a decision for a set of participating processes, based on the internal state of those processes. Every process P_i can decide locally whether or not it can make the changes made in a transaction permanent. The protocol decides to commit iff all processes can do so. If one or more processes cannot, it should decide to abort in order to keep the data at the different processes consistent.

The decision should be made known to all processes which will then take the appropriate actions. The internal state is reflected in a vote yes or no for every process i, such that $vote_i = \texttt{yes}$ iff changes can be made permanent. Such a protocol can be specified as the following knowledge transition. Let $total\_vote = \texttt{yes}$ iff $\wedge_{i \in G} vote_i = \texttt{yes}$.

$$\bigwedge_{i \in G} K_i vote_i \quad \rightsquigarrow \quad \bigwedge_{i \in G} K_i(total\_vote \ \wedge \ (dec_i = \texttt{commit} \Leftrightarrow total\_vote = \texttt{yes})).$$

Here, $K_i vote_i$ means that i knows the value of $vote_i$. This in fact abbreviates $K_i(vote_i = \texttt{yes}) \vee K_i(vote_i = \texttt{no})$. Note that all these specifications used only the set of processes involved and local information.

The approach we introduce in this paper is the following. Given a (knowledge-based) *specification* of a problem, we refine this specification to a *sequence of knowledge transitions*. For example, the above transition can be split into three simpler transitions are follows:

$$\begin{aligned}\bigwedge_{i \in G} K_i vote_i \ &\rightsquigarrow \ K_c total\_vote \\ &\rightsquigarrow \ E_G total\_vote \\ &\rightsquigarrow \ \bigwedge_{i \in G} K_i(total\_vote \ \wedge \ (dec_i = \texttt{commit} \Leftrightarrow total\_vote = \texttt{yes})).\end{aligned}$$

These knowledge transitions are then *instantiated* with protocol layers that are suited for the architecture under consideration and implement the knowledge transitions specified. The result of this is an algorithm that consists of a sequence of layers.
Such an algorithm can then be *transformed* to a parallel or distributed algorithm, using the techniques developed by Janssen, Poel and Zwiers as discussed in, for example, [15, 24, 30, 12]. This transformation is based on the principle of *communication closed layers* as introduced by Elrad and Francez [8], translated to an algebraic setting. After some optimizations the transformed system results in an algorithm that solves the original problem and is tailored to a certain implementation architecture.
In order to be able to take this approach, we give a *classification* of knowledge transitions, and of the ways such transitions can be implemented for different architectures. As such, the knowledge transitions serve as a vehicle for the abstract specification of protocol layers. By taking different refinements of the problem specification using different transitions, different implementations of the problem can be obtained along the same lines, thus emphasizing the similarities and characteristics of the implementations.
The applicability of such layered approaches in general (not particularly using knowledge transitions) has been shown by numerous examples, such as distributed minimum weight spanning tree algorithms, parallel parsing, parts of caching algorithm, pipelining, real-time mutual exclusion and minimal distance algorithms.

The outline of this paper is as follows. We first discuss our language, the knowledge based logic, and a transformation principle for programs. Thereafter we introduce knowledge transitions and classify well-known communication structures for different networks as knowledge transitions. Finally we explain how to derive algorithms as sequences of knowledge transitions and apply this to different versions of the Two-Phase Commit protocol. A full version of this paper, including some other examples as well, is available as [13].

2 Programs, communication closedness and knowledge

Many protocols and distributed algorithms are given in a setting of asynchronous message passing. In this paper we restrict ourselves to this form of communication, in order to simplify technical details that would divert the attention from the main issues of this paper.

Systems consist of a number of components with local variables that communicate using *send*(c, e) and *receive*(c, x) actions, where c is a channel, e is an expression and x is a variable. Channels connect two unique processes and are unidirectional. Often a channel is therefore represented as a pair (v, v') of nodes or processes. Channels are viewed as *single place* buffers.
Besides communication actions and assignments $x := e$ to local variables, our programming language includes conditionals of the form **if** b **then** S **else** T **fi**. If T is omitted it is assumed to be **skip** (do nothing). Actions can be composed by means of parallel composition "$\|$" and sequential composition " **;** ".

Any process that is composed out of the constructs above is called a *layer.* Layers can be composed by means of *layer composition* "$\bullet$". Informally speaking, when we compose two layers S and T by means of layer composition the resulting process $S \bullet T$ executes actions a of S before actions b of T iff a and b are *dependent.* Two actions are dependent, denoted by $a \leftrightsquigarrow b$, iff they access the same (local) variables, or access the same channel. So layer composition can be seen as an intermediate between sequential composition, where full ordering between S and T would be specified, and parallel composition, where ordering between dependent actions of S and T can be in an arbitrary direction, not necessarily from S to T. As such, layer composition cannot directly be translated in well-known program constructs, but it serves as a specification construct in the initial and intermediate design stages. Moreover, layer composition has nice algebraic properties that make it well-suited for a transformational style of program derivation. Please refer to the work by Janssen, Poel and Zwiers, for example [14, 30, 12], for detailed discussions thereof.

Layered programs and communication closed layers

One of the most important algebraic properties that relies on the use of layer composition is the so-called ccl law. It is based on the principle of *communication closed layers* as introduced by Elrad and Francez [8]. This law states that under a certain side condition, a *layered* or sequentially structured system $(P \| R) \bullet (Q \| S)$ behave the same (has the same semantics) as the *parallel* system $(P \bullet Q) \| (R \bullet S)$. The side condition is that there exist no "cross-dependencies" between components in different layers. Formally, assume for processes P, Q, R, and S, that P and S are independent, and that Q and R are independent ($P \not\leftrightsquigarrow S$ and $Q \not\leftrightsquigarrow R$). Under this assumption we have

$$(P \| R) \bullet (Q \| S) \quad = \quad (P \bullet Q) \| (R \bullet S) \qquad\qquad (\mathsf{ccl})$$

This law can be generalized to more processes and more layers of course.

The idea is to derive layered implementations that satisfy this side condition, and to transform these to distributed implementations. In general this side condition does not hold for systems consisting of a number of layers, as different layers can have common channels leading to cross-dependencies. In order to circumvent these problems, we temporarily introduce *virtual channels* per layer, for example by replacing every channel C in layer l by a channel C_l. The resulting process is equivalent to the original one. Thereafter the ccl law trivially applies, as all dependencies are either within a single layer or between different layers but within the same process.
After transforming the renamed system to a parallel system, we can replace the layer composition by sequential composition and replace the virtual channels by again a single channel per edge by means of *multiplexing techniques* (see [12, 30]). These multiplexing techniques do not always apply. In this setting a sufficient condition is to ensure that in every layer every *send* is matched by a *receive* action for the same channel for every possible evaluation of conditionals. Informally speaking this implies that channels are empty at the end of a layer, and therefore *receive* actions in other layers will read the values sent in the layer

they belong to. Multiplexing and replacing layer composition by sequential composition do not preserve semantic equality. They do however preserve the input/output behavior of systems, that is, if viewed as pairs of initial and final states the systems are the same. The is called *IO equivalence.*

The combined result of the above steps is summarized by the following transformation principle.

Let S be a system consisting of a number of layers
$S \triangleq L(0) \bullet L(1) \bullet \cdots \bullet L(n)$,
where every layer is of the form
$L(l) \triangleq \textbf{for } i \in G \textbf{ par } P(i,l) \textbf{ rof}$,
with every *send* action matched by a *receive* action, for all possible evaluations of the conditionals. Assume all components communicate by means of asynchronous message passing only. Then S is IO-equivalent to the system S'
$S' \triangleq \textbf{for } i \in G \textbf{ par } P(i) \textbf{ rof}$,
where
$P(i) \triangleq P(i,0) \,;\, P(i,1) \,;\, \cdots \,;\, P(i,n)$.

Knowledge based logic

Knowledge based or epistemic logic [10, 9, 20] is a class of modal logics that allow to add some notion of locality to formulae. We cannot only say that φ holds, but also that φ holds for a process or agent i, or is a fact that holds for the combined states of a group G of processes, so-called *distributed* or *group* knowledge.

The basic modality is K, which stands for knowledge. The formula $K_i\varphi$ states that process i knows φ. $K_i\varphi$ holds in a states s such that the local state part s_i of s for process i satisfies φ. Knowledge of different processes can be combined. We say that φ is distributed knowledge for a group of processes G iff φ holds for the combined states of all processes $i \in G$. For example, if $K_ix = 1$ and $K_jy = 2$, then $D_{\{i,j\}}y = x + 1$.
Formally speaking, we use the following logic. Assume a given non-empty set P of propositional constants and let A be a finite set of agents or processes. The set $L_A(P)$ of epistemic formulae $\varphi, \psi, \ldots$ is the smallest set closed under

- If $p \in P$ then $p \in L_A(P)$;
- If $\varphi, \psi \in L_A(P)$, then $\varphi \wedge \psi \in L_A(P)$ and $\neg\varphi \in L_A(P)$;
- If $G \subseteq A, i \in A, \varphi \in L_A(P)$, then $K_i\varphi \in L_A(P), D_G\varphi \in L_A(P)$.

As usual, we define implication "$\Rightarrow$" and disjunction "$\vee$" as abbreviations. Also, *true* and *false* abbreviate $p_0 \vee \neg p_0$ and $p_0 \wedge \neg p_0$ for some constant $p_0 \in P$ respectively. Finally the modalities "E" and "S" are defined as abbreviations as well. The modality E_G states that *everybody* in G knows a certain proposition, and the modality S_G states that *somebody* in G knows a certain proposition. They are defined as

$$\begin{aligned} E_G\varphi &\equiv \bigwedge_{i\in G} K_i\varphi, \\ S_G\varphi &\equiv \bigvee_{i\in G} K_i\varphi. \end{aligned}$$

The basic modalities are characterized by a number of axioms and rules. (See, for example, Meyer, van der Hoek and Vreeswijk [20] or Fagin et al. [9] for detailed discussions.)

$$K_i\varphi \Rightarrow \varphi$$
$$D_G\varphi \Rightarrow \varphi \qquad \text{knowledge axioms}$$

$$(K_i\varphi \wedge K_i(\varphi \Rightarrow \psi)) \Rightarrow K_i\psi$$
$$(D_G\varphi \wedge D_G(\varphi \Rightarrow \psi)) \Rightarrow D_G\psi \qquad \text{consequence closure}$$

$$K_i\varphi \Rightarrow K_iK_i\varphi$$
$$D_G\varphi \Rightarrow D_GD_G\varphi \qquad \text{positive introspection}$$

$$\neg K_i\varphi \Rightarrow K_i\neg K_i\varphi$$
$$\neg D_G\varphi \Rightarrow D_G\neg D_G\varphi \qquad \text{negative introspection}$$

$$\frac{\varphi}{K_i\varphi, \qquad D_G\varphi} \qquad \text{knowledge generalization}$$

In the following we use a number of properties of the logic. Let $i \in G$, and let G be a subset of G'.

$$\begin{aligned}
K_i\varphi &\Rightarrow D_G\varphi,\\
K_i\varphi &\Rightarrow S_G\varphi,\\
D_G\varphi &\Rightarrow D_{G'}\varphi,\\
E_{G'}\varphi &\Rightarrow E_G\varphi,\\
E_G\varphi &\Rightarrow S_G\varphi,\\
K_i(\varphi \wedge \psi) &\Leftrightarrow K_i\varphi \wedge K_i\psi,\\
D_G(\varphi \wedge \psi) &\Leftrightarrow D_G\varphi \wedge D_G\psi,\\
K_i(\varphi \vee \psi) &\Leftarrow K_i\varphi \vee K_i\psi.
\end{aligned}$$

Note that the latter implication is not an equivalence. As a counter example that $K_i\mathit{true}$, which obviously holds. However, $K_i\varphi \vee K_i\neg\varphi$ is *not* a tautology. Process i need not know whether φ holds or whether its negation holds.

We use K_ix to state that i knows the value of x, which abbreviates $\bigvee_{v\in \mathit{Val}} K_ix = v$, if x takes its value from Val. The semantics of the logic and of the programming language are both given as sets of partially ordered multisets. These are discussed in [13] and omitted here due to space limitations.

3 Knowledge transitions and communication structures

We have discussed our programming language and knowledge based logic. In the introduction we have argued informally that protocols or protocol layers can sometimes be viewed as transitions from one state of knowledge to another. In this section we give an overview of possible knowledge transitions and classify well-known communication structures as knowledge transitions.

Not all knowledge transitions make equally much sense. The transition $K_i\varphi \rightsquigarrow E_G\varphi$ intuitively corresponds a broadcast-like protocol where φ is sent to all processes. A transition such as $K_i\varphi \rightsquigarrow D_G\varphi$ however makes less sense: if $i \in G$ this is immediately fulfilled without any communication.
In table 1 the transitions are summarized. Every entry in the table gives the relation

$\rightsquigarrow$	$K_j\psi$	$S_{G'}\psi$	$\bigwedge_{G'} K_i\psi_i$	$E_{G'}\psi$	$D_{G'}\psi$
$K_i\varphi$	$i = j : \varphi \Rightarrow \psi$, skip; $i \neq j : \varphi \Rightarrow \psi$, notify	$i \in G'$: $\varphi \Rightarrow \psi$, skip	$\varphi \Rightarrow \psi_i$, broadcast	$\varphi \Rightarrow \psi$, broadcast	$i \in G'$: $\varphi \Rightarrow \psi$, skip
$S_G\varphi$	$\varphi \Rightarrow \psi$, search	$G \subseteq G'$: $\varphi \Rightarrow \psi$, skip	$\varphi \Rightarrow \psi_i$, broadcast	$G \subseteq G'$: $\varphi \Rightarrow \psi$, broadcast	$G \subseteq G'$: $\varphi \Rightarrow \psi$, skip
$\bigwedge_G K_i\varphi_i$	$(\wedge_G\varphi_i) \Rightarrow \psi$, centralize	$(\wedge_G\varphi_i) \Rightarrow \psi$, elect	$G \subseteq G'$: $(\wedge_G\varphi_i) \Rightarrow \psi_i$, confer	$G \subseteq G'$: $(\wedge_G\varphi_i) \Rightarrow \psi$, confer	$G \subseteq G'$: $(\wedge_G\varphi_i) \Rightarrow \psi$, skip
$E_G\varphi$	$j \in G$: $\varphi \Rightarrow \psi$, skip; $j \notin G$: $\varphi \Rightarrow \psi$, notify	$G \cap G' \neq \emptyset$: $\varphi \Rightarrow \psi$, skip; $G \cap G' = \emptyset$: $\varphi \Rightarrow \psi$, notify	$G' \subseteq G$: $\varphi \Rightarrow \psi_i$, skip; $G' \not\subseteq G$: $\varphi \Rightarrow \psi_i$, distribute	$G' \subseteq G$: $\varphi \Rightarrow \psi$, skip; $G' \not\subseteq G$: $\varphi \Rightarrow \psi$, distribute	$G \cap G' \neq \emptyset$: $\varphi \Rightarrow \psi$, skip
$D_G\varphi$	$\varphi \Rightarrow \psi$, centralize	$\varphi \Rightarrow \psi$, elect	$G \subseteq G'$: $\varphi \Rightarrow \psi_i$, confer	$\varphi \Rightarrow \psi$, confer	$G \subseteq G'$: $\varphi \Rightarrow \psi$, skip; $G \not\subseteq G'$: $\varphi \Rightarrow \psi$, centralize

Table 1: Knowledge transitions

between φ and ψ for which that knowledge transition makes sense for different relations between i, j, G and G'. In the general case, protocol layers will also lead to an increase in knowledge due to the fact that, for example, a process knows from whom it has received messages. This increase in knowledge is not reflected in this table, only the way φ directly relates to ψ is given. Transitions that have a non-trivial implementation are named in this table. The entry “skip” means that the transition is trivially satisfied by doing nothing.

The roles of $\wedge K_i$ and of E_G are often similar, due to the similarities in their definitions. Furthermore there is a correspondence between D_G and $\wedge K_i$, as we can observe in the table. We can roughly distinguish four different types of transitions:

- *Broadcast, distribute* or *notify* transitions. These distribute information that is known for a certain process or set of processes to a larger set of processes, possibly all processes.
- *Centralize* or *search* transitions. In this case different information of different processes is gathered to a single process or other set of processes.
- *Elect* transitions. In this case again distributed information is gathered, but resulting not towards a certain process or set of processes, but leading to an in general unknown “winner.”
- *Confer* transitions. For confer transitions distributed information is made known to all or to a larger set of processes.

We would like to give instantiations of all non-trivial knowledge transitions with layers that implement that transition for a certain architecture. Some transitions are more difficult to implement than others for certain architectures. Take, for example, the transition $K_i\varphi \rightsquigarrow E_G\varphi$. In a fully connected network a single round of send actions suffices. In an arbitrary network one needs message diffusion or other more complicated algorithms.
Also, some knowledge transitions can be built up from other transitions. In order to confer

one can, for example, combine a centralizing phase with a distribution phase. Here we restrict ourselves to a few characteristic transitions, needed in the examples.

From the literature many communication structures are known. Broadcasts, waves, phases, heartbeats, logic pulsing, rooted tree communication, message diffusion all correspond to certain types of protocols for different network architectures. (See Raynal and Helary [25] and Andrews [1] for overviews.) Such protocols or protocol layers correspond to (sequences of) knowledge transitions. We give a classification of such layers for different architectures. This classification is by no means complete; not all communication structures are discussed. It should however be possible to classify other communication structures along the same lines. Proof rules to do so are given at the end of this section.

We discuss two different types of network architectures: rooted trees or sets of rooted trees, and connected graphs. Other architectures, such as linear lists or fully connected graphs, are special cases of these two.
Assume we have a finite set of nodes V, and a subset $Root \subseteq V$ of root nodes. Let $root(v) \equiv v \in Root$. Every node v has a set of directed downward edges $down(v)$ and a set of successor nodes $S(v)$, and every non-root node v has an upward edge $up(v)$ pointing towards its root node. We discuss some generic instantiations of knowledge transitions, going from top-left to bottom-right in the table. For exact implementations we refer to [13] or to the literature mentioned.

$K_i\varphi \rightsquigarrow K_j\varphi$. This is the most elementary transition. A process i notifies some process j of a certain fact known to i. If there exists an edge from i to j it can be implemented by a pair of communication actions. If not, some kind of relaying or forwarding protocol is to be used.
In general, this transition can be implemented abstractly by a so-called notify action as proposed by Moses and Kislev [21]. An action notify(j, φ) by process i ensures that eventually j knows φ. However, such a notify action only specifies what should be done, but not how this result can be obtained.

$K_v\varphi \rightsquigarrow E_V\varphi$. This coincides with $K_v\varphi \rightsquigarrow \wedge K_v\psi_v$ for $\varphi \Rightarrow \psi_v$. Under this category fall *broadcast* protocols for arbitrary graphs, and direct *distribution layers* for fully connected graphs and two-level trees with a single root. Broadcast algorithms in general are more complicated. See Cristian et al. [7] or Mullender [22] for more details.

$\bigwedge_G K_i\varphi_i \rightsquigarrow K_j\psi$. Centralizers captured by the transition above occur frequently in protocols. Information that is located at different nodes is to be gathered to a single node. For tree-structured networks this can be done by means of the so-called *wave concept.* (See Raynal and Helary, [25].) The root node initiates a request wave down the tree, which is returned from the leaves upwards, gathering the information. The first phase of a wave is called a *downward wave,* whereas the second phase is called the *upward wave.* If all nodes know that the information is to be sent, the downward part can be omitted. The downward part can also be used to broadcast information throughout the tree.
Fully connected networks can handled similarly, as if they were two-level trees.

$\bigwedge_G K_i\varphi_i \rightsquigarrow E_G\psi$. This transition can be implemented by adding the downward part of a wave to a full wave for tree-structured networks. For fully connected graphs this can be implemented much simpler: every node sends its information to every other node. This is, for example, used in decentralized Two-Phase Commit algorithms [4].

$D_G\varphi \rightsquigarrow K_j\varphi$. This transition is a special case of the centralizer transition $\wedge_G K_i\varphi_i \rightsquigarrow K_j\psi$ if φ_i is the information of node i to compute φ. It should therefore be known in what way the information to compute φ is distributed over the nodes.

$D_G\varphi \rightsquigarrow E_G\varphi$. Again this case is similar to the case for $\wedge_G K_i\varphi_i \rightsquigarrow E_G\psi$.

How to classify layers?

Above we have given a classification of certain layers as knowledge transitions. An intuitive explanation has been given of why those layers belong to that transition class. In principle we have to *prove* that an algorithm satisfies a certain specification or knowledge transition. To give such a prove we would like stay as much as possible within the limits of well-known proof systems for parallel algorithms, such as Owicki-Gries style proofs [23, 2]. The programs we use in this paper can be treated as programs with an **await** construct to implement *send* and *receive* actions, and channels plus disjoint sets of variables. Thus we can give (non-knowledge based) proof outlines for programs in the usual way (see Apt and Olderog [2] for a extensive overview, and Janssen [13] for a discussion of the rules in our setting). In order to be able to prove knowledge based properties of programs we add the following rule, based on proof outlines for parallel programs, the rule for knowledge generalization, and the definition of $K_i\varphi$. Let φ_i and ψ_i be basic (not using the knowledge modalities) assertions, and let $S \vdash \Phi \rightsquigarrow \Psi$ denote that program S satisfies the knowledge transition $\Phi \rightsquigarrow \Psi$. We have the following proof rule.

$$\frac{\begin{array}{l}\{\varphi_i\}S_i\{\psi_i\},\ \text{for all } 1 \le i \le n,\\ \text{There exist valid proof outlines } \{\varphi_i\}S_i^{\dagger}\{\psi_i\} \text{ that}\\ \text{are interference free,}\end{array}}{S_1 \parallel \cdots \parallel S_n \quad \vdash \quad \bigwedge_{1\le i\le n} K_i\varphi_i \quad \rightsquigarrow \quad \bigwedge_{1\le i\le n} K_i\psi_i} \qquad \text{(knowledge and parallelism)}$$

Using rules for disjunction and conjunction, and the properties of our modalities, we can give derived rules for the other modalities, such as D_G and S_G. Soundness of the knowledge based rules follows in a rather straightforward way from the soundness of the rule for parallelism and the definition of validity of K_i [13].

4 Two-Phase Commit

The Two Phase Commit protocol is an example of *atomic commit protocols* that are used in distributed databases to guarantee *consistency* of the database. A distributed database consists of a number of sites connected by some network, where every site has a local database. Data are therefore distributed over a number of sites. In such a distributed database system *transactions,* consisting of a series of read and write actions, are executed. Reading and writing database items is be done by forwarding the action to the site where the item is stored. Terminating the transaction however involves *all* sites accessed in the transaction, as all sites must agree on the decision to be taken—which is either to *commit* or to *abort*—in order to guarantee consistency. In the case of an abort all changes made by the transaction are discarded, in the case of a commit they are made permanent. A protocol that guarantees such consistency is called an atomic commit protocol (ACP). We refer to Bernstein, Hadzilacos and Goodman [4] for more details.
In an ACP every participating process has one vote: `yes` or `no`, and every process can reach one out of two decisions: `commit` or `abort`. Here we do not take into account the possibility of communication failures or site failures, that is, we assume that every message sent is eventually delivered and that sites are working correctly.

First of all we should give a specification of the atomic commit problem as a knowledge transition. Thereafter we refine this transition to a sequence of (simpler) transitions. As we do not take failures into account the requirements can be phased as follows: *Given the votes of every participating process, each process should decide to* `commit` *iff every process* *has voted* yes. This is represented by the following knowledge transition. Let G be the set of participating processes and define $total\_vote = \texttt{yes}$ iff $\wedge_{i \in G} vote_i = \texttt{yes}$. So $total\_vote$ is not a variable but represents the combined values of all local variables $vote_i$.

$$\bigwedge_{i \in G} K_i vote_i \quad \leadsto \quad \bigwedge_{i \in G} K_i(total\_vote \ \wedge \ (dec_i = \texttt{commit} \Leftrightarrow total\_vote = \texttt{yes})).$$

Using the definitions of D_G and $total\_vote$ (given the distribution of the variables over the processes) this can be rewritten to

$$D_G total\_vote \quad \leadsto \quad \bigwedge_{i \in G} K_i(total\_vote \ \wedge \ (dec_i = \texttt{commit} \Leftrightarrow total\_vote = \texttt{yes})).$$

Deriving layered implementations

To derive implementations for knowledge transitions the following strategy is employed. We first check whether the transition under consideration has an immediate implementation for a certain network architecture. If this is the case, we're done. If not so we split the transition into two or more smaller transitions and continue with them. This "transition splitting" is in fact a real design step which can have consequences for the eventual implementation. The resulting *layered* algorithm is thereafter transformed to a *distributed* system using the transformation principle discussed in section 2.

In order to simplify matters, we first split of a transition "$\overset{2}{\leadsto}$," to be implemented by a final layer TPC_2 from the transition specified, where the decision is "executed," from the rest. In this final layer only local changes need to be performed, so its implementation is straightforward. This results in the following two transitions.

$$\begin{aligned} D_G total\_vote \quad & \overset{1}{\leadsto} \quad E_G total\_vote \\ & \overset{2}{\leadsto} \quad \bigwedge_{i \in G} K_i(total\_vote \ \wedge \ (dec_i = \texttt{commit} \Leftrightarrow total\_vote = \texttt{yes})). \end{aligned}$$

The first transition is a *confer transition* (see table 1), and can immediately be implemented for fully connected networks, by means of sending the votes to all other nodes, as was mentioned in the previous section. This would result in the following layer implementing "$\overset{1}{\leadsto}$."

```
$TPC_1 \triangleq$
  $\{D_G total\_vote\}$
  for $i \in G$ par
    for $i' \in G - \{i\}$ par $send((i, i'), vote_i)$ rof ;
    for $i' \in G - \{i\}$ par $receive((i, i'), vote_{ii'})$ rof
  rof
  $\{E_G total\_vote\}$
```

A second possibility is that we do *not* have a fully connected network, but some kind of tree structured network. In that case there is no apparent immediate solution to the above transition. So we split the transition again, and do so in the following way. Let $c \in G$ be some participating process.

$$D_G total\_vote \quad \overset{3}{\leadsto} \quad K_c total\_vote \quad \overset{4}{\leadsto} \quad E_G total\_vote.$$

The question now is what a sensible choice for c would be. Under the given assumption that we have a tree structured network an obvious choice is to take for c the root of the tree. There is however—under additional conditions—a second possibility. For a *linear tree* or chain, that is, a tree where every node has at most one downward edge, we can also take the (unique) leaf of the tree! We assume that the tree has at least two nodes. We first

discuss the former possibility.
To obtain a $D_G total\_vote \rightsquigarrow K_c total\_vote$ transition, which is a *centralize transition,* we can use a full wave as discussed in the previous section. This would result in the following implementation.

```
$TPC_3 \triangleq$
  $\{D_G total\_vote\}$
  **for** $i \in G$ **par**
    **if** $\neg root(i)$ **then** $receive(up(i), req_i)$ **fi** ;
    **for** $j \in down(i)$ **par** $send(j, req_i)$ **rof** ;
    **for** $j = (i, i') \in down(i)$ **par** $receive(j, vote_{ii'})$ **rof** ;
    **if** $(\forall i' \neq i \in S(i).\ vote_{ii'} = \mathtt{yes}) \wedge vote_i = \mathtt{yes}$ **then** $rep_i := \mathtt{yes}$ **else** $rep_i := \mathtt{no}$ **fi** ;
    **if** $\neg root(i)$ **then** $send(up(i), rep_i)$ **fi**
  **rof**
  $\{K_c total\_vote\}$
```

The value of *total_vote* is stored in rep_c.

In the linear case we have the following implementation. Let here $down(i)$ denote the unique edge downward for every node i. For the leaf of the linear tree this is nil, and $send(nil, e) \overset{\text{def}}{=}$ **skip** .

```
$TPC_{3'} \triangleq$
  $\{D_G total\_vote\}$
  **for** $i \in G$ **par**
    **if** $\neg root(i)$ **then** $receive(up(i), v_i)$ **fi** ;
    **if** $(root(i) \wedge vote_i = \mathtt{yes}) \vee (v_i = \mathtt{yes} \wedge vote_i = \mathtt{yes})$ **then** $send(down(i), \mathtt{yes})$
                                                        **else** $send(down(i), \mathtt{no})$ **fi**
  **rof**
  $\{K_c total\_vote\}$
```

The *total_vote* follows in this case from the values of v_c and $vote_c$.

To implement the second transition in this layer, "$\overset{4}{\rightsquigarrow}$", we can use a downward wave for the first case, and an upward wave in the linear case, as it is a *broadcast transition,* leading to the following implementations:

```
$TPC_4 \triangleq$
  $\{K_c total\_vote\}$
  **for** $i \in G$ **par**
    **if** $\neg root(i)$ **then** $receive(up(i), rep_i)$ **fi** ;
    **for** $j \in down(i)$ **par** $send(j, rep_i)$ **rof**
  **rof**,
  $\{E_G total\_vote\}$

$TPC_{4'} \triangleq$
  $\{K_c total\_vote\}$
  (**if** $v_c = \mathtt{yes} \wedge vote_c = \mathtt{yes}$ **then** $rep_c := \mathtt{yes}$ **else** $rep_c := \mathtt{no}$ **fi** ;
   $send(up(c), rep_c)$ )   $\|$
  **for** $i \in G - \{c\}$ **par**
    $receive(down(i), rep_i)$ ;
    $send(up(i), rep_i)$
  **rof**
  $\{E_G total\_vote\}$
```

The first two lines of $TPC_{4'}$ correspond to the process for the leaf node c.
A third possible network configuration is a special case of general networks: the ring. In this case we could again take a similar approach as in the previous case by appointing one node to gather all votes, and send the result through the ring (see [13]).

Transforming sequences of layers to parallel processes

We have derived a number of layered implementations for the Two-Phase Commit protocol consisting of two or three layers, where every layer is a parallel composition over all participants. The actual implementation we should arrive at must be of the form **for** $i \in G$ **par** $P(i)$ **rof**, that is, a single (sequential) process for every participant. The transformation from the layered to the distributed structure can be carried out using the ccl law, or more precisely, the transformation principle discussed in section 2. Using this principle we transform the layered implementations given above. As an example take the layered implementation for tree-structured networks. This layered implementation is

$$TPC_l \triangleq TPC_3 \bullet TPC_4 \bullet TPC_2.$$

Transforming this system immediately results in the distributed process TPC given below, which is IO-equivalent to the layered implementation. Therefore it satisfies the same initial knowledge transition specification.

```
TPC ≜
  for $i \in G$ par
    if $\neg root(i)$ then $receive(up(i), req_i)$ fi ;
    for $j \in down(i)$ par $send(j, req_i)$ rof ;
    for $j = (i, i') \in down(i)$ par $receive(j, vote_{ii'})$ rof ;
    if $(\forall i' \neq i \in S(i).\ vote_{ii'} = \texttt{yes}) \wedge vote_i = \texttt{yes}$ then $rep_i := \texttt{yes}$ else $rep_i := \texttt{no}$ fi ;
    if $\neg root(i)$ then $send(up(i), rep_i)$ fi ;
    if $\neg root(i)$ then $receive(up(i), rep_i)$ fi ;
    for $j \in down(i)$ par $send(j, rep_i)$ rof ;
    if $rep_i = \texttt{yes}$ then $dec_i := \texttt{commit}$ else $dec_i := \texttt{abort}$ fi
  rof.
```

This algorithm can be optimized by combining the two conditionals in the sixth and seventh line, but the basic structure remains the above.

Similarly, we can transform the layered implementation of the linear algorithm,

$$TPC'_l \triangleq TPC_{3'} \bullet TPC_{4'} \bullet TPC_2,$$

using the transformation rule to the following distributable algorithm TPC'.

```
TPC' ≜
  (if $\neg root(c)$ then $receive(up(c), v_c)$ fi ;
   if $v_c = \texttt{yes} \wedge vote_c = \texttt{yes}$ then $rep_c := \texttt{yes}$ else $rep_c := \texttt{no}$ fi ;
   $send(up(c), rep_c)$ ;
   if $rep_c = \texttt{yes}$ then $dec_i := \texttt{commit}$ else $dec_i := \texttt{abort}$ fi
  )
  ||
  for $i \in G - \{c\}$ par
    if $\neg root(i)$ then $receive(up(i), v_i)$ fi ;
    if $(root(i) \wedge vote_i = \texttt{yes}) \vee (v_i = \texttt{yes} \wedge vote_i = \texttt{yes})$ then $send(down(i), \texttt{yes})$
                                                                  else $send(down(i), \texttt{no})$
     fi ;
    $receive(down(i), rep_i)$ ;
    $send(up(i), rep_i)$ ;
    if $rep_i = \texttt{yes}$ then $dec_i := \texttt{commit}$ else $dec_i := \texttt{abort}$ fi
  rof.
```

Note that for the networks under consideration, which have at least two participants, the first guard ($\neg root(c)$) always evaluates to *true* and can therefore be removed.

These protocols correspond to a generalization of the *decentralized Two-Phase Commit* and the *linear Two-Phase Commit* as they are known from the literature. The result for the fully connected network is known as *centralized Two-Phase Commit.*

The same derivation style can be applied to other systems that have an underlying logical structure that is layered. In the full report some other examples are discussed as well, such as waves as sequences of layers, and an algorithm for computing minimal distances in networks.

5 Concluding remarks

In this paper we have discussed how to use knowledge based logics in the layered design of distributed systems. The contribution of this paper is twofold. First of all we have given a classification scheme for protocol layers as knowledge transitions. Secondly we have shown how such knowledge transitions can be used to derive layered implementations of protocols. Thus we have use knowledge based logics to give generic specifications of program layers and protocols.
We have shown that this design principle applies to a number of algorithms. In principle, any algorithm that can be viewed as a layered system should fit in this framework, which concerns a substantial class of algorithms. There exist however algorithms that cannot be written as layered systems, for example, highly interactive systems such as memories, or so-called retro-active systems (see Janssen [12] for a discussion of these problems).

The role of the logic has been limited to the specification of knowledge transitions. It might be interesting to use that logic to prove the layers correct themselves, possible in the style of van Hulst and Meyer [28]. Possibly such ideas would allow the approach presented to be extended to non-layered systems as well. The advantage of the approach presented here however is that the extensions to well-known techniques for program verification needed in this approach are rather limited.
We have used epistemic logic primarily as a logic to express locality of information. In [29] Wieczorek proposes a logic with modalities that directly express location. This logic however is weaker in the sense that it does not allow to combine information of different locations using the properties of the knowledge modalities.

Another interesting approach using knowledge based logics is to use program constructs that allow for the use of knowledge based expressions. The notify constructs as introduced by Moses and Kislev [21] is an example thereof. Furthermore one can use actions guarded by knowledge based expressions instead of normal boolean guards. Such programs are discussed by Fagin et al. in [9] and Moses and Kislev [21]. One of the difficulties with these programs is however that the transition of a knowledge based program to an ordinary program has not yet been formalized. Possibly classification schemes as introduced here combined with layered derivation can be of help to formalize this transition.

Acknowledgements. The author would like to thank Mannes Poel for detailed reading of the manuscript, and Yoram Moses, Wim Koole and John-Jules Meyer for useful comments on this work.

References

[1] G. Andrews. *Concurrent Programming — Principles and Practice.* The Benjamin/Cummings Publishing Company, 1991.

[2] K. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs.* Springer-Verlag, 1991.

[3] R. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[5] R. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[6] C. Chou and E. Gafni. Understanding and verifying distributed algorithms using stratified decomposition. In *Proceeding 7th ACM Symposium on Principles of Distributed Computing*, 1988.

[7] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proceedings 15th International Symposium on Fault-Tolerant Computing*, 1985.

[8] T. Elrad and N. Francez. Decomposition of distributed programs into communication closed layers. *Science of Computer Programming*, 2:155–173, 1982.

[9] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning About Knowledge.* MIT Press, 1995. To appear.

[10] J. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.

[11] J. Halpern and L. Zuck. A little knowledge goes a long way: Knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM*, 39(3):449–478, 1992.

[12] W. Janssen. *Layered Design of Parallel Systems.* PhD thesis, University of Twente, 1994.

[13] W. Janssen. Layers as knowledge transitions in the design of distributed systems. Technical Report 94-71, University of Twente, 1994.

[14] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In *Proceedings of CONCUR '91, LNCS 527*, pages 298–316. Springer-Verlag, 1991.

[15] W. Janssen and J. Zwiers. Protocol design by layered decomposition, a compositional approach. In J. Vytopil, editor, *Proceedings Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 571*, pages 307–326. Springer-Verlag, 1992.

[16] B. Jonsson. Modular verification of asynchronous networks. In *Proceedings 6th ACM Symposium on Principles of Distributed Computing*, pages 152–166, 1987.

[17] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2), 1992.

[18] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions.* Morgan Kaufman Publishers, 1994.

[19] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings 6th ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[20] J.-J. Meyer, W. van der Hoek, and G. Vreeswijk. Epistemic logic for computer science: A tutorial. *Bulletin of the EATCS, numbers 44 and 45*, 1991.

[21] Y. Moses and O. Kislev. Knowledge-oriented programming, (extended abstract). In *Proceedings 12th ACM Symposium on Principles of Distributed Computing*, pages 261–270. ACM, 1993.

[22] S. Mullender, editor. *Distributed Systems.* Addison-Wesley, second edition, 1993.

[23] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[24] M. Poel and J. Zwiers. Layering techniques for development of parallel systems. In G. v. Bochmann and D. Probst, editors, *Proceedings Computer Aided Verification, LNCS 663*, pages 16–29. Springer–Verlag, 1992.

[25] M. Raynal and J.-M. Helary. *Synchronization and control of distributed systems and programs.* John Wiley & Sons, 1990.

[26] F. Stomp and W.-P. de Roever. A correctness proof of a distributed minimum-weight spanning tree algorithm (extended abstract). In *Proceedings of the 7th ICDCS*, 1987.

[27] F. Stomp and W.-P. de Roever. A principle for sequential reasoning about distributed systems. *Formal Aspects of Computing*, 6(6):716–737, 1994.

[28] M. van Hulst and J.-J. Meyer. An epistemic proof system for parallel processes. In R. Fagin, editor, *Proceedings 5th TARK*, pages 243–254. Morgan Kaufmann, 1994.

[29] M. Wieczorek. *Locative Temporal Logic and Distributed Real-Time Systems.* PhD thesis, Catholic University of Nijmegen, 1994.

[30] J. Zwiers and W. Janssen. Partial order based design of concurrent systems. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX School/Symposium "A Decade of Concurreny", Noordwijkerhout, 1993, LNCS 803*, pages 622–684. Springer-Verlag, 1994.

Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs[*]

Jens Knoop [†] Bernhard Steffen[*] Jürgen Vollmer [‡]

Abstract

In this paper we show how to construct optimal bitvector analysis algorithms for parallel programs with shared memory that are as efficient as their purely sequential counterparts, and which can easily be implemented. Whereas the complexity result is rather obvious, our optimality result is a consequence of a new Kam/Ullman-style Coincidence Theorem. Thus using our method, the standard algorithms for sequential programs computing liveness, availability, very business, reaching definitions, definition-use chains, or performing partially redundant expression and assignment elimination, partial dead code elimination or strength reduction, can straightforward be transferred to the parallel setting at almost no cost.

Keywords: Parallelism, interleaving semantics, synchronization, program optimization, data flow analysis, bitvector problems, definition-use chains, partially redundant expression elimination, partial dead code elimination.

1 Motivation

Parallel implementations are of growing interest, as they are more and more supported by modern hardware environments. However, despite its importance [SHW, SW, WS], there is currently very little work on classical data flow analysis for parallel languages. Probably, the reason for this deficiency is that a naive adaptation fails [MP] and the straightforward correct adaptation needs an unacceptable effort, which is caused by considering all interleavings that manifest the possible executions of a parallel program.

Thus, either heuristics are proposed to avoid the consideration of all the interleavings [McD], or restricted situations are considered, which do not require to consider the interleavings at all. E.g., in [GS] data independence of parallel components is required. Thus the result of a parallel execution does not depend on the particular choice of the interleaving, which is exploited for the construction of an optimal and efficient algorithm determining the reaching-definition information. Completely different is the approach of abstract interpretation-based state space reduction proposed in [CH1, CH2], which allows general synchronization mechanisms but still requires the construction of an appropriately reduced version of the global state space which is often still unmanageable.

In this paper we show how to construct arbitrary bitvector analysis algorithms for parallel programs with shared memory that

1. optimally cover the phenomenon of *interference*

2. are as *efficient* as their sequential counterparts and

[*]For an extended version of this paper see [KSV1].

[†]Fakultät für Mathematik und Informatik, Universität Passau, Innstrasse 33, D-94032 Passau, Germany. E-mail: {knoop,steffen}@fmi.uni-passau.de

[‡]Institut für Programm- und Datenstrukturen, Universität Karlsruhe, Vincenz-Prießnitz-Straße 3, D-76128 Karlsruhe, Germany. E-mail: vollmer@ipd.info.uni-karlsruhe.de

3. easy to implement.

The first property is a consequence of a Kam/Ullman-style ([KU]) Coincidence Theorem for bitvector analyses stating that the *parallel meet over all paths (PMOP)* solution, which specifies the desired properties, coincides with our *parallel bitvector maximal fixed point (PMFP$_{BV}$)* solution, which is the basis of our algorithm. This result is rather surprising, as it states that although the various interleavings of the executions of parallel components are semantically different, they need not be considered during bitvector analysis, which is the key observation of this paper.

The second property is a simple consequence of the fact that our algorithms behave like standard bitvector algorithms. In particular, they do *not* require the consideration of any kind of global state space. This is important, as even the corresponding reduced state spaces would usually still be exponential in size.

The third property is due to the fact, that only a minor modification of the sequential bitvector algorithm needs to be applied after a preprocess consisting of a single fixed point routine (cf. Section 3.4).

Thus all the well-known algorithms for liveness, availability, very business, reaching definitions, definition-use chains (cf. [He]), partially redundant expression elimination (cf. [DRZ, KRS1, MR]), partial dead code elimination (cf. [KRS3]), partially redundant assignment elimination (cf. [KRS4]), or strength reduction (cf. [Dh, JD, KRS2]) can be adapted for parallel programs at almost no cost on the runtime and the implementation side.

The next section will recall the sequential situation, while Section 3 develops the corresponding notions for parallel programs. Subsequently, Section 4 sketches some applications of our algorithm and Section 5 contains our conclusions. The Appendix, finally, contains the detailed algorithm.

2 Sequential Programs

In this section we summarize the sequential setting of data flow analysis.

2.1 Representation

In the sequential setting it is common to represent procedures as *directed flow graphs* $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set N and edge set E (cf. [He]). Nodes $n \in N$ represent the statements, edges $(n, m) \in E$ the nondeterministic branching structure of the procedure under consideration, and $\mathbf{s}$ and $\mathbf{e}$ denote the unique *start node* and *end node* of G, which are assumed to possess no predecessors and successors, respectively, and to represent the empty statement `skip`. $pred_G(n)=_{df} \{ m \,|\, (m, n) \in E \}$ and $succ_G(n)=_{df} \{ m \,|\, (n, m) \in E \}$ denote the set of all immediate predecessors and successors of a node n, respectively. A *finite path* in G is a sequence $(n_1, \ldots, n_q)$ of nodes such that $(n_j, n_{j+1}) \in E$ for $j \in \{1, \ldots, q-1\}$. $\mathbf{P}_G[m, n]$ denotes the set of all finite paths from m to n, and $\mathbf{P}_G[m, n[$ the set of all finite paths from m to a predecessor of n. Moreover, $\lambda(p)$ denotes the number of node occurrences of p, and ε the unique path of length 0. Finally, every node $n \in N$ is assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$.

2.2 Data Flow Analysis

Data flow analysis (DFA) is concerned with the static analysis of programs in order to support the generation of efficient object code by "optimizing" compilers (cf. [He, MJ]). For imperative languages, DFA provides information about the program states that may occur at some given program points during execution. Theoretically well-founded are DFAs that are based on *abstract* *interpretation* (cf. [CC1, Ma]). The point of this approach is to replace the "full" semantics by a simpler more abstract version, which is tailored to deal with a specific problem. Usually, the abstract semantics is specified by a *local semantic functional*

$$[\![\ ]\!] : N \to (\mathcal{C} \to \mathcal{C})$$

which gives abstract meaning to every program statement in terms of a transformation function from a complete lattice $(\mathcal{C}, \sqcap, \sqsubseteq, \bot, \top)$ into itself, where the elements of $\mathcal{C}$ express the DFA-information of interest.[1]

Since $\mathbf{s}$ and $\mathbf{e}$ are assumed to represent the empty statement `skip` they are associated with the identity $Id_{\mathcal{C}}$ on $\mathcal{C}$. A local semantic functional $[\![\ ]\!]$ can easily be extended to cover finite paths as well. For every path $p = (n_1, \ldots, n_q) \in \mathbf{P}_G[m, n]$, we define:

$$[\![\, p \,]\!] =_{df} \begin{cases} Id_{\mathcal{C}} & \text{if } p \equiv \varepsilon \\ [\![\, (n_2, \ldots, n_q) \,]\!] \circ [\![\, n_1 \,]\!] & \text{otherwise} \end{cases}$$

2.2.1 The *MOP*-Solution of a DFA

The *MOP*-solution — the solution of the *meet over all paths (MOP)* strategy in the sense of Kam and Ullman [KU] — defines the intuitively desired solution of a DFA. This strategy directly mimics possible program executions in that it "meets" (intersects) all informations belonging to a program path reaching the program point under consideration.

The *MOP*-Solution: $\forall\, n \in N\ \forall\, c_0 \in \mathcal{C}.\ MOP_{(G,[\![\ ]\!])}(n)(c_0) = \bigsqcap \{\, [\![\, p \,]\!](c_0) \mid p \in \mathbf{P}_G[\mathbf{s}, n[\,\}$

In fact, this directly reflects our desires, but is in general not effective.

2.2.2 The *MFP*-Solution of a DFA

The point of the *maximal fixed point (MFP)* strategy in the sense of Kam and Ullman [KU] is to iteratively approximate the greatest solution of a system of equations which specifies the consistency between pre-conditions expressed in terms of $\mathcal{C}$:

Equation System 2.1

$$\mathbf{pre}(n) \;=\; \begin{cases} c_0 & \text{if } n = \mathbf{s} \\ \bigsqcap \{\, [\![\, m \,]\!](\mathbf{pre}(m)) \mid m \in pred_G(n) \,\} & \text{otherwise} \end{cases}$$

Denoting the greatest solution of Equation System 2.1 with respect to the start information $c_0 \in \mathcal{C}$ by $\mathbf{pre}_{c_0}$, the solution of the *MFP*-strategy is defined by:

The *MFP*-Solution: $\forall\, n \in N\ \forall\, c_0 \in \mathcal{C}.\ MFP_{(G,[\![\ ]\!])}(n)(c_0) = \mathbf{pre}_{c_0}$

For monotonic functionals,[2] this leads to a suboptimal but algorithmic description (see Algorithm A.1 in Appendix A). The question of optimality of the *MFP*-solution was elegantly answered by Kam and Ullman [KU]:

Theorem 2.2 (The (Sequential) Coincidence Theorem)
Given a flow graph $G = (N, E, \mathbf{s}, \mathbf{e})$, the MFP-solution and the MOP-solution coincide, i.e. $\forall\, n \in N\ \forall\, c_0 \in \mathcal{C}.\ MOP_{(G,[\![\ ]\!])}(n)(c_0) = MFP_{(G,[\![\ ]\!])}(n)(c_0)$, whenever all the semantic functions $[\![\, n \,]\!]$, $n \in N$, are distributive.[3]

[1] In the following $\mathcal{C}$ will always denote a complete lattice.

[2] A function $f : \mathcal{C} \to \mathcal{C}$ is called *monotonic* iff $\forall\, c, c' \in \mathcal{C}.\ c \sqsubseteq c'$ implies $f(c) \sqsubseteq f(c')$.

[3] A function $f : \mathcal{C} \to \mathcal{C}$ is called *distributive* iff $\forall\, C' \subseteq \mathcal{C}.\ f(\bigsqcap C') = \bigsqcap \{f(c) \mid c \in C'\}$. It is well-known that distributivity is a stronger requirement than monotonicity in the following sense: A function $f : \mathcal{C} \to \mathcal{C}$ is monotonic iff $\forall\, C' \subseteq \mathcal{C}.\ f(\bigsqcap C') \sqsubseteq \bigsqcap \{f(c) \mid c \in C'\}$.

2.2.3 The Functional Characterization of the *MFP*-Solution

From interprocedural DFA, it is well-known that the *MFP*-solution can alternatively be defined by means of a functional approach [SP]. Here, one iteratively approximates the greatest solution of a system of equations specifying consistency between functions $[\![n]\!]$, $n \in N$. Intuitively, a function $[\![n]\!]$ transforms data flow information that is assumed to be valid at the start node of the program into the data flow information being valid before the execution of n.

Definition 2.3 (The Functional Approach)
The functional $[\![\ ]\!] : N \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ *is defined as the greatest solution of the equation system given by:*

$$[\![n]\!] = \begin{cases} Id_{\mathcal{C}} & \textit{if } n = \mathbf{s} \\ \bigsqcap \{ [\![m]\!] \circ [\![m]\!] \,|\, m \in pred_G(n) \} & \textit{otherwise} \end{cases}$$

The following equivalence result is important [KS]:

Theorem 2.4 $\forall n \in N \ \forall c_0 \in \mathcal{C}.\ MFP_{(G,[\![\ ]\!])}(n)(c_0) = [\![n]\!](c_0)$

The functional characterization of the *MFP*-solution will be the (intuitive) key for computing the parallel version of the maximal fixed point solution. As we are only dealing with Boolean values later on, this characterization can easily be coded back into the standard form.

3 Parallel Programs

As usual, we consider a parallel imperative programming language with an interleaving semantics. Formally, this means that we view parallel programs semantically as 'abbreviations' of usually much larger nondeterministic programs, which result from a product construction between parallel components (cf. [CC2, CH1, CH2]). In fact, in the worst case, the size of the nondeterministic 'product' program grows exponentially in the number of parallel components of the corresponding parallel program. This immediately clarifies the dilemma of data flow analysis for parallel programs: even though it can be reduced to standard data flow analysis on the corresponding nondeterministic program, this approach is unacceptable in practice for complexity reasons. Fortunately, as we will see in Section 3.3, bitvector analyses, which are most relevant in practice, can be performed as efficiently on parallel programs as on sequential programs.

The following section establishes the notational background for the formal development and the proofs. One could therefore try to immediately continue with Section 3.3 and to 'backtrack' to Section 3.1 at need.

3.1 Representation

Syntactically, parallelism is expressed by means of a `par` statement whose components are assumed to be executed independently and in parallel on a shared memory.[4] As usual, we assume that there are neither jumps leading into a component of a `par` statement from outside nor vice versa.

Similarly to [GS], we represent a parallel program by a nondeterministic *parallel flow graph* $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ with node set N^* and edge set E^*. Except for subgraphs representing `par` statements a parallel flow graph is a nondeterministic flow graph in the sense of Section 2,

[4]Integrating a replicator statement in order to allow a dynamical process creation is straightforward (cf. [CH2, Vo2]).

i.e., nodes $n \in N^*$ represent the statements, edges $(m,n) \in E^*$ the nondeterministic branching structure of the procedure under consideration, and $\mathbf{s}^*$ and $\mathbf{e}^*$ denote the distinct *start node* and *end node*, which are assumed to possess no predecessors and successors, respectively. As in Section 2, we assume that every node $n \in N^*$ lies on a path from $\mathbf{s}^*$ to $\mathbf{e}^*$, and that the start and the end nodes of parallel flow graphs represent the empty statement `skip`. Additionally, $pred_{G^*}(n) =_{df} \{\, m \,|\, (m,n) \in E^* \,\}$ and $succ_{G^*}(n) =_{df} \{\, m \,|\ (n,m) \in E^* \,\}$ denote the set of all immediate predecessors and successors of a node $n \in N^*$, respectively.

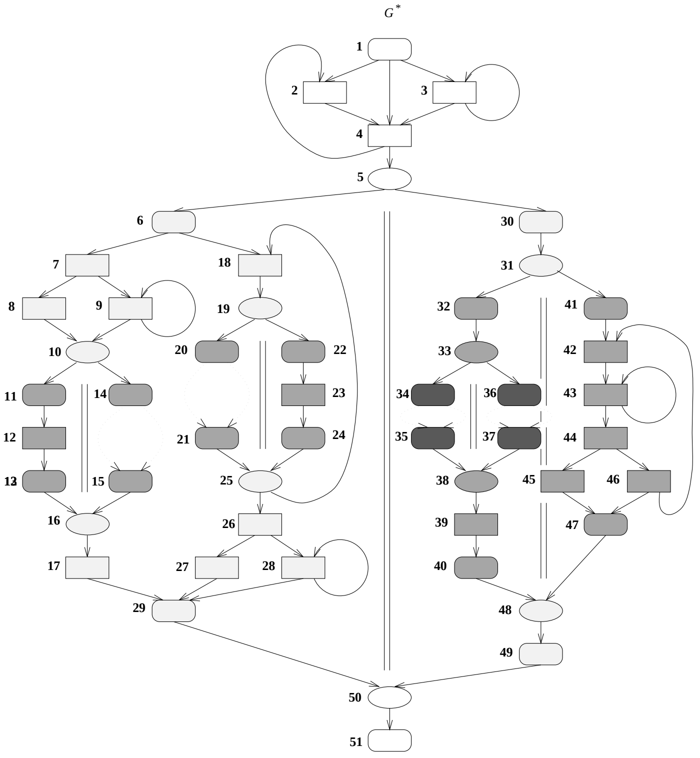


Figure 1: The Parallel Flow Graph G^*

A `par` statement as well as every of its components are also considered parallel flow graphs (cf. Figure 1 for illustration). The start node and the end node of a graph representing a `par` statement have the start nodes and the end nodes of the component flow graphs as their only successors and predecessors, respectively. The set of all subgraphs of G^* representing a `par` statement is denoted by $\mathcal{G}_\mathcal{P}(G^*)$. Additionally,

$$\mathcal{G}_\mathcal{P}^{max}(G^*) =_{df} \{\, G \in \mathcal{G}_\mathcal{P}(G^*) \,|\, \forall\, G' \in \mathcal{G}_\mathcal{P}(G^*).\ G \subseteq G' \Rightarrow G = G' \,\}$$

denotes the set of *maximal* graphs of $\mathcal{G}_\mathcal{P}(G^*)$.[5] Moreover, for $G' \in \mathcal{G}_\mathcal{P}(G^*)$, $\mathcal{G}_\mathcal{C}(G')$ denotes the set of component flow graphs of G', and $CpNodes(G') =_{df} N' \backslash \{\mathbf{s}', \mathbf{e}'\}$ the set of nodes of its component flow graphs.[6] It is worth noting that for $G \in \mathcal{G}_\mathcal{P}(G^*)$ every component flow graph $G' \in \mathcal{G}_\mathcal{C}(G)$ and also G itself is a single-entry/single-exit region of G^*. Moreover, we introduce the following abbreviations for the sets of start nodes and end nodes of graphs of $\mathcal{G}_\mathcal{P}(G^*)$:

$$N^*_N =_{df} \{\, \mathbf{s} \,|\, G \in \mathcal{G}_\mathcal{P}(G^*) \,\} \quad \text{and} \quad N^*_X =_{df} \{\, \mathbf{e} \,|\, G \in \mathcal{G}_\mathcal{P}(G^*) \,\}$$

[5]For parallel flow graphs G and G' we define: $G \subseteq G'$ if and only if $N \subseteq N'$ and $E \subseteq E'$.

[6]We use the convention that the node set and the edge set, and the start node and the end node of a flow graph carry the same marking as the flow graph itself. Hence, G and G' stand for the expanded versions $G = (N, E, \mathbf{s}, \mathbf{e})$ and $G' = (N', E', \mathbf{s}', \mathbf{e}')$, respectively.

Additionally, we need the functions *Nodes*, *start*, *end*, *pfg*, and *cfg*. The functions *Nodes*, *start* and *end* map a flow graph to its node set, and its start node and end node, respectively. The function *pfg* maps a node n occurring in some flow graph $G' \in \mathcal{G}_{\mathcal{P}}(G^*)$ to the smallest flow graph of $\mathcal{G}_{\mathcal{P}}(G^*)$ containing n; and it maps the remaining nodes n of N^* to G^*, i.e.,

$$pfg(n) =_{df} \begin{cases} \bigcap\{\, G' \in \mathcal{G}_{\mathcal{P}}(G^*) \mid n \in Nodes(G') \,\} & \text{if } n \in Nodes(\mathcal{G}_{\mathcal{P}}^{max}(G^*)) \\ G^* & \text{otherwise} \end{cases}$$

Similarly, the function *cfg* maps a node n occurring in a component flow graph of some graph $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ to the smallest component flow graph containing n; and it maps the remaining nodes n of N^* to G^*, i.e.,

$$cfg(n) =_{df} \begin{cases} \bigcap\{\, G' \in \mathcal{G}_{\mathcal{C}}(\mathcal{G}_{\mathcal{P}}(G^*)) \mid n \in Nodes(G') \,\} & \text{if } n \in CpNodes(\mathcal{G}_{\mathcal{P}}^{max}(G^*)) \\ G^* & \text{otherwise} \end{cases}$$

Both *pfg* and *cfg* are well-defined, since `par` statements in a program are either unrelated or properly nested.

Finally, given a parallel flow graph G we define an associated sequential flow graph G_{seq}, which results from G by replacing all nodes belonging to a component flow graph of some graph $G' \in \mathcal{G}_{\mathcal{P}}^{max}(G)$ together with all edges starting or ending in such a node by an edge leading from $start(G')$ to $end(G')$. Note that G_{seq} is a nondeterministic sequential flow graph in the sense of Section 2. This is illustrated in Figure 2, which shows the sequentialized version of the parallel flow graph of Figure 1.

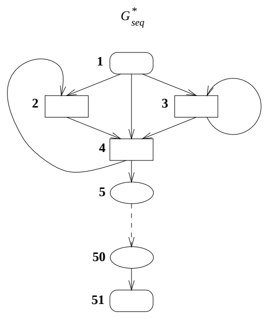


Figure 2: G^*_{seq}

Interleaving Predecessors

Given a sequential flow graph G, the set of nodes that might dynamically precede a node n is precisely given by the set of its static predecessors $pred_G(n)$. Given a parallel flow graph, however, the interleaving of statements of parallel components must be taken care of. In fact, nodes n occurring in a component of some `par` statement additionally can have all nodes as dynamic predecessors, whose execution may be interleaved with that of n. For example, in the program of Figure 1 the execution of node **24**, whose only static predecessor is node **23**, may be interleaved with the execution of the nodes **20**, **21**, and **30**, ..., **49**. We denote these 'potentially parallel' nodes as *interleaving predecessors*. The set of all interleaving predecessors of a node $n \in N^*$ is recursively defined by means of the function $Pred_{G^*}^{Itlvg} : N^* \rightarrow \mathcal{P}(N^*)$, where $\mathcal{P}$ denotes the power set operator and *mpe-pfg* a function, which maps a node $n \in N^*$ to its *minimal properly enclosing* graph of $\mathcal{G}_{\mathcal{P}}(G^*) \cup \{G^*\}$:

$$Pred^{Itlvg}_{G^*}(n) =_{df} \begin{cases} \emptyset & \text{if } N^* \backslash CpNodes(\mathcal{G}^{max}_{\mathcal{P}}(G^*)) \\ \\ CpNodes(mpe\text{-}pfg(n)) \backslash Nodes(cfg(n)) \cup & \\ Pred^{Itlvg}_{G^*}(start(cfg(start(mpe\text{-}pfg(n))))) & \text{otherwise} \end{cases}$$

where *mpe-pfg* is defined by:

$$mpe\text{-}pfg(n) =_{df} \begin{cases} pfg(start(cfg(n))) & \text{if } n \in N^*_N \cup N^*_X \\ \\ pfg(n) & \text{otherwise} \end{cases}$$

Program Paths of Parallel Programs

As mentioned already, the interleaving semantics of an imperative parallel programming language can be defined via a translation that reduces parallel programs to (much larger) nondeterministic programs. However, there is also an alternative way to characterize the node sequences constituting a parallel (program) path, following in spirit the definition of an interprocedural program path as proposed by Sharir and Pnueli [SP]. They start by interpreting every branch statement purely nondeterministically, which allows to simply use the definition of *finite path* as introduced in Section 2. This results in a superset of the set of all interprocedurally valid paths, which they now define by means of an additional consistency condition. In our case, we are forced to define our consistency condition on arbitrary node sequences, as the consideration of interleavings invalidates the first step. Here, the following notion of well-formedness is important.

Definition 3.1 (G-Well-Formedness)
Let G be a (parallel) flow graph, and $p =_{df} (n_1, \ldots, n_q)$ be a sequence of nodes. Then p is G-well-formed *if and only if*

1. *the projection $p\downarrow_{G_{seq}}$ of p onto G_{seq} lies in $\mathbf{P}_{G_{seq}}[start(G_{seq}), end(G_{seq})]$*
2. *for all node occurrences $n_i \in N^*_N$ of the sequence p there exists a $j \in \{i+1, \ldots, q\}$ such that*
   - *(a) $n_j \in N^*_X$,*
   - *(b) n_j is the successor of n_i on $p\downarrow_{G_{seq}}$ and*
   - *(c) the sequence $(n_{i+1}, \ldots, n_{j-1})$ is G'-well-formed for all $G' \in \mathcal{G}_{\mathcal{C}}(pfg(n_i))$.*

Now the set of parallel paths is defined as follows.

Definition 3.2 (Parallel Path)
Let $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ be a parallel flow graph, and $p =_{df} (n_1, \ldots, n_q)$ be a sequence of nodes of N^*. Then:

1. *p is a* parallel path from $\mathbf{s}^*$ to $\mathbf{e}^*$ *if and only if p is G^*-well-formed.*
2. *p is a* parallel path from n_1 to n_q *if it is a subpath of some parallel path from $\mathbf{s}^*$ to $\mathbf{e}^*$.*

$\mathbf{PP}_{G^*}[m, n]$ *denotes the set of all parallel paths from m to n, and* $\mathbf{PP}_{G^*}[m, n[$ *the set of all parallel paths from m to a (static or interleaving) predecessor of n, defined by*

$$\mathbf{PP}_{G^*}[m, n[=_{df} \{(n_1, \ldots, n_q) \mid (n_1, \ldots, n_q, n_{q+1}) \in \mathbf{PP}_{G^*}[m, n]\}$$

3.2 Data Flow Analysis of Parallel Programs

As for a sequential program, a DFA for a parallel program is completely specified by means of a local semantic functional

$$[\![\ ]\!] : N^* \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$$

which gives abstract meaning to every node n of a parallel flow graph G^* in terms of a function from $\mathcal{C}$ to $\mathcal{C}$.

As in the sequential case it is straightforward to extend a local semantic functional to cover also finite parallel paths. Thus, given a node n of a parallel program G^*, the parallel version of the *MOP*-solution is clear, and as in the sequential case, it marks the desired solution to the considered data flow problem:

The *PMOP*-Solution:

$$\forall\, n \in N^*\ \forall\, c_0 \in \mathcal{C}.\ \mathit{PMOP}_{(G^*,[\![\ ]\!])}(n)(c_0) = \bigsqcap \{\, [\![\, p\,]\!](c) \mid p \in \mathbf{PP}_{G^*}[\mathbf{s}^*, n[\ \}$$

Referring to the nondeterministic 'product program', which explicitly represents all the possible interleavings, would allow us to straightforward adapt the sequential situation and to state a Coincidence Theorem. However, this would not be of much practical use, as this approach would require to define the *MFP*-solution relative to the potentially exponential product program. Fortunately, as we will see in the next section, for bitvector algorithms there exists an elegant and efficient way out.

3.3 Bitvector Analyses

Bitvector problems can be characterized by the simplicity of their local semantic functional

$$[\![\ ]\!] : N^* \rightarrow (\mathcal{B} \rightarrow \mathcal{B})$$

which specifies the effect of a node n on a particular component of the bitvector (see Section 4 for illustration). Here $\mathcal{B}$ is the lattice $(\{\mathit{ff}, \mathit{tt}\}, \sqcap, \sqsubseteq)$ of Boolean truth values with $\mathit{ff} \sqsubseteq \mathit{tt}$ and the logical 'and' as meet operation $\sqcap$, or its dual counterpart with $\mathit{tt} \sqsubseteq \mathit{ff}$ and the logical 'or' as meet operation $\sqcap$.

Despite their simplicity, bitvector problems are highly relevant in practice, as they include problems like liveness, availability, very business, reaching definitions, definition-use chains, partially redundant expression and assignment elimination, partial dead code elimination or strength reduction.

We are now going to show, how to optimize the effort for computing the *PMOP*-solution. This requires the consideration of the semantic domain $\mathcal{F}_{\mathcal{B}}$ consisting of the monotonic Boolean functions $\mathcal{B} \rightarrow \mathcal{B}$. Obviously we have:

Proposition 3.3 *1. $\mathcal{F}_{\mathcal{B}}$ simply consists of the constant functions $Const_{tt}$ and $Const_{ff}$, together with the identity $Id_{\mathcal{B}}$ on $\mathcal{B}$.*

2. $\mathcal{F}_{\mathcal{B}}$, together with the pointwise ordering between functions, forms a complete lattice with least element $Const_{ff}$ and greatest element $Const_{tt}$, which is closed under function composition.

3. All functions of $\mathcal{F}_{\mathcal{B}}$ are distributive.

The key to the efficient computation of the 'interleaving effect' is based on the following simple observation, which pinpoints the specific nature of a domain of functions $M \rightarrow M$, M any set, that only consists of constant functions and the identity.

Lemma 3.4 (Main-Lemma)
Let $f_i : \mathcal{F}_{\mathcal{B}} \rightarrow \mathcal{F}_{\mathcal{B}}$, $1 \leq i \leq q$, $q \in \mathbb{N}$, *be functions from* $\mathcal{F}_{\mathcal{B}}$ *to* $\mathcal{F}_{\mathcal{B}}$. *Then we have:*

$$\exists\, k \in \{1, \ldots, q\}.\ f_q \circ \ldots \circ f_2 \circ f_1 = f_k \ \wedge\ \forall\, j \in \{k+1, \ldots, q\}.\ f_j = Id_{\mathcal{B}}$$

The essence of this lemma for our application is that it restricts the way of possible interference within a parallel program: if there is any interference than this interference is due to a single statement within a parallel component. Combining this observation with the fact that for $m \in Pred_{G^*}^{Itlvg}(n)$, there exists a parallel path leading to n whose last step requires the execution of m, we obtain that the potential of interference, which in general would be given in terms of paths, is fully characterized by the set $Pred_{G^*}^{Itlvg}(n)$. In fact, considering the computation of universal properties that are described by maximal fixed points (the computation of minimal fixed points requires the dual argument), the obvious existence of a path to n that does not require the execution of any statement of $Pred_{G^*}^{Itlvg}(n)$ implies that the only effect of interference is 'destruction'. This motivates the introduction of the following predicate:

$NonDestructed : N^* \rightarrow \mathcal{B}$ defined by

$$\forall\, n \in N^*.\ NonDestructed(n) =_{df} \bigwedge \{\, [\![\, m \,]\!](tt) \ \mid\ m \in Pred_{G^*}^{Itlvg}(n) \,\}$$

which indicates that no node of a parallel component destroys the property under consideration, i.e. $[\![\, m \,]\!] \neq Const_{ff}$ for all $m \in Pred_{G^*}^{Itlvg}(n)$. Note that only the constant function induced by this predicate is used in Definition 3.7 to model interference, and in fact, Theorem 3.8 guarantees that this modelling is sufficient. Obviously, this predicate is easily and efficiently computable. Algorithm B.1 computes it as a side result.

Besides taking care of possible interference, we also need to take care of the synchronization required by nodes in N_X^*: in order to leave a parallel statement, all parallel components are required to terminate. The information that is necessary to model this effect can be computed by a hierarchical algorithm that only considers purely sequential programs. The central idea coincides with that of interprocedural analysis [KS]: we need to compute the effect of complete subgraphs, or in this case of complete parallel components. This information is computed in an 'innermost' fashion and then propagated to the next surrounding parallel statement. The following definition describes the complete three-step procedure:

1. Terminate, if G does not contain any parallel components. Otherwise, select successively all maximal flow graphs $G' \in \mathcal{G}_{\mathcal{P}}(G)$ that do not contain a parallel statement, and determine the effect $[\![\![\, G' \,]\!]\!]$ of this (purely sequential) graph according to the equational system of Definition 2.3 with respect to the local semantic functional $[\![\ ]\!]'_{seq} : N'_{seq} \rightarrow \mathcal{F}_{\mathcal{B}}$ given by

$$[\![\, n \,]\!]'_{seq} =_{df} \begin{cases} Id_{\mathcal{B}} \sqcap Const_{NonDestructed(n)} & \text{if } n \in N_N^* \\ [\![\![\, pfg(n) \,]\!]\!]^* & \text{if } n \in N_X^* \\ [\![\, n \,]\!] & \text{otherwise} \end{cases}$$

2. Compute the effect $[\![\![\, G'' \,]\!]\!]^*$ of the innermost parallel statements G'' of G by

$$[\![\![\, G'' \,]\!]\!]^* = \bigsqcap \{\, [\![\![\, end(G'_{seq}) \,]\!]\!] \ \mid\ G' \in \mathcal{G}_{\mathcal{C}}(G'') \,\}$$

3. Transform G by replacing all innermost parallel statements $G'' = (N'', E'', \mathbf{s}'', \mathbf{e}'')$ by $(\{\mathbf{s}'', \mathbf{e}''\}, \{(\mathbf{s}'', \mathbf{e}'')\}, \mathbf{s}'', \mathbf{e}'')$, and replace the local semantics of $\mathbf{s}''$ and $\mathbf{e}''$ by $Id_{\mathcal{B}} \sqcap \bigsqcap \{\, [\![\, n \,]\!] \mid n \in N'' \}$ and $[\![\![\, G'' \,]\!]\!]^*$, respectively. Continue with step 1.

This three step algorithm is a straightforward hierarchical adaptation of the algorithm for computing the functional version of the *MFP*-solution for the sequential case. Only the third step realizing the synchronization at nodes in N^*_X needs some explanation, which is summarized in the following lemma.

Lemma 3.5 *The PMOP-solution of a parallel flow graph G that only consists of purely sequential parallel components $G_1, \ldots, G_k$ is given by:*

$$\mathit{PMOP}_{(G,[\![\ ]\!])}(\mathit{end}(G)) = \bigsqcap\{\, [\![\ \mathit{end}(G_i)\ ]\!] \mid 1 \leq i \leq k \,\}$$

Also the proof of this lemma is a consequence of the Main Lemma 3.4. As a single statement is responsible for the entire effect of a path, the effect of each complete path through a parallel statement is already given by some path through one of the parallel components (the one containing the vital statement). Thus in order to model the effect (or *PMOP*-solution) of a parallel statement, it is sufficient to meet the effects of all paths that are local to one of the components, and it is exactly this fact, which is formalized in Lemma 3.5.

Now the following theorem can be proved by means of a straightforward inductive extension of the functional version of the sequential Coincidence Theorem 2.2, which is tailored to cover complete paths, i.e. paths going from the start to the end of a parallel statement:

Theorem 3.6 (The Hierarchical Coincidence Theorem)
Let $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ be a parallel flow graph, and $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ a local semantic functional. Then we have:

$$\mathit{PMOP}_{(G,[\![\ ]\!])}(\mathit{end}(G)) = [\![\, G\,]\!]^*$$

After this hierarchical preprocess the following modification of the equation system for sequential bitvector analyses is optimal:

Definition 3.7 *The functional $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ is defined as the greatest solution of the equation system given by:*[7]

$$[\![\, n\,]\!] = \begin{cases} \mathit{Id}_{\mathcal{B}} & \text{if } n = \mathbf{s}^* \\ [\![\, \mathit{pfg}(n)\,]\!]^* \circ [\![\, \mathit{start}(\mathit{pfg}(n))\,]\!] \sqcap \mathit{Const}_{\mathit{NonDestructed}(n)} & \text{if } n \in N^*_X \\ \bigsqcap\{\, [\![\, m\,]\!] \circ [\![\, m\,]\!] \mid m \in \mathit{pred}_{G^*}(n)\} \sqcap \mathit{Const}_{\mathit{NonDestructed}(n)} & \text{otherwise} \end{cases}$$

This allows us to define the PMFP_{BV}-solution, a fixed point solution for the bitvector case, in the following fashion:

The PMFP_{BV}-Solution:

$$\mathit{PMFP}_{BV(G^*,[\![\ ]\!])} : N^* \to \mathcal{F}_{\mathcal{B}} \text{ defined by } \forall\, n \in N^*\ \forall\, b \in \mathcal{B}.\ \mathit{PMFP}_{BV(G^*,[\![\ ]\!])}(n)(b) = [\![\, n\,]\!](b)$$

As in the sequential case the PMFP_{BV}-strategy is practically relevant, because it can efficiently be computed (see Algorithm B.1 in Appendix B). The following theorem, whose proof can be found in [KSV1], now establishes that it also coincides with the desired *PMOP*-solution.

Theorem 3.8 (The Parallel Bitvector Coincidence Theorem)
Let $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ be a parallel flow graph, and $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ a local semantic functional. Then we have that the PMOP-solution and the PMFP_{BV}-solution coincide, i.e.,

$$\forall\, n \in N^*.\ \mathit{PMOP}_{(G^*,[\![\ ]\!])}(n) = \mathit{PMFP}_{BV(G^*,[\![\ ]\!])}(n)$$

[7]Note that $[\![\ ]\!]$ is the straightforward extension of the functional defined in Definition 2.3. Thus the overloading of notation is harmless, as no reference to the sequential version is made in this definition.

3.4 Performance and Implementation

Our algorithm is based on a functional version of an *MFP*-solution, as it is common for interprocedural analyses. However, as bitvector algorithms only deal with Boolean values, proceeding argument-wise, would simply require to apply a standard bitvector algorithm twice. In particular, for regular program structures, all the nice properties of bitvector algorithms apply. In fact, for the standard version of Algorithm B.1 a single execution is sufficient, as we can start here with the same start information as the standard sequential analysis. Thus, even if we count the effort for computing the predicate *NonDestructed* separately, our analysis would simply be a composition of four standard bitvector analyses. In practice, however, our algorithm behaves much better, as the existence of a single destructing statement allows us to skip the analysis of large parts of the program. In fact, in our experience, the parallel version often runs faster than the sequential version on a program of similar size.

The same argumentation also indicates a way for a cheap implementation on top of existing bitvector algorithms. However, we recommend the direct implementation of the functional version, which to our experience, runs even faster than the decomposed standard version. This is not too surprising, as the functional version only needs to consider one additional value and does not require the argumentwise application.

4 Applications

As mentioned in Section 1 and Section 3.3, bitvector problems have a broad scope of applications. In this section we present the local semantic functionals of four bitvector problems in order to give the flavour of a typical bitvector analysis. Moreover, these analyses are all practically relevant, since they are the central components of two algorithms for the computationally optimal placement of computations and assignments in a program, which eliminate all partially redundant expressions [KRS1] and all partially dead assignments in a program [KRS3], respectively.

According to [KRS1] a computationally optimal placement of computations in a program requires to compute the set of program points where a computation is *up-safe*, i.e., where it has been computed on every program path reaching the program point under consideration, and *down-safe*, i.e., where it will be computed on every program continuation reaching the end node of the program.[8] The DFA-problems for up-safety and down-safety are specified by the local semantic functionals $[\![\, n \,]\!]_{us}$ and $[\![\, n \,]\!]_{ds}$, respectively:

$$[\![\, n \,]\!]_{us} =_{df} \begin{cases} Const_{tt} & \text{if } Transp(n) \wedge Comp(n) \\ Id_{\mathcal{B}} & \text{if } Transp(n) \wedge \neg Comp(n) \\ Const_{ff} & \text{if } \neg Transp(n) \end{cases} \qquad [\![\, n \,]\!]_{ds} =_{df} \begin{cases} Const_{tt} & \text{if } Comp(n) \\ Id_{\mathcal{B}} & \text{if } \neg Comp(n) \wedge Transp(n) \\ Const_{ff} & \text{if } \neg(Comp(n) \vee Transp(n)) \end{cases}$$

Details on the complete placement transformation for parallel programs can be found in [KSV2].

According to [KRS3] all partially dead assignments in a program can be eliminated by successively moving assignments as far as possible in the direction of the control flow and by subsequently removing all assignments whose left hand side variable is dead after the execution of the assignment under consideration. In order to capture the second order effects of partial dead code elimination, this two step procedure is repeated until the programs eventually stabilizes. Below the local semantic functionals specifying the DFA-problems for the sinking of assignments $[\![\, n \,]\!]_{dl}$ and the detection of dead variables $[\![\, n \,]\!]_{dd}$ are presented, which are the central components of the algorithm of [KRS3]:

[8]Up-safety and down-safety are also known as *availability* and *anticipability (very business)*, respectively.

$$[\![n]\!]_{dd} =_{df} \begin{cases} Const_{tt} & \text{if } \neg Used(n) \wedge Mod(n) \\ Id_{\mathcal{B}} & \text{if } \neg(Used(n) \vee Mod(n)) \\ Const_{ff} & \text{if } Used(n) \end{cases} \qquad [\![n]\!]_{dl} =_{df} \begin{cases} Const_{tt} & \text{if } LocDelay(n) \\ Id_{\mathcal{B}} & \text{if } \neg(LocDelay \vee LocBlock(n)) \\ Const_{ff} & \text{if } \neg LocDelay \wedge LocBlock(n) \end{cases}$$

5 Conclusions

We have shown how to construct optimal bitvector analysis algorithms for parallel programs with shared memory that are as efficient as their purely sequential counterparts, and which can easily be implemented. At the first sight, the existence of such an algorithm is rather surprising, as the interleaving semantics underlying our programming language is an indication for an exponential effort. However, the restriction to bitvector analysis constrains the possible ways of interference in such a way that we could construct a fixed point algorithm that directly works on the parallel program without taking any interleavings into account. The algorithm is implemented on the *Fixpoint Analysis Machine* of [SCKKM]. Moreover, a variant of the computationally optimal placement algorithm for computations sketched in Section 4 is implemented in the ESPRIT project COMPARE [Vo1, Vo2].

References

[CC1] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th International Symposium on Principles of Programming Languages (POPL'77)*, Los Angeles, California, 1977, 238 - 252.

[CC2] Cousot, P., and Cousot, R. Invariance proof methods and analysis techniques for parallel programs. In *Biermann, A. W., Guiho, G., and Kodratoff, Y. (eds.) Automatic Program Construction Techniques*, chapter 12, 243 - 271, Macmillan Publishing Company, 1984.

[CH1] Chow, J.-H., and Harrison, W. L. Compile time analysis of parallel programs that share memory. In *Conference Record of the 19th International Symposium on Principles of Programming Languages (POPL'92)*, Albuquerque, New Mexico, 1992, 130 - 141.

[CH2] Chow, J.-H., and Harrison, W. L. State Space Reduction in Abstract Interpretation of Parallel Programs. In *Proceedings of the International Conference on Computer Languages, (ICCL'94)*, Toulouse, France, May 16-19, 1994, 277-288.

[Dh] Dhamdhere, D. M. A new algorithm for composite hoisting and strength reduction optimisation (+ Corrigendum). *Internat. J. Computer Math. 27*, (1989), 1 - 14 (+ 31 - 32).

[DRZ] Dhamdhere, D. M., Rosen, B. K., and Zadeck, F. K. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92)*, San Francisco, California, *SIGPLAN Notices 27*, 7 (1992), 212 - 223.

[GS] Grunwald, D., and Srinivasan, H. Data flow equations for explicitely parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Parallel Programming (PPOPP'93)*, *SIGPLAN Notices 28*, 7 (1993).

[He] Hecht, M. S. Flow analysis of computer programs. Elsevier, North-Holland, 1977.

[JD] Joshi, S. M., and Dhamdhere, D. M. A composite hoisting-strength reduction transformation for global program optimization. Part I & II. *Internat. J. Computer Math. 11*, (1982), 21 - 41, 111 - 126.

[KRS1] Knoop, J., Rüthing, O., and Steffen, B. Optimal code motion: Theory and practice. *Transactions on Programming Languages and Systems 16*, 4 (1994), 1117 - 1155.

[KRS2] Knoop, J., Rüthing, O., and Steffen, B. Lazy strength reduction. *Journal of Programming Languages 1*, 1 (1993), 71 - 91.

[KRS3] Knoop, J., Rüthing, O., and Steffen, B. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94)*, Orlando, Florida, *SIGPLAN Notices 29*, 6 (1994), 147 - 158.

[KRS4] Knoop, J., Rüthing, O., and Steffen, B. The power of assignment motion. To appear in *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implemantion (PLDI'95)*, La Jolla, California, June 18 - 21, 1995.

[KS] Knoop, J., and Steffen, B. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction (CC'92)*, Paderborn, Germany, Springer-Verlag, LNCS 641 (1992), 125 - 140.

[KSV1] Knoop, J., Steffen, B., and Vollmer, J. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. Fakultät für Mathematik und Informatik, Universität Passau, Germany, MIP-Bericht Nr. 9409 (1994), 29 pages.

[KSV2] Knoop, J., Steffen, B., and Vollmer, J. Optimal code motion for parallel programs. To appear in *Proceedings of the 12th Workshop on "Alternative Konzepte für Sprachen und Rechner"*, Physikzentrum Bad Honnef, Germany, May 2 - 4, 1995.

[KU] Kam, J. B., and Ullman, J. D. Monotone data flow analysis frameworks. *Acta Informatica 7*, (1977), 309 - 317.

[Ma] Marriot, K. Frameworks for abstract interpretation. *Acta Informatica 30*, (1993), 103 - 129.

[McD] McDowell, C. E. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing 6*, 3 (1989), 513 - 536.

[MJ] Muchnick, S. S., and Jones, N. D. (Eds.). Program flow analysis: Theory and applications. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[MR] Morel, E., and Renvoise, C. Global optimization by suppression of partial redundancies. *Communications of the ACM 22*, 2 (1979), 96 - 103.

[MP] Midkiff, S. P., and Padua, D. A. Issues in the optimization of parallel programs. In *Proceedings of the International Conference on Parallel Processing, Volume II*, St. Charles, Illinois, (1990), 105 - 113.

[SCKKM] Steffen, B., Claßen, A., Klein, M., Knoop, J., and Margaria, T. The fixpoint analysis machine. To appear in *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95)*, Philadelphia, Pennsylvania, USA, August 21-24, 1995.

[SHW] Srinivasan, H., Hook, J., and Wolfe, M. Static single assignment form for explicitly parallel programs. In *Conference Record of the 20th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'93)*, Charleston, South Carolina, 1993, 260 - 272.

[SP] Sharir, M., and Pnueli, A. Two approaches to interprocedural data flow analysis. In [MJ], (1981), 189 - 233.

[SW] Srinivasan, H., and Wolfe, M. Analyzing programs with explicit parallelism. In *Proceedings of the 4th International Conference on Languages and Compilers for Parallel Computing*, Santa Clara, California, Springer-Verlag, LNCS 589 (1991), 405 - 419.

[Vo1] Vollmer, J. Data flow equations for parallel programs that share memory. Tech. Rep. 2.11.1 of the ESPRIT Project COMPARE (1994), Fakultät für Informatik, Universität Karlsruhe, Germany.

[Vo2] Vollmer, J. Data flow analysis of parallel programs. To appear in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'95)*, 1995. Extended version available as: Interner Bericht 19/95 (1995), 28 pages, Fakultät für Informatik, Universität Karlsruhe, Germany.

[WS] Wolfe, M, and Srinivasan, H. Data structures for optimizing programs with explicit parallelism. In *Proceedings of the 1st International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, Springer-Verlag, LNCS 591 (1991), 139 - 156.

A Computing the *MFP*-Solution

Algorithm A.1 (Computing the *MFP*-Solution)

Input: *A flow graph* $G = (N, E, \mathbf{s}, \mathbf{e})$*, a local semantic functional* $[\![\ ]\!] : N \to \mathcal{F}_{\mathcal{B}}$*, and a function* $f_{init} \in \mathcal{F}_{\mathcal{B}}$ *reflecting the assumptions on the context in which the procedure under consideration is called. Usually,* f_{init} *is given by* $Id_{\mathcal{B}}$*.*

Output: *An annotation of* G *with functions* $[\![\, n \,]\!] \in \mathcal{F}_{\mathcal{B}}$*,* $n \in N$*, representing the greatest solution of the equation system of Definition 2.3. In fact, after termination of the algorithm the functional* $[\![\ ]\!]$ *satisfies:* $\forall\, n \in N.\ [\![\, n \,]\!] = MFP_{(G,[\![\ ]\!])}(n) = MOP_{(G,[\![\ ]\!])}(n)$

BEGIN $MFP(G, [\![\ ]\!], f_{init})$ **END**.

where

```
PROCEDURE MFP ($G = (N, E, \mathbf{s}, \mathbf{e}) : SequentialFlowGraph$;
                    $[\![\ ]\!] : N \to \mathcal{F}_{\mathcal{B}}$ : $LocalSemanticFunctional$;  $f_{start} : \mathcal{F}_{\mathcal{B}}$);
VAR $f : \mathcal{F}_{\mathcal{B}}$;
BEGIN
    (Initialization of the annotation array gtr and the variable workset)
    FORALL $n \in N \backslash \{\mathbf{s}\}$ DO $[\![\, n \,]\!] := Const_{tt}$ OD;
    $[\![\, \mathbf{s} \,]\!] := f_{start}$; $workset := \{\, n \mid n = \mathbf{s} \vee [\![\, n \,]\!] = Const_{ff} \,\}$;
    (Iterative fixed point computation)
    WHILE $workset \neq \emptyset$ DO
        LET $n \in workset$
            BEGIN
                $workset := workset \backslash \{\, n \,\}$; $f := [\![\, n \,]\!] \circ [\![\, n \,]\!]$;
                FORALL $m \in succ_G(n)$ DO
                    IF $[\![\, m \,]\!] \sqsupset f$ THEN $[\![\, m \,]\!] := f$; $workset := workset \cup \{\, m \,\}$ FI OD END
    OD
END.
```

B Computing the *PMFP*$_{BV}$-Solution

Algorithm B.1 (Computing the *PMFP*$_{BV}$-Solution)

Input: *A parallel flow graph* $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$*, a local semantic functional* $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$*, a function* $f_{init} \in \mathcal{F}_{\mathcal{B}}$ *and a Boolean value* $b_{init} \in \mathcal{B}$*, where* f_{init} *and* b_{init} *reflect the assumptions on the context in which the procedure under consideration is called. Usually,* f_{init} *and* b_{init} *are given by* $Id_{\mathcal{B}}$ *and* $f\!f$*, respectively.*

Output: *An annotation of* G^* *with functions* $[\![\, G \,]\!]^* \in \mathcal{F}_{\mathcal{B}}$*,* $G \in \mathcal{G}_{\mathcal{P}}(G^*)$*, representing the semantic functions computed in step 2 of the three step procedure of Section 3.3, and with functions* $[\![\, n \,]\!] \in \mathcal{F}_{\mathcal{B}}$*,* $n \in N^*$*, representing the greatest solution of the equation system of Definition 3.7. In fact, after the termination of the algorithm the functional* $[\![\ ]\!]$ *satisfies:* $\forall\, n \in N^*.\ [\![\, n \,]\!] = PMFP_{BV(G^*,[\![\ ]\!])}(n) = PMOP_{(G^*,[\![\ ]\!])}(n)$

Remark: *The global variables* $[\![\, G \,]\!]^*$*,* $G \in \bigcup \{\mathcal{G}_{\mathcal{C}}(G') \mid G' \in \mathcal{G}_{\mathcal{P}}(G^*)\,\}$*, each of which is storing a function of* $\mathcal{F}_{\mathcal{B}}$*, are used during the hierarchical computation of the* $PMFP_{BV}$*-solution for storing the global effect of graphs that are a component of some graph* $G \in \mathcal{G}_{\mathcal{P}}(G^*)$*. Additionally, the global variables* $harmful(G)$*,* $G \in \bigcup \{\, \mathcal{G}_{\mathcal{C}}(G') \mid G' \in \mathcal{G}_{\mathcal{P}}(G^*)\,\}$*, store whether* G *contains a node* n *with* $[\![\, n \,]\!] = Const_{ff}$*. These variables are used to compute the value of the predicate* NonDestructed *of Section 3.3. Finally, every flow graph* $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ *is assumed to have a* rank*, which is recursively defined by:*

$$rank(G) =_{df} \begin{cases} 0 & \text{if } G \in \mathcal{G}_{\mathcal{P}}^{min}(G^*) \\ max\{\, rank(G') \mid G' \in \mathcal{G}_{\mathcal{P}}(G^*) \wedge G' \subset G \,\} + 1 & \text{otherwise} \end{cases}$$

where $\mathcal{G}_{\mathcal{P}}^{min}(G^*) =_{df} \{\, G \in \mathcal{G}_{\mathcal{P}}(G^*) \mid \forall\, G' \in \mathcal{G}_{\mathcal{P}}(G^*).\ G' \subseteq G \Rightarrow G' = G \,\}$ *denotes the set of* minimal *graphs of* $\mathcal{G}_{\mathcal{P}}(G^*)$*.*

```
BEGIN
   $GLOBEFF(G^*, \overline{\llbracket\;\rrbracket})$;  ( Synchronization: Computing $\llbracket G \rrbracket^*$ for all $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ )
   $PMFP_{BV}(G^*, \overline{\llbracket\;\rrbracket}, f_{init}, b_{init})$  ( Interleaving: Computing the $PMFP_{BV}$-Solution $\llbracket n \rrbracket$ for all $n \in N^*$ )
END.

where

PROCEDURE $GLOBEFF$ ($G = (N, E, \mathbf{s}, \mathbf{e})$ : $ParallelFlowGraph$;
                                  $\overline{\llbracket\;\rrbracket} : N \rightarrow \mathcal{F}_{\mathcal{B}}$ : $LocalSemanticFunctional$);
VAR $i$ : $integer$;
BEGIN
    FOR $i := 0$ TO $rank(G)$ DO
       FORALL $G' \in \{ G'' \,|\, G'' \in \mathcal{G}_{\mathcal{P}}(G) \wedge rank(G'') = i \}$ DO
          FORALL $G'' \in \{ G'''_{seq} \,|\, G''' \in \mathcal{G}_{\mathcal{C}}(G') \}$ where $G'' = (N'', E'', \mathbf{s}'', \mathbf{e}'')$ DO
             LET $\forall\, n \in N''.\ \llbracket n \rrbracket'' = \begin{cases} Id_{\mathcal{B}} \sqcap Const_{\forall\, \bar{G} \in \mathcal{G}_{\mathcal{C}}(pfg(n)).\ \neg harmful(\bar{G})} & \text{if } n \in N^*_N \\ \llbracket pfg(n) \rrbracket^* & \text{if } n \in N^*_X \\ \overline{\llbracket n \rrbracket} & \text{otherwise} \end{cases}$
                BEGIN
                   $harmful(G'') := (\ |\{ n \in N'' \,|\, \llbracket n \rrbracket'' = Const_{ff} \}| \geq 1\ )$;
                   $MFP(G'', \llbracket\;\rrbracket'', Id_{\mathcal{B}})$;  $\llbracket G'' \rrbracket^* := \llbracket end(G'') \rrbracket^*$
                END OD;
          $\llbracket G' \rrbracket^* := \bigsqcap \{ \llbracket G'' \rrbracket^* \,|\, G'' \in \mathcal{G}_{\mathcal{C}}(G') \}$ OD OD
END.

PROCEDURE $PMFP_{BV}$ ($G = (N, E, \mathbf{s}, \mathbf{e})$ : $ParallelFlowGraph$;
                                  $\overline{\llbracket\;\rrbracket} : N \rightarrow \mathcal{F}_{\mathcal{B}}$ : $LocalSemanticFunctional$; $f_{start} : \mathcal{F}_{\mathcal{B}}$; $harmful : \mathcal{B}$);
VAR $f : \mathcal{F}_{\mathcal{B}}$;
BEGIN
    IF $harmful$ THEN FORALL $n \in N$ DO $\llbracket n \rrbracket := Const_{ff}$ OD
    ELSE
       ( Initialization of the annotation arrays $\llbracket\;\rrbracket$ and the variable workset )
       FORALL $n \in N \backslash \{\mathbf{s}\}$ DO $\llbracket n \rrbracket := Const_{tt}$ OD;
       $\llbracket \mathbf{s} \rrbracket := f_{start}$; $workset := \{ n \,|\, n = \mathbf{s} \vee \overline{\llbracket n \rrbracket} = Const_{ff} \}$;
       ( Iterative fixed point computation )
       WHILE $workset \neq \emptyset$ DO
          LET $n \in workset$
             BEGIN
                $workset := workset \backslash \{ n \}$;
                IF $n \in N \backslash N^*_N$
                   THEN
                      $f := \overline{\llbracket n \rrbracket} \circ \llbracket n \rrbracket$;
                      FORALL $m \in succ_G(n)$ DO
                         IF $\llbracket m \rrbracket \sqsupset f$ THEN $\llbracket m \rrbracket := f$; $workset := workset \cup \{ m \}$ FI OD
                   ELSE
                      FORALL $G' \in \mathcal{G}_{\mathcal{C}}(pfg(n))$ DO
                         $PMFP_{BV}(G', \overline{\llbracket\;\rrbracket}, \llbracket n \rrbracket, \sum_{G'' \in \mathcal{G}_{\mathcal{C}}(pfg(n)) \backslash \{G'\}} harmful(G''))$ OD;
                      $f := \llbracket pfg(n) \rrbracket^* \circ \llbracket n \rrbracket$;
                      IF $\llbracket end(pfg(n)) \rrbracket \sqsupset f$
                         THEN $\llbracket end(pfg(n)) \rrbracket := f$; $workset := workset \cup \{ end(pfg(n)) \}$ FI FI
             END OD FI
END.
```

Let $\llbracket n \rrbracket_{alg}$, $n \in N^*$, denote the final values of the corresponding variables after the termination of Algorithm B.1, and $\llbracket n \rrbracket$, $n \in N^*$, the greatest solution of the equation system of Definition 3.7, then we have: $\forall\, n \in N^*.\ \llbracket n \rrbracket_{alg} = \llbracket n \rrbracket$

Author Index