Integrated Formal Methods

Second International Conference, IFM 2000
Dagstuhl Castle, Germany, November 2000
Proceedings
Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo
Wolfgang Grieskamp  Thomas Santen
Bill Stoddart  (Eds.)

Integrated
Formal Methods

Second International Conference, IFM 2000
Dagstuhl Castle, Germany, November 1-3, 2000
Proceedings
Preface

IFM 2000, the second in a series of international conferences on Integrated Formal Methods, was held at the 18th-century château of Schloss Dagstuhl, Saarland, Germany, from the 1st to the 3rd of November 2000.

The conference programme consisted of invited talks from Sir Tony Hoare FRS and Wolfram Schulte, along with 22 papers selected from 58 submissions.

Applying formal methods may involve the modelling of different aspects of a system that are expressed through different paradigms. This motivates us to research the combination of different viewpoints of a system, either by the creation of hybrid notations, by extending existing notations, by translating between notations, or by incorporating a wider perspective with the innovative use of an existing notation.

The integration of formal methods promises great benefits to systems modelling and software development. Regardless of the approach taken, however, significant issues can arise in areas such as semantic integration, the tractability of our notations, the integration of tool support, the integration of proof systems, consistency, and completeness. Issues arise equally in our conceptualisation of systems at different levels of abstraction and the development of these conceptualisations through the process of refinement.

The stated theme of IFM’99 was the integration of state-based and behavioural formalisms. For IFM 2000 this was widened, and the submitted papers have been grouped in five technical sessions, covering the linking and extending of notations, methodology, the foundation of one formalism in another, semantics, and aspects of verification and validation.

We hope that these proceedings will be of benefit both to the conference participants and to the wider community of workers in the field. The production of these proceedings would not have been possible without the invaluable help of the programme committee and external referees, and of all the contributors who submitted papers to the conference.

November 2000

Wolfgang Grieskamp
Thomas Santen
Bill Stoddart
Organization

Wolfgang Grieskamp, Technical University of Berlin, Germany
Thomas Santen, Technical University of Berlin, Germany
Bill Stoddart, University of Teesside, UK

Program Committee

Keijiro Araki (Kyushu, Japan)          Mike Hinchey (Omaha, USA)
Didier Bert (Grenoble, France)        Bernd Krieg-Brückner
Egon Börger (Pisa, Italy)             (Bremen, Germany)
Jonathan Bowen (London, UK)           Michel Lemoine (Toulouse, France)
Micheal Butler (Southampton, UK)      Shaoying Liu (Hiroshima, Japan)
Jim Davies (Oxford, UK)               John McDermid (York, UK)
John Derrick (Kent, UK)               Dominique Méry (Nancy, France)
Jin Song Dong (Singapore)             Thomas Santen (Berlin, Germany)
Heiko Dörr (Berlin, Germany)          Steve Schneider (London, UK)
John Fitzgerald (Newcastle, UK)       Wolfram Schulte (Redmond, USA)
Andy Galloway (York, UK)              Jane Sinclair (Warwick, UK)
Chris George (Macao)                  Graeme Smith (Brisbane, Australia)
Wolfgang Grieskamp                    Kenji Taguchi (Uppsala, Sweden)
(Hamburg, Germany)                    W J (Hans) Toetenel (Delft, Holland)
Henri Habrias (Nantes, France)        Heike Wehrheim
Susumu Hayashi (Kobe, Japan)          (Oldenburg, Germany)
Maritta Heisel (Magdeburg, Germany)   

Additional Reviewers

Richard Banach                       Dang van Hung                        Mark Saaltink
Christie Bolton                     Tomasz Janowski                     Holger Schlingloff
Dominique Cansell                   Markus Lepper                       Carron Shankland
David Carrington                    Liu Jing                           Kim Soon-Kyeong
Marc Cavazza                        Brendan Mahony                     Carsten Sühl
Michael Cebulla                     Stephan Merz                        Ulrich Ultes-Nitsche
Steve Dunne                         Michael Meyer                     Jacob Wieland
Carla Ferreira                      zu Hoerste                       Qiwen Xu
Dennis Furey                        Tim Mossakowski                   Hirokazu Yatsu
He Jieng                            Steve Paynter                      Shoji Yuen
Monika Heiner                       Jean-Claude Reynaud              
Steffen Helke                       Dean Rosenzweig


# Table of Contents

**Invited Talk**

Assertions .......................................................... 1

Tony Hoare

**Linking and Extending Notations**

State-Based Extension of CASL ........................................... 3

Hubert Baumeister and Alexandre Zamulin

Linking DC Together with TRSL ........................................... 25

Anne Elisabeth Haxthausen and Xia Yong

Formalizing Timing Diagrams as Causal Dependencies for Verification Purposes .......................................................... 45

Jörg Fischer and Stefan Conrad

A Process Compensation Language ....................................... 61

Michael Butler and Carla Ferreira

Activity Graphs and Processes ........................................... 77

Christie Bolton and Jim Davies

Structuring Real-Time Object-Z Specifications ......................... 97

Graeme Smith and Ian Hayes

ISpec: Towards Practical and Sound Interface Specifications ......... 116

Hans B.M. Jonkers

**Methodology**

Cooperation of Formal Methods in an Engineering Based Software Development Process ........................................... 136

Yamine Ait-Amour

Developing Control Systems Components ............................. 156

Luigia Petre and Kaisa Sere

Specification and Analysis of Automata-Based Designs ............... 176

Jeremy Bryans, Lynne Blair, Howard Bowman, and John Derrick

Structural Refinement in Object-Z / CSP ............................. 194

John Derrick and Graeme Smith
Towards a Unified Development Methodology for Shared-Variable Parallel and Distributed Programs ........................................... 214
Jürgen Dingel

Foundation of One Formalism by Another

Construction of Finite Labelled Transition Systems from B Abstract Systems .......................................................... 235
Didier Bert and Francis Cave

μ-Charts and Z: Hows, Whys, and Wherefores ........................ 255
Greg Reeve and Steve Reeves

Combining Operational Semantics, Logic Programming and Literate Programming in the Specification and Animation of the Verilog Hardware Description Language .................................................. 277
Jonathan P. Bowen

Invited Talk

Why Doesn’t Anyone Use Formal Methods? ............................ 297
Wolfram Schulte

Semantics

How to Write a Healthiness Condition ................................ 299
Yifeng Chen

A Concurrent and Compositional Petri Net Semantics of Preemption ..... 318
Hanna Klaudel and Franck Pommereau

Verification and Validation

An Approach to Symbolic Test Generation ............................. 338
Vlad Rusu, Lydie du Bousquet, and Thierry Jéron

Behavioral Conformance Verification in an Integrated Approach Using UML and B ....................................................... 358
Eric Meyer and Thomas Santen

Predicate Diagrams for the Verification of Reactive Systems ........ 380
Dominique Cansell, Dominique Méry, and Stephan Merz

Modular Verification for a Class of PLTL Properties .............. 398
Pierre-Alain Masson, Hassan Mountassir, and Jacques Julliand

Towards Model Checking Stochastic Process Algebra ............... 420
Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle

Author Index .............................................................. 441
Assertions

Tony Hoare
Senior Researcher
Microsoft Research Ltd.
1 Guildhall St., Cambridge, CB2 3NH
thoare@microsoft.com

An assertion is a Boolean formula written in the text of a program, which the programmer asserts will always be true when that part of the program is executed. It specifies an internal interface between that part of the program that comes before it and all that follows it. In the software industry today, assertions are conditionally compiled in test runs of a program, and help in the detection and diagnosis of errors. Alan Turing first proposed assertions as a means of checking a large routine. They were rediscovered independently by Naur as generalised snapshots, and by Floyd, who used them to assign meanings to programs. Floyd suggested that if the internal assertions were strong enough, they would constitute a formal proof of the correctness of a complete program. In this lecture, I will summarise the subsequent development of the idea, and describe some of its practical impact.

In the early seventies, I developed an axiomatic approach for proofs of programs that use all the main constructions of a high-level programming language - iterations, local variables, procedures and parameters, recursion, and even jumps. Following Dijkstra, I always took a top-down view of the task of software construction, with assertions formulated as part of program specification, and with proofs conducted as part of program design. I hoped that this research would help to reduce the high costs of programming error, and the high risks of using computers in critical applications. But the real attraction for me was that the axioms underlying program proofs would provide an objective and scientific test of the quality of programming language design: a language described by a small collection of obvious rules, easily applied, would be better than one that required many rules with complex side-conditions. In collaboration with Wirth, we tried out the idea on the Pascal language; and later it inspired the design of Euclid by a team in Xerox PARC.

In scaling proof methods from small sequential algorithms to large software systems, it was necessary to extend the power of the assertion language. The Z specification language was developed by Abrial on the basis of Zermelo’s set theory, which Frankel showed to be essentially adequate for expression of all concepts known to mathematics. It should therefore be adequate to express all the abstractions useful to computing, and prove the correctness of their representations. Dijkstra dealt with non-determinism, by imagining the choice to be exercised maliciously by a demon. Jones and his fellow designers of VDM included initial as well as final values of program variables. All these ideas were successfully tested by IBM in specifying the internal interfaces of a large system, CICS.
The next challenge was to extend the technology to concurrent programs. Milner suggested that their meaning could be specified by the collection of tests which they passed. Following Popper’s criterion of falsifiability, Roscoe and Brookes concentrated on failures of a test, and constructed a non-deterministic model of concurrency, following the paradigm of Communicating Sequential Processes. This was applied industrially by the British start-up microchip Company Inmos in the design of the occam programming language, and the architecture of the transputer which implemented it. Finally, Hehner showed how Roscoe’s results could be coded directly in the language of assertions, so that any kind of program, concurrent as well as sequential, could be interpreted as the strongest assertion that describes all its possible behaviours.

This insight has inspired all my subsequent research. With the aid of He Jifeng, it has been applied to wider varieties of programming paradigm and language, including hardware and software, parallel and sequential, declarative and procedural. Ignoring radical differences in syntax, and abstracting from implementation technology, very similar definitions and mathematical laws apply in many different paradigms; perhaps Computing Science has achieved a level of maturity to undertake the challenge that drives the progress of many other sciences, namely unification of theories of programming.
State-Based Extension of CASL

Hubert Baumeister\(^1\) and Alexandre Zamulin\(^2\)

1 Institute of Computer Science, University of Munich  
baumeist@informatik.uni-muenchen.de
2 Institute of Informatics Systems  
Siberian Division of Russian Academy of Sciences  
zam@iis.nsk.su

Abstract. A state-based extension of the algebraic specification language CASL is presented. It permits the specification of the static part of a complex dynamic system by means of CASL and the dynamic part by means of the facilities described in the paper. The dynamic system is defined as possessing a number of states and a number of operations (procedures) for transforming one state into another. Each state possesses one and the same static part specified by CASL and a varying part specified by additional tools. The varying part includes dynamic sorts/functions/predicates and dependent functions/predicates. The dependent functions/predicates are specified by formulae using the names of the dynamic functions/predicates so that each time one of the last ones is updated the corresponding former ones are also updated. The updates of the dynamic entities are produced by procedures which are specified by means of preconditions, postconditions, and dynamic equations.

1 Introduction

The Common Framework Initiative (CoFI) [18] is an open collaborative effort to design a common framework for algebraic specifications. The rational behind CoFI is that the lack of such a framework greatly hinders the dissemination and application of research results in algebraic specification. The aim is to base the common framework as much as possible on a critical selection of features that have already been explored in various contexts. The common framework will provide a family of languages centered around a single, reasonably expressive common specification language called CASL [17]. Some of these languages will be extensions of CASL, e.g. oriented to particular programming paradigms, while others will be sub-languages of CASL, e.g. executable.

In this paper we define SB-CASL, a state-based extension of CASL [17] which is based on algebraic specifications and the concept of implicit state à la Z, B, or VDM, also known as the state-as-algebra approach. In contrast to Z, VDM, and B, this approach does not constrain a specifier by a fixed number of basic types and type constructors used for the representation of application data, and gives a formal semantics for all notions used in the method. SB-CASL brings

\(^*\) This research has been partially supported by ESPRIT working group 29432 (CoFI WG) and the Russian Foundation for Basic Research under Grant 98-01-00682

© Springer-Verlag Berlin Heidelberg 2000
together ideas from Typed Gurevich Machines of Zamulin [19], based on the original work of Gurevich [13,14], Algebraic Specifications with Implicit State of Dauchy and Gaudel, first presented in [8] and further developed in [15,9], D- oids by Astesiano and Zucca [2,20,21], and the work of Baumeister [3,4,5]. The formalism serves for the specification of dynamic systems possessing a state and a number of operations for accessing and updating the state.

The novelty of SB-CASL is that it combines the operational style for the specification of state transformations with the declarative style in a practical specification language. In the operational style one defines how one state is transformed into another; in contrast, in the declarative style only the properties that the successor state has to possess are specified and not how the state is constructed. Up to now either only the operational style was used, like in ASM’s and the Implicit State approach, or only the declarative style was used as in the approach by Baumeister. A notable exception is the approach by Zucca [21] which also allows both styles of specifications; however her intention was not to provide a specification language.

The paper is organized as follows. The CASL institution is briefly described in Sec. 2. States and state updates are defined in Sec. 3. Dynamic systems are introduced in Sec. 4. Transition terms, serving for the construction of dynamic formulae, are described in Sec. 5 and dynamic formulae in Sec. 6. The structure of a dynamic system specification and supporting examples are given in Sec. 7. Some related work is discussed in Sec. 8, and in Sec. 9 some conclusions are drawn.

2 The CASL Institution

A basic specification in CASL consists of a many-sorted signature $\Sigma$ together with a set of sentences. The (loose) semantics of a basic specification is the class of those models in $\text{Mod}(\Sigma)$ which satisfy all the specified sentences. For reasons of simplicity we restrict ourselves to the many-sorted part of CASL and leave out the order-sorted part. However, all the subsequent constructions in this paper can also be performed in the presence of a subsorting relationship.

A many-sorted signature $\Sigma = (S, TF, PF, P)$ consists of:

- a set $S$ of sorts;
- sets $TF_{w,s}$, $PF_{w,s}$, of total function symbols, respectively partial function symbols, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$, for each function profile $(w, s)$ consisting of a sequence of argument sorts $w \in S^*$ and a result sort $s \in S$ (constants are treated as functions with no arguments);
- sets $P_w$ of predicate symbols, for each predicate profile consisting of a sequence of argument sorts $w \in S^*$.

Here and in the sequel a function (predicate) symbol is a name accompanied with a profile. Names may be overloaded, occurring in more than one of the above sets.

In this paper we write a total function symbol as $f : s_1, \ldots, s_n \rightarrow s$ and a partial function symbol as $f : s_1, \ldots, s_n \rightarrow ? s$. When the list of argument values is empty, we write $s$ and $?s$, respectively.
For a many-sorted signature $\Sigma = (S, TF, PF, P)$ a many-sorted model $A \in \text{Mod}(\Sigma)$ is a many-sorted first-order structure consisting of a many-sorted partial algebra:

- a non-empty carrier set $|A|_s$ for each sort $s \in S$ (let $|A|_w$ denote the cartesian product $|A|_{s_1} \times \cdots \times |A|_{s_n}$ when $w = s_1 \ldots s_n$),
- a partial function $f^A$ from $|A|_w$ to $|A|_s$ for each function symbol $f \in TF_{w,s}$ or $f \in PF_{w,s}$, the function being required to be total in the former case,
- together with a predicate $p^A \subseteq |A|_w$ for each predicate symbol $p \in P_w$.

Many-sorted terms on a signature $\Sigma = (S, TF, PF, P)$ and a set of sorted, non-overloaded variables $X$ are built from:

- universally quantified variables from $X$, introduced by

\[
\text{var} \; v_{11}, \ldots, v_{1k} : s_1, \ldots, v_{nm} : s_n;
\]

- applications of function symbols in $TF \cup PF$ to argument terms of appropriate sorts.

For a many-sorted signature $\Sigma = (S, TF, PF, P)$, the set of $\Sigma$-sentences consists of sort-generation constraints and the usual closed many-sorted first-order logic formulae, built from atomic formulae using quantification (over sorted variables) and logical connectives. The atomic formulae are:

- applications of predicate symbols $p \in P$ to argument terms of appropriate sorts;
- assertions about the definedness of terms, written $\text{def } t$;
- existential and strong equations between terms of the same sort, written $t_1 \equiv t_2$ and $t_1 = t_2$, respectively.

The satisfaction of a sentence in a structure $A$ is determined as usual by the holding of its atomic formulae w.r.t. assignments of (defined) values to all the variables that occur in them. The value of a term w.r.t. a variable assignment may be undefined due to the application of a partial function during the evaluation of the term, or because some arguments of a function application are undefined. The satisfaction of sentences, however, is 2-valued.

The application of a predicate symbol $p$ to a sequence of argument terms holds in $A$ iff the values of all the terms are defined and give a tuple belonging to $p^A$. A definedness assertion concerning a term holds iff the value of the term is defined. An existential equation holds iff the values of both terms are defined and identical, whereas a strong equation holds also when the values of both terms are undefined. A sort-generation constraint $(S', F')$ is satisfied in a $\Sigma$-model $A$ if the carriers of the sorts in $S'$ are generated by the function symbols in $F'$.

\section{States and State Updates}

The signature of a system defined by SB-CASL includes a part

$$\Sigma_{\text{stat}} = (S_{\text{stat}}, TF_{\text{stat}}, PF_{\text{stat}}, P_{\text{stat}})$$
which defines some data types (sorts and operations) using the standard CASL facilities. These data types are used for the specification of system’s states and the description of possible state updates. A \( \Sigma_{\text{stat}} \)-structure is called a static structure in the sequel.

The system’s states are defined by dynamic sorts, dynamic functions, and dynamic predicates. The names and profiles of these sorts/functions/predicates, \( \Delta_{\text{dyn}} = (S_{\text{dyn}}, TF_{\text{dyn}}, P F_{\text{dyn}}, P_{\text{dyn}}) \), form a signature extension of the static signature \( \Sigma_{\text{stat}} \).

We require that each dynamic function \( f : w \rightarrow s \) where \( s \) is a dynamic sort from \( S_{\text{dyn}} \) or \( w \) contains a dynamic sort from \( S_{\text{dyn}} \) is in \( P F_{\text{dyn}} \). The reason is that a function having a dynamic sort in its profile may become partial if elements are added or removed from this sort.

In the rest of this paper we denote by \( \Sigma_{\text{dyn}} = (S', TF', PF', P') \) the union of \( \Sigma_{\text{stat}} \) with \( \Delta_{\text{dyn}} \).

**Example 1.** The first example is a specification of an identifier table. The identifier table stores some data for each identifier definition. It can be block-structured according to block nesting. Typical functions are creating an empty identifier table, inserting identifier data in the current block, checking whether an identifier is defined in the current block, checking whether an identifier is defined in the program, fetching identifier data, and deleting all identifier definitions of the current block.

The following specification defines the state of the identifier table. The static signature is given by the union of the signatures of NAT, NAME, and DEFDATA; and \( \text{id\_table} \) and \( \text{cur\_level} \) are dynamic functions:

**System ID,TABLE**
use NAT, NAME, DEFDATA ** The specifications used
dynamic
  func \( \text{id\_table} \) : Name, Pos \( \rightarrow \) ? Defdata;
  func \( \text{cur\_level} \) : Pos; ** the current level of block nesting

**Example 2.** The second example is taken from one of the latest work of Zucca [21]. The procedures (dynamic operations in her paper), whose “intended interpretation” is described at the model level in that paper, will be formally specified here.

**System CIRCLES**
use REAL, COLOUR ** The spec. COLOUR has only two constants
  ** "green" and "red" of sort "Colour"
dynamic
  sort Circle;
  func X, Y: Circle \( \rightarrow \) Real;
  func radius: Circle \( \rightarrow \) Real;
  func col: Circle \( \rightarrow \) Colour;

A dynamic sort/function/predicate can be different in different states.

**Definition 1.** A \( \Sigma_{\text{dyn}} \)-state is a \( \Sigma_{\text{dyn}} \)-structure where \( \Sigma_{\text{dyn}} = \Sigma_{\text{stat}} \cup \Delta_{\text{dyn}} \).
The restriction of any $\Sigma_{\text{dyn}}$-state $A$ to $\Sigma_{\text{stat}}$, $A|_{\Sigma_{\text{stat}}}$, is a static structure called the base of $A$. Several $\Sigma_{\text{dyn}}$-states can have the same base. Following [15], we denote the set of all $\Sigma_{\text{dyn}}$-states with the same base $B$ by $\text{state}_B(\Sigma_{\text{dyn}})$ and mean by a $\Sigma_{\text{dyn}}^B$-state a $\Sigma_{\text{dyn}}$-state with the static structure $B$. Thus, the carrier of any static sort $s \in S_{\text{stat}}$ in a $\Sigma_{\text{dyn}}$-state $A$ is the same as the carrier of $s$ in $B$, that is $|A|_s = |B|_s$.

### 3.1 Update-Sets

One state can be transformed into another by a state update which is either a function update or a predicate update or a sort update.

**Definition 2.** Let $B$ be a static structure over $\Sigma_{\text{stat}}$. A function update in a $\Sigma_{\text{dyn}}^B$-state $A$ is a triple $(f, \bar{a}, a)$ where $f_{\bar{w}s}$ is a dynamic function (constant) symbol in $\Delta_{\text{dyn}}$, $w = s_1 \ldots s_n$, $\bar{a} = <a_1, \ldots, a_n> \in |A|_w$ ($\bar{a}$ is the empty tuple $<$ when $n$ is equal to zero), and $a$ is either an element of $|A|_s$ or the symbol $\perp$. A function update $(f, \bar{a}, \perp)$ is valid only if $f$ is the symbol of a partial function.

A function update $\alpha = (f, \bar{a}, a)$ serves for the transformation of a $\Sigma_{\text{dyn}}^B$-state $A_0$ in the following way:

- $g^A' = g^A$ for any $g_{\bar{w}s} \in TF' \cup PF'$ different from $f$;
- $f^{A\alpha}(\bar{a}) = a$ if $a$ is not $\perp$, $f^{A\alpha}(\bar{a})$ becomes undefined otherwise;
- $f^{A\alpha}(\bar{a}') = f^A(\bar{a}')$ for any tuple $\bar{a}' = <a_1', \ldots, a_n'>$ different from $\bar{a}$;
- $p^{A\alpha} = p^A$ for any $p_{\bar{w}s} \in P'$;
- $|A\alpha|_s = |A|_s$ for any $s \in S'$.

Following Gurevich [13], we say that $A_0$ is obtained by firing the update $\alpha$ on $A$. Roughly speaking, firing a function update either inserts an element in the definition domain of a dynamic function or modifies the value of such a function at one point in its domain or removes an element from the definition domain.

**Definition 3.** Let $B$ be a static structure over $\Sigma_{\text{stat}}$. A predicate update in a $\Sigma_{\text{dyn}}^B$-state $A$ is either a triple $(+, p, \bar{a})$ or a triple $(-, p, \bar{a})$ where $p_{\bar{w}s}$ is a predicate symbol in $\Delta_{\text{dyn}}$, $w = s_1 \ldots s_n$, and $\bar{a} = <a_1, \ldots, a_n> \in |A|_w$.

A predicate update $\beta = (\xi, p, \bar{a})$ serves for the transformation of a $\Sigma_{\text{dyn}}^B$-state $A_0$ into a new $\Sigma_{\text{dyn}}^B$-state $A_\beta$ in the following way:

- $p^{A\beta}(\bar{a})$ holds if $\xi$ is “$+$” and $p^{A\beta}(\bar{a})$ does not hold if $\xi$ is “$-$”;
- $p^{A\beta}(\bar{a}')$ iff $p^A(\bar{a}')$ for any tuple $\bar{a}' = <a_1', \ldots, a_n'>$ different from $\bar{a}$;
- $q^{A\beta} = q^A$ for any $q_{\bar{w}s} \in P_{\text{dyn}}$ different from $p$;
- $f^{A\beta} = f^A$ for any $f_{\bar{w}s} \in TF' \cup PF'$; and
- $|A\beta|_s = |A|_s$ for any $s \in S'$.

**Definition 4.** Let $s$ be the name of a dynamic sort. A sort-update $\delta$ in $A$ is either a triple $(+, s, \text{id})$ where $\text{id}$ is an element such that $\text{id} \notin |A|_s$ or a triple $(-, s, \text{id})$ where $\text{id}$ is an element such that $\text{id} \in |A|_s$. 


A sort update $\delta = (+, s, id)$ transforms a $\Sigma^B_{\text{dyn}}$-state $A$ into a new $\Sigma^B_{\text{dyn}}$-state $A\delta$ in the following way:

- $|A\delta|_s = |A|_s \cup \{id\}$,
- $|A\delta|_{s'} = |A|_{s'}$ for any $s' \in S'$ different from $s$,
- $f^{A\delta} = f^A$ for any $f_{ws} \in TF \cup PF$, and
- $p^{A\delta} = p^A$ for any $p_w \in P$.

A sort update $\delta = (-, s, id)$ transforms a $\Sigma^B_{\text{dyn}}$-state $A$ into a new $\Sigma^B_{\text{dyn}}$-state $A\delta$ in the following way:

- for any function $f^A : |A|_{s_1} \times \ldots \times |A|_{s_n} \rightarrow |A|_{s_{n+1}}$ if $|A|_{s_i}, i = 1, \ldots, n+1$, is a dynamic sort associated with the sort name $s$ and $id \in |A|_{s_i}$, then if there is a maplet $<a_1, \ldots, a_n \mapsto a_{n+1}> \in f^A$ such that $a_i = id$, then $f^{A\delta} = f^A \setminus \{<a_1, \ldots, a_{n+1}>\}$; $f^{A\delta} = f^A$ otherwise;
- for any predicate $p^A : |A|_{s_1} \times \ldots \times |A|_{s_n}$, if $|A|_{s_i}, i = 1, \ldots, n$, is a dynamic sort associated with the sort name $s$ and $id \in |A|_{s_i}$, then if there is a tuple $<a_1, \ldots, a_n> \in p^A$ such that $a_i = id$, then $p^{A\delta} = p^A \setminus \{<a_1, \ldots, a_n>, \} ; p^{A\delta} = p^A$ otherwise;
- $|A\delta|_s = A_s \setminus \{id\}$ and $|A\delta|_{s'} = |A|_{s'}$ for any $s' \neq s$.

Thus, the sort update $\delta = (-, s, a)$ contracts the set of elements of a certain sort and deletes the corresponding entries from the graphs of all dynamic functions and predicates using and/or producing the element indicated.

Note that it is possible to create algebras with empty carrier sets using sort updates. For example, if $A$ is an algebra with $|A|_s = \{a\}$ and $\delta = (-, s, a)$, then $|A\delta|_s = \{\}$. This is in contradiction with the requirement that within the CASL institution carrier sets are non-empty. To solve this problem we introduce the institution SB-CASL which is the same as the CASL institution, but allows their models to have empty carrier sets. Note that the introduction of empty carriers does not pose any problems with the satisfaction relation in case of partial logic (cf. [7]), and that any many-sorted model of the CASL institution is also a model of the SB-CASL institution.

**Definition 5.** Let $\Gamma$ be a set of function/predicate/sort updates. The set $\Gamma$ is inconsistent if it contains either

- two contradictory function updates of the following kind: $\alpha_1 = (f, a, a')$ and $\alpha_2 = (f, a, a')$, where $a \neq a'$ (two contradictory function updates define the function differently at the same point), or
- two contradictory predicate updates of the following kind: $\beta_1 = (+, p, a)$ and $\beta_2 = (+, p, a)$ (two contradictory predicate updates define the predicate differently at the same tuple of arguments), or
- two sort updates of the following kind: $\delta_1 = (+, s, id)$ and $\delta_2 = (-, s, id)$ (two contradictory sort updates insert in the sort and delete from the sort the same element), or
- either an $\alpha = (f, <a_1, \ldots, a_n>, a_{n+1})$ for an $f^A : |A|_{s_1} \times \ldots \times |A|_{s_n} \rightarrow |A|_{s_{n+1}}$ or a $\beta = (\xi, p, <a_1, \ldots, a_n>)$ for a $p^A : |A|_{s_1} \times \ldots \times |A|_{s_n}$, and $\delta = (-, s, id)$ such that $s$ is $s_i$ for some $i = 1, \ldots, n+1$ and $id = a_i$ (a sort element is removed while a function/predicate is forced to use it).

the update set is consistent otherwise.
A consistent update-set $\Gamma$ applied to a $\Sigma^B_{\text{dyn}}$-state $A$ transforms $A$ into a new $\Sigma^B_{\text{dyn}}$-state $A'$ by simultaneous firing all $\alpha \in \Gamma$, all $\beta \in \Gamma$, and all $\delta \in \Gamma$. If $\Gamma$ is inconsistent, the new state is not defined. If $\Gamma$ is empty, $A'$ is the same as $A$. Following [16], we denote the application of $\Gamma$ to a state $A$ by $\mathit{AF}$. The set of all consistent sets of updates in $\text{state}_B(\Sigma)$ is denoted by $\text{update}_B(\Sigma_{\text{dyn}})$ in the sequel.

**Definition 6.** Let $\Gamma_1$ and $\Gamma_2$ be two consistent update-sets in a $\Sigma^B_{\text{dyn}}$-state $A$, $\alpha_1 = (f, \langle a_1, \ldots, a_n \rangle, a)$, $\alpha_2 = (f, \langle a_1, \ldots, a_n \rangle, a')$, $\beta_1 = (\xi_1, p, \langle a_1, \ldots, a_n \rangle)$, $\beta_2 = (\xi_2, p, \langle a_1, \ldots, a_n \rangle)$, $\delta_1 = (+, s, \mathit{id})$, and $\delta_2 = (\sim, s, \mathit{id})$, where $a \neq a'$ and both $\xi_1$ and $\xi_2$ are either "+" or "−" such that $\xi_1$ is different from $\xi_2$. The sequential union of $\Gamma_1$ and $\Gamma_2$, denoted by $\Gamma_1 \cap \Gamma_2$, is defined as follows: $u \in \Gamma_1 \cap \Gamma_2$ if $u \in \Gamma_1$ or $u \in \Gamma_2$, except the following cases:

- if $\alpha_1 \in \Gamma_1$ and $\alpha_2 \in \Gamma_2$, then $\alpha_2 \in \Gamma_1 \cap \Gamma_2$ and $\alpha_1 \notin \Gamma_1 \cap \Gamma_2$;
- if $\beta_1 \in \Gamma_1$ and $\beta_2 \in \Gamma_2$, then $\beta_2 \in \Gamma_1 \cap \Gamma_2$ and $\beta_1 \notin \Gamma_1 \cap \Gamma_2$;
- if $\delta_1 = (\sim, s, a) \in \Gamma_2$, then for any $\alpha = (g, \langle a_1, \ldots, a_n, a_{n+1} \rangle) \in \Gamma_1$, where $g^A : |A|_{s_1} \times \ldots \times |A|_{s_n} \to |A|_{s_{n+1}}$, and for any $\beta = (\xi, p, \langle a_1, \ldots, a_n \rangle) \in \Gamma_1$, where $p^A : |A|_{s_1} \times \ldots \times |A|_{s_n}$: if $s$ is $s_i$, $i = 1, \ldots, n$, and $a_i = \mathit{id}$, then $\alpha \notin \Gamma_1 \cap \Gamma_2$ and $\beta \notin \Gamma_1 \cap \Gamma_2$.

Thus, in a sequential union of update sets, each next update of a function/predicate at a certain point waives each preceding update of this function/predicate at the same point. If there are sequential creations of elements of the same sort, the set of elements of this sort will be expanded accordingly. If there is a deletion of an element, the corresponding sort will be contracted accordingly and all function/predicate updates involving this element will be ignored. Note that if an element is first created and then deleted, then the resulting update set will contain no trace of this element.

### 3.2 Dependent Functions and Predicates

A function update of the form $\alpha = (f, \tilde{a}, a)$ applied to a $\Sigma^B_{\text{dyn}}$-state $A$ does not change any other function $g$, only $f$ at point $\tilde{a}$ is changed. Consider now the following partial function $\text{find}: \text{Name} \to ?$. Defdata fetching data for a name in ID_TABLE. Clearly the result of $\text{find}$ depends on the state of ID_TABLE; thus $\text{find}$ has to be a dynamic function. Further, $\text{find}$ depends on the dynamic function $\text{id\_table}$ because, starting from the current block level, $\text{find}$ looks for the definition of an identifier at each block level. Thus we expect that whenever $\text{id\_table}$ is changed so is $\text{find}$. However, the semantics of update-sets and dynamic functions does not provide us with such an automatism.

To still allow such functions as $\text{find}$, we introduce *dependent functions* and *dependent predicates* as a second set of state components. They form a signature extension $\Delta_{\text{dep}} = (\emptyset, TF_{\text{dep}}, PF_{\text{dep}}, P_{\text{dep}})$ of $\Sigma_{\text{dyn}}$ where $TF_{\text{dep}} \cap PF_{\text{dep}} = \emptyset$. Note that the set of sort names of $\Delta_{\text{dep}}$ is empty; we don’t allow for dependent sorts.

As with dynamic functions, we require that each dependent function $f : w \to s$ where $s$ is a dynamic sort from $S_{\text{dyn}}$ or $w$ contains a dynamic sort from $S_{\text{dyn}}$ is in $PF_{\text{dep}}$. For example, the following dependent functions/predicates can be defined in the system ID_TABLE:
depend
pred defined_current: Name; ** checks whether an id is defined in the current block
pred is_defined: Name; ** checks whether an id is defined in the table
func find: Name \rightarrow ? Defdata; ** fetches data from the table

Definition 7. A $\Sigma_{\text{dep}}^B$-state is a $\Sigma_{\text{dep}}$-structure with the static structure $B$ where $\Sigma_{\text{dep}} = \Sigma_{\text{dyn}} \cup \Delta_{\text{dep}}$.

Thus, dependent functions/predicates extend a $\Sigma_{\text{dyn}}^B$-state to a $\Sigma_{\text{dep}}^B$-state. The set of all $\Sigma_{\text{dep}}^B$-states with the same static structure $B$ is denoted by $\text{state}_B(\Sigma_{\text{dep}})$.

4 Dynamic Systems

A state update modifies the dynamic and dependent functions/predicates. Possible state updates are specified by procedures declared in the fourth part of the system’s signature, $\Delta_{\text{proc}}$, which consists of sets $TP_w, PP_w$ of total procedure symbols, respectively partial procedure symbols, such that $TP_w \cap PP_w = \emptyset$, for each procedure profile $w$ consisting of a sequence of argument sorts from $\Sigma_{\text{dep}}$.

Example 3. For example, the following procedures can be defined in the system ID_TABLE:

```
proc initialize; ** construction of an empty identifier table
insert_entry; ? Name, Defdata; ** insertion of a new entry in the identifier table
new_level; ** creation of a new level of nesting in the identifier table
delete_level ?; ** deletion of the last block in the identifier table
```

Example 4. The following procedures can be declared in the system CIRCLES:

```
proc
  start; ** create one circle with default attributes
  move: Circle, Real, Real; ** change the coordinates of a circle
  moveAll: Real, Real; ** change the coordinates of all circles
  resize: Circle, Real; ** change the radius of a circle
  changeCol: Circle; ** change the colour of a circle
  copy: circle; ** create a new circle with the attributes of the argument circle
  delCreen; ** delete all green circles
```

Definition 8. The signature $D\Sigma = (\Sigma_{\text{stat}}, \Delta_{\text{dyn}}, \Delta_{\text{dep}}, \Delta_{\text{proc}})$ of a dynamic system consists of

- a static signature $\Sigma_{\text{stat}}$;
- a signature extension $\Delta_{\text{dyn}}$ of $\Sigma_{\text{stat}}$ by symbols of dynamic sorts, functions, and predicates such that the profiles of total functions do not contain a dynamic sort;
- a signature extension $\Delta_{\text{dep}}$ of $\Sigma_{\text{stat}} \cup \Delta_{\text{dyn}}$ by symbols of dependent functions and predicates, but without dependent sorts; and
two families of sets $\Delta_{\text{proc}} = (TP, PP)$ of total and partial procedure symbols.

Definition 9. A dynamic system, $DS(B)$, of signature $D\Sigma$ consists of

- a set of states $|DS(B)| \subseteq \text{state} B(\Sigma_{\text{dep}})$, called the carrier of the system;
- a partial surjective function $\text{map}^{DS(B)} : \text{state} B(\Sigma_{\text{dyn}}) \to |DS(B)|$ such that if $\text{map}^{DS(B)}(A)$ is defined, then $\text{map}^{DS(B)}(A)|_{\Sigma_{\text{dep}}} = A$ for each $A \in \text{state} B(\Sigma_{\text{dyn}})$;
- for each procedure symbol $p : s_1, \ldots, s_n$, a (partial) map $p^{DS(B)}$ associating an update set $\Gamma \in \text{update} B(\Sigma_{\text{dyn}})$ with a state $A$ of $DS(B)$ and a tuple $<a_1, \ldots, a_n>$ where $a_i \in |A|_i, i = 1, \ldots, n$.

Given a dynamic system $DS(B)$, we call a $\Sigma_{\text{dep}}$-structure $A$ a state of $DS(B)$ if $A \in |DS(B)|$.

We write $p^{DS(B)}(A, \bar{a})$ for the application of a procedure $p^{DS(B)}$ to a pair consisting of a state $A$ and a tuple $\bar{a}$ where $A \in |DS(B)|$ and $\bar{a} = <a_1, \ldots, a_n>$.

For a procedure $p$, we say that $p$ is a constant procedure if the result of $p^{DS(B)}(A, \bar{a})$ does not depend on $A$. This kind of procedure can be used for the initialization of a dynamic system.

5 Transition Terms

State updates are specified by means of special transition terms. The interpretation of a transition term $TT$ in a dynamic system $DS(B)$ at a state $A$ w.r.t. a variable assignment $\sigma : X \to |A|$ produces an update set $\Gamma$ or is undefined. The corresponding state $A'$ after firing the update set can be obtained by

$$A' = \text{map}^{DS(B)}((A|_{\Sigma_{\text{dep}}})\Gamma)$$

for which we will simply write, in abuse of notation, $A' = A\Gamma$.

5.1 Basic Transition Terms

Basic transition terms are update instructions, procedure call, sort contraction instruction, and the skip instruction.

Update instructions. Let $f$ be the name of a dynamic function with the profile $s_1, \ldots, s_n \to s$, $g$ the name of a partial dynamic function with the profile $s_1, \ldots, s_n \to s$, $p$ the name of a dynamic predicate with the profile $s_1, \ldots, s_n$, $X$ a set of sorted variables, $t_i$ a term of sort $s_i$ over signature $\Sigma_{\text{dep}}$ with variables $X$ for $i = 1, \ldots, n$. Then

$$f(t_1, \ldots, t_n) := t,$$
$$g(t_1, \ldots, t_n) := \text{undef},$$
$$p(t_1, \ldots, t_n) := \text{true},$$
$$p(t_1, \ldots, t_n) := \text{false}$$

are transition terms called update instructions.
Interpretation. If \( A \) is a state of \( DS(B) \), \( \sigma \) a variable assignment, \( t \) and \( t_i, i = 1, \ldots, n \), are defined terms in \( A \) under \( \sigma \), then

\[
[f(t_1, \ldots, t_n) := t_1^A, \ldots, t_n^A, t^A, \sigma)] \subset \{(f, \langle t_1^A, \ldots, t_n^A, t^A, \sigma \rangle)\},
\]

\[
g(t_1, \ldots, t_n) := \text{undef}[A, \sigma) \subset \{(g, \langle t_1^A, \ldots, t_n^A, \perp \rangle)\},
\]

\[
p(t_1, \ldots, t_n) := \text{true}[A, \sigma) \subset \{(\text{true}, \langle t_1^A, \ldots, t_n^A, t^A, \sigma \rangle)\},
\]

\[
p(t_1, \ldots, t_n) := \text{false}[A, \sigma) \subset \{(\text{false}, \langle t_1^A, \ldots, t_n^A, t^A, \sigma \rangle)\},
\]

If at least one of \( t, t_i, i = 1, \ldots, n \), is not defined in \( A \) under \( \sigma \), then the interpretation of the above transition terms is undefined.

Example 5. Let \( x \) be a variable of sort \( Nat \) and \( f \) be a dynamic function from \( Nat \) to \( Nat \). The execution of the transition term \( f(x) := f(x) + 1 \) under the variable assignment \( \sigma = \{x \mapsto a\} \) transforms a state \( A \) into a state \( A' \) so that

\[
f_{A'}(a) = f_A(a) + 1 \quad \text{and} \quad f_{A'}(n) = f_A(n) \quad \text{for all} \quad n \neq a.
\]

If \( c \) is a partial dynamic constant, a transition term \( c := \text{undef} \) will make \( c \) undefined in the new state.

Procedure call. If \( p : s_1, \ldots, s_n \) is a procedure symbol and \( t_1, \ldots, t_n \) are terms of sorts \( s_1, \ldots, s_n \) over \( \Sigma_{\text{dep}} \) with variables from \( X \), then \( p(t_1, \ldots, t_n) \) is a transition term called a procedure call.

Interpretation. Let \( DS(B) \) be a dynamic system of signature \( D\Sigma \), \( A \) a state in \( |DS(B)| \), and \( \sigma \) a variable assignment. The interpretation of a procedure call is defined as follows:

\[
[p(t_1, \ldots, t_n)]^A, \sigma = p_{DS(B)}(A, \langle t_1^A, \ldots, t_n^A, \sigma \rangle)
\]

if each \( t_i^A, i = 1, \ldots, n \), is defined and \( p_{DS(B)} \) is defined for the state \( A \) and the tuple \( \langle t_1^A, \ldots, t_n^A, \sigma \rangle \); \([p(t_1, \ldots, t_n)]^A, \sigma \) is undefined otherwise.

Sort contraction. If \( t \) is a term of a dynamic sort \( s \), then \( \text{drop } t \) is a transition term called a sort contraction.

Interpretation: \([\text{drop } t]^A, \sigma = 0\) where \( \delta = (\perp, s, t^A, \sigma) \).

Skip. The transition term \( \text{skip} \) causes no state update, i.e. \([\text{skip}]^A, \sigma = 0\).

5.2 Transition Term Constructors

Complex transition terms are constructed recursively from basic transition terms by means of several term constructors, e.g., sequence constructor, set constructor, condition constructor, guarded update, loop constructor, import constructor, and massive update.
**Sequence constructor.** If \( TT_1, TT_2, \ldots, TT_n \) are transition terms, then

\[
\text{seq } TT_1, TT_2, \ldots, TT_n \text{ end}
\]

is a transition term called a sequence of transition terms.

**Interpretation.** Let \( A \) be a state, \( \Gamma_1 = \llbracket TT_1 \rrbracket^A, \sigma, A_1 = A \Gamma_1, \Gamma_2 = \llbracket TT_2 \rrbracket^{A_1, \sigma}, A_2 = A_1 \Gamma_2, \ldots, \Gamma_n = \llbracket TT_n \rrbracket^{A_{n-1}, \sigma} \). Then

\[
\llbracket \text{seq } TT_1, TT_2, \ldots, TT_n \text{ end} \rrbracket^{A, \sigma} = \Gamma,
\]

where \( \Gamma = \Gamma_1 \cup \ldots \cup \Gamma_n \) and each \( \llbracket TT_i \rrbracket^{A_{i-1}, \sigma} \) is defined.

Thus, to execute a sequence of transition terms starting with a state \( A \), it is sufficient to create the sequential union of their update sets and use it for the transformation of \( A \) (which is equivalent to the sequential execution of the terms one after another).

**Set constructor.** If \( TT_1, \ldots, TT_n \) are transition terms, then

\[
\text{set } TT_1, \ldots, TT_n \text{ end}
\]

is a transition term called a set of transition terms.

**Interpretation.** Let \( A \) be a state and \( \Gamma_1 = \llbracket TT_1 \rrbracket^A, \sigma, \ldots, \Gamma_n = \llbracket TT_n \rrbracket^A, \sigma \). Then

\[
\llbracket \text{set } TT_1, \ldots, TT_n \text{ end} \rrbracket^{A, \sigma} = \Gamma_1 \cup \ldots \cup \Gamma_n
\]

if each \( \llbracket TT_i \rrbracket^A, \sigma \) is defined and \( \Gamma_1 \cup \ldots \cup \Gamma_n \) is consistent; the semantics is not defined otherwise.

In other words, to execute a set of transition terms, execute all of them in parallel and unite the results if they are consistent.

**Condition constructor.** If \( k \) is a natural number, \( g_0, \ldots, g_k \) are formulae, and \( TT_0, \ldots, TT_k \) are transition terms, then the following expression is a transition term called a conditional transition term:

\[
\text{if } g_0 \text{ then } TT_0 \\
\text{elseif } g_1 \text{ then } TT_1 \\
\ldots \\
\text{elseif } g_k \text{ then } TT_k \\
\text{endif}
\]

If \( g_k \) is the formula \( \text{true} \), then the last elseif clause can be replaced with "else \( TT_k \)". We also write if \( g \) then \( TT \) for if \( g \) then \( TT \) else skip endif.

**Interpretation.** Let \( A \) be a state, \( \sigma \) a variable assignment, and \( TT \) a conditional transition term, then

\[
\llbracket TT \rrbracket^{A, \sigma} = \llbracket TT \rrbracket^A, \sigma
\]

if \( g_i \) holds in \( A \) w.r.t. \( \sigma \), but every \( g_j \) with \( j < i \) fails in \( A \) w.r.t. \( \sigma \). \( \llbracket TT \rrbracket^{A, \sigma} = \emptyset \) if every \( g_i \) fails in \( A \) w.r.t. \( \sigma \).
Loop constructors. The condition constructor together with the sequence constructor gives us a possibility to define some loop constructors. If $TT$ is a transition term and $g$ is a formula, then while $g$ do $TT$ and do $TT$ until $g$ are transition terms.

Interpretation. $\llbracket\text{while } g \text{ do } TT\rrbracket^{A,\sigma} = \llbracket\text{if } g \text{ then } \text{seq } TT, \text{ while } g \text{ do } TT \text{ end}\rrbracket^{A,\sigma}$; $\llbracket\text{do } TT \text{ until } g\rrbracket^{A,\sigma} = \llbracket\text{seq } TT, \text{ if } \neg g \text{ then } \text{do } TT \text{ until } g\rrbracket^{A,\sigma}$.

Import constructor. If $x$ is a variable, $s$ is a dynamic sort name and $TT$ is a transition term, then import $x : s$ in $TT$ is a transition term called an import term.

Interpretation. Let $A' = A\delta, \delta = (+, s, a)$ for some $a \notin A_s$, and $\sigma' = \sigma \oplus \{x \mapsto a\}$ where $\oplus$ is the overriding union, i.e. $(\sigma \oplus \sigma')(x) = \sigma''(x)$ if $x$ is in the domain of $\sigma''$ and $(\sigma \oplus \sigma')(x) = \sigma(x)$ otherwise, then $\llbracket\text{import } x : s \text{ in } TT\rrbracket^{A',\sigma'} = \{\delta\}; \llbracket TT\rrbracket^{A',\sigma'}$.

Massive update. Let $x$ be a variable of sort $s$ and $TT$ a transition term. A massive update

$$\text{forall } x : s . TT$$

permits the specification of a parallel update of one or more sorts/functions/predicates at several points.

Interpretation. Let $A$ be a state such that $|A|_s$ is not empty, let $\sigma$ and $\sigma' = \{x\} \to |A|_s$ be variable assignment, and let $\sigma'' = \sigma \oplus \sigma'$. Then

- if $\llbracket TT\rrbracket^{A,\sigma''}$ is defined and $\Gamma = \bigcup\{\llbracket TT\rrbracket^{A,\sigma''}\}$ is consistent, then $\llbracket\text{forall } x : s . TT\rrbracket^{A,\sigma} = \Gamma$;
- $\llbracket\text{forall } x : s . TT\rrbracket^{A,\sigma}$ is not defined otherwise.

If $|A| = \emptyset$, then $\llbracket\text{forall } x : s . TT\rrbracket^{A,\sigma} = \emptyset$. That is, the massive update over the empty sort produces nothing.

Example 6. Let $f$ be a dynamic function from Nat to Nat. A transition term

$$\text{forall } x : \text{Nat} . f(x) := f(x) + 1$$

interpreted in a state $A$ yields the update-set

$$\{(f, n, f^A(n) + 1) \mid n \in |A|_{\text{Nat}}\}$$

if $f^A(n)$ is defined for all $n$.

Example 7. The execution of the update set produced by the transition term

$$\text{forall } x : s . \text{drop } x$$

at the current state $A$ will remove all elements from $|A|_s$ and will make empty all functions and predicates using $s$ in their profiles.
6 Dynamic Formulae

For the specification of dynamic systems we introduce dynamic formulae which can be dynamic equations, precondition formulae, or postcondition formulae. A dynamic equation serves for the specification of a behaviour of a procedure in terms of a transition rule. A precondition formula allows us to define the domain of a procedure. Finally, a postcondition formula is used to specify the behaviour of a procedure similar to VDM or Z.

6.1 Dynamic Equations

A dynamic equation is of the form $TT_1 = TT_2$ where $TT_1$ and $TT_2$ are transition terms over variables from $X$. A dynamic system $DS(B)$ satisfies a dynamic equation if for all states $A$ of $DS(B)$ and variable assignments $\sigma : X \rightarrow |A|:

$$A[TT_1]^{A,\sigma} = A[TT_2]^{A,\sigma} \text{ and } A[TT_{1/2}]^{A,\sigma} \in |DS(B)|.$$

The most common use of dynamic equations is in the form:

$$p(x_1, \ldots, x_n) = TT$$

where $TT$ does not contain a direct or indirect call of $p$. This defines the semantics of $p$ in a dynamic system to be a function mapping a state $A$ and a tuple $<a_1, \ldots, a_n>$ to the update set given by the interpretation of $TT$ w.r.t. $A$ and the variable assignment mapping each $x_i$ to $a_i$.

Example 8. For a simple example consider a dynamic constant $\text{counter} : \text{Int}$ and procedures $\text{Inc}$ and $\text{Dec}$. The procedure $\text{Inc}$ can be defined by the following dynamic equation:

$$\text{Inc} = \text{counter} := \text{counter} + 1.$$ 

Similarly, $\text{Dec}$ can be defined using the dynamic equation

$$\text{Dec} = \text{counter} := \text{counter} - 1.$$ 

However, dynamic equations need not follow the pattern $p(x_1, \ldots, x_n) = TT$. For example, an alternative way to define $\text{Dec}$ from the previous example is by the following dynamic equation:

$$\text{seq Dec, Inc end} = \text{skip}.$$ 

This means that whenever a $\text{Dec}$ procedure is followed by an $\text{Inc}$ procedure, the state of the system should not change.

6.2 Precondition Formulae

Let $p$ be an element of $PP_{s_1, \ldots, s_n}$, i.e. a partial procedure symbol, and $t_1, \ldots, t_n$ are terms over $\Sigma_{dep}$ with variables from $X$. A precondition formula of the form
pre $p(t_1, \ldots, t_n) : \varphi$ can be used to state under which conditions a partial
procedure $p$ is guaranteed to be defined.

A dynamic system $DS(B)$ satisfies a precondition formula iff the value of
$p^{DS(B)}(A, \llbracket t_1 \rrbracket^A \sigma, \ldots, \llbracket t_n \rrbracket^A \sigma >)$ is defined in exactly those states $A$ and for
those variable assignments $\sigma$ for which $\varphi$ holds.

The following precondition formula

$$pre \ insert\_entry(id, d) : \neg defined\_current(id)$$

states that the partial procedure $insert\_entry$ declared in Ex. 3 for the dynamic
system $ID\_TABLE$ must be defined only in those states $A$ and only for those
arguments $id$ for which the interpretation of $defined\_current$ is false.

### 6.3 Postcondition Formulae

Dynamic equations of the form $p(x_1, \ldots, x_n) = TT$ can be used to specify
dynamic systems in an operational style similar to Gurevich’s ASMs.

However, sometimes it is convenient to use a declarative style similar to the
one used by the specification languages Z and VDM. In the declarative style, the
values of a dynamic/dependent component before and after the execution of a
procedure are related by a first-order formula. Usually this formula only defines
the relationship between the values and does not provide an algorithm how to
change the value. For example, the $Dec$ operation of Ex. 8 could be defined by
a postcondition formula

$$post \ Dec : counter = \text{counter}' + 1$$

where $counter$ refers to the value of $counter$ in the state before executing the
$Dec$ operation, and $\text{counter}'$ refers to the value of $counter$ after executing $Dec$.
Note that this formula does not prescribe how the value of $counter$ is computed
after performing the $Dec$ operation. In contrast, a dynamic equation of the form

$$Dec = counter := counter - 1$$

defines an update-set used for changing the value of $counter$.

To be more precise, let $A$ be a state of a dynamic system $DS(B)$ containing
the dynamic constant $counter$ and a procedure $Dec$. Then the interpretation of
$Dec$ in $DS(B)$ yields the update-set $Dec^{DS(B)}(1) = \Gamma$ which applied to a state
$A$ yields the state $A\Gamma$. Then $A\Gamma$ has the following property: the interpretation
of $counter$ in $A$ has to be the same as the interpretation of $counter$ in $A\Gamma$ plus 1:

$$counter^A(1) = counter^{A\Gamma}(1) + 1.\text{ The general syntax of a postcondition formula is:}$$

$$post \ p(t_1, \ldots, t_n) : \varphi$$

where $p$ is a (partial) procedure, $t_1, \ldots, t_n$ are terms over $\Sigma_{dep}$, and $\varphi$ is a
formula over the signature $\bar{\Sigma}$ which is constructed as follows. First consider
the case where there are no dynamic sorts, then the sorts of $\bar{\Sigma}$ are the static
sorts of \( \Sigma_{\text{stat}} \). The operation symbols (total or partial) of \( \Sigma \) are those defined in the static signature plus two copies of the dynamic and dependent function symbols: one copy denoting the function before the execution of a procedure and one copy, decorated by a prime (\( \prime \)), denoting the function after the execution of a procedure. Similar for the predicate symbols of \( \Sigma \).

To define whether \( \varphi \) is satisfied by a state \( A \) and an update-set \( \Gamma \), we construct from \( A \) and \( A\Gamma \) a \( \Sigma \)-model \( A^\mathbb{F} \) as follows. The interpretation of a static signature component, i.e., a sort, operation, or predicate symbol from \( \Sigma_{\text{stat}} \), is the same as the interpretation of that component in \( A \). Note that this is the same as the interpretation of that symbol in \( A\Gamma \) and \( B \), because \( A|_{\Sigma_{\text{stat}}} = B = A\Gamma|_{\Sigma_{\text{stat}}} \).

The interpretation of a dynamic or dependent signature component in \( A^\mathbb{F} \) is either the interpretation of that component in \( A \), or, if this is a component with a prime in \( \Sigma \), the interpretation of the corresponding unprimed component in \( A\Gamma \). Then a state \( A \) and an update-set \( \Gamma \) satisfy a formula \( \varphi \) iff \( A^\mathbb{F} \models \varphi \).

In the case of dynamic sorts the construction is similar. That is, \( \Sigma \) contains the static sorts plus two copies of the dynamic sorts, one copy decorated with a prime. However, the profile of a primed dynamic or dependent function/predicate in \( \Sigma \) having a dynamic sort in its profile changes. Each dynamic sort in the profile has to be replaced by its primed version. Note that the profiles of unprimed function/predicate symbols remain the same.

As an example consider the dynamic system CIRCLES, which has a dynamic sort \( \text{Circle} \) and a dynamic function \( \text{X} : \text{Circle} \to \text{Real} \) (among others). Now the signature \( \Sigma \) contains two sorts \( \text{Circle} \) and \( \text{Circle}' \) and two function symbols \( \text{X} : \text{Circle} \to \text{Real} \) and \( \text{X}' : \text{Circle}' \to \text{Real} \). Note that in the profile of \( \text{X}' \) the dynamic sort \( \text{Circle} \) is also decorated with a prime.

There is still another problem with dynamic sorts. Consider the following specification of the \text{move} operation in the dynamic system CIRCLES:

\[
\text{post } \text{move}(c, x, y) : X'(c) = X(c) + x \land Y'(c) = Y(c) + y
\]

Note that the formula above is not well-formed w.r.t. \( \Sigma \) because \( X' \) is a function from \( \text{Circle}' \) to \( \text{Real} \) while \( c \) is of sort \( \text{Circle} \), and similar for \( Y' \) and \( c \). The solution is to introduce in \( \Sigma \) a partial function \( \text{tm}_{\text{Circle}} : s \to s' \) called a tracking map for each dynamic sort \( s \). The notion of tracking map was first introduced with d-oids (cf. [1,2,20,21]). In our example we use the function \( \text{tm}_{\text{Circle}} : \text{Circle} \to \text{Circle}' \) and write

\[
\text{post } \text{move}(c, x, y) : X'(\text{tm}_{\text{Circle}}(c)) = X(c) + x \land Y'(\text{tm}_{\text{Circle}}(c)) = Y(c) + y
\]

In the sequel we leave the application of the tracking map implicit whenever possible. If \( t[r] \) is a term with a subterm \( r \) and \( r \) is required to be of dynamic sort \( s' \), then we allow \( r \) to be of dynamic sort \( s \) and understand this as an abbreviation for \( t[\text{tm}_s(r)] \). This allows us to write the above postcondition formula as:

\[
\text{post } \text{move}(c, x, y) : X'(c) = X(c) + x \land Y'(c) = Y(c) + y
\]

For the interpretation of the tracking maps in \( A^\mathbb{F} \) we have to define the tracking map associated with a dynamic sort and an update-set. This tracking map is undefined for elements that are removed and is the identity otherwise.

\[
\text{tm}^A\Gamma : |A|_s \to |A\Gamma|_s
\]
\[ tnm^A_\Gamma(a) = \begin{cases} \perp & \text{if } \delta = (-, s, a) \in \Gamma \\ a & \text{else} \end{cases} \]

The definition of satisfaction remains the same, i.e. \( A \) and \( \Gamma \) satisfy \( \varphi \) if \( A^\Gamma \models \varphi \). The formal definition of \( \Sigma \) and \( A^\Gamma \) is given in App. A.

### 7 Specification of Dynamic Systems

**Definition 10.** A dynamic system specification

\[ DSS = (SPEC, \Delta_{\text{dyn}}, (\Delta_{\text{dep}}, Ax), (\Delta_{\text{proc}}, Ax_{\text{proc}})) \]

has four levels:

- The first level is a CASL specification \( SPEC \) with semantics \( (\Sigma_{\text{stat}}, M) \).

- The second level defines those aspects of the system’s state which are likely to change. It includes a signature extension, \( \Delta_{\text{dyn}} \), which declares some dynamic sorts/functions/predicates.

- The third level defines some dependent functions/predicates and indicates state invariants. It does not introduce new sorts and uses the names of dynamic functions/predicates from \( \Delta_{\text{dyn}} \), the names of dependent functions/predicates from \( \Delta_{\text{dep}} \), and the operations of \( \Sigma_{\text{stat}} \). The formulae in \( Ax \) restrict the set of possible states of a dynamic system satisfying \( DSS \).

- The fourth level, \( (\Delta_{\text{proc}}, Ax_{\text{proc}}) \), defines some procedures. \( Ax_{\text{proc}} \) is a set of dynamic formulae.

A dynamic system specification \( DSS \) defines a dynamic signature

\[ D\Sigma = (\Sigma_{\text{stat}}, \Delta_{\text{dyn}}, \Delta_{\text{dep}}, \Sigma_{\text{proc}}) \]

and a dynamic system \( DS(B) \) over signature \( D\Sigma \) satisfies a dynamic specification \( DSS \) iff

- \( B \) is a model of \( SPEC \).
- \( |DS(B)| \) is the set \( \{ A \mid A \in state_B(\Sigma_{\text{dep}}) \land A \models Ax \} \).
- \( DS(B) \) satisfies each dynamic formula in \( Ax_{\text{proc}} \).

**Example 9.**

**Specification of the "ID\_TABLE" system**

**System ID\_TABLE**

use NAT, NAME, DEFDATA ** The specifications used
dynamic id_table: Name, Pos \rightarrow? Defdata;
cur_level: Pos; - the current level of block nesting
depend pred defined\_current, is\_defined: Name;
pred local\_defined: Name, Pos;
func local\_find: Name, Pos \rightarrow? Defdata;
func find: Name →? Defdata;
var id: Name, k: Pos
• defined current(id) ⇔ def id_table(id, cur_level));
• local defined(id, 0) ⇔ false;
• local defined(id, k) ⇔ def id_table(id, k)∨ local defined(id, k-1);
• is defined(id) ⇔ local defined(id, cur_level);
• local find(id, k) = id_table(id, k) when def id_table(id, k)
  else local find (id, k-1));
• find(id) = local find(id, cur_level) if is defined(id);

proc initialize; ** construction of an empty identifier table
insert entry: ? Name, Defdata;
new level;
delete level?;
var id: Name, k: Pos, d: Defdata
• pre delete level: cur level > 1;
• pre insert entry(id, d): ¬ defined current(id);
• initialize = set cur level := 1,
    for all id: Name, x: Pos. id_table(id, x) := undef end;
• post insert entry(id, d): id_table'(id, cur level) = d;
• post new level: cur level' = cur level + 1;
• delete level = set cur level := cur level − 1,
  for all id: Name. id_table(id, cur level) := undef end;

Example 10.

System CIRCLES
use REAL, COLOUR ** The spec. COLOUR has only two constants
  ** "green" and "red" of sort "Colour"

dynamic
  sort Circle;
  func X, Y: Circle →? Real;
  func radius: Circle →? Real;
  func col: Circle →? Colour

proc
  start; ** creation of one circle with default attributes
  move: Circle, Real, Real; ** change the coordinates of a circle
  moveAll: Real, Real; ** change the coordinates of all circles
  resize: Circle, Real; ** change the radius of a circle
  changeCol: Circle; ** change the colour of a circle
  copy: circle; ** create a new circle with the attributes of the argument circle
  delCreen; ** delete all green circles
var x, y, r: Real, cir: Circle
• start = seq for all c: Circle. drop c,
  import c: Circle in
    set X(c) := 0, Y(c) := 0, radius(c) := 1, col(c) := green end
end;
• \textbf{post} \textit{move}(\textit{cir}, x, y): X'(\textit{cir}) = X'(\textit{cir}) + x \land Y'(\textit{cir}) = Y(\textit{cir}) + y;

• \textit{moveAll}(x, y) = \textit{forall} \textit{c: Circle. set} X(\textit{c}) := X(\textit{c}) + x, Y(\textit{c}) := Y(\textit{c}) + y \textit{end;}

• \textbf{post} \textit{resize}(\textit{cir}, r): \textit{radius}'(\textit{cir}) = r;

• \textit{changeCol}(\textit{cir}) = \textbf{if} \textit{colour}(\textit{cir}) = \textit{green} \textbf{then} \textit{colour}(\textit{cir}) := \textit{red} \\
\textbf{else} \textit{colour}(\textit{cir}) := \textit{green} \textbf{endif};

• \textit{copy}(\textit{cir}) = \textbf{import} \textit{c: Circle in set} X(\textit{c}) := X(\textit{cir}), Y(\textit{c}) := Y(\textit{cir}), \textit{radius}(\textit{c}) := \textit{radius}(\textit{cir}), \textit{col}(\textit{c}) := \textit{col}(\textit{cir}) \textit{end;}

• \textit{delGreen} = \textbf{forall} \textit{c: Circle. if} \textit{colour}(\textit{c}) = \textit{green} \textbf{then} \textit{drop} \textit{c;}

\textbf{end}

\section{8 Related Work}

As mentioned in the introduction, this extension of CASL is based on several works using the concept of implicit state. Currently none of them offers the full set of tools needed for the specification of a broad range of dynamic systems. Therefore the natural combination of the facilities offered by these works and their adaptation to the CASL institution has been one of our main goals.

We have liked very much the concept of update set introduced by Gurevich in [13] for the explanation of state transitions. It was used later by him in [14] for giving denotational semantics of transition rules of Abstract State Machines (ASMs). We also give the semantics of our transition terms in terms of update sets. However, ASMs are based on total universal algebras treating predicates as Boolean functions. Since our states are partial many-sorted structures, we have extended $\alpha$-updates of ASMs with $\beta$- and $\delta$-updates representing, respectively, the updates over predicates and sorts. Our notion of transition term is an extension and generalization of the ASM notion of transition rule. The amendments are the procedure call, the \textit{drop} rule allowing a sort to shrink, the sequence constructor and based on it the loop constructors (see also [6] for another proposition of sequence and loop constructors). ASMs do not have such features as dependent functions and procedures, nor is the notion of dynamic system defined for them.

Dependent functions and procedures are borrowed from [8,15] through intermediate steps of [19,9]. However, their semantics is different. In [19,9] dependent functions are not part of the state; they belong to the dynamic system, and this caused some problems with the use of their names in terms. In [8,15] dependent functions are part of the state with a very complex semantics of their redefinition in different states. The introduction of the function \textit{map} has allowed us to treat dependent functions as part of the state with very simple semantics of their redefinition. The semantics of modifiers (analogue of our procedures) and update expressions (analogue of our transition terms) is given in [8,15] operationally (there is no notion of update set in this approach), while we have done it denotationally. We have also provided the means for working with partial structures (only total many-sorted algebras are used in [8,15]).

The notion of dynamic system, as it is defined in our paper, stems from the notion of \textit{d-oid} introduced in [1,2] and further developed in [20,21]. A d-oid is a set of instance structures (e.g., algebras), a set of dynamic functions resembling
our dependent functions and procedures and a tracking map indicating relationships between instance structures. The approach deals with models and does not address the issue of d-oid specification. Therefore, we had to borrow and adapt for our purpose a specification technique of another source.

Another approach are “Transition Categories” presented in [10,11]. Here the algebra of data types is equipped with sorts called reference sorts in addition to ordinary sorts, and the state extends this algebra by partial functions called contents functions for each reference sort which map elements of reference sorts to their contents. State transformations are defined by conditional parallel assignment redefining the contents functions. Though based on different semantical foundations, Große-Rhode’s approach appears as a special class of dynamic system defined in this paper where the static part contains the reference sorts and the only components of the dynamic part are the contents functions. Then the effect of conditional parallel assignment can be achieved using dynamic equations.

In a later work [12], Große-Rhode defines Algebra Rewrite Systems. A rewrite rule r is of the form $P_l \leftrightarrow P_r$ where $P_l = (X_l, E_l)$ and $P_r = (X_r, E_r)$ are presentations consisting of sets of generators $X_l$ and $X_r$ and sets of equations $E_l$ and $E_r$, respectively. A rule is applied to a partial algebra $A$ by first removing the elements of $X_l$ from the carrier sets of $A$ together with the equalities in $E_l$, and then adding the elements in $X_r$ and the equalities in $E_r$. This allows the modelling of update instructions $f(t_1) := t_2$ by first undefining a function entry and then adding a new function entry by a rule $\{x : s\} \leftrightarrow \{f(t_1) = x\} \leftrightarrow \{x : s\}$. Deletion of an element of sort $s$ is modeled by a rule $\{x : s\} \leftrightarrow \{\emptyset, \emptyset\}$, and creation of an element by a rule $\{\emptyset, \emptyset\} \leftrightarrow \{x : s\}$.

However, the approach by Große-Rhode is restricted to partial algebras and thus cannot be used in the context of CASL, which is order-sorted and in addition to partial function contains total functions and predicates. Further, the application of rewrite rules and also the interaction between axioms defining the static part with rewrite rules may yield unexpected results, like the identification of elements in the result state.

While all the above approaches favour an operational style of writing specifications, with the exception of Zucca [21], that is, they specify how a state is transformed into another state, the approach by Baumeister [3,4,5] uses a declarative approach, defining how the states before and after the execution of a state transformation are related. This allows writing specifications which are similar to those written in specification languages like Z or VDM. Baumeister’s approach does not require states to be modeled as algebras; states can be structures from any suitable institution. From this approach we have used the idea how to interpret postcondition formulae.

9 Conclusion

In this paper we have defined an extension of CASL for the specification of state-based software systems intended to be part of the common framework defined by the Common Framework Initiative. It permits the specification of the static part of a complex dynamic system by means of CASL and the dynamic part by means of the facilities described in the paper. This is one level of integration of
two different specification paradigms. The specification of the dynamic part can also be done either by means of transition rules or by means of postconditions (where it is appropriate). This provides the second level of integration. Moreover, the use of update sets for describing the semantics of both transition rules and postconditions has permitted us to define the semantics in a simple way and easily solve the well-known frame problem.

The next step in the development of the described specification technique is the introduction of structuring facilities permitting the specifications to be united, localized, parameterized, etc.

References

State-Based Extension of CASL 23

A Postcondition Formulae

In this section we give a formal definition of $\Sigma$, $\Gamma$, and postcondition formulae.

Let

$$\Delta_{\text{dyn}} = (S_{\text{dyn}}, TF_{\text{dyn}}, PF_{\text{dyn}}, P_{\text{dyn}})$$

be a signature extension w.r.t. $\Sigma_{\text{stat}} = (S_{\text{stat}}, TF_{\text{stat}}, PF_{\text{stat}}, P_{\text{stat}})$,

$$\Delta_{\text{dep}} = (\emptyset, TF_{\text{dep}}, PF_{\text{dep}}, P_{\text{dep}})$$

be a signature extension w.r.t. $\Sigma_{\text{stat}} \cup \Delta_{\text{dyn}}$, then

$$\Delta = (S, TF, PF, P) = (S_{\text{dyn}} \cup TF_{\text{dyn}}, TF_{\text{dep}} \cup PF_{\text{dyn}} \cup PF_{\text{dep}}, P_{\text{dyn}} \cup P_{\text{dep}})$$

is a signature extension w.r.t. $\Sigma_{\text{stat}}$. We define $\bar{\Sigma} = (\bar{S}, \bar{TF}, \bar{PF}, \bar{P})$ as follows:

$$\bar{S} = S_{\text{stat}} \cup S_{\text{dyn}} \cup \{s' \mid s \in S_{\text{dyn}}\}$$

$$\bar{TF} = \{f : w \rightarrow s_0 \mid f : w \rightarrow s_0 \in (TF_{\text{stat}} \cup TF)\} \cup \{f' : \bar{s}_1, \ldots, \bar{s}_n \rightarrow \bar{s}_0 \mid f : s_1, \ldots, s_n \rightarrow s_0 \in TF, \bar{s}_i = s'_i \text{ if } s_i \in S_{\text{dyn}}, \bar{s}_i = s_i \text{ else, } 0 \leq i \leq n\}$$

$$\bar{PF} = \{f : w \rightarrow s_0 \mid f : w \rightarrow s_0 \in (PF_{\text{stat}} \cup PF)\} \cup \{f' : \bar{s}_1, \ldots, \bar{s}_n \rightarrow \bar{s}_0 \mid f : s_1, \ldots, s_n \rightarrow s_0 \in PF, \bar{s}_i = s'_i \text{ if } s_i \in S_{\text{dyn}}, \bar{s}_i = s_i \text{ else, } 0 \leq i \leq n\} \cup \{tm_s : s \rightarrow s' \mid s \in S_{\text{dyn}}\}$$

$$\bar{P} = \{p : w \mid p : w \in (P_{\text{stat}} \cup P)\} \cup \{p' : \bar{s}_1, \ldots, \bar{s}_n \mid p : s_1, \ldots, s_n \in P, \bar{s}_i = s'_i \text{ if } s_i \in S_{\text{dyn}}, \bar{s}_i = s_i \text{ else, } 0 \leq i \leq n\}$$
Definition 11. A postcondition formula is a formula of the form

$$\text{post}\, p(t_1, \ldots, t_n) : \varphi$$

where \( p : s_1, \ldots, s_n \) is a procedure symbol, \( s_1, \ldots, s_n \) are sorts in \( S_{\text{stat}} \cup S_{\text{dyn}} \), \( t_1, \ldots, t_n \) are terms over variables \( X \) of sorts \( s_1, \ldots, s_n \), and \( \varphi \) is a formula over \( \bar{\Sigma} \) with variables \( X \).

Given a state \( A \) in a dynamic system \( DS(B) \) and an update-set \( \Gamma \), we define a \( \bar{\Sigma} \)-structure \( A\Gamma \) by:

- For all sort symbols in \( \bar{S} \):
  - \( |A\Gamma|_s = |A|_s \) if \( s \in S_{\text{stat}} \cup S_{\text{dyn}} \)
  - \( |A\Gamma|_{s'} = |A\Gamma|_s \) if \( s' \in \bar{S} \setminus (S_{\text{stat}} \cup S_{\text{dyn}}) \)

- For all total function symbols in \( PF \):
  - \( f^{A\Gamma} = f^A \) if \( f \in TF_{\text{stat}} \cup TF \)
  - \( f^{A\Gamma} = f^{A\Gamma} \) if \( f' \in TF \setminus (TF_{\text{stat}} \cup TF) \)

- For all partial function symbols in \( \bar{PF} \),
  - \( f^{A\Gamma} = f^A \) if \( f \in PF_{\text{stat}} \cup PF \)
  - \( f^{A\Gamma} = \text{tm}^{A\Gamma}_s \) if \( f = \text{tm}_s \)
  - \( f^{A\Gamma} = f^{A\Gamma} \) if \( f' \in \bar{PF} \setminus (PF_{\text{stat}} \cup PF \cup \{ \text{tm}_s | s \in S_{\text{dyn}} \}) \)

- For all predicate symbols in \( \bar{P} \),
  - \( p^{A\Gamma} = p^A \) if \( p \in P_{\text{stat}} \cup P \)
  - \( p^{A\Gamma} = p^{A\Gamma} \) if \( p' \in \bar{P} \setminus (P_{\text{stat}} \cup P) \)

Note that one can write \( f^{A\Gamma} = f^{A\Gamma} \) and \( p^{A\Gamma} = p^{A\Gamma} \), though different sort symbols are used in the profiles of \( f \) and \( f' \) (\( p \) and \( p' \)), since the carrier-set of each \( s' \) in \( A\Gamma \) is the same as the carrier set of the corresponding \( s \) in \( A\Gamma \).

A dynamic system \( DS(B) \) satisfies a postcondition formula

$$\text{post}\, p(t_1, \ldots, t_n) : \varphi$$

iff for any state \( A \) and variable assignment \( \sigma : X \to A \) if the update-set

$$\Gamma = p^{DS(B)}(A, <[t_1]^{A,\sigma}, \ldots, [t_n]^{A,\sigma}>)$$

exists, then

$$\varphi^{A\Gamma,\sigma} = \text{true}.$$
Linking DC Together with TRSL

Anne Elisabeth Haxthausen and Xia Yong

1 Department of Information Technology, Techn. Univ. of Denmark, DK-2800 Lyngby
e-mail: ah@it.dtu.dk
2 Department of Information Technology, University of Zürich, CH-8057 Zürich
e-mail: xia@ifi.unizh.ch

Abstract. Duration Calculus (DC) is an interval-based real-time logic, which can be used in capturing and eliciting users’ real-time requirements. The Timed RAISE Specification Language (TRSL) is an extension of the RAISE Specification Language with real-time features. This paper links DC and TRSL together in a method for real-time developments. An operational semantics with behavior is specified for TRSL. It is defined what it means for a TRSL process to satisfy a DC requirement, and a method for verifying whether the satisfaction relation holds or not is provided. Our contribution also demonstrates a general approach for linking state-based real-time logics together with event-based, timed process algebra languages.

Keywords: Formal methods, RAISE, Duration Calculus, integration of specification formalisms.

1 Introduction

Duration Calculus (DC) [12] is an interval-based real-time logic, which captures and elicits users’ real-time requirements in the form of constraints on the durations of states of the system, i.e. at a high level of abstraction. However, as a state-based logic, it lacks the ability to specify sequential programs and communicating concurrent processes at a concrete level.

The Timed RAISE Specification Language (TRSL) [11] has this ability. TRSL is a recent real-time extension of the RAISE Specification Language (RSL) [9] which together with its associated method [10] and tools has shown to be very useful in the industrial development of software systems. However, TRSL lacks the ability of DC to specify timing properties at a high level of abstraction.

Therefore, a promising approach for the development of real-time systems could be to use DC for high-level specifications of real-time requirements and TRSL for specifying real-time implementations in the form of communicating concurrent processes. To be more precise the main idea for a development method for real-time systems is as follows (see figure 1):

1. The RAISE method [10] is used for stepwise developing a specification of the untimed properties of the system, starting with an abstract, property-oriented RSL specification and ending with a concrete, implementation-oriented RSL specification.
2. In parallel with the RSL development of the untimed system, a DC requirement specification of the real-time properties of that system is developed. State variables in the DC specification are variables defined (at least) in the last RSL specification.

3. Timing information is added to the RSL specification achieving a TRSL specification of a real-time implementation.

This method rises two questions: when is the TRSL specification a correct development of the RSL specification, and when does it satisfy the DC requirements. The first question has been addressed in [2], while the goal of this paper is to address the second question.

First, in section 2 we give a brief summary of the DC variant (DC with Super-Dense Chop) that we have used, and in section 3 we introduce an operational semantics with behavior for TRSL. Then, in section 4, we use the operational semantics with behavior to define a satisfaction relation between TRSL and DC, and in section 5 we present an associated verification method. In section 6 we illustrate our method on a case study, and finally, in section 7, we state conclusions and topics for future work. Appendices A and B contain the rules of the operational semantics and proof rules of the verification method.

2 Duration Calculus with Super-Dense Chop

The classical Duration Calculus (DC) [12] is an interval temporal logic, which can be used to specify constraints on the durations of states, but not to specify super-dense computations, i.e. events that are instantaneous and may happen simultaneously. Hence, classical DC can’t be used to describe TRSL events like input, output and assignments that according to the TRSL time model are assumed to be instantaneous and may happen simultaneously. However, there exists variants of DC that can be used for this purpose, e.g. Duration Calculus with Super-dense Chop [13] and Duration Calculus of Weakly Monotonic Time [7]. We have chosen to use the first of these (but could equally well have chosen the second one). Duration Calculus with Super-dense Chop provides a new chop modality • called super-dense chop and left and right neighborhood properties
\( \neg S \) and \( \neg S \), where \( S \) is a state. The idea is to express instantaneous events as neighborhood properties and combine these sequentially using the super-dense chop modality which chops a time point in the grand time space into a non-point interval in a fine time space. For instance, the behavior of \( x := x + 1 \) can be described as \( \exists v \cdot \neg (x = v) \land \neg (x = v + 1) \).

Below we give a short survey of the syntactical categories of states, terms and formulas. They are all constructed from a given set \( \mathcal{SV} \) of typed state variables. The detailed semantics and axioms and theorems can be found in [13].

**States:** The set \( \text{State} \) of \( \mathcal{SV} \)-states is constructed from the set of state variables \( \mathcal{SV} \) using lifted versions of operators like \( = \) and \( + \). For instance, if \( x \) and \( y \) are state variables of type \( \mathbb{R} \) then \( x = 1 \) and \( x = y + 1 \) are states. State variables of a type \( T \) denote \( T \)-valued functions of time. States denote Boolean-valued functions of time.

**Terms:** The set \( \text{Term} \) of \( \mathcal{SV} \)-terms is constructed from the set of \( \mathcal{SV} \)-states and a given set of constants \( \text{Var} \) according to the following grammar, where \( S \in \text{State}, v \in \text{Var}, \) and \( r \in \text{Term} \):

\[
r :: v \mid \int S \mid r + r \mid r - r \mid r \cdot r
\]

A term denotes a real-valued interval function. For instance, the term \( \int S \) denotes the duration of \( S \) on the interval of consideration.

**Formulas:** The set \( \text{Formula} \) of \( \mathcal{SV} \)-formulas is constructed from the set of \( \mathcal{SV} \)-states and the set of \( \mathcal{SV} \)-terms according to the following grammar, where \( S \in \text{State}, r \in \text{Term}, \) and \( \phi \in \text{Formula} \):

\[
\phi :: r = r \mid r > r \mid \neg r \mid \phi \lor \phi \mid \phi \land \phi \mid \phi \Rightarrow \phi \mid \exists v. \phi \mid \phi \bullet \phi \mid \sqcup \phi
\]

Formulas denote Boolean-valued interval functions. For instance, the formula \( \neg \phi \) is true for a given interval, if there is a left neighborhood of the interval in which \( S \) is true, and \( \sqcup \phi \) is true, if \( \phi \) is true for all subintervals.

The following abbreviations are frequently used:

\[
\ell \triangleq [1] \quad \text{(length of interval)} \\
[\ell] \triangleq (\ell = 0) \quad \text{(is a point interval)} \\
[S] \triangleq (\ell > 0) \land (\int S = \ell) \quad \text{(S holds essentially)} \\
[S]^* \triangleq [\ell] \lor [S] \quad \text{(is a point or S holds essentially)} \\
\neg S \triangleq [\ell = 0] \land \neg S \quad \text{(S holds in left neighborhood of point)} \\
\neg S \triangleq [\ell = 0] \land \neg S \quad \text{(S holds in right neighborhood of point)}
\]

3 An Operational Semantics with Behavior for TRSL

An operational semantics [11] in the style of Plotkin [8] has been given for a core of TRSL. In order to define a satisfaction relation between TRSL specifications and DC formulas, below we suggest an extension of the original operational semantics.

3.1 TRSL Core Syntax

A core TRSL specification consists of a set \( \Sigma \) of declarations of program variables, channels and values (functions), and a set \( A \) of axioms defining the values.
An example of a TRSL specification can be found in section 6. The syntax for core TRSL \( \Sigma \)-expressions used in the axioms is:

\[
E ::= \text{true} \mid \text{false} \mid r \mid T \mid () \mid \text{skip} \mid \text{stop} \mid \text{chaos} \mid \text{id} \mid x \mid \lambda \text{id} : \tau \bullet E \mid EE \mid \text{let } \text{id} = E \text{ in } E \mid \text{if } E \text{ then } E \text{ else } E \mid x := E \mid \text{while } E \text{ do } E \mid E \mid E ⌈⌉ \mid E ⌉⌊⌋ \mid E \parallel E \mid E – E \mid E;E \mid \text{wait } E \mid \text{let } t = c ? x \text{ in } E \mid \text{let } t = c ! E \text{ in } E
\]

where \( x \) is a variable name from \( \Sigma \), \( c \) a channel name from \( \Sigma \), \( t \) and \( \text{id} \) value names, \( r \) a real and \( T \) a time duration (i.e. positive real). The three last constructs are the real-time constructs of TRSL. A wait statement \text{wait } E \), where \( E \) denotes a time duration \( T \), causes the time to elapse \( T \) time units. An input expression \text{let } t = c ? x \text{ in } E \) offers to input from the channel \( c \). If this leads to a communication on \( c \), the value input from \( c \) will be stored in the variable \( x \), \( t \) will be bound to a time value representing the time elapsed between input being ready and the communication taking place, and \( E \) will be executed in the scope of \( t \). The meaning of output expressions are defined similarly. The remaining constructs are described in [9]. Here we just give a few remarks. () is equivalent to \text{skip}. There are two concurrency operators: the parallel operator \( \parallel \) and the interlocking operator \( – \parallel \), where the latter differs from the former by not allowing its arguments to communicate with external processes, but forcing them to communicate with each other or deadlock if that is not possible and none of them can terminate. We assume that the components \( E_1 \) and \( E_2 \) of concurrent expressions \( E_1 \parallel E_2 \) and \( E_1 – \parallel E_2 \) do not access the same variables.

In addition to the above syntax, we introduce the following shorthands: \( c!E \) is short for \( \text{let } t = c!E \text{ in skip} \), \( c?x \) is short for \( \text{let } t = c?x \text{ in skip} \) and \( c? \) is short for \( c?x \), when \( x \) has type \text{Unit}.

### 3.2 Definitions and Notation

Here, we introduce some definitions and notation, which will be used in the operational semantics.

**Stores** A \( \Sigma \)-store \( s \) is a map from variables in \( \Sigma \) to values.

**Environments** An environment \( \rho \) is a finite map from identifiers (of formal function parameters) to values (actual parameters).

**Events** There are three kinds of \( \Sigma \)-events: Visible events (input events: \( c?v \) and output events: \( c!v \), where \( c \) is a channel in \( \Sigma \)), time measurable events (\( \varepsilon(d) \), \( d > 0 \)) and silent events (\( \varepsilon \)). In the semantics, “\( \diamond \)” denotes any event; “\( \Delta \)” denotes any visible or silent event; \( a \) denotes any visible event and \( \overline{a} \) its complement event (i.e. \( c\overline{?}v = c?v \) and \( c\overline{!}v = c!v \)).

**Configurations** The operational semantics is based on the evolution of configurations. A basic \( \Sigma \)-configuration is a pair \( < E, s > \), where \( E \) is a \( \Sigma \)-expression and \( s \) a \( \Sigma \)-store. Following the approach of [4,1], in the evolution of configurations, it is sometimes necessary to distribute the store to its sub expressions so that they can evolve separately. Hence, in addition to basic
configurations, we need a more advanced notion of configurations which are built from basic configurations in a similar way as composite TRSL expressions are built from basic TRSL expressions. For instance, if $\alpha_1$ and $\alpha_2$ are (basic) configurations then $\alpha_1 \Box \alpha_2$ is a configuration. The detailed mathematical definition can be found in [11].

**Behaviors** A $\Sigma$-behavior is a DC $\Sigma^\#$-formula, where $\Sigma^\#$ consists of the following DC state variables:

- For each channel $c$ in $\Sigma$, DC state variables $c?a$ and $c!$, which record when a process is willing to receive an input and send an output, respectively, on channel $c$.
- For each program variable $x$ in $\Sigma$, a DC state variable $x$, which records the value of $x$ as a function of time.

The purpose of a $\Sigma$-behavior is to record the history about the evolution of the observables (the state variables in $\Sigma^\#$) of a TRSL $\Sigma$-process.

In the operational rules, $[s]$ is an abbreviation for $[x_1 = v_1] \land ... \land [x_n = v_n]$, where $s$ is a store equal to $[x_1 \mapsto v_1, ..., x_n \mapsto v_n]$. $\lhd s$, $\rhd s$, $\lhd^\circ s$, and $\rhd^\circ s$ are similar abbreviations.

In the sequents and side conditions of the operational rules we use a number of auxiliary functions the meaning of which is described below.

- $\text{var}(E)$ denotes the set of variables occurring in expression $E$.
- $s \setminus \text{set}$, where set is a set of names, denotes the store obtained from store $s$ by removing the names in set from the domain of $s$. $\text{Sort}_d(\alpha)$ denotes the set of ports (channel names tagged with a '?' or '!' symbol) that corresponds to the next possible visible (input or output) events that $\alpha$ can initially evolve with within the next $d$ units of time. The detailed mathematical definition of $\text{Sort}_d(\alpha)$ can be found in [11].
- $\text{SORT}_d$ denotes the set of ports that the environment can initially evolve with within the next $d$ units of time.

### 3.3 Operational Rules

The operational rules define a labeled transition relation between configurations with behaviors. The basic form of the transition relation is:

$$\rho \vdash \alpha \text{ with } \phi \overset{e}{\rightarrow} \alpha' \text{ with } \phi'$$

where $\rho$ is an environment, $\alpha$ and $\alpha'$ are $\Sigma$-configurations, $\phi$ and $\phi'$ are $\Sigma$-behaviors, and $e$ is a $\Sigma$-event. The transition expresses that the first step of execution of $\alpha$ when the event $e$ happens will lead to a new configuration $\alpha'$ with a new behavior $\phi'$. The new behavior (history) $\phi'$ is an extension of the old behavior (history) $\phi$ with the history of what could be observed during the event $e$. Other possible forms of the transition relation include:

$$\rho \vdash \alpha \text{ with } \phi \mid \beta \text{ with } \phi'' \overset{e}{\rightarrow} \alpha' \text{ with } \phi' \mid \beta' \text{ with } \phi'''$$

$$\rho \vdash \alpha \text{ with } \phi \parallel s \parallel \beta \text{ with } \phi'' \overset{e}{\rightarrow} \alpha' \text{ with } \phi' \parallel s \parallel \beta' \text{ with } \phi'''$$

The collection of operational rules can be found in appendix A. Here we just show one of the rules (without premises):
\[
\rho \vdash \text{let } t = c?x \text{ in } E, s > \quad \text{with } \phi \quad \xrightarrow{c?x} \\
< E[0/t], s \upharpoonright [x \mapsto v] > \quad \text{with } \phi \bullet (c?s \land \bigwedge (s[x \mapsto v]))
\]

It states that if a process described by an input expression \(\text{let } t = c?x \text{ in } E\) in the context of a store \(s\) inputs a value \(v\) on channel \(c\) then this will lead to the new process described by the expression \(E[0/t]\) in an updated store in which \(x\) is bound to \(v\). Furthermore, the new behavior will be an extension of the old one with a DC formula expressing that the time interval of the input event is of length \(0\) and in the left neighborhood of this interval the variables are bound to values as described by the old store \(s\) and in the right neighborhood the variables are bound to values as described by the new store \(s^\dagger\) \((i.e. the value of \(x\) is changed to \(v\), while all other variables are stable).

3.4 Validation of the Operational Rules

The original operational rules without behaviors describe (implicitly) possible interpretations of a \(\Sigma\)-store (cf. [11]). We have validated that the behaviors (DC formulas) added to the original operational rules describe these intended interpretations. For instance, for any conclusion of the form \(\rho \vdash \alpha\) \(\text{with } \phi \xrightarrow{\cdot} \alpha'\) \(\text{with } \phi'\), the new behavior \(\phi'\) must be equivalent to \(\phi \bullet \phi''\) for some \(\phi''\) such that \(\phi''\) conforms with the event \(e\) and the states \(s\) and \(s'\) of the pre and post configurations \(\alpha\) and \(\alpha'\). This means for instance that \(\phi''\) must describe an interval for which (1) the length fits with \(e\) (i.e. if \(e\) is instantaneous then \(\phi'' \Rightarrow \ell = 0\), and if \(e = \varepsilon(d)\) then \(\phi'' \Rightarrow \ell = d\)), (2) the left and right neighborhoods are \(s\) and \(s'\) (i.e. \(\phi'' \Rightarrow \uparrow s \land \bigwedge s')\), and (3) the state is stable if the event is time measurable (i.e. if \(e = \varepsilon(d)\) then \(s = s'\) and \(\phi'' \Rightarrow \lceil s \rceil\)).

4 Satisfaction Relation

In order to define the satisfaction relation, we first define the notion of a behavior of a TRSL expression as a finite prefix of one of its possible behavior histories that can be derived by repeated use of the operational rules:

**Definition 4.1** A DC formula \(\text{bh}\) is called a behavior of a TRSL \(\Sigma\)-expression \(E\) wrt. an initial \(\Sigma\)-store \(s_0\), iff there exists a configuration \(\alpha\), such that\(^1\)

\[ [\alpha] \vdash < E, s_0 > \quad \text{with } \bigwedge (\hat{\alpha})^* \quad \alpha \text{ with } \text{bh} \]

Moreover, if \(\alpha\) is of the form \(< v, s >\), where \(v\) is a computational value (i.e. a value literal or a lambda expression), \(\text{bh}\) is called a terminated behavior.

**Definition 4.2** A TRSL \(\Sigma\)-expression \(E\) satisfies a DC \(\Sigma^\theta\)-formula \(\phi\), (written \(E \text{ sat}_\Sigma \phi\), iff for any initial \(\Sigma\)-store, \(s_0\), for any behavior \(\text{bh}\) of \(E\) wrt. \(s_0\), there exists a DC formula \(tl\) such that \(\text{bh} \bullet tl\) is also a behavior of \(E\) wrt. \(s_0\) and \(|=_{dc} \text{bh} \bullet tl \Rightarrow \phi\). And when \(\text{bh}\) is a terminated behavior of \(E\), \(tl = \bigwedge\).

\(^1\) Below, \((\hat{\alpha})^*\) denotes the transitive closure of the transition relation.
5 Verification Method

It is not always practical directly to use Definition 4.2 to prove whether a TRSL expression satisfies a DC requirement or not. The reason is that some TRSL expressions have many (sometimes even infinitely many) behaviors. This section gives our solution to cope with this problem.

5.1 Common Histories

Our main idea is to use a common history formula (or characteristic formula) of a TRSL expression, instead of its behaviors to prove whether the expression satisfies a DC formula. A common history formula of a TRSL expression is a DC formula which is an abstraction of all the possible behaviors of the expression. It expresses what is common to all its behaviors.

Definition 5.1 A DC formula $\psi$ is called a common $\Sigma$-history (formula) of a TRSL $\Sigma$-expression $E$, iff for any initial $\Sigma$-store, $s_0$, for every behavior formula $\varphi$ of $E$ wrt. $s_0$, there exists a DC formula $\theta$, such that $\varphi \cdot \theta$ is also a behavior wrt. $s_0$, and $|=dc \varphi \cdot \theta \Rightarrow \psi$.

Notation: If $\psi$ is a common history of a $\Sigma$-expression $E$, we write $H_\Sigma(E) \ni \psi$. We drop the subscript $\Sigma$, when it is obvious from the context.

We can now use the following theorem to prove that an expression satisfies a DC requirement:

Theorem 5.2 $E$ sat$_\Sigma$ $\phi$, if there exists a common $\Sigma$-history formula $\psi$ of $E$, such that $|=dc \psi \Rightarrow \phi$.

The next problem is how we can derive a common history formula of a TRSL expression. Below, we will give some examples and in next subsection we introduce a proof system.

Assignments. In a context where only one variable $x$ has been declared, the behavior of the TRSL expression $x := x + 1$ depends on the value of $x$ before the assignment, i.e. on the initial store. Abstracting from the initial store, a common history of this expression is

$$\exists v \cdot \land (x = v) \land \lor (x = v + 1)$$

A common history of $x := 2$ reduces to $\lor (x = 2)$

Wait. In a context where only one variable $x$ has been declared, a common history of wait 5 is

$$\ell = 5 \land \exists v \cdot ([x = v] \land \land (x = v) \land \land (x = v))$$
Sequence 1. In a context where only one variable $x$ has been declared, a common history of $x := 2; \text{wait} \ 5$ is

$$\bigwedge(x = 2) \bullet (\ell = 5 \land \exists v \cdot ([x = v] \land \tau_0(x = v) \land \rho(x = v)))$$

which according to the rules for DC with super dense chop can be reduced to

$$\ell = 5 \land [x = 2] \land \rho(x = 2)$$

A common history of an expression of the form $\ldots; x := 2; \text{wait} \ 5; \ldots$ is of the form $\ldots \bullet \ell = 5 \land [x = 2] \bullet \ldots$

Input. The behaviors of $c?x$ wrt. an initial store $[x \mapsto v_0]$, include the following major forms:

1. $\tau_c(x = v_0) \land \bigwedge(x = v)$
   corresponding to the case where the environment is willing immediately to output $v$ on channel $c$
2. $[x = v_0] \land [c?] \land \ell = d \land \tau_0(x = v_0) \land \rho(x = v_0)$
   corresponding to the case where the environment is not willing to output on $c$ for $d$ time units
3. $[x = v_0] \land [c?] \land \ell = d \land \tau_0(x = v_0) \land \rho(x = v)$
   corresponding to the case where the environment is willing after $d$ time units to output the value $v$ on $c$

A common history which generalizes all these forms is:

$$[c?]^* \land \exists v_0, v \cdot ([x = v_0]^* \land \tau_0(x = v_0) \land \rho(x = v))$$

Sequence 2 (input before "!"). Consider the expression $x := 1;c?; x := 2; \text{wait} \ 5; \ldots$ and assume that $x$ is the only considered variable.

For the cases where the environment is willing to output on $c$, we can generalize its possible behaviors as: $[x = 1 \land c?] \bullet (\ell = 5 \land [x = 2]) \bullet \ldots$. For the cases where the environment is not willing to output on $c$, we can generalize its possible behaviors as: $[x = 1 \land c? \land \rho(x = 1)]^*$. Hence, all kinds of behaviors can be generalized by the following common history:

$$([x = 1 \land c?] \bullet (\ell = 5 \land [x = 2]) \bullet \ldots) \lor [x = 1 \land c? \land \rho(x = 1)]^*$$

Concurrency. In order to find a common history of the concurrent expression $\text{wait} \ 5;c?x \parallel c3$, we first consider common histories of the sequential components:

$$H(\text{wait} \ 5;c?x) \ni$$

$$(\ell = 5 \land \exists v_0 \cdot ([x = v_0] \land \tau_0(x = v_0) \land \rho(x = v_0)))) \bullet$$

$$[c?]^* \land \exists v_0, v \cdot ([x = v_0]^* \land \tau_0(x = v_0) \land \rho(x = v))$$

and

$$H(c3) \ni [c!]^*$$
(Note that the common history of \(c !3\) is found in a context with no variables.) TRSL has a \textit{maximal progress} assumption, so the two sequential components must communicate as early as possible. Now we “match” the two histories against each other and find that both first have to wait 5 time units and then they can communicate. So a common history of \(\text{wait} \ 5; c ? x \parallel c !3\) is
\[
(\ell = 5 \land \exists v_0 \cdot ([x = v_0] \land r_\Sigma(x = v_0)) \cdot \mathcal{A}(x = 3)
\]

## 5.2 Rule System

For TRSL expressions being in a certain standard form \cite{3}, we have developed proof rules for deriving their common histories. Some expressions not being in this standard form can be converted into the standard form using the TRSL proof rules in \cite{11}, so in this sense we are able to find the common history for more general expressions.

The rule system defines an expansion relation \(\rightarrow\) between terms which can be DC formulas and applications of some special functions, \(H_{\Sigma}, assign_{\Sigma}, input_{\Sigma}, output_{\Sigma}, wait_{\Sigma}, waitinput_{\Sigma}, waitoutput_{\Sigma}, \circ, I_N_{\Sigma}\) and \(OR_{\Sigma}\). The idea is that, if \(H_{\Sigma}(E) \rightarrow^* \phi\) for a TRSL \(\Sigma\)-expression \(E\) and a pure DC formula \(\phi\) then \(\phi\) is a common \(\Sigma\)-history of \(E\), i.e., \(H_{\Sigma}(E) \triangleright \phi\). The side conditions in some of the rules use other expansion relations, \(\rightarrow_{\text{COMM}}\) and \(\rightarrow_{\text{WAIT}}\), which are defined by their own rules.

In appendix B we list some of the most important rules needed for the case study in section 6.

## 6 A Case Study

In this section, we illustrate our verification method on an example: a simplified version of the gas burner system proposed in \cite{6}. A gas burner is either heating when the flame is burning or idling when the flame is not burning, and it alternates indefinitely between heating and idling. Usually no gas is flowing while it is idling. However, when changing from idling to heating, gas must be flowing for a short time before it can be ignited. Hence, there may be a period where gas is flowing and a flame is not burning, i.e., gas is leaking. A design of a safe gas burner must ensure that the time intervals where gas is leaking do not get too long.

### 6.1 TRSL Specification

The specification below defines a design for a gas burner process and assumptions about its environment. The \textit{GasBurner} consists of three parallel components: \textit{MainController}, \textit{HeatController} and \textit{FlameController}. The variables \textit{gas} and \textit{flame} are equal to 1 when the gas and flame, respectively, are on, and 0 otherwise. The \textit{HeatController} senses whether the environment is giving a request (on

2 The subscript \(\Sigma\) can be dropped when the considered \(\Sigma\) is obvious from the context.
channel HeatOn or HeatOff) for heat to be turned on or off, and signals this (on channels hon and hoff) to the MainController. The main controller controls the flow of gas based on the heat requests it receives and informations (on channel flon) about the appearance of a flame. The FlameController controls the flame. The environment, Env, has a fixed pattern, parameterized with four parameters $x_1, x_2, x_3$ and $x_4$.

Figure 2 illustrates the communication relations among the four processes.
Env() ≡ wait \( x_2 \); HeatOn(); GasOn?
\( ((\text{wait } x_1; \text{FlOn!}()); \text{wait } x_4; \text{HeatOff!}()) \mid \text{wait } x_3) \)
GBSystem() ≡ \textbf{while true do} (Env() \parallel \text{GasBurner()})

6.2 DC Requirements

The gas burner system must satisfy the following DC formulas expressing that
(1) a leak must not last longer than 1 second, and (2) the time distance between
two leaks must be at least 30 seconds:

1. \( \Box([\text{Leak}] \Rightarrow \ell \leq 1) \)
2. \( \Box(([\text{Leak}] \cdot [\neg \text{Leak}] \cdot [\text{Leak}]) \Rightarrow \ell \geq 30) \)

Here \( \text{Leak} \hat{=} \text{gas} = 1 \land \text{flame} = 0 \), where \( \text{gas} \) and \( \text{flame} \) are variables appearing
in the TRSL specification.

6.3 Verification

Below we sketch how our verification method can be used to prove that the
TRSL specification of the gas burner system, \( \text{GBSystem()} \), satisfies the DC
requirements given in Section 6.2, i.e.

1. \( \text{GBSystem()} \text{ sat } \Box([\text{Leak}] \Rightarrow \ell \leq 1) \)
2. \( \text{GBSystem()} \text{ sat } \Box(([\text{Leak}] \cdot [\neg \text{Leak}] \cdot [\text{Leak}]) \Rightarrow \ell \geq 30) \).

A Common History of the Gas Burner System

First we look for a common history formula of the gas burner system. Using the
proof rules we can derive:

\[
\begin{align*}
H(\text{MainController}()) & \rightarrow h_{mc} \\
H(\text{HeatController}()) & \rightarrow h_{hc} \\
H(\text{FlameController}()) & \rightarrow h_{fc} \\
H(\text{Env}()) & \rightarrow h_{env} \\
H(\text{Env}() \parallel \text{GasBurner}()) & \rightarrow I N(h_{env}, h_{mc}, h_{hc}, h_{fc}) \rightarrow h_{gb} \\
H(\text{GBSystem}()) & \rightarrow (h_{gb})^ω
\end{align*}
\]

where\(^4\)

\( h_{mc} \hat{=} \text{assign}_\Sigma([\text{gas} \mapsto 0]) \bullet \text{input}(\text{hon}, \_ \cdot \text{wait}(30) \bullet \text{output}(\text{GasOn,}()) \circ \text{assign}_\Sigma([\text{gas} \mapsto 1]) \bullet
((\text{input}(\text{flon}, \_ \cdot \text{input}(\text{hoff}, \_ \cdot \text{assign}_\Sigma([\text{gas} \mapsto 0]) \bullet
\text{output}(\text{floff,}())))
\lor
(\text{wait}(1) \bullet \text{output}(\text{gasoff,}()) \circ \text{assign}_\Sigma([\text{gas} \mapsto 0]) \circ
\text{output}(\text{noflame,}())))\)

\(^3\) \text{input}_\Sigma(c, \_) is a special form of \text{input}_\Sigma(c, x). It is short for \([c?]^* \land \exists s : \Sigma \cdot ([s]^* \land
\neg \Phi s \land \neg \Phi s)\)
36 A.E. Haxthausen and X. Yong

\[ h_{hc} \hat{=} \text{input}(\text{HeatOn},.) \circ \text{output}(\text{hon}(.) ) \circ \\
(\text{input}(\text{noflame},.) \lor \text{input}(\text{HeatOff},.) \circ \text{output}(\text{hoff}(.))) \]

\[ h_{fc} \hat{=} \text{assign}([\text{flame} \mapsto 0] \bullet \\
(\text{input}(\text{FlOn},.) \circ \text{assign}([\text{flame} \mapsto 1] \bullet \\
\text{output}(\text{flon}(.) ) \circ \text{input}(\text{floff},.) \circ \text{assign}([\text{flame} \mapsto 0] )) \lor \\
\text{input}(\text{gasoff},.) ) \]

\[ h_{env} \hat{=} \text{wait}(x_2) \bullet \text{output}(\text{HeatOn},.) \circ \text{input}(\text{GasOn},.) \circ \\
\text{OR}(\text{wait}(x_1) \bullet \text{output}(\text{FlOn}) \circ \text{wait}(x_4) \bullet \text{output}(\text{HeatOff},.), \\
\text{wait}(x_3)) \]

\[ h_{gb} \hat{=} ((\ell = x_2 \land [\text{hon} \land \text{HeatOn} \land \text{Floff}] \land [\text{gas} = 0 \land \text{flame} = 0]) \bullet \\
(\ell = x_1 \land [\text{FlOn} \land \text{HeatOff} \land \text{FlOn}] \land [\text{gas} = 1 \land \text{flame} = 0]) \bullet \\
(\ell = x_4 \land [\text{HeatOff} \land \text{heatoff} \land \text{FlOn}] \land [\text{gas} = 1 \land \text{flame} = 1] \\
\land \not\rho([\text{gas} = 0 \land \text{flame} = 0])) \lor \\
(\ell = x_2 \land [\text{hon} \land \text{HeatOn} \land \text{gasoff}] \land [\text{gas} = 0 \land \text{flame} = 0]) \bullet \\
(\ell = x_3 \land [\text{noflame} \land \text{gasoff} \land \text{FlOn}] \land [\text{gas} = 1 \land \text{flame} = 0]) \bullet \\
(\ell = 1 \land [\text{noflame} \land \text{gasoff}] \land [\text{gas} = 1 \land \text{flame} = 0]) \bullet \\
(\ell = x_3 - 1 \land [\text{gas} = 0 \land \text{flame} = 0] \land \not\rho([\text{gas} = 0 \land \text{flame} = 0])) \\
\lor \\
(\ell = x_1 \land [\text{noflame} \land \text{FlOn}] \land [\text{gas} = 0 \land \text{flame} = 0]) \bullet \\
(\ell = 30 \land [\text{noflame} \land \text{gasoff}] \land [\text{gas} = 0 \land \text{flame} = 0]) \bullet \\
(\ell = 1 \land [\text{noflame} \land \text{gasoff}] \land [\text{gas} = 1 \land \text{flame} = 0]) \bullet \\
(\ell = x_3 - 1 \land [\text{gas} = 0 \land \text{flame} = 0] \land \not\rho([\text{gas} = 0 \land \text{flame} = 0])) \]

Hence, \((h_{gb})^\omega\) is a common history formula of the whole gas burner system \(GBSystem()\).

Application of theorem 5.2.

According to theorem 5.2 we can now verify the proof obligations by proving

\[ \models_{dc} (h_{gb})^\omega \Rightarrow \omega ((\text{Leak} \Rightarrow \ell \leq 1) \land \\
\models_{dc} (h_{gb})^\omega \Rightarrow \omega ((\text{Leak} \land \not\text{Leak} \land \text{Leak}) \Rightarrow \ell \geq 30). \]

This can be done by using the following \(\omega\) rule

\[
\text{for any } n \geq 0, \text{ if } \models_{dc} \phi^n \Rightarrow \psi \text{ then } \models_{dc} \phi^{n+1} \Rightarrow \psi \]

and the standard proof rules for DC with super-dense chop.

7 Discussion

In this paper, we have proposed how DC and TRSL can be linked together in a real-time development method. Our approach for linking these two formalisms together was to extend the operational semantics of TRSL with behaviors which are DC formulas describing the history of the observables of the system, and
then define a satisfaction (or refinement) relation between sentences of the two languages in terms of behaviors.

Our approach can also be used as a general way of linking state-based real-time logics together with timed, event-based process algebra languages and is new to our knowledge. DC has e.g. previously been linked together with subsets of other event-based process algebra languages (e.g. OCCAM [13], HCSP [14], SDL [5]), but using another approach: the event-based languages were given a denotational semantics in terms of DC formulas. It is very straightforward to extend an operational semantics with behaviors and much easier (simpler) than giving a denotational DC semantics. This becomes especially evident for larger languages providing more advanced language constructs. For instance, using the operational approach rather than the denotational approach, it became possible to handle lambda expressions and to allow concurrent components to share output channels and input channels (both features provided by TRSL, but not by the previously considered languages). The behaviors record fewer observations than in the denotational approach making our resulting DC common history formulas much shorter, and hence, it also becomes easier to verify that a satisfaction relation holds. In the ProCoS approach [6], DC has been linked together with an event-based language via a transformational approach: by some rules, a DC design (in a certain subset of DC) can be transformed into a program which by construction satisfies the DC design. This approach has the advantage over our invent-and-verify approach that no proofs should be made, however, it has the disadvantage that it is less general: it does not provide the possibility of testing whether a given program satisfies a given DC formula.

Topics for future work include tool support for our verification method. A quite promising approach is to define an appropriate subset of TRSL for which the common history formulas of TRSL programs can be derived mechanically and for which these formulas are in a decidable subset of DC. Hence, if also the real-time requirements are formulated in the decidable subset of DC, the whole verification process can be done totally mechanically by a model checking tool.

Acknowledgment. The authors thank Zhou Chaochen, Michael R. Hansen and Chris George for their advice on this work and the anonymous referees for their comments.

References

A Operational Rules

Basic Expressions

\( \rho \vdash <\text{skip}, s> \) with \( \phi \vdash <\text{(),} s> \) with \( \phi \)

\( \rho \vdash <\text{stop}, s> \) with \( \phi \vdash <\text{stop}, s> \) with \( \phi \left[ \{ \phi \} \land \varepsilon \equiv \delta \land ^{\downarrow} s \land \triangleleft \right] \)

\( \rho \vdash <\text{chaos}, s> \) with \( \phi \vdash <\text{chaos}, s> \) with \( \phi \)

Configuration Fork

\( \rho \vdash E_1 \text{ op } E_2 ; s > \) with \( \phi \vdash <E_1, s> \) with \( \phi \) \( \text{ op } <E_2, s> \) with \( \phi \)

where \( \text{ op } = \text{ [ ]} \). [ ]

Look Up

\( \rho \vdash [\text{id} \rightarrow v] ; <\text{id}, s> \) with \( \phi \vdash <v, s> \) with \( \phi \)

\( \rho \vdash <\text{id}, s> \vdash [\text{id} \rightarrow v] ; <v, s> \) with \( \phi \vdash <\text{id} \rightarrow v] ; <v, s> \) with \( \phi \)
Sequencing

\[ \rho \vdash E_1 : E_2, s > \text{ with } \varphi \rightarrow \rho \varphi (E_1, s > : E_2) \text{ with } \varphi \]

\[ \rho \vdash \alpha \text{ with } \varphi \rightarrow \rho \varphi \alpha' \text{ with } \varphi' \]

\[ \rho \vdash (\alpha ; E) \text{ with } \varphi \rightarrow \rho \varphi (\alpha' ; E) \text{ with } \varphi' \]

\[ \rho \vdash (\langle v, s > ; E \rangle) \text{ with } \varphi \rightarrow \rho \varphi \langle , s > \text{ with } \varphi \]

Assignments

\[ \rho \vdash x := E, s > \text{ with } \varphi \rightarrow \rho \varphi x := E \text{ with } \varphi \]

\[ \rho \vdash \alpha \text{ with } \varphi \rightarrow \rho \varphi \alpha' \text{ with } \varphi' \]

\[ \rho \vdash (\langle x := \alpha \rangle) \text{ with } \varphi \rightarrow \rho \varphi (\langle x := \alpha' \rangle) \text{ with } \varphi' \]

\[ \rho \vdash (\langle x := \langle v, s > \rangle \rangle) \text{ with } \varphi \rightarrow \rho \varphi \langle , s > \| \text{ with } \varphi \]

Waiting

\[ \rho \vdash \text{wait } E, s > \text{ with } \varphi \rightarrow \rho \varphi \text{wait } E \text{ with } \varphi \]

\[ \rho \vdash \alpha \text{ with } \varphi \rightarrow \rho \varphi \alpha' \text{ with } \varphi' \]

\[ \rho \vdash (\text{wait } \alpha) \text{ with } \varphi \rightarrow \rho \varphi (\text{wait } \alpha') \text{ with } \varphi' \]

\[ \rho \vdash \text{wait } (d + d'), s > \text{ with } \varphi \rightarrow \rho \varphi \text{wait } E \text{ with } \varphi \]

when \( d > 0 \)

\[ \rho \vdash \text{wait } \langle 0 \rangle, s > \text{ with } \varphi \rightarrow \rho \varphi \langle 0 \rangle, s > \text{ with } \varphi \]

Input

\[ \rho \vdash \text{let } t = c ? x \text{ in } E, s > \text{ with } \varphi \rightarrow \rho \varphi \text{let } t = c ? x \text{ in } E \text{ with } \varphi \]

\[ \rho \vdash \text{let } t = c ? x \text{ in } E, s > \text{ with } \varphi \rightarrow \rho \varphi \text{let } t = c ? x \text{ in } E \text{ with } \varphi \]

Output

\[ \rho \vdash \text{let } t = c ! E \text{ in } E, s > \text{ with } \varphi \rightarrow \rho \varphi \text{let } t = c ! E \text{ in } E \text{ with } \varphi \]

\[ \rho \vdash \alpha \text{ with } \varphi \rightarrow \rho \varphi \alpha' \text{ with } \varphi' \]

\[ \rho \vdash \text{let } t = c ! \alpha \text{ in } E \text{ with } \varphi \rightarrow \rho \varphi \text{let } t = c ! \alpha' \text{ in } E \text{ with } \varphi' \]

\[ \rho \vdash \text{let } t = c ! \langle v, s > \rangle \text{ in } E \text{ with } \varphi \rightarrow \rho \varphi \text{let } t = c ! \langle v, s > \rangle \text{ in } E \text{ with } \varphi \]

\[ \rho \vdash \text{let } t = c ! x \text{ in } \langle v, s > \rangle \text{ in } E \text{ with } \varphi \rightarrow \rho \varphi \text{let } t = c ! x \text{ in } \langle v, s > \rangle \text{ in } E \text{ with } \varphi \]
Internal Choice

\[ \rho \vdash (\alpha \cap \beta) \text{ with } \phi \quad \xrightarrow{\downarrow} \alpha \text{ with } \phi \]

\[ \xrightarrow{\uparrow} \beta \text{ with } \phi \]

External Choice

\[ \rho \vdash \alpha \text{ with } \phi \quad \xrightarrow{\alpha'} \alpha' \text{ with } \phi' \]

\[ \rho \vdash (\beta \cap \alpha') \text{ with } \phi' \]

\[ \beta \text{ with } \phi' \quad \xrightarrow{\beta'} \beta' \text{ with } \phi'' \]

\[ \rho \vdash \alpha \text{ with } \phi \quad \xrightarrow{\epsilon(d)} \alpha' \text{ with } \phi' \]

\[ \rho \vdash \beta \text{ with } \phi'' \quad \xrightarrow{\beta''} \beta' \text{ with } \phi'' \]

\[ \rho \vdash \alpha \text{ with } \phi \quad \xrightarrow{\epsilon(d)} \alpha' \text{ with } \phi' \]

\[ \rho \vdash \beta \text{ with } \phi'' \quad \xrightarrow{\beta''} \beta' \text{ with } \phi'' \]

Parallel Combinator

\[ \rho \vdash <E_1, E_2, s \succ \text{ with } \phi \]

\[ <E_1, s \ \text{var}(E_2) \succ \text{ with } \phi \ | \ s \ \text{var}(E_1) \cup \text{var}(E_2) \succ <E_2, s \ \text{var}(E_1) > \text{ with } \phi \]

\[ \rho \vdash \alpha \text{ with } \phi \quad \xrightarrow{\alpha'} \alpha' \text{ with } \phi' \]

\[ \beta \text{ with } \phi' \quad \xrightarrow{\beta'} \beta' \text{ with } \phi'' \]

\[ \rho \vdash \alpha \text{ with } \phi \quad \xrightarrow{\epsilon(d)} \alpha' \text{ with } \phi' \]

\[ \beta \text{ with } \phi'' \quad \xrightarrow{\epsilon(d)} \beta' \text{ with } \phi'' \]

\[ \rho \vdash \alpha \text{ with } \phi \quad \xrightarrow{\epsilon(d)} \alpha' \text{ with } \phi' \]

\[ \beta \text{ with } \phi'' \quad \xrightarrow{\epsilon(d)} \beta' \text{ with } \phi'' \]

\[ \rho \vdash \alpha \text{ with } \phi \quad \xrightarrow{\epsilon(d)} \alpha' \text{ with } \phi' \]

\[ \beta \text{ with } \phi'' \quad \xrightarrow{\epsilon(d)} \beta' \text{ with } \phi'' \]

\[ \rho \vdash \alpha \text{ with } \phi \quad \xrightarrow{\epsilon(d)} \alpha' \text{ with } \phi' \]

\[ \beta \text{ with } \phi'' \quad \xrightarrow{\epsilon(d)} \beta' \text{ with } \phi'' \]

\[ \text{when} \]

\[ \begin{align*}
\text{Sort}_2(\alpha) \cap \text{Sort}_2(\beta) &= \emptyset; \\
\text{Sort}_2(\alpha) \cap \text{SORT}_2 &= \emptyset; \\
\text{Sort}_2(\beta) \cap \text{SORT}_2 &= \emptyset.
\end{align*} \]

\[ \rho \vdash \alpha \text{ with } \phi \quad \xrightarrow{\epsilon(d)} \alpha' \text{ with } \phi' \]

\[ \beta \text{ with } \phi'' \quad \xrightarrow{\epsilon(d)} \beta' \text{ with } \phi'' \]

\[ \rho \vdash \alpha \text{ with } \phi \quad \xrightarrow{\epsilon(d)} \alpha' \text{ with } \phi' \]

\[ \beta \text{ with } \phi'' \quad \xrightarrow{\epsilon(d)} \beta' \text{ with } \phi'' \]

\[ \rho \vdash \alpha \text{ with } \phi \quad \xrightarrow{\epsilon(d)} \alpha' \text{ with } \phi' \]

\[ \beta \text{ with } \phi'' \quad \xrightarrow{\epsilon(d)} \beta' \text{ with } \phi'' \]

\[ \rho \vdash \alpha \text{ with } \phi \quad \xrightarrow{\epsilon(d)} \alpha' \text{ with } \phi' \]

\[ \beta \text{ with } \phi'' \quad \xrightarrow{\epsilon(d)} \beta' \text{ with } \phi'' \]
\[\rho \vdash \alpha \text{ with } \phi \vdash \alpha' \text{ with } \phi'\]

\[\rho \vdash \alpha \text{ with } s \vdash s' \text{ with } \phi' \vdash \alpha' \text{ with } \phi'\]

\[\rho \vdash <v, s'' > \text{ with } \phi \vdash s \vdash s' \text{ with } \phi' \vdash \alpha' \text{ with } \phi'\]

\[\rho \vdash <v, s'' > \text{ with } \phi \vdash s \vdash s' \text{ with } \phi' \vdash \alpha' \text{ with } \phi'\]

\[\rho \vdash <\lambda id : \tau \bullet E, s > \text{ with } \phi \vdash \lambda id : \tau \bullet E, s > \text{ with } \phi\]

\[\rho \vdash < [\lambda id : \tau \bullet E_1, \rho_1 ], s > \text{ with } \phi \vdash [\lambda id : \tau \bullet E_1, \rho_1 ] < E_2, s > \text{ with } \phi\]
42 A.E. Haxthausen and X. Yong

\[ \rho \vdash \alpha \text{ with } \phi \xrightarrow{\gamma} \alpha' \text{ with } \phi' \]

\[ \rho \vdash ([\lambda \text{id} : \tau \bullet E, \rho_1] \alpha) \text{ with } \phi \xrightarrow{\gamma} ([\lambda \text{id} : \tau \bullet E, \rho_1] \alpha') \text{ with } \phi' \]

\[ \rho \vdash ([\lambda \text{id} : \tau \bullet E, \rho_1] \text{ < } v, s \text{ >}) \text{ with } \phi \xrightarrow{\gamma} ([\lambda \text{id} : \tau \bullet E, \rho_1] \text{ < } v', s \text{ >}) \text{ with } \phi' \]

**Let Expressions**

\[ \rho \vdash \text{let id = } E_1 \text{ in } E_2, s \text{ >} \text{ with } \phi \xrightarrow{\gamma} \text{ (let id = } E_1, s \text{ > in } E_2) \text{ with } \phi \]

\[ \rho \vdash \alpha \text{ with } \phi \xrightarrow{\gamma} \alpha' \text{ with } \phi' \]

\[ \rho \vdash \text{let id = } \alpha \text{ in } E \text{ with } \phi \xrightarrow{\gamma} \text{ (let id = } \alpha', \text{ in } E \text{ ) with } \phi' \]

\[ \rho \vdash \text{let id = } v, s \text{ > in } E \text{ with } \phi \xrightarrow{\gamma} E[v/\text{id}], s \text{ > with } \phi \]

**If Expressions**

\[ \rho \vdash \text{if } E \text{ then } E_1 \text{ else } E_2, s \text{ > with } \phi \xrightarrow{\gamma} \text{ (if } E, s \text{ > then } E_1 \text{ else } E_2) \text{ with } \phi \]

\[ \rho \vdash \alpha \text{ with } \phi \xrightarrow{\gamma} \alpha' \text{ with } \phi' \]

\[ \rho \vdash \text{if } \alpha \text{ then } E_1 \text{ else } E_2 \text{ with } \phi \xrightarrow{\gamma} \text{ (if } \alpha', \text{ then } E_1 \text{ else } E_2 \text{ ) with } \phi' \]

\[ \rho \vdash \text{if } \text{true, } s \text{ > then } E_1 \text{ else } E_2 \text{ with } \phi \xrightarrow{\gamma} \text{ (true, } s \text{ > then } E_1 \text{ else } E_2 \text{ ) with } \phi \]

**While Expressions**

\[ \rho \vdash \text{while } E_1 \text{ do } E_2, s \text{ > with } \phi \xrightarrow{\gamma} \text{ (if } E_1, s \text{ > then } E_2, \text{ while } E_1 \text{ do } E_2 \text{ else skip) with } \phi \]

**B Proof Rules**

**Some Shorthands**

\[ (\phi)^{\omega} \doteq \bigvee_{j \in \text{Nat}} \phi^j \]

\[ \phi^{n} \doteq (\phi \circ \ldots \circ \phi), \text{ for } n \geq 1 \]
Rules for \( H \)

\[
\begin{align*}
H_S(x := v) & \rightarrow \text{assign}_{\Sigma}([x \mapsto v]) & H_S(\text{wait } d) & \rightarrow \text{wait}_{\Sigma}(d) \\
H_S(c ? x) & \rightarrow \text{input}_{\Sigma}(c, x) & H_S(c ! v) & \rightarrow \text{output}_{\Sigma}(c, v) \\
H_S(E_1) & \rightarrow \phi_1, H_S(E_2) \rightarrow \phi_2 & H_S(E) & \rightarrow \phi \\
\implies H_S(E_1; E_2) & \rightarrow \phi_1 \circ \phi_2 & H_S(\text{while true do } E) & \rightarrow \phi^2 \\
H_S(E_1) & \rightarrow \phi_1, H_S(E_2) \rightarrow \phi_2 & & \\
H_S(E_1 \parallel E_2) & \rightarrow \text{OR}_{\Sigma}(\phi_1, \phi_2), H_S(E_1 \| E_2) & \rightarrow \phi_1 \lor \phi_2 & \\
H_S(E_i) & \rightarrow \phi_i, i \in \{1, ..., n\}, \Sigma_i = \Sigma \setminus \bigcup_{j \in \{1, ..., n\} \setminus \{i\}} \text{var}(E_i) & H_S(E_1 \parallel (E_2 \| ... \| E_n)) & \rightarrow \text{IN}_{\Sigma}(\phi_1, ..., \phi_n)
\end{align*}
\]

Rules for Various Functions \(^4\)

1. \( \text{assign}_{\Sigma}(s_1) \rightarrow \exists s : \Sigma \cdot (\Sigma_s \land s \uparrow (s \uparrow s_1)) \)
2. \( \text{input}_{\Sigma}(c, x) \rightarrow [c?]^* \land \exists s : \Sigma, v \cdot ([s]^* \land \Sigma_s \land \not\mathcal{R}(s \uparrow [x \mapsto v])) \)
3. \( \text{output}_{\Sigma}(c, v) \rightarrow [c]^* \land \exists s : \Sigma \cdot ([s]^* \land \Sigma_s \land \not\mathcal{R} s) \)
4. \( \text{wait}_{\Sigma}(d) \rightarrow \ell = d \land \exists s : \Sigma \cdot ([s]^* \land \Sigma_s \land \not\mathcal{R} s) \)
5. \( \text{waitinput}_{\Sigma}(c) \rightarrow [c?]^* \land \exists s : \Sigma \cdot ([s]^* \land \Sigma_s \land \not\mathcal{R} s) \)
6. \( \text{waitoutput}_{\Sigma}(c) \rightarrow [c]^* \land \exists s : \Sigma \cdot ([s]^* \land \Sigma_s \land \not\mathcal{R} s) \)

Rules for \( \circ \)

“\( \phi \circ \phi' \)” describes a common history for a sequential composition (taking into account that the first expression may not terminate).

1. \( \text{input}_{\Sigma}(c, x) \circ \phi \rightarrow \text{input}_{\Sigma}(c, x) \bullet \phi \lor \text{waitinput}_{\Sigma}(c) \)
2. \( \text{output}_{\Sigma}(c, v) \circ \phi \rightarrow \text{output}_{\Sigma}(c, v) \bullet \phi \lor \text{waitoutput}_{\Sigma}(c) \)
3. \( \text{waitinput}_{\Sigma}(c) \circ \phi \rightarrow \text{waitinput}_{\Sigma}(c) \)
4. \( \text{waitoutput}_{\Sigma}(c) \circ \phi \rightarrow \text{waitoutput}_{\Sigma}(c) \)
5. \( \text{wait}_{\Sigma}(d) \circ \phi \rightarrow \text{wait}_{\Sigma}(d) \bullet \phi \)
6. \( (\phi_1 \lor \phi_2) \circ \phi \rightarrow (\phi_1 \circ \phi) \lor (\phi_2 \circ \phi) \)
7. \( \phi_1 \circ \phi_2 \rightarrow \phi_1 \bullet \phi_2, \text{ when } \phi_1 \Rightarrow (\ell = 0) \)

Rules for \( \text{OR} \)

“\( \text{OR}_{\Sigma}(\phi_1, \phi_2) \)” describes a common history of an internal choice.

1. \( \phi \circ \text{OR}_{\Sigma}(\phi_1, \phi_2) \rightarrow \text{OR}_{\Sigma}(\phi \circ \phi_1, \phi \circ \phi_2) \)

Rules for \( \text{IN} \)

“\( \text{IN}_{\Sigma}(\phi_1, \phi_2, ..., \phi_n) \)” describes a common history of a collection of parallel sequential processes interlocked with its complete environment.

1. \( \text{IN}_{\Sigma} \left( \text{OR}_{\Sigma}(\phi_1, \phi_2), \phi_3, ..., \phi_n \right) \rightarrow \text{IN}_{\Sigma}(\phi_1, \phi_3, ..., \phi_n) \lor \text{IN}_{\Sigma}(\phi_2, \phi_3, ..., \phi_n) \)

---

\(^4\) Note, \( \exists s : \Sigma \cdot \phi[s] \) means that there exists a \( \Sigma \)-store \( s \) such that \( \phi[s] \) is true.
2. $IN_{\Sigma}(\phi_1, ..., \phi_n) \rightarrow^5 \bigvee_{i,j \in \{1, ..., n\}, i \neq j} \phi_i \cdot IN_{\Sigma}(\phi_1, ..., \phi_i', ..., \phi_j', ..., \phi_n)$

\[ IN_{\Sigma}(\phi_i, \phi_j) \equiv COMM \phi_0 \cdot IN_{\Sigma}(\phi_i', \phi_j') \]

when there exists $i,j \in \{1, ..., n\}$ with $i \neq j$ for which

\[ IN_{\Sigma}(\phi_i, \phi_j) \equiv COMM \phi_0 \cdot IN_{\Sigma}(\phi_i', \phi_j') \]

for some $\phi_0, \phi_i$, and $\phi_j$.

3. $IN_{\Sigma}(wait_{\Sigma}(\delta) \bullet \phi_1 \vee \theta_{12} \vee ... \vee \theta_{1m_1}, \theta_{21} \vee ... \vee \theta_{2m_2}, ... \theta_{n1} \vee ... \vee \theta_{nm_n}) \rightarrow^6 (wait_{\Sigma}(\delta) \land (\theta_{12} \vee ... \vee \theta_{1m_1}) \land (\theta_{21} \vee ... \vee \theta_{2m_2}) \land ... \land (\theta_{n1} \vee ... \vee \theta_{nm_n}))$

\[ IN_{\Sigma}(\phi_i \vee \theta_{12} \vee ... \vee \theta_{1m_1}, \theta_{21} \vee ... \vee \theta_{2m_2}, ... \theta_{n1} \vee ... \vee \theta_{nm_n}) \]

when:

a) for all $(i,j) \in \{1, ..., n\} \times \{1, ..., m_i\}$ with $(i,j) \neq (1,1)$,

\[ IN_{\Sigma}(wait_{\Sigma}(\delta) \bullet \phi_i, \theta_{ij}) \rightarrow_{WAIT} (wait_{\Sigma}(\delta) \land \theta_{ij}) \bullet IN_{\Sigma}(\phi_i, \theta_{ij}'), \] for some $\phi_i$ and $\theta_{ij}';$

b) there exists no $(h,i), (j,k) \in \{1, ..., n\} \times \{1, ..., n\}$ with $h \neq j$, for which

\[ IN_{\Sigma}(\theta_{hi}, \theta_{jk}) \equiv COMM \theta_0 \bullet IN_{\Sigma}(\theta_{hi}', \theta_{jk}') \] for some $\theta_0, \theta_{hi}', \theta_{jk}'.$

4. $IN_{\Sigma}(assign_{\Sigma_1}(s) \bullet \phi_1, \phi_2, ..., \phi_n) \rightarrow assign_{\Sigma}(s) \bullet IN_{\Sigma}(\phi_1, \phi_2, ..., \phi_n)$

5. $IN_{\Sigma}(\perp, \phi_2, ..., \phi_n) \rightarrow IN_{\Sigma}(\phi_2, ..., \phi_n)$

6. $IN_{\Sigma}(\phi) \rightarrow \phi$

7. $IN_{\Sigma}(\phi_1, \phi_2, ..., \phi_j, ...) \rightarrow IN_{\Sigma}(\phi_1, \phi_2, ..., \phi_i, ...)$

8. ...

Rules for $\equiv COMM$

"$IN_{\Sigma}(\psi_1, \psi_2) \equiv COMM \phi \bullet IN_{\Sigma}(\phi_1, \phi_2)$" holds if two concurrent components (described by $\psi_1$ and $\psi_2$) are able to make a communication (described by $\phi$) leading to two new concurrent components (described by $\phi_1$ and $\phi_2$).

1. $IN_{\Sigma}(input_{\Sigma_1}(c, x) \vee \phi_1) \circ \phi_1, (output_{\Sigma_2}(c, v) \vee \phi_2) \circ \phi_2) \equiv COMM assign_{\Sigma}(x \mapsto v) \bullet IN(\phi_1, \phi_2)$

2. ...

Rules for $\rightarrow_{WAIT}$

"$IN_{\Sigma}(\psi_1, \psi_2) \rightarrow_{WAIT} \phi \bullet IN_{\Sigma}(\phi_1, \phi_2)$" holds if two concurrent components (described by $\psi_1$ and $\psi_2$) evolve a time measurable event (described by $\phi$) leading to two new concurrent components (described by $\phi_1$ and $\phi_2$).

1. $IN_{\Sigma}(wait_{\Sigma_1}(\delta) \bullet \phi_1, input_{\Sigma_2}(c, x) \circ \phi_2) \rightarrow_{WAIT} (wait_{\Sigma_1}(\delta) \land [c?] \bullet IN(\phi_1, input_{\Sigma_2}(c, x) \circ \phi_2)$

2. $IN_{\Sigma}(wait_{\Sigma_1}(\delta) \bullet \phi_1, output_{\Sigma_2}(c, v) \circ \phi_2) \rightarrow_{WAIT} (wait_{\Sigma_1}(\delta) \land [c!] \bullet IN(\phi_1, output_{\Sigma_2}(c, v) \circ \phi_2)$

3. $IN_{\Sigma}(wait_{\Sigma_1}(\delta_1) \bullet \phi_1, wait_{\Sigma_2}(\delta_2 + \delta_3) \circ \phi_2) \rightarrow_{WAIT} wait_{\Sigma_1}(\delta_1) \bullet IN(\phi_1, wait_{\Sigma_2}(\delta_2) \circ \phi_2)$

5 This rule expands the common history of a concurrent composition as a disjunction of its possible communication behaviors, when an initial communication is possible.

6 This rule expands the common history of a concurrent composition in which all components can wait for $\delta$ time units and no initial communication is possible.
Formalizing Timing Diagrams as Causal Dependencies for Verification Purposes

Jörg Fischer\textsuperscript{1} and Stefan Conrad\textsuperscript{2}

\textsuperscript{1} Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik
Postfach 4120, D-39016 Magdeburg, Germany
jfischer@iti.cs.uni-magdeburg.de

\textsuperscript{2} Ludwig-Maximilians-Universität München, Institut für Informatik
Oettingenstr. 67, D-80538 München, Germany
conrad@informatik.uni-muenchen.de

Abstract. In this paper we investigate timing diagrams as a means to specify causal dependencies. We introduce a stylized graphical representation of timing diagrams for which we define a formal basis. Furthermore, we compare our approach with well-known approaches from the area of program verification and show the semantic relationships. The major aim we follow by this work is a seamless integration of hardware design and software development providing a common semantic basis e.g. for verification. Therefore, the semantic relationships to frameworks for program verification show that the combination of these approaches is a good starting point for further development.

Keywords: causal dependencies, timing diagrams, hardware and software design, formal semantics, integrated verification, relational semantics, dynamic logic.

1 Introduction

The development of rigorous and formal methods is marked by the fact that nearly each area started to develop its own methods from scratch. Thereby, we are dealing with a bunch of different formal methods mostly developed for treating specific aspects of systems, e.g. for programming, databases, telecommunication equipment, and hardware devices. Not only the large number of these approaches and their variants but also the fact that systems do not just consist of single “aspects” cause the necessity to integrate approaches for different aspects in order to offer a more homogeneous way of supporting the design and development of systems, in particular when hardware and software have to be considered together at the same time.

The design of embedded systems, telecommunication systems and other computer controlled devices can roughly be divided into two major parts: hardware design and software development. Both areas have their own peculiarities. Hardware devices, e.g. integrated circuits, are mostly non-sequential and have to meet

\footnote{Supported by the DFG under project no. NE 592/4-2.}

© Springer-Verlag Berlin Heidelberg 2000
real-time constraints and time dependencies like synchronization. Special issues considered in software development are termination of programs, usage of recursion etc. The investigations of common theoretical foundations is a must for overcoming the above-mentioned heterogeneity because all aspects of engineering base on the same concepts: on processes, on states, on data etc.

Timing diagrams form a simple mechanism which can be used in hardware design. For example, specification of integrated circuits (ICs) is commonly based on timing diagrams so that electronic engineers are supposed to be familiar with this specification technique. The next section informally introduces the concept of causal dependencies in order to stylize timing diagrams and to complete their expressibility for the use in specification. A graphical notation will be presented for the specification with causal dependencies which is easy to interpret. Section 3 presents a more precise introduction to the principles used herein by defining syntax and semantics of that graphical notation. The main goal of this paper is tackled in Section 4. In this section the relationship of our specification technique using causal dependencies and widely spread program verification techniques is investigated. We conclude by briefly discussing further related work and pointing out future work.

2 Timing Diagrams and Causal Dependencies

Timing diagrams are diagrams like the one in Fig. 1 depicting a serial 5 bit character transmission. A low level bit which is called start bit announces a character. Then, the five data bits follow, and a high level bit called the stop bit closes the transmission.

![Fig. 1. Asynchronous serial character transmission — 5 data bits, one stop bit.](image)

However, not only the behavior on one wire is interesting but also the temporal relationship between several wires, i.e. synchronization. Fig. 2 shows how to express this by means of a dashed line. The left picture could be part of a diagram contained in a technical specification of an integrated circuit and the right picture is the stylized form of the left picture. The intuitive meaning of these pictures could be the following one: The level on the lower wire switches from high level to low level and then a data transmission on the upper wire begins. The dashed line equates time points on several processes, thus expressing synchronization.

Pictures like those of Fig. 2 can express scenarios. But this is usually not sufficient for specification purposes. Specifications have to express e.g. that data
transmission \textit{requires} the falling level on the lower wire. This is a logical consequence. In order to emphasize this relationship we will frame the premises as shown in Fig. 3. These logical consequences are called “causal dependencies” in the sequel.

There are two kinds of causal dependencies: causal dependencies of type \textit{requires} and of type \textit{causes}. Fig. 4 contains an example for each of these kinds.

3 Syntax and Semantics of Causal Dependencies

3.1 Signatures and Models

Firstly, we fix ground sorts and methods. Intuitively, methods are actions representing time intervals, i.e. which can begin and can end. The term “method” can be used, e.g. in order to formalize database transactions or computer programs.

\textbf{Definition 1 (Signature.)}. A signature is a pair $\Sigma \coloneqq (S_\Sigma, \Omega_\Sigma)$ where $S_\Sigma$ is a set of ground sorts and $\Omega_\Sigma \coloneqq (\Omega_{\Sigma,u,v} \mid u, v \in S_\Sigma^*)$ is a family of methods.
Herein and hereafter, \( S^* \) refers to the free monoid over any set \( S \), i.e. the set of sequences \( X_1 \ldots X_m \) with \( m \geq 0 \) and \( X_i \in S \) for each \( i \in \{1, \ldots, m\} \). These sequences of ground sorts are called \text{sorts}.

Methods \( R \in \Omega_{\Sigma,u_1 \ldots u_m,v_1 \ldots v_n} \) where \( u_i \in S^*_\Sigma \) and \( v_j \in S^*_\Sigma \) for all \( i \in \{1, \ldots, m\} \) and \( j \in \{1, \ldots, n\} \) are presented by boxes as shown in Fig. 5. The left-hand side of a box represents the time before execution of the action corresponding to the method and the right-hand side of a box represents the time after executing this action. Please note that \( u_i \in S^\Sigma \) and \( v_j \in S^\Sigma \) were never demanded, i.e. arrow bundles can be presented by one arrow as shown in the picture on the right-hand side of Fig. 5.

The semantic representations of methods are binary relations.

**Definition 2 (Structure.).** Given a signature \( \Sigma \). A structure w.r.t. this signature or briefly a \( \Sigma \)-structure \( \mathcal{M} \) is a map assigning a set \( X^\mathcal{M} \) of states for each ground sort \( X \in S^\Sigma \), which is freely continued to sorts so that \( u^\mathcal{M} \overset{\text{def}}{=} X^\mathcal{M}_1 \times \ldots \times X^\mathcal{M}_m \), for each sort \( u \overset{\text{def}}{=} X_1 \ldots X_m \) and assigning a binary relation \( R^\mathcal{M} \in \mathcal{P}(u^\mathcal{M} \times v^\mathcal{M}) \) of actions for each method \( R \in \Omega_{\Sigma,u,v} \).

Hereafter, the following notation will be used for relations of actions:

\[
x \xrightarrow{R} y \overset{\text{def}}{=} (x,y) \in R
\]
$x \xrightarrow{R} y$ states that the state $y$ is reached by action $R$ from $x$ as well as in many other approaches to defining processes like transition systems [23]. However, there is a subtle difference to usual interpretation of that relation in order to more easily deal with parallelism. Suppose as an example two actions ON and OFF, respective states and a binary relation shown in Fig. 6. Herein, its interpretation is a process which creates two parallel successor processors one by action ON and the other by action OFF in each state. In the most other process definition techniques like transition systems the branchings would express non-deterministic choice rather than parallelism.

\begin{align*}
  x \xrightarrow{id} y & \iff x = y \\
  x \xrightarrow{R,S} y & \iff \exists y. \left( x \xrightarrow{R} y \land y \xrightarrow{S} z \right) \\
  (x_i | i \in I) \underbrace{\otimes_{R_i}} & \iff (y_i | i \in I) \iff \forall i \in I. x \xrightarrow{R_i} y_i \\
  x \rightarrow_{\bigwedge_{R_i}} y & \iff \forall i \in I. x \xrightarrow{R_i} y
\end{align*}

The shadowed boxes in Table 1 represent not only methods but also terms so that this is a recursive definition.

The constructions introduced in Table 2 and Table 3 are defined as macros. In the following definitions $\text{tt}_{u,v}$ is the par-operation with $m = 0$, cf. Table 1.

\begin{align*}
  l\{k\}_{(v_i | i \in \{1, \ldots, m\})} & \overset{\text{def}}{=} \prod_{i=1}^{m} \begin{cases}
    \text{id}_{u_k} & i = k \\
    \text{tt}_{(\_),v_i} & i \neq k
  \end{cases} \\
  r\{k\}_{(u_i | i \in \{1, \ldots, m\})} & \overset{\text{def}}{=} \prod_{i=1}^{m} \begin{cases}
    \text{id}_{u_k} & i = k \\
    \text{tt}_{u_i,(\_)} & i \neq k
  \end{cases}
\end{align*}

---

Fig. 6. Infinitary Binary Tree.

### 3.2 Terms

Methods can be composed by the means shown in Table 1 to terms whose interpretation in a structure $M$ is defined as follows: The superscript $\_^M$ will be omit herein and hereafter:

\begin{align*}
  x \xrightarrow{id} y & \iff x = y \\
  x \xrightarrow{R,S} y & \iff \exists y. \left( x \xrightarrow{R} y \land y \xrightarrow{S} z \right) \\
  (x_i | i \in I) \underbrace{\otimes_{R_i}} & \iff (y_i | i \in I) \iff \forall i \in I. x \xrightarrow{R_i} y_i \\
  x \rightarrow_{\bigwedge_{R_i}} y & \iff \forall i \in I. x \xrightarrow{R_i} y
\end{align*}

The shadowed boxes in Table 1 represent not only methods but also terms so that this is a recursive definition.

The constructions introduced in Table 2 and Table 3 are defined as macros. In the following definitions $\text{tt}_{u,v}$ is the par-operation with $m = 0$, cf. Table 1.
### Table 1. Operations on Hypergraphs.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>skp</td>
<td>$u \rightarrow u$</td>
</tr>
<tr>
<td>seq</td>
<td>$u \rightarrow r \rightarrow v \rightarrow s \rightarrow w \rightarrow R; S$</td>
</tr>
<tr>
<td>con</td>
<td>$u_1 \rightarrow n_1 \rightarrow v_1$</td>
</tr>
<tr>
<td>par</td>
<td>$u \rightarrow \vdots \rightarrow v \rightarrow \bigwedge_{i \in I} R_i$</td>
</tr>
</tbody>
</table>

Intuitively, terms represent scenarios. The skp-operation in Table 1 refers to an operation which does nothing. The sequential composition seq of terms $R$ and $S$ correlates to the action where $S$ begins as soon as $R$ ends. Independent partial scenarios are represented by the con-operation; and parallel composition of methods $R$ and $S$ whose corresponding action is the simultaneous execution of $R$ and $S$ is presented as par-operation in Table 1. Fig. 7 shows an example of a term. The frames in terms $T_{\Sigma, u, v}$ should cut off the nodes determining the sorts $u$ and $v$. This is necessary, sometimes, because these sorts could be ambiguous.

### 3.3 Propositions

Now, we can start to stipulate requirements of which three kinds can be distinguished. However, the propositions of the first kind, the equations, can be defined via the propositions of the other kinds, which are called causal dependencies.
Table 2. Derived Terms — part i.

\[ u \rightarrow t_k \rightarrow v_k \rightarrow v_{k+1} \rightarrow \cdots \rightarrow v_m \]

\[ u \rightarrow t_k \rightarrow v_k \rightarrow v_{k+1} \rightarrow \cdots \rightarrow v_m \]

\[ \text{def} = f_k \cup t[v_i \mid i \in \{1, \ldots, m\}] \]

\[ \text{def} = p[u_i \mid i \in \{1, \ldots, m\}] \]

Table 3. Derived Terms — part ii.

\[ t_1 \rightarrow v_1 \rightarrow \cdots \rightarrow t_m \rightarrow v_m \]

\[ u_1 \rightarrow t_1 \rightarrow \cdots \rightarrow u_m \rightarrow t_m \]

\[ \text{def} = \Delta_{m,u} ; \bigotimes_{i=1}^{m} t_i \]

\[ \text{def} = \bigotimes_{i=1}^{m} t_i ; \bigcirc_{m,v} \]

Fig. 7. Producing an Electronic Device.
As mentioned before, causal dependencies can be of two types (cf. Fig. 8 and Fig. 9).

\[
\begin{align*}
\forall x, y. \exists z. x & \xrightarrow{R} y \\
& \xrightarrow{S} z
\end{align*}
\]

Fig. 8. Causal Dependency of type causes.

\[
\begin{align*}
\forall y, z. \exists x. y & \xrightarrow{S} z \\
x & \xrightarrow{R} y
\end{align*}
\]

Fig. 9. Causal Dependency of type requires.

4 Relationship to Program Verification Systems

In this section we show that the semantic foundations of well-known program verification techniques as well as the relational semantics given by the structure concept from Definition 2 are equivalent.

The representation of timing diagrams as causal dependencies benefits from this strong relationship to program verification techniques. This correspondence helps to overcome the heterogeneity between representations via timing diagrams and program verification techniques.

4.1 Axiomatic form of Hoare Triple Logics

Hoare triples like \{X\} R \{Y\} express partial correctness of programs R under the precondition X and postcondition Y. Regarding R as method of the approach in this paper, partial correctness is given by the following statement:

\[
\{X\} R \{Y\} \overset{\text{def}}{=} \forall x, y. x \in X \quad x \xrightarrow{R} y \\
y \in Y
\]

i. e. whenever the current program state fulfills the precondition X and the program R terminates then a state is reached fulfilling postcondition Y. Partial correctness does not require termination of the program.
Formalizing Timing Diagrams as Causal Dependencies 53

The new structure concept. Below, another equivalent form of Definition 2 is presented which bases on an axiomatization of partial correctness. The notions “Galois connection”, “CABA” and “tensor” will be explained subsequently.

Definition 3 (Structure.). Given a signature Σ. A structure w. r. t. Σ or, briefly a Σ-structure M is a map

- assigning a complete atomic Boolean algebra (CABA) \( X^{M} \) to each ground sort \( X \in S_{\Sigma} \)
- freely continued to a map assigning \( u^{M} \text{ def } = X_{1}^{M} \otimes \ldots \otimes X_{m}^{M} \) to each sort \( u \text{ def } = X_{1} \ldots X_{m} \)
- and assigning a Galois connection \( R^{M} \) between \( u^{M} \) and \( v^{M} \) for each method \( R \in \Omega_{\Sigma,u,v} \)

State conditions. The preconditions and postconditions will be formalized as usual by Boolean algebras. Boolean algebras which are isomorphic to a power set algebra \( \mathcal{P}(A) \) with a set \( A \) are called complete atomic Boolean algebra (CABA).
All finite Boolean algebras are by virtue of Stones representation lemma \([25]\) complete atomic Boolean algebras. The singletons \( \{a\} \) with \( a \in A \) are called atoms. Subsequently the attention is restricted to power set algebras.

The elements \( a \in A \) represent the states which correspond to the states of the structure concept according Definition 2. \( a \in X \) with \( X \in \mathcal{P}(A) \) expresses that the state \( a \) obeys the condition \( X \).

The tensor of a family

\[
\left( B_{i} \text{ def } = \mathcal{P}(A_{i}) \mid i \in I \right)
\]

of power set algebras is the following power set algebra:

\[
\bigotimes_{i \in I} B_{i} \text{ def } = \mathcal{P} \left( \prod_{i \in I} A_{i} \right)
\]

Galois connections. Hoare triples are axiomatized by Galois connections between complete atomic Boolean algebras. This subsection introduces the concepts for the more general case of complete lattices. \([1]\) introduces the most general form of the concept of Galois connection.

Given two complete lattices \( \mathcal{U} \) and \( \mathcal{V} \). In this context a Galois correspondence is a relation \( R \subseteq \mathcal{U} \times \mathcal{V} \) fulfilling the following axioms where \( \subseteq \) denotes the lattice order, \( \cap \) denotes the infimum operation and \( \cup \) denotes the supremum operation.

\( \{X\} \ R \ \{Y\} \) states \( (X,Y) \in R \):

\[
\forall X, X', Y, Y'. \ X' \subseteq X \ \{X\} \ R \ \{Y\} \ \ Y \subseteq Y' \\
\forall X, Y. \ \forall X \in \mathcal{X}. \ \{X\} \ R \ \{Y\} \\
\forall X, \mathcal{X}. \ \forall \mathcal{X} \in \mathcal{P}(X). \ \{\mathcal{X}\} \ R \ \{\emptyset\}
\]

\[
\forall X, \mathcal{X}. \ \forall \mathcal{Y}. \ \forall \mathcal{Y} \in \mathcal{P}(X). \ \{X\} \ R \ \{\emptyset\}
\]
The terminus of Galois connection between complete lattices could be defined also in two other ways. There are three forms of a Galois connection $R$ between complete lattices $U$ and $V$:

- the already mentioned relation $R$,
- infimums (\(=\ \cap\)-operation) preserving functions $R^\cap$ between $V$ and $U$, i.e. functions fulfilling for each $\mathcal{Y} \subseteq Y$ the equation $R^\cap(\cap \mathcal{Y}) = \cap \{R^\cap(Y) \mid Y \in \mathcal{Y}\}$,
- supremaums (\(=\ \cup\)-operation) preserving functions $R^\cup$ between $U$ and $V$, i.e. functions fulfilling for each $\mathcal{X} \subseteq X$ the equation $R^\cup(\cup \mathcal{X}) = \cup \{R^\cup(X) \mid X \in \mathcal{X}\}$.

The latter functions can be obtained by the equations:

\[
R^\cap(Y) \overset{\text{def}}{=} \bigcup \{X \mid \{X\} R \{Y\}\} \quad (2)
\]

\[
R^\cup(X) \overset{\text{def}}{=} \bigcap \{Y \mid \{X\} R \{Y\}\} \quad (3)
\]

The following equation describes how to obtain the relation from the functional forms of Galois connection:

\[
\{X\} R \{Y\} \overset{\text{def}}{\Longleftrightarrow} R^\cup(X) \subseteq Y \quad (4)
\]

\[
\overset{\text{def}}{\Longleftrightarrow} X \subseteq R^\cap(Y) \quad (5)
\]

**Lemma 1.** (2) and (5) as well as (3) and (4) are bijections.

**The equivalence of the structure concepts.** A structure according Definition 2 can be obtained by this equation for all methods $R$:

\[
x R y \overset{\text{def}}{\Longleftrightarrow} y \in R^\cup(\{x\}) \quad (6)
\]

A structure according Definition 3 can be obtained from a structure according Definition 2 by this equation:

\[
R^\cup(X) = \left\{y \mid \exists x \in X. x R y \right\} \quad (7)
\]

**Lemma 2.** (7) defines a supremums preserving function.

**Lemma 3.** (7) is equivalent to (1)

**Lemma 4.** (1) and (6) form a bijection.
Now, for all methods and terms there are four forms to define their semantics. The four forms for the identity are

\[
\begin{align*}
\{X\} \text{id}_u \{Y\} & \iff X \subseteq Y \\
(id_u)^\cap (Y) & = Y \\
(id_u)^\cup (X) & = X \\
x \overset{id_u}{\to} y & \iff x = y
\end{align*}
\]

The sequential composition can be described as follows in all four forms:

\[
\begin{align*}
\{X\} R; S \{Z\} & \iff \exists Y. (\{X\} R \{Y\} \land \{Y\} S \{Z\}) \\
(R; S)^\cap (Z) & = R^\cap (S^\cap (Z)) \\
(R; S)^\cup (X) & = S^\cup (R^\cup (X)) \\
x R; S \to z & \iff \exists y. (x R \to y \land y S \to z)
\end{align*}
\]

The equation proposition between terms can be defined in these four forms:

\[
\begin{align*}
R \subseteq S & \iff \forall X, Y. \frac{\{X\} S \{Y\}}{\{X\} R \{Y\}} \\
& \iff \forall X. (R^\cup (X) \subseteq S^\cup (X)) \\
& \iff \forall Y. (S^\cap (Y) \subseteq R^\cap (Y)) \\
& \iff \forall x, y. \frac{x R \to y}{x S \to y}
\end{align*}
\]

4.2 Dijkstra’s wlp Calculus [5]

For a Galois connection \( R \) representing a program the infimum preserving function \( R^\cap \) is the wlp operator which is used as semantic base for Dijkstra’s wlp calculus (wlp stands for \textit{weakest liberal precondition}). A liberal precondition for a program \( R \) is a condition which is fulfilled if \( R \) does not terminate.

4.3 Quantalian Semantics [2,21]

Given a signature the expressions obtained from methods and the operations \text{id} and \text{seq} of Table 1 and a construction principle which is not contained in Table 1 yield a quantalian structure on methods.

\textbf{Definition 4 (Quantai.)}. A quantal \( Q \) is a complete lattice with a constant \( id_Q \in Q \) and an operation \( \_; \_ : Q \times Q \to Q \) forming a monoid, i.e. obeying these conditions for all \( R, S, T \in Q \):

\begin{align*}
\text{Associativity}.
\end{align*}

\[
R; S; T \overset{\text{def}}{=} (R; S); T = R; (S; T)
\]
Neutrality of $id_Q$.

\[
R; id_Q = R \\
\text{id}_Q; S = S
\]

and fulfilling for each $Q \subseteq \mathcal{Q}$

\[
R; \bigvee Q = \bigvee \{R; Q \mid Q \in \Omega\} \\
\bigvee Q; S = \bigvee \{Q; S \mid Q \in \Omega\}
\]

The additional construction principle has the following semantics in all four forms:

\[
\{X\} \bigvee_{i \in I} R_i \{Y\} \iff \forall i. \{X\} R_i \{Y\}
\]

\[
\left(\bigvee_{i \in I} R_i\right) (Y) = \bigcap_{i \in I} R_i^= (Y)
\]

\[
\left(\bigvee_{i \in I} R_i\right) (X) = \bigcup_{i \in I} R_i^= (X)
\]

\[
x \in \bigvee_{i \in I} R_i y \iff \exists i. x R_i \rightarrow y
\]

4.4 Dynamic Logics [12]

Dynamic logics are considered only semantically, herein: $X_0$ and $Y_0$ stand for sets of states. Any set $Y \in \mathcal{P}(Y_0)$ is mapped to $[R] Y \in \mathcal{P}(X_0)$ and to $\langle R \rangle Y \in \mathcal{P}(X_0)$ for programs $R$, such that $[R]$ is an intersection preserving map and $\langle R \rangle$ is an union preserving map. The intuitive meanings of these junctors are: $\langle R \rangle Y$ is the set containing all those states from which one can get by $R$ to a state obeying condition $Y$ and $[R] Y$ is the set containing all those states from which one cannot get by $R$ to states which do not fulfill $Y$. More formally, using the relational semantics as introduced in Definition 2 these junctors are defined as follows:

\[
x \in [R] Y \iff \forall y. x R \rightarrow y, y \in Y
\]

\[
x \in \langle R \rangle Y \iff \exists y. y \in Y \wedge (x R \rightarrow y)
\]

There are analogous constructions for statements about the past:

\[
y \in \overline{[R]} X \iff \forall x. x R \rightarrow y, x \in X
\]

\[
y \in \overline{\langle R \rangle} X \iff \exists x. x \in X, (x R \rightarrow y)
\]
Now, the relationship to the semantics of this paper can easily be provided:

\[ X \subseteq [R]Y \iff [\overline{R}]X \subseteq Y \]
\[ \iff \{X\} R \{Y\} \]
\[ \langle R \rangle Y \subseteq X \iff Y \subseteq [\overline{R}]X \]
\[ \iff \{\neg X\} R \{\neg Y\} \]

Obviously, for each Galois connection \( R \) the map \( X \) assigning to \( \neg R \cup (\neg X) \) is an infimum preserving function. For each Galois connection by virtue of logical negation another Galois connection correlated to this infimum preserving function as mentioned before, the Galois connection \( \overline{R} \) can be defined as follows in all four equivalent forms:

\[ \{Y\} \overline{R} \{X\} \iff \{\neg X\} R \{\neg Y\} \]
\[ \overline{R}^\uparrow(X) = \neg R^\downarrow(\neg X) \]
\[ \overline{R}^\downarrow(Y) = \neg R^\uparrow(\neg Y) \]
\[ y \xrightarrow{\overline{R}} x \iff x \xrightarrow{R} y \]

This map is involutionary:

\[ \overline{\overline{R}} = R \]

Using this map all four above-mentioned constructions can be expressed by means of Galois connections:

\[ [R]Y = R^\uparrow(Y) \]
\[ [\overline{R}]X = R^\downarrow(X) \]
\[ \overline{[R]}X = \overline{R}^\uparrow(X) \]
\[ \langle R \rangle Y = \overline{R}^\downarrow(Y) \]

Beside these statements, in dynamic logic, there are further expression means for iteration and conditionals in programming languages. They are also important for usage in timing diagrams. However, this topic is worth to be dedicated to an own investigation. For the purposes of this paper it is not necessary to consider such expression means, just because there is no counterpart in the kind of timing diagrams we are interested in.

### 4.5 The Semantics of Causal Dependencies

Causal dependencies of type \textit{causes} like that in Fig. 8 can also be expressed by the formula \([R] \langle S \rangle \texttt{tt}\) in dynamic logics where \texttt{tt} is the “true” proposition. \([S][R] \texttt{tt}\) expresses causal dependencies of type \textit{requires} as shown in Fig. 9. Causal dependencies can also be expressed by Hoare triples if we accept second order statements:
\( \forall X. \{ \neg X \} S \{ \text{ff} \} \)
\( \{ \text{tt} \} R \{ X \} \)

is the semantics of causal dependencies of type \emph{causes} and causal dependencies of the type \emph{requires} can be expressed as follows:

\( \forall Y. \{ \text{tt} \} R \{ Y \} \)
\( \{ \neg Y \} S \{ \text{ff} \} \)

The expressibility of dynamic logics is exceeded by the fact that the par-construct in Table 1 cannot be expressed in dynamic logic. However, it is not hard to find a respective extension of dynamic logics because there is a direct correspondence of dynamic logic and the relational semantics used in Definition 2 can be recycled by this definition from the junctors of dynamic logics:

\[ x R y \overset{\text{def}}{\iff} y \in \langle R \rangle \{ x \} \]

5 Conclusion

5.1 Related Work

There is a flock of approaches to describe parallel processes. [23] contains an extract of the entirety of “models of concurrency”. Processes can be described declaratively, e.g. using temporal [8,24], dynamic [12] and other logics and using Hoare triples [13]. On the other hand, processes can be defined operationally, just as in programming languages, CCS [15], CSP [14], ACP [4], the “chemical abstract machine” [3] and other so-called process algebras, and Petri nets [19].

Causal dependencies play an important role in this paper. Other approaches to define processes using causality concepts are event structures [16,17,26] and Chu spaces [11,18]. The causality concept which is used in these approaches corresponds with the type \emph{requires} of our approach. We added the causal dependencies of type \emph{causes} because we cannot do without them in most of the practically relevant cases. The principal concept of both event structures and Chu spaces is the event concept. Events are supposed to happen at mostly once during a process lifetime. Thus, for non-trivial processes the number of events being involved during the process lifetime is infinite. Specifying processes this fact has to be considered because in specifications only finitely presented statements can be used. This is another reason why we introduced the concept of causal dependencies. A counterpart of the \emph{conflict relation} treating non-determinacy postulated by event structures is still under investigation.

Timing diagrams were already formalized for instance in [22] for the same purposes as in this paper. [22] contains also a semantic description using a temporal logic. We decided to go another way. Roughly speaking, temporal logics are logics describing the properties of temporally ordered states. Causal dependencies as well as the program verification techniques emphasize the dynamic aspects. They handle with methods, their composition to complex scenarios and synchronization. These issues are also to the fore for specification based on timing diagrams.
5.2 Discussion

This paper is the result of searching for a precise semantics of timing diagrams. Timing diagrams form a declarative specification style already used by electronic engineers, e.g. describing the functionality of integrated circuits. Timing diagrams are characterized in [22] as “graphical specification language with an intuitive semantics, which is especially appropriate for the description of asynchronous distributed systems such as hardware designs”. There are other formalizations for timing diagrams, for instance in [22] where timing diagrams are translated to temporal logics. In contrast, we prefer a relational semantics closely related to program verification techniques like dynamic logics. For that, there are two main reasons: The close relationship to program verification techniques raises the hope to overcome the heterogeneity between hardware and software design by a common semantic base. The other reason is that timing diagrams rather express dynamic aspects as scenarios and their order (just like the relational semantics introduced in this paper) than properties of temporally ordered states as in temporal logics.

In addition to the work presented in this paper, we have already developed a module concept for our approach [9] using the well-known principles from algebraic specifications [6,7,20]. Real time issues and temporal ordering are still subject of investigations.

References

A Process Compensation Language

Michael Butler and Carla Ferreira

Declarative Systems and Software Engineering Group (DSSE)
Dept. of Electronics and Computer Science, Univ. of Southampton,
Highfield, Southampton SO17 1BJ, United Kingdom
{mjb,cf}@ecs.soton.ac.uk

Abstract. This paper presents a formal language for the design of component-based enterprise system. The language (StAC) allows the usual parallel and sequential behaviours, but most significant is the concept of compensation that allows a previous action to be undone. The semantics of the language is given by an operational approach. The specification of a system is composed by a set of StAC processes that describe the behaviour of the system and a set of B operations that describe basic computations. Operational semantics is used to justified the integration of StAC processes with B operations.

1 Introduction

The work presented in this paper is the result of the collaboration of the DSSE Group with the Transaction Processing Design and New Technology Development Group at IBM UK Laboratories, Hursley. The collaboration has been concerned with approaches and techniques for component-based enterprise system development.

So far, with support from IBM UK Laboratories, we have defined a formal design language suitable for a heterogeneous distributed environment. IBM imposed some particular features on the language:

- The target enterprise solution should be built by stitching together Enterprise Java Beans [7] (EJBs).
- The language should allow sequential and parallel composition of behaviours.
- It should be possible for earlier actions to be undone, which is called “compensation”, whereby the system keeps track of the compensations that need to be executed if part of a process is to be aborted.

The IBM group believes that compensation gives more flexibility than the traditional commit-rollback approach to transaction processing. This flexibility is necessary for the heterogeneous distributed environment on which modern enterprise systems operate. Instead of restoring the system to the state before activities where performed in the case of abnormal events, activities can have a compensation activity associated with them.

The main goal of our collaboration with IBM was to define the semantics of compensation, which is a complex concept. The complexity arises in particular because of the combination of compensation with parallel execution.
In this paper we define a formal language for the design of heterogeneous distributed systems, which we called StAC (Structured Activity Compensation). In this approach the description of a system determines the way to “connect” simple components in order to create a complete system. In the StAC language the components are B [1] operations. B is a model-oriented formal notation. We use B instead of EJBs because we want the language to have a formal semantics.

A system specification has two components, the StAC specification that describes the execution order of the operations, and a B specification that describes the state of the system and also its basic operations. Since there are commercial tools available to support the development in B, the B specification can be animated and also proof obligations can be generated and proved.

An operational semantics is defined for the StAC language and its integration with B is also formalised through an operational approach.

In Section 2 we present informally the StAC language and specify in StAC an e-bookstore. In Section 3 we complete the example specification by describing the B operations. Section 4 describes the semantics of StAC, and in the last section, we discuss the integration of StAC with B.

2 The StAC Language

We can say informally that in StAC a system is specified as a process, and such a process will be decomposed into several sub-processes in a top-down approach. At the bottom level there will only be activities (each activity is a atomic computation), so they can not be further decomposed. Formally a system is described by a set of equations of the form

\[ N = P, \]

where \( N \) is a process identifier and \( P \) is an expression that can contain several process identifiers, including \( N \), since the equations can have recursion. We have determined, for simplicity reasons, that the first equation describes the overall system being specified.

The distinctive characteristic about the StAC language is the concept of compensation. This concept can have many interpretations, the most common being recovering from an error. Each process can have a corresponding compensation process, so executing a process has two consequences: the execution of the process itself, and its compensation process must be preserved by the system\(^1\).

Next we present two informal definitions of the concepts of compensation process and compensate. Later on these two concepts will be formally defined.

**Definition 1 (Compensation Process).** Process representing activities to be performed to compensate some behaviour.

**Definition 2 (Compensate).** Action of invoking the compensation process.

\(^1\) Later on we will define what the expression “preserved by the system” exactly means.
The StAC language has some usual process combinators, like sequence and parallel. Besides these it has specific combinators to deal with the compensation. Next we present the syntax of StAC terms in Backus-Naur form:

\[
\text{Process ::= } A \quad \text{(activity label)} \\
\quad | 0 \quad \text{(skip)} \\
\quad | b \rightarrow P \quad \text{(condition)} \\
\quad | \text{rec}(N) \quad \text{(recursion)} \\
\quad | P;Q \quad \text{(sequence)} \\
\quad | P \parallel Q \quad \text{(parallel)} \\
\quad | \| x \in X.P_x \quad \text{(generalised parallel)} \\
\quad | P][Q \quad \text{(choice)} \\
\quad | [] x \in X.P_x \quad \text{(generalised choice)} \\
\quad | P \div Q \quad \text{(compensation pair)} \\
\quad | \Xi \quad \text{(compensate)} \\
\quad | \Box \quad \text{(commit)} \\
\quad | [P] \quad \text{(compensation scoping)}
\]

Each activity label \( A \) has an associated activity \( A \rightarrow \) representing an atomic change in the state: if \( \Sigma \) is the set of all possible states, then \( A \rightarrow \) is a relation on \( \Sigma \). In Section 3, we show how the B notation is used to describe state and activities. In the rest of the paper we consider the capital letters \( A \) to \( D \) are activities, and the letters \( P \) and \( Q \) are processes.

In the conditional operator, process \( P \) is guarded by a boolean function \( b \). This boolean function can consult the state, i.e., \( b : \Sigma \rightarrow \text{Bool} \) and \( b \rightarrow P \) is enabled only if \( b \) is true. The recursive operator \( \text{rec}(N) \) enables the use of a process identifier \( N \) of the right-side of an equation to be used in the left-side term of an equation. The sequence \( P;Q \) forces an order in the execution of \( P \) and \( Q \): \( P \) must be executed first and only when \( P \) finishes can \( Q \) be executed. In parallel process \( P \parallel Q \), the execution of the activities of \( P \) and \( Q \) is interleaved. Generalised parallel extends the parallel operator over a set \( X \), which can either be finite or infinite. The choice \( P][Q \) selects whichever of \( P \) or \( Q \) is enabled. If both \( P \) and \( Q \) are enabled, the choice is made by the environment. Notice that the \( [] \) operator causes non-determinism in some cases. If we consider the following example:

\[
(A; B)] (A; C)
\]

when activity \( A \) occurs its not possible to determine which one of the two behaviours \( A; B \) or \( A; C \) will be chosen. In this case, the choice is made by the system rather than the environment. Generalised choice extends choice over a set of processes.

The next set of operators is related to the compensation concept. The compensation pair \( P \div Q \) expresses that \( P \) has \( Q \) as its compensation process. The compensation process is constructed in the reverse order of process execution, for example:

\[
(A \div A'); (B \div B')
\]
behaves as \( A; B \) and has the compensation process \( B'; A' \). A compensation process can be viewed as a stack where processes are pushed into the top of the stack.

The compensate operator \( \mathit{\checkmark} \) causes the compensation process to be executed. Consider the process (1) followed by \( \mathit{\checkmark} \):

\[
(A \div A'); (B \div B'); \mathit{\checkmark}.
\]  

(2)

As we saw before, process (1) behaves as \( A; B \), and then \( \mathit{\checkmark} \) operator causes the compensation process to be executed, so the overall behaviour of process (2) is \( (A; B); (B'; A') \) which we write as \( A; B; B'; A' \).

The commit operator \( \square \) clears the compensation process, meaning that after a commit the compensation process is set to 0. Consider again process (1). If now we append to it the \( \square \) operator we have:

\[
(A \div A'); (B \div B'); \square.
\]

As we already saw the compensation process of process (1) is \( B'; A' \), but after the \( \square \) operator the compensation process is 0. Another example,

\[
(A \div A'); (B \div B'); \square; \mathit{\checkmark}
\]

behaves as \( A; B \) because when the \( \mathit{\checkmark} \) operator is called the compensation process \( B'; A' \) has already been cleared by the \( \square \) operator.

As we mentioned before the complexity of StAC language arises from the combination of compensation with parallel execution. Given the following parallel processes:

\[
(A \div A') \parallel (B \div B')
\]

the execution of processes \( A \div A' \) and \( B \div B' \) is interleaved, implying that the execution of their compensation process should also be interleaved, so the resulting compensation process is \( A' \parallel B' \).

Next we will consider the combination of compensation with choice. The process:

\[
(A \div A') \parallel (B \div B')
\]

behaves as \( A \) or \( B \) and the compensation process in the first case is \( A' \) and in the second case is \( B' \).

The compensation scoping operator \( [P] \) creates a new compensation process within the square brackets. In the following process:

\[
(A \div A'); [(B \div B')],
\]

the compensation process within the brackets is just \( B' \), and \( A' \) is excluded because it's outside the brackets. If we added the compensate operator as follows:

\[
(A \div A'); [(B \div B'); \mathit{\checkmark}],
\]
the overall process would behave as $A; B; B'$, since the $\Xi$ just executes the compensation process within the brackets. Also, the process:

$$(A \div A'); [(B \div B'); \square]; C \div C')$$

has $C'; A'$ as compensation. Since the commit operator is inside the brackets, it just clears the compensation process $B'$ that is within the brackets. Another feature of the compensation scoping operator is that compensation is remembered if compensate is not performed, as in the example:

$$(A \div A'); [(B \div B')]; (C \div C').$$

Here, the compensation process is $C'; B'; A'$, which includes the compensation process $B'$ defined inside the brackets. $B'$ is retained because there is no commit with the brackets.

The StAC language permits nested compensation pairs as in the next example:

$$A \div (B \div C). \quad (3)$$

The process (3) behaves as $A$ and has the compensation process $B \div C$. Let's review what happens when the compensate operator is appended to process (3),

$$(A \div (B \div C)); \Xi. \quad (4)$$

First $A$ is executed, after the $\Xi$ operator invokes the compensation process $B \div C$, so the behaviour of process (4) is $A; B$ with compensation process $C$.

**Example: E-Bookstore**

The E-Bookstore is a typical example of an e-business. In this example each client defines a limited budget and has an e-basket where the selected books are kept. Every time the client selects a book, the budget is checked to see if it was exceeded, in this case the book is returned to the e-shelf. When the client finishes shopping he can either pay or abandon the bookstore, in the later case all selected books have to be returned to the shelf. Next, we present the StAC specification of the e-bookstore:

\[
\text{Bookstore} = \parallel c \in \text{Client}(c) \\
\text{Client}(c) = \text{Arrive}(c) ; \text{ChooseBooks}(c) ; ((\text{Quit}(c) ; \Xi) \parallel \text{Pay}(c)) ; \square ; \text{Exit}(c) \\
\text{ChooseBooks}(c) = \text{Checkout}(c) \parallel ((\text{ChooseBook}(c) ; \text{ChooseBooks}(c))) \\
\text{ChooseBook}(c) = \parallel b \in B ; (\text{AddBook}(c, b) \div \text{ReturnBook}(c, b)) ; \text{overBudget}(c) \Rightarrow \Xi \\
\text{Pay}(c) = \text{ProcessCard}(c) ; \neg \text{accepted}(c) \Rightarrow \Xi
\]

Notice that some processes are written with a different font, e.g., \text{AddBook}, this means that those processes are activity labels, and as mentioned before they can not be further decomposed. Activities will be specified as B operations in
the next section. Both overBudget\((c)\) and accepted\((c)\) are boolean expressions which will also be defined in B.

In the above specification the bookstore is represented by an infinite set of parallel Client processes each indexed from a set C. This implies that the process Bookstore never ends, so it is always available and if a client exits the bookstore he can later return with a new Client process. Each Client has a thread of compensation independent from all the other Client parallel processes. Also, each Client is a sequence of five processes. The first one is Arrive that creates and initialises the client information, setting the budget to a value determined by the client. The next process is ChooseBooks, followed by a choice between paying the books in the basket or abandon the bookstore without buying any books. If the client chooses to quit, the process ⊥ is invoked causing the return of all books in the client’s basket to the shelf. The fourth process is ⊥ which discharges all compensation information. The last process is Exit that removes the client from the bookstore, clearing all the information related to that client. The process ChooseBooks is a recursive process where the client chooses between selecting individual books (process ChooseBook) and returning afterwards to ChooseBooks process or stop selecting books, choosing Checkout\((c)\). ChooseBook creates a new thread of compensation, using scoping brackets. This new thread is only related to one book. Within ChooseBook there is a compensation pair, AddBook compensated by ReturnBook, and the compensation is executed only if adding that book to the basket exceeds the budget. In this case executing the compensation implies returning the book that has just been added to the basket rather than all books in the basket. In the process Pay, the client’s card is processed, and if the card is rejected, the compensation is executed, so all selected books are returned to the shelf.

3 Describing Activities in B

B AMN is a model-oriented formal notation and is part of the B-method developed by Abrial [1]. In the B-method, a system is defined as an abstract machine consisting of some state and some operations acting on the state.

```
MACHINE Bookstore
VARIABLES v
INVARIANT I
OPERATIONS ...
END
```

Fig. 1. Bookstore abstract machine
The bookstore abstract machine has the structure outlined in Figure 1. The abstract machine consists of some variables which are described using set-theoretic constructs. The invariant is a set of first-order predicates. Operations act on the variables while preserving the invariant and can have input and output parameters. Initialisation and operations are written in the generalised substitution notation of B AMN which includes constructs such as assignment, guarded statements and unbounded choice.

Machine State

Next we describe the machine state of our example, which is defined by the aggregation of the clauses VARIABLES and INVARIANT:

MACHINE Bookstore
VARIABLES budget, basket, price, accepted
INVARIANT
  budget ∈ C → N ∧
  basket ∈ C → F(B) ∧
  price ∈ B → N1 ∧
  accepted ∈ C → BOOL ∧
  ...
END

The clause VARIABLES names the variables of the abstract machine such as budget, basket, price and accepted. In the INVARIANT part we specify the types of the variables introduced in the previous clause. The variable budget is a function that for each client returns the maximum amount of money the client intends to spend in the bookstore. Variable basket is also a function but in this case it returns the books selected by each client. The price function contains the price of each book, which is necessary in order to verify if a client has exceed his budget. The variable accepted is a function that for each client determines if the client's card was accepted or rejected.

Machine Operations

The operations defined in the B specification are the activities of the process Bookstore described in Section 2. The activities are Arrive, Checkout, AddBook, ReturnBook, Quit, ProcessCard and Exit.

We will describe in detail some operations, the first one is the operation AddBook,

\[
\text{AddBook}(c, b) \triangleq \text{SELECT } c ∈ C ∧ b ∈ B ∧ b \notin \text{basket}(c) \text{ THEN}
\]

\[
\text{basket}(c) ::= \text{basket}(c) ∪ \{b\}
\]

END

The SELECT construct enables the operation only if all conditions are true. In this case c must be in the set C of clients, b must be a book of the set B, and also
the book $b$ must not be already in the basket of the client. The last condition was just added for simplicity reasons, we could extend the basket variable to return a bag instead of a set of books. If all conditions are met, book $b$ is added to the basket of client $c$. The operations ReturnBook, Arrive and Exit are similar to the operation described above in the sense that they are enabled by certain conditions and all of them cause a change in the machine state.

Next we describe the operation Checkout:

$$\text{Checkout}(c) \doteq \text{SELECT } c \in C \text{ THEN } \text{skip END}$$

This operation is enabled if client $c$ is in the set $C$. Since Checkout is used in the StAC process Bookstore to exit the recursive definition of ChooseBooks, this operation does not need to perform any explicit action. Operation Quit is similar to operation Checkout, and is used to determine which action the client wants to perform, quit the bookstore or pay the books.

The boolean function overBudget is specified as a B definition, since it only consults the machine state:

$$\text{overBudget}(c) \overset{\text{def}}{=} \text{budget}(c) \geq \sum b . (b \in \text{basket}(c) \mid \text{price}(b))$$

This boolean expression calculates if the total price of the books in the client’s basket exceeds the initial budget.

Operation ProcessCard sets the variable accepted. The variable accepted is used in the process Pay to trigger the compensation process when the card is rejected.

$$\text{ProcessCard}(c) \doteq \text{SELECT } c \in C \text{ THEN } \text{CHOICE accepted := TRUE OR accepted := FALSE END}$$

Operation ProcessCard is described as a choice between attributing the value TRUE or FALSE to the variable accepted depending on the card being accepted or rejected. The need for this variable is due to the fact that conditional processes must only use boolean expressions as guards, and ProcessCard is not a boolean function, because for the same state it can assign different values to variable accepted.

We can state that any guards of conditional processes should be specified in B as boolean expressions of the variables of the state machine.

4 The StAC\textsubscript{i} Language

In this section we introduce the StAC\textsubscript{i} (Structured Activity Compensation with indexes) language which extends the StAC language by adding different threads of compensation to a process. The main reasons for creating this new language is that StAC\textsubscript{i} has a clear semantics for compensation and makes it easier to describe parallel compensation.
We define formally the syntax and the semantics of StAC\textsubscript{i} language. This new language will be used to define the semantics of StAC. To achieve that we constructed a translation function from StAC to StAC\textsubscript{i} terms.

The main purpose in defining the operational semantics was the formalisation of the concepts present in the StAC language, which was the aim of the collaboration with IBM. Also, we will justify (section 5) the integration of StAC and B through an operational approach.

4.1 Abstract Syntax

The StAC\textsubscript{i} language extends the concept of compensation present in StAC language. In StAC\textsubscript{i} a process can have several compensation threads, each one with an independent compensation process. The new operators reflect the extension of the StAC language. Operator \( P \div i Q \) denotes that \( Q \) is the compensation process of \( P \) within the thread \( i \). In same way operators \( \oplus i \) and \( \ominus i \) compensate and commit, not the general compensation process, but the compensation process of the thread \( i \). The new operator \( J \triangleright i \) merges all compensation processes of the threads belonging to \( J \) into the compensation process of thread \( i \).

A process in the StAC\textsubscript{i} language is defined by the following Backus-Naur form:

\[
\text{Process ::= } \begin{align*}
A & \quad \text{(activity label)} \\
0 & \quad \text{(skip)} \\
b \rightarrow P & \quad \text{(condition)} \\
rec(N) & \quad \text{(recursion)} \\
P; Q & \quad \text{(sequence)} \\
P \parallel Q & \quad \text{(parallel)} \\
\parallel x \in X.P_x & \quad \text{(generalised parallel)} \\
P||Q & \quad \text{(choice)} \\
\llbracket x \in X.P_x \rrbracket & \quad \text{(generalised choice)} \\
P \div i Q & \quad \text{(compensation pair)} \\
\oplus i & \quad \text{(compensate)} \\
\ominus i & \quad \text{(commit)} \\
J \triangleright k & \quad \text{(merge)}
\end{align*}
\]

As expected both StAC and StAC\textsubscript{i} languages are very similar, StAC\textsubscript{i} has some additional symbols \( i, k, J \) and \( \triangleright \). Symbols \( i \) and \( k \) are members of the set of indexes \( I \), also \( J \) is a subset of \( I \). \( J \triangleright k \) is a new operator and it will be described formally in the next section.

4.2 Operational Semantics

The semantic domain of an StAC\textsubscript{i} program is a tuple,

\[
(P, C, \sigma) \in \text{Process} \times (I \rightarrow \text{Process}) \times \Sigma
\]
which we call a Configuration. In the above tuple, \( C \) is the set of compensation threads for the process \( P \), such that for each index \( i \), \( C(i) \) represents the compensation process of thread \( i \). \( \Sigma \) represents the state of the B machine. We will write a labelled transition

\[
(P, C, \sigma) \xrightarrow{A} (P', C', \sigma')
\]

to denote that executing activity \( A \) may cause a configuration transition from \( (P, C, \sigma) \) to \( (P', C', \sigma') \).

In any configuration, the choice between enabled transitions with different labels is made by the environment while the choice between enabled transitions with the same label is made by the system.

Some rules of the operation semantics are silent rules, where the transition is not labeled. Those rules do not introduce non-determinism because their applicability is disjoint from the rules for labelled transitions and from each other.

Next we give a set of operational rules for StAC \( i \) programs.

**Activity**

We assume that an activity is a relation from states to states, and write \( \sigma \xrightarrow{A} \sigma' \) when \( \sigma \) is related to \( \sigma' \) by \( A \). The execution of an activity within an StAC \( i \) process imposes a change in the state, leaving the compensation function unchanged:

\[
\sigma \xrightarrow{A} \sigma' \\
(A, C, \sigma) \xrightarrow{A} (0, C, \sigma')
\]

**Condition**

In this case the execution of a process \( P \) is guarded by a boolean function \( b \). If the result of applying \( b \) to the current state is \( true \), then \( P \) may be executed:

\[
(P, C, \sigma) \xrightarrow{A} (P', C', \sigma') \land b(\sigma) = true \\
(b \rightarrow P, C, \sigma) \xrightarrow{A} (P', C', \sigma')
\]

If the result of \( b \) is false then \( b \rightarrow P \) is replaced by skip and both the compensation function and the state remain unchanged:

\[
b(\sigma) = false \\
(b \rightarrow P, C, \sigma) \rightarrow (0, C, \sigma)
\]

**Recursion**

In the recursive call of a process \( N \) (\( N \) is a process identifier), \( rec(N) \) is substituted by the term of the left-side of the equation \( N = P \):

\[
N = P \\
(rec(N), C, \sigma) \rightarrow (P, C, \sigma)
\]
Sequence
This rule states that in a sequence of processes $P; Q$, an order is imposed, so the process $P$ is executed first and only then $Q$ can be executed:

\[
\frac{(P, C, \sigma) \xrightarrow{A} (P', C', \sigma')}{(P; Q, C, \sigma) \xrightarrow{A} (P'; Q, C', \sigma')}
\]

Executing 0 in sequence with $P$ is the same as executing $P$ alone:

\[
(0; P, C, \sigma) \rightarrow (P, C, \sigma)
\]

Parallel
Parallel processes can be executed in an arbitrary order:

\[
\frac{(P, C, \sigma) \xrightarrow{A} (P', C', \sigma')}{(P \parallel Q, C, \sigma) \xrightarrow{A} (P', \parallel Q, C', \sigma')}
\]

\[
\frac{(P, C, \sigma) \xrightarrow{A} (P', C', \sigma')}{(Q \parallel P, C, \sigma) \xrightarrow{A} (Q \parallel P', C', \sigma')}
\]

Executing a process $P$ in parallel with 0 is equivalent to just executing $P$:

\[
(P \parallel 0, C, \sigma) \rightarrow (P, C, \sigma)
\]

\[
(0 \parallel P, C, \sigma) \rightarrow (P, C, \sigma)
\]

Note that the parallel process $P \parallel Q$ terminates (i.e., reduces to 0) when both $P$ and $Q$ terminate.

Generalised Parallel
The rules for parallel are generalised for set $X$:

\[
\frac{(P_{x_1}, C, \sigma) \xrightarrow{A} (P'_{x_1}, C', \sigma')}{(\parallel x \in X. P_x, C, \sigma) \xrightarrow{A} (\parallel x \in (X - \{x_1\}). P'_x \parallel P'_{x_1}, C', \sigma')}
\]

Choice
In the process $P || Q$ only one of the processes $P$ or $Q$ is executed:

\[
\frac{(P, C, \sigma) \xrightarrow{A} (P', C', \sigma')}{(P || Q, C, \sigma) \xrightarrow{A} (P', C', \sigma')}
\]

\[
\frac{(P, C, \sigma) \xrightarrow{A} (P', C', \sigma')}{(Q || P, C, \sigma) \xrightarrow{A} (P', C', \sigma')}
\]

The choice between process 0 and any other process $P$ is equivalent to the single process $P$:

\[
(0 || 0, C, \sigma) \rightarrow (0, C, \sigma)
\]

\[
(0 || P, C, \sigma) \rightarrow (P, C, \sigma)
\]
Generalised Choice
This operator extends choice over a set $X$:

$$\left( (P_{x_1}, C, \sigma) \xrightarrow{A} (P'_{x_1}, C', \sigma') \land x_1 \in X \right)$$

Compensation Pair
In the compensation pair $P \div Q$, an evolution in process $P$ does not alter process $Q$:

$$\left( (P, C, \sigma) \xrightarrow{A} (P', C', \sigma') \land (P \div_i Q, C, \sigma) \xrightarrow{A} (P' \div_i Q, C', \sigma') \right)$$

The rule below adds the compensation process $Q$ to the compensation function $C$, which only happens when the process $P$ finishes:

$$\left( (0 \div_i Q, C, \sigma) \rightarrow (0, C[i := Q; C(i)], \sigma) \right)$$

Compensate
In the next rule, the operator $\llcorner i$ causes the compensation process of level $i$ to be executed, and also resets that compensation process to 0:

$$\left( (\llcorner_i, C, \sigma) \rightarrow (C[i), C[i := 0], \sigma) \right)$$

Commit
The operator $\lbrack_{i}$ clears the compensation process of level $i$ to skip:

$$\left( (\lbrack_{i}, C, \sigma) \rightarrow (0, C[i := 0], \sigma) \right)$$

Merge
The operator $J \triangleright i$ merges all compensation processes threads of set $J$ in parallel on to the front of compensation process of thread $i$:

$$\left( (J \triangleright i, C, \sigma) \rightarrow (0, C[i := (\{ j \in J.C(j) \}; C(i), J := 0), \sigma) \right)$$

In the above rule the expression $J := 0$ denotes attributing to all elements of set $J$ the process 0, i.e., $\{ j := 0 \mid j \in J \}$. $J$ must be disjoint from $i$. 
4.3 Translation from StAC to StAC_i

Instead of defining the semantics of the StAC language, we define a translation from StAC to StAC_i, so the interpretation of an StAC process P is given in terms of StAC_i by the translation function.

Next we present function T that translates an StAC process into an StAC_i process,

\[ T : L(\text{StAC}) \times I \rightarrow L(\text{StAC}_i) \]

where L(StAC) and L(StAC_i) represent StAC and StAC_i languages, and I is an infinite set of indexes. The parameter I is necessary in order to define T recursively. To translate a process P we have to select an index \( i \in I \), and then \( T(P, i) \) will return a StAC_i process.

The next translation rules are fairly simple. For example, to translate a sequential process \( P; Q \) with an index \( i \) is necessary to translate each process \( P \) and \( Q \) with the same index \( i \):

\[
\begin{align*}
T(A, i) &= A \\
T(b \rightarrow P, i) &= b \rightarrow T(P, i) \\
T(\text{rec}(P), i) &= \text{rec}(T(P, i)) \\
T(P; Q, i) &= T(P, i); T(Q, i) \\
T(P[Q, i]) &= T(P, i)[T(Q, i)] \\
T([x \in X.P_x], i) &= \{ j \in X \mid T(P_x, j) \} \\
T(P \div Q, i) &= P \div i Q \\
T(\sqcap, i) &= \sqcap_i \\
T(0, i) &= 0
\end{align*}
\]

The following set of rules are more complex. The main difficulty in the translation is parallel processes and their compensation information. Since we do not know the order of execution of \( P \parallel Q \) it implies that we also do not know in which order their compensation should be executed. The solution is to create a new thread for each parallel process, so their compensation process is also a parallel process. In the end both new threads, \( j \) and \( k \), are merged into the previous thread \( i \).

\[
\begin{align*}
T(P \parallel Q, i) &= \sqcap_{\{j, k\}}; (T(P, j) \parallel T(Q, k)); \{j, k\} \triangleright i \\
T(\{x \in X.P_x, i\}) &= \sqcap_{j}; (\{x \in X.T(P_x, j)\}; J \triangleright i
\end{align*}
\]

where \( j \) and \( k \) are new distinct indexes, and \( J = \{j_x \mid x \in X\} \) is a set of new indexes such that \( x \neq x' \Rightarrow j_x \neq j_{x'} \). The notation \( \sqcap_J \) is a simplification for \( \sqcap_{j_x \in J, \sqcap_{j_{x'}}} \). The final merge means that the compensation processes of the parallel processes are retained (unless they have been explicitly committed).

In the last rule we translate the compensation scoping \([P]\). The scoping brackets are translated to a new thread of compensation that in the end is merged in to the previous index:

\[
T([P], i) = \sqcap_{j}; T(P, j); \{j\} \triangleright i
\]
where \( j \) is a new index.

To clarify the translation described above, we will exemplify the translation of a simple process,

\[
\begin{align*}
\mathcal{T}(A \div A' \parallel B \div B'); C \div C', i) &= \mathcal{T}(A \div A' \parallel B \div B', i); (j, k) \triangleright i; C \div C', i) \\
&= \square\{j, k\}; \{T(A \div A', j) \parallel T(B \div B', k); \{j, k\} \triangleright i; C \div C', i) \\
&= \square\{j, k\}; \{A \div j A' \parallel B \div k B', i); \{j, k\} \triangleright i; C \div C', i) 
\end{align*}
\]

The first rule applies the function \( \mathcal{T} \) to both sequential processes \( A \div A' \parallel B \div B' \) and \( C \div C' \). To translate the parallel process it is necessary to create a new thread for each parallel process. After the translation of both parallel processes, the new threads are merged into the initial thread.

5 Integrating StAC\(_i\) and \( B \)

The operational semantics rules at Section 5 allow us to consider a process as a Labelled Transition System (LTS) as in [6]. Furthermore, [2] shows how a B machine can be viewed as an LTS. For those reasons, the semantics of the integration of StAC\(_i\) and B will be defined based on the operational semantics.

A B machine can be viewed as an LTS, where the state space is represented by the cartesian product of the types of state of the machine variables, labels are represented by the operations names and the transitions are represented by the operations.

The semantics of B operations is given in terms of weakest preconditions. For a statement \( S \) and postcondition \( Q \), \([S]Q\) represents the weakest precondition under which \( S \) is guaranteed to terminate in a state satisfying \( Q \).

In order to define when a transition is allowed by a B operation, we use the notion of conjugate weakest precondition defined as follows:

\[
\langle S \rangle Q \triangleq \neg[S]\neg Q.
\]

\( \langle S \rangle Q \) represents the weakest precondition under which it is possible for \( S \) to establish \( Q \) (as opposed to the guarantee offered by \([S]Q\)). Rules for \( [S] \) and \( \langle S \rangle \) are shown in figure 2.

Suppose the B machine represents activity \( A \) with an operation of the form

\[
A \triangleq S,
\]

where \( A \) is the operation identifier and \( S \) is a B AMN statement on the machine state \( \sigma \), then the transition

\[
\sigma \xrightarrow{A} \sigma'
\]

is possible provided

\[
v = \sigma \Rightarrow \langle S \rangle(v = \sigma').
\]
Here \( v \) represents the variables of the state machine.

A parameterised B operation of the form

\[
A(x) = S
\]

represents a set of activity definitions with labels of the form \( A.i \), and the operation corresponding to activity \( A.i \) is given by the statement \( x := i; S \).

6 Conclusions

The aim of the collaboration with IBM was to formalise the concepts of the structured ordering of activities and activity compensation. We have achieved that by defining a language \( \text{StAC}_i \) that specifies a system as a set of process equations. The language permits choice, parallel and sequential behaviours. But most important, \( \text{StAC}_i \) formalises the concept of compensation. Each process can have a compensation process, that is invoked when its actions need to be compensated. There are special processes called activities that represent atomic computations. Instead of extending \( \text{StAC}_i \) to include variables and expressions we used the B notation to specify activities. The specification of a system has two components, a set of process equations and a B machine describing the activities. The B machine includes a state, operations on the state and boolean expressions. The semantics of the \( \text{StAC}_i \) language and its integration with B is justified through an operational semantics.

The work presented in [3] has similar aims, but it does not have the concept of compensation.

Future work includes the translation of \( \text{StAC}_i \) processes into a B machine. Having the overall system specified in B would allow the specification to be animated and appropriate proof obligations to be generated (which is already possible for the activities). An experimental translation has been devised, but it is necessary to provide a formal proof that the translated specification is equivalent to the combined \( \text{StAC}_i \) and B specification.

Another important extension to the present work is to support the refinement of the B machine, that specifies the overall system, to compositions of EJBs.
This would guarantee the correctness of all of the development steps, from the abstract specification (StACi and B) to the implementation (Java code).

Acknowledgments. We would like to thank Peter Henderson of DSSE and Iain Houston, Mandy Chessell, David Vines and Catherine Griffin of IBM UK Labs for their contribution.

References

Activity Graphs and Processes

Christie Bolton and Jim Davies

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD
{christie,jdavies}@comlab.ox.ac.uk

Abstract. The widespread adoption of graphical notations for software design has created a demand for formally-based methods to support and extend their use. A principal focus for this demand is the Unified Modeling Language (UML), and, within UML, the diagrammatic notations for describing dynamic properties.

This paper shows how one such notation, that of Activity Graphs, can be given a process semantics in the language of Communicating Sequential Processes (CSP). Such a semantics can be used to demonstrate the consistency of an object model and to provide a link to other methods and tools. A small example is included.

1 Introduction

The Unified Modeling Language (UML) [22] is a collection of graphical notations, each of which can be used to specify different aspects of structure, state or system behaviour at varying levels of rigour and abstraction. It has been widely adopted for object-oriented software development. UML has an extensively-documented structural semantics but, as [8] observe, lacks any formal static or behavioural semantics.

A significant amount of work has been done on the formal semantics of UML, notably: a type semantics of class models [11]; dynamic semantics for state diagrams [14] and state machines [4]; and the combined work of the precise UML group [19]. Closely related to our work is that of [16] and [1]. In addition, some of the notations, being adapted from existing methods, have inherited a formal semantics by association. Activity graphs, diagrams used to describe the flow of behaviour within a system, are an example of this inheritance. They combine features of State Charts [12] with the intuition of Petri Nets.

The contribution of this paper is two-fold. Firstly we provide a formal semantics for activity graphs, describing them in terms of the process language CSP [13]. Secondly, based on results presented in [11] and a refinement-preserving translation from abstract data types to processes described in [3], and along similar lines to the work of [9] and [5], we describe how the final class description of a system can be expressed as an abstract data type and then translated to its corresponding CSP process.
Using refinement we can then compare the process corresponding to a class description against processes corresponding to activity graphs constructed in its development process; we can verify that the final class description is consistent with its specification.

An earlier attempt at formalising the semantics of activity graphs, by the current authors, is included in [2]. The intent there was quite different: to demonstrate that a uniform process semantics could be used to check the consistency of different parts of a process model. The treatment of activity graphs, in particular, was rather more superficial than that described here. Two notable differences are that the graph transitions in [2] could not be labelled with actions and events and that the semantics only applied to graphs with a restricted topology. For instance, the semantics of [2] could not cope with cross-synchronisation, as illustrated in Figure 1.

![Fig. 1. An activity graph with cross-synchronisation.](image)

In this paper, we take an entirely different approach to the graph notation: states, pseudo-states and transitions are all now modelled in terms of processes. Furthermore, the graph syntax of [2] has been discarded and replaced with one much closer to the *de facto* standard of the Rose interchange format [20].

The paper begins with a brief introduction to the mathematical notations used throughout the document. In Section 3, we introduce the graphical notation used in activity graphs and our corresponding syntactic definitions before presenting our behavioural semantics in Section 4. In Section 5, we give an example of a simple class description and an activity graph that might have been constructed during its development process. We show how our behavioural semantics may be used to verify that they are consistent. We conclude the paper with a discussion on the applicability of this work and highlight potential areas for future research.
2 Notation

The mathematical notation used in this paper is a combination of Z [24], CSP [13] and Object-Z [7,23]. In Section 6 we assume a little knowledge of Petri nets (see, for example, [18]).

2.1 Z and Object-Z

The language of Z is widely used for state-based specification. A characteristic feature is the schema, a pattern of declaration and constraint. The name of a schema can be used anywhere that a declaration, constraint or set could appear. Schemas may be named using the following syntax:

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>declaration</td>
</tr>
<tr>
<td>constraint</td>
</tr>
</tbody>
</table>

If S is a schema, then $\theta S$ denotes the characteristic binding of S in which each declared component is associated with its current value. This semantics supports the definition of a schema calculus, in which specification and analysis can be conducted at the level of schemas, abstracting and encapsulating the components declared within them; we can refer to the collection of identifiers in a particular schema without introducing a named instance of that schema type.

Schemas can be used as declarations. For example, the expression $\lambda S \cdot t$ denotes a function from the schema type underlying S, a set of bindings, to the type of term expression t.

The Z notation has a standard syntax for set expressions, predicates and definitions. Basic types, maximal sets within the specification, may be introduced as follows.

| Type |

Using the axiomatic definition we may introduce a global constant $x$, drawn from a set $S$, satisfying predicate p.

<table>
<thead>
<tr>
<th>$x : S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
</tr>
</tbody>
</table>

Object-Z is an object-oriented extension of Z adding notions of classes, objects, inheritance and polymorphism. A class is represented by a named box containing local types, constant definitions and schemas describing state, initialisation and methods on the class.

<table>
<thead>
<tr>
<th>Class Name[generic parameters]</th>
</tr>
</thead>
<tbody>
<tr>
<td>local type and constant definitions</td>
</tr>
<tr>
<td>state schema</td>
</tr>
<tr>
<td>initial state schema</td>
</tr>
<tr>
<td>operations</td>
</tr>
</tbody>
</table>
2.2 Processes

A process, as defined in [13], is a pattern of communication. We may use processes to represent components in terms of their communicating behaviour, building up descriptions using the standard operators of the CSP language.

Processes themselves are defined in terms of events: synchronous, atomic communications between a process and its environment. Compound events may be constructed using ‘.’ the dot operator; a family of compound events is called a channel, and may be used to represent the passing of a value between components.

Processes may be defined by sets of mutually-recursive equations, that may be indexed to allow parameterised definitions. Parameters may be used to represent aspects of the process state, and may appear in guards: we write $B \Rightarrow P$ to denote the process that behaves as $P$ if $B$ is true, and can perform no events otherwise.

The atomic process $\text{Skip}$ denotes successful termination, the end of a pattern of communication. If $P$ is a process and $a$ is an event then $a \rightarrow P$ is a process that is initially ready to engage in $a$. If this event occurs then the subsequent behaviour is that of $P$. If $P$ and $Q$ are both processes, then the process $P \parallel Q$ first behaves as $P$ and then, if $P$ terminates, behaves as $Q$.

If $P$ and $Q$ are processes, then $P \cap Q$ represents an internal choice between $P$ and $Q$; this choice is resolved by the process without reference to its environment. An internal choice over a set of indexed processes $\{i : I \cdot P(i)\}$ is written $\cap_i : I \cdot P(i)$. An external choice between two processes, written $P \sqcap Q$, may be influenced by the environment; the choice is resolved by the first event to occur. An external choice over a set of indexed processes is written $\square_i : I \cdot P(i)$; if each begins with a different event, then this is a menu of processes for the environment to choose from.

We write $\| i : I \cdot [A(i)] \cdot P(i)$ to denote an indexed parallel combination of processes in which each process $P(i)$ can evolve independently but must synchronise upon every event from the set $A(i)$. The combination may terminate only when every process is ready to do so. In an interleaving parallel combination, no synchronisation is required; in the combination $P \parallel Q$, the two processes evolve independently. We also employ an indexed version of the operator; $\| | i : I \cdot P(i)$. As with ordinary parallel combination, termination requires the agreement of all parties. Processes may also be partially-interleaved, in the sense that they must synchronise upon certain events but may perform other common events independently. In the combination $P \parallel | A \parallel Q$, the two processes must synchronise upon every event from the set $A$, but may perform all others independently.

Finally, if $P$ is a process and $A$ is a set of events then $P \setminus A$ is a process that behaves as $P$, except that both the requirement to synchronise upon and the ability to observe events from the set $A$ has been removed.

2.3 Behavioural Semantics of Processes

Several standard semantic models exist for the process language of CSP; see, for example, [21]. For the purposes of this paper we will employ the traces model;
it may be inappropriate to infer precise availability information from an activity
graph. In this model, each process is associated with a set of traces, or finite
sequences of events. The presence of a trace in the semantic set of a process
indicates that it is possible for that process to engage in that sequence of events.

Letting $\Sigma$ denote the set of all event names and $CSP$ denote the syntactic
domain of process terms, we may define the semantic function $T$ that takes a
CSP process and returns the set of all traces of the given process.

$$T : CSP \rightarrow \mathcal{P}(\text{seq } \Sigma)$$

The traces model admits a refinement ordering based upon reverse contain-
ment.

$$\subseteq_T : CSP \leftrightarrow CSP$$

$$\forall P, Q \in CSP \quad P \subseteq_T Q \iff T(Q) \subseteq T(P)$$

Refinement may be established through a combination of structural induction [6],
data-independence [15], and exhaustive, mechanical model-checking as described
in [21].

### 3 Graphical and Syntactic Descriptions of Activity

### Graphs

An activity graph is a special type of state diagram that is used to model the
flow of activities within a procedure; essentially it describes a collection of use
cases. An activity graph is constructed from a combination of action states,
(sub)activity states, start and finish states and pseudostates merge, decision,
fork and join. A table showing the graphical representation of each type of state
is presented in Figure 2.

Syntactically we define these states as follows:

$$Type ::= \text{action}\langle\text{Action}\rangle \mid \text{activity}\langle\text{ActivityLabel}\rangle \mid \text{start} \mid \text{finish} \mid \text{merge} \mid \text{decision} \mid \text{fork} \mid \text{join}$$

where Action and ActivityLabel are given types.

$$[\text{Action}, \text{ActivityLabel}]$$

Note that activity states and action states although graphically indistinguish-
able, as illustrated in Figure 2, are very different semantically. An action may
be thought of as a simple task that cannot be broken down any further, whereas
an activity is a task that can be broken down, and may itself be expressed as
an activity graph. An ActivityLabel within an activity state points to another
activity graph within the specification; an example of an activity state is “Build
House” within the “Create House” activity graph illustrated in Figure 3.

The states within an activity graph are linked together by transition lines.
These lines may have associated guards and actions as illustrated in Figure 4.
Given the types \textit{Line} and \textit{Guard},

\[ \text{[Line, Guard]} \]

the schema definition of a transition is as follows:
Transition

\[
\text{guard} : \text{Guard} \\
\text{action} : \text{Action} \\
\text{line} : \text{Line}
\]

If the graphical representation shows no guard or no action then the schema description of the transition records the default values true and null respectively.

\[
\text{true} : \text{Guard} \\
\text{null} : \text{Action}
\]

Each state within an activity graph records the type of its contents, the set of incoming lines it requires and the set of outgoing transitions it enables.

State

\[
\text{lines} : P \text{Line} \\
\text{transitions} : P \text{Transition} \\
\text{type} : \text{Type}
\]

The type State allows all possible states, including those which are not permissible within a UML specification. The schema WellFormed describes the subset of possible states that are permissible according to the constraints within the official documentation [17].

WellFormedState

State

\[
\begin{align*}
\text{type} = \text{start} &\iff \text{lines} = \emptyset \\
\text{type} = \text{finish} &\iff \text{transitions} = \emptyset \\
\text{type} \in \{\text{start, merge, join}\} &\Rightarrow \#\text{transitions} = 1 \\
\text{type} \in \{\text{finish, decision, fork}\} &\Rightarrow \#\text{lines} = 1 \\
(\exists \text{tr} : \text{transitions} \bullet \text{tr.line} \in \text{lines}) &\Rightarrow \text{type} \notin \{\text{start, finish, merge, decision, fork, join}\}
\end{align*}
\]

An activity graph is built up from a finite set of well-formed states. The environment of a specification is a function mapping the name of each graph to the associated graph. These are described formally below.

\[
\begin{align*}
\text{Graph} &::= \text{states} \triangleright F \text{WellFormedState} \triangleright \\
\text{Env} &::= \text{ActivityLabel} \rightarrow \text{Graph}
\end{align*}
\]

4 Behavioural and Semantic Descriptions

In this section we define our semantic function which takes the name of an activity graph within a specification and returns the CSP process that models the
behaviour of that graph. Essentially, this function returns the parallel combination of the processes corresponding to the states within the named graph each synchronising on its own alphabet; the synchronisation of the line events ensures that the actions occur in the correct order.

We introduce the types Process and Event

\[ [\text{Process}, \text{Event}] \]

and define the functions \( \varepsilon_{\text{line}} \) and \( \varepsilon_{\text{action}} \) which, respectively, map line names and actions to events. We insist that events corresponding to each line and to each action are distinct. Having defined an Action to be a simple task that cannot be broken down any further we may justifiably model it as a single event.

\[
\begin{align*}
\varepsilon_{\text{line}} : & \text{Line} \rightarrow \text{Event} \\
\varepsilon_{\text{action}} : & \text{Action} \rightarrow \text{Event} \\
\text{ran} \varepsilon_{\text{line}} \cap \text{ran} \varepsilon_{\text{action}} & = \emptyset
\end{align*}
\]

4.1 Alphabets

In order to define the alphabet for each state—the events on which it must synchronise—we must consider the events associated with each line, type and transition. We introduce functions \( \alpha_{\text{line}}, \alpha_{\text{tran}}, \alpha_{\text{type}} \) and \( \alpha_{\text{state}} \) which respectively take a line, a transition, the type of a state, and a state and return the associated set of events.

The set of events associated with a line is simply the relational image of the line under \( \varepsilon_{\text{line}} \). The set of events associated with a transition is the set containing the events corresponding to its line and its action.

\[
\begin{align*}
\alpha_{\text{line}} : & \text{Line} \rightarrow \mathbb{P} \text{Event} \\
\alpha_{\text{tran}} : & \text{Transition} \rightarrow \mathbb{P} \text{Event} \\
\alpha_{\text{line}} & = (\lambda \text{Line} . \{ \varepsilon_{\text{line}} \}) \\
\alpha_{\text{tran}} & = (\lambda \text{Transition} . \{ \varepsilon_{\text{line}}, \varepsilon_{\text{action}} \})
\end{align*}
\]

The set of events associated with the type action state with given action \( \text{actn} \) is the set containing the singleton event corresponding to \( \text{actn} \), whereas the set of events associated with the type activity state with given ActivityLabel \( \text{actL} \) is the alphabet of the graph corresponding to \( \text{actL} \); this is simply the union of the alphabets of all the states in the identified graph. The set of events associated with all other types of state is simply the empty set.
The alphabet of a given state is the union of the alphabets of all of its incoming lines, the alphabet of its type and the alphabets of all of its outgoing transitions.

### 4.2 Processes

We observed at the beginning of Section 4 that the process corresponding to the behaviour of a named activity graph within its specification environment was simply the parallel combination of the processes corresponding to its constituent states, each synchronising on its own alphabet. All line events are then hidden.

We formally define the function as follows.

The function $\rho_{state}$ returns the process corresponding to the behaviour of the given state. $\rho_{state}$ is essentially the sequential composition of the processes corresponding to the lines, type and transitions of the given state. In order define it formally we must consider these constituent processes and the effect of loops on the system.

The process that corresponds to the behaviour of the type action state with given action $actn$, is simply the event associated with $actn$ followed by $Skip$, followed by the lines, type and transitions of the given state.
whilst the process that corresponds to the behaviour of the type activity state with given ActionLabel $\text{actL}$, is the process corresponding to the graph associated with $\text{actL}$. The process that corresponds to the behaviour of all other types of states is simply $\text{Skip}$.

\[
\rho_{\text{type}} : \text{State} \rightarrow \text{Env} \rightarrow \text{Process}
\]

\[
\forall \text{state : State}; \text{env : Env}; \text{actn : Action}; \text{actL : ActivityLabel} \bullet
\]

\[
\text{state.type} = \text{action actn} \Rightarrow \\
\rho_{\text{type}} \text{state env} = (\varepsilon_{\text{action actn}}) \rightarrow \text{Skip}
\]

\[
\wedge \\
\text{state.type} = \text{activity actL} \Rightarrow \\
\rho_{\text{type}} \text{state env} = \text{semantics (actL) env}
\]

\[
\wedge \\
\text{state.type} \in \{\text{start, finish, merge, decision, fork, join}\} \\
\Rightarrow \\
\rho_{\text{type}} \text{state env} = \text{Skip}
\]

We observe that the difference between a $\text{merge}$ state and a $\text{join}$ state is that a $\text{merge}$ state can be reached provided any one of its incoming lines is enabled whereas a $\text{join}$ state can be reached only when all of its incoming lines have been enabled. In order to reflect these differences we define the following processes.

\[
\text{RequireAny} (ls) = \Box e : \varepsilon_{\text{line}(ls)} \bullet e \rightarrow \text{Skip}
\]

\[
\text{RequireAll} (ls) = ||| e : \varepsilon_{\text{line}(ls)} \bullet e \rightarrow \text{Skip}
\]

Similarly, in order to reflect the difference between a $\text{fork}$ state where all outgoing transitions are enabled and other states where only one transition may be chosen, we define the following processes.

\[
\text{Fork} (ts) = ||| \text{Transition} | \theta \text{Transition} \in ts \bullet
\]

\[
\text{guard} & \ (\text{if (action = null) then (} \varepsilon_{\text{line line}} \rightarrow \text{Skip})
\]

\[
\text{else (} \varepsilon_{\text{action action}} \rightarrow (\varepsilon_{\text{line line}}) \rightarrow \text{Skip} )
\]

\[
\text{Choice} (ts) = \Diamond \text{Transition} | \theta \text{Transition} \in ts \bullet
\]

\[
\text{guard} & \ (\text{if (action = null) then (} \varepsilon_{\text{line line}} \rightarrow \text{Skip})
\]

\[
\text{else (} \varepsilon_{\text{action action}} \rightarrow (\varepsilon_{\text{line line}}) \rightarrow \text{Skip} )
\]

Observe that if there is no action event on the transition line, that is $\text{action = null}$, then the corresponding process has no action event.

Given these definitions we define the function $\rho_{\text{trans}}$ that takes a state and returns the process that corresponds to the behaviour of its outgoing transitions.
Activity Graphs and Processes

\( \rho_{\text{trans}} : \text{State} \rightarrow \text{Process} \)

\[ \rho_{\text{trans}} = ( \lambda \text{State} \cdot \begin{cases} \text{Skip} & \text{if } (\text{component} = \text{finish}) \\ \text{Fork}(\text{transitions}) & \text{if } (\text{component} = \text{fork}) \\ \text{Choice}(\text{transitions}) & \text{else} \end{cases} ) \]

Note that we define the process corresponding to the transitions of a finish state to be Skip. This avoids having to choose one from an empty set of transitions that would give rise to deadlock.

Before we define the function \( \rho_{\text{lines}} \), consider the graph illustrated in Figure 5. Since line2 is a self-transition, that is, it loops back directly to the same state,

![Fig. 5. An activity looping back to itself.](image)

the event \( \epsilon_{\text{line2}} \) will not be in the alphabet of the process corresponding to any other state. This event will therefore not be constrained by synchronisation and can occur at any time. Thus, we must treat incoming lines which are part of a self-transition separately when considering the process corresponding to the incoming lines. We define the function \( \alpha_{\text{loops}} \) to take a state and return the set of all self-transitions, and the function \( \alpha_{\text{loops}} \) to return the set of all lines contained in a self-transition.

\[ \alpha_{\text{loops}} : \text{State} \rightarrow \mathbb{P} \text{ Event} \]

\[ \text{loops} = ( \lambda \text{state} : \text{State} \cdot \{ \text{tr} : \text{state.transitions} \mid \text{tr.line} \in \text{state.lines} \} ) \]

\[ \alpha_{\text{loops}} = ( \lambda \text{state} : \text{State} \cdot \{ \text{tr} : \text{loops state} \cdot \epsilon_{\text{line}}(\text{tr.line}) \} ) \]

Given these definitions, and recalling that only action states and activity can have direct loops back to themselves in a well-formed graph, we define the function \( \rho_{\text{lines}} \) which takes a state and returns the process describing the behaviour of all the incoming lines as follows.

\[ \rho_{\text{lines}} : \text{State} \rightarrow \text{Process} \]

\[ \rho_{\text{lines}} = ( \lambda \text{state} : \text{State} \cdot \begin{cases} \text{RequireAll}(\text{lines}) & \text{if } (\text{type} = \text{join}) \\ \text{RequireAny}(\text{lines} \setminus (\alpha_{\text{loops}} \text{ state})) & \text{else} \end{cases} ) \]

Finally, we define the function \( \rho_{\text{state}} \) which takes a state within a given specification environment and returns the process which models the behaviour of the entire state; the incoming lines, the type of state and the outgoing transitions.
\( \rho_{\text{state}} : \text{State} \rightarrow \text{Env} \rightarrow \text{Process} \)

\[
\rho_{\text{state}} = \\
( \lambda \text{state} : \text{State} \bullet ( \lambda \text{env} : \text{Env} \bullet \\
 \text{if } (\text{type} = \text{start}) \text{ then } \rho_{\text{trans state}} \\
 \text{else} \\
 \text{if } (\alpha_{\text{loops state}} = \emptyset) \text{ then} \\
 \text{let} \\
 X = \rho_{\text{lines state}} \triangledown \\
 ( \rho_{\text{type state env trans state}} \triangledown X \\
 ||| \\
 X ) \\
 \text{within} \\
 X \\
 \text{else} \\
 \text{let} \\
 Y = \rho_{\text{lines state}} \triangledown ( Z ||| Y ) \\
 Z = \rho_{\text{type state env}} \triangledown \\
 ( \text{Choice (loops state)} \triangledown Z \\
 \Box \\
 \text{Choice (transitions \setminus (loops state))} \triangledown Y ) \\
 \text{within} \\
 Y ) )
\]

Observe that the process corresponding to a start state is the only non-recursive process; a start activity can occur only once. For all other states, through interleaving the process with itself after the events corresponding to the incoming lines have occurred, we enable the incoming lines events to occur many times before the events corresponding to the state type and the state transitions occur\(^1\), thereby mirroring the intended behaviour of the graph and hence the system.

5 Example

In this section we present an Object-Z description of a simple ticket machine. We assume that this model was derived as the result of a UML specification and present an activity graph that might have been constructed in the development process. We show how both the Object-Z description and the activity graph can be translated to their process equivalents in CSP. We are then able to do a refinement check to determine whether or not the final model is consistent with the activity graph.

\(^1\) This corresponds closely to Petri nets in which each place can contain multiple tokens.
5.1 Activity Graph Describing a Ticket Machine

We assume that the activity graph illustrated in Figure 6 was constructed during the development process of a ticket machine (although we have added line names to our graph). We observe that a coin can be inserted at any time, and that for each coin that has been inserted either the red button or the green button may be pressed. If the red button is pressed then a coin will be returned. If the green button is pressed then the machine will non-deterministically choose either to issue a ticket or to return a coin.

We define the following set $I$ which indexes the processes corresponding to the states within the graph.

$I = \{\text{start}, \text{insert}, \text{fork}, \text{choice1}, \text{choice2}, \text{return}, \text{issue}, \text{stopR}, \text{stopG}\}$

In addition, we identify the following sets of events corresponding to the actions and lines described in the graph.

$Actions = \{\text{insertCoin}, \text{pressRed}, \text{pressGreen}, \text{issueTicket}, \text{returnCoin}\}$

$Lines = \{l:1 \ldots 10 \bullet \text{line} l\}$

Our syntax corresponds closely to the Rational Rose interchange format and we may therefore readily obtain our syntactical interpretation of the graph. Applying our semantic function to this syntactic description we obtain the process corresponding to the activity graph.

$GraphProcess = (\parallel i : I \bullet [A(i)] P(i)) \setminus Lines$

where for each $i$ in $I$, the process $P(i)$ is as defined below:

$P(\text{start}) = \text{line.1} \rightarrow \text{Skip}$

$P(\text{insert}) = (\text{line.1} \rightarrow \text{Skip} \sqcup \text{line.4} \rightarrow \text{Skip})$

$\parallel \parallel (\text{insertCoin} \rightarrow \text{line.2} \rightarrow P(\text{insert}))$

$P(\text{insert})$
\( P(fork) = \text{line}2 \rightarrow ((\text{line}3 \rightarrow \text{Skip} || \text{line}4 \rightarrow \text{Skip}) \circ P(fork)) \)

\( P(\text{choice1}) = \text{line}3 \rightarrow (\text{green} \rightarrow \text{line}5 \rightarrow P(\text{choice1}) \circ \text{red} \rightarrow \text{line}9 \rightarrow P(\text{choice1})) \)

\( P(\text{choice2}) = \text{line}5 \rightarrow (\text{line}6 \rightarrow P(\text{choice2}) \circ \text{line}8 \rightarrow P(\text{choice2})) \)

\( P(\text{issue}) = \text{line}6 \rightarrow (\text{issueTicket} \rightarrow \text{line}7 \rightarrow P(\text{issue})) \)

\( P(\text{stopG}) = \text{line}7 \rightarrow (\text{Skip} || P(\text{stopG})) \)

\( P(\text{return}) = (\text{line}8 \rightarrow \text{Skip} \circ \text{line}9 \rightarrow \text{Skip}) \circ \text{returnCoin} \rightarrow \text{line}10 \rightarrow P(\text{return}) \)

\( P(\text{stopR}) = \text{line}7 \rightarrow (\text{Skip} || P(\text{stopR})) \)

and for each \( i \) in \( I \), the set of events \( A(i) \) is as defined below:

\( A(\text{start}) = \{\text{line}1\} \)

\( A(\text{insert}) = \{\text{line}1, \text{line}2, \text{line}4, \text{insertCoin}\} \)

\( A(\text{fork}) = \{\text{line}2, \text{line}3, \text{line}4\} \)

\( A(\text{choice1}) = \{\text{line}3, \text{line}5, \text{line}9, \text{red}, \text{green}\} \)

\( A(\text{choice2}) = \{\text{line}5, \text{line}6, \text{line}8\} \)

\( A(\text{issue}) = \{\text{line}6, \text{line}7, \text{issueTicket}\} \)

\( A(\text{stopG}) = \{\text{line}7\} \)

\( A(\text{return}) = \{\text{line}8, \text{line}9, \text{line}10, \text{returnCoin}\} \)

\( A(\text{stopR}) = \{\text{line}10\} \).

5.2 **Object-Z Description of a Ticket Machine**

In this section we give an Object-Z description of a ticket machine which might have been constructed based on a UML specification containing the activity graph illustrated in Figure 6.

Four state variables have been introduced during the development process: \( \text{coins} \) is the number of coins that have been inserted for which the user has not yet selected the red button or the green button; \( \text{tickets} \) is the number of tickets left in the machine; \( \text{toReturn} \) is the number of coins to be returned; and
toIssue is the number of tickets to be issued. Note that the non-determinism in the activity graph—the machine could either return a coin or issue a ticket if the green button was pressed—has been resolved. In the Object-Z description, if the green button is pressed then the machine will issue a ticket if there is one left; otherwise it will return a coin.

Applying the methods described in [2] for translating a Z or Object-Z description, via an abstract data type to its corresponding process, and invoking the theorem presented in [3] that this translation preserves refinement:

**Theorem** One abstract data type is refined by another precisely when the process corresponding to the first is refined by the process corresponding to the second.
we see that the process corresponding to our Object-Z description is described by ClassDescProcess below.

\[
\text{ClassDescProcess} = \\
\text{let} \\
P(c, t, \text{iss}, \text{ret}) = \\
\text{insertCoin} \to P(c + 1, t, \text{iss}, \text{ret}) \\
\text{c > 0 & red} \to P(c - 1, t, \text{iss}, \text{ret} + 1) \\
\text{c > 0 & green} \to \begin{cases} \\
\text{if } t > 0 \text{ then } P(c - 1, t, \text{iss} + 1, \text{ret}) \\
\text{else } P(c - 1, t, \text{iss}, \text{ret} + 1) \\
\end{cases} \\
\text{iss} > 0 \& \text{issueTicket} \to P(c, t - 1, \text{iss} - 1, \text{ret}) \\
\text{ret} > 0 \& \text{returnCoin} \to P(c, t, \text{iss}, \text{ret} - 1) \\
\text{within} \\
\bigcap t : \{0 \ldots T\} \cdot P(0, t, 0, 0)
\]

The integer \( T \) is the machine’s ticket capacity. On initialisation it may be filled with any number of tickets not exceeding that capacity. Initially no coins have been inserted, no coins need returning and there are no tickets waiting to be issued.

5.3 Verifying Consistency

We prove that our final model of a ticket machine is consistent with the activity graph constructed in its development process using refinement.

In attempting to analyse a UML specification, we must take account of the context in which each diagram is presented. In general, activity graphs are intended only to illustrate a particular use case and so it would be inappropriate to infer availability information; hence we use the traces model to compare GraphProcess with ClassDescProcess.

\[
\text{ClassDescProcess} \sqsubseteq_T \text{GraphProcess}
\]

This refinement check tells us that every sequence of methods allowed by the activity graph is a possible behaviour of the class description.

For specifications in which the activity graph is intended to convey availability information, we would use the stable failures refinement [21] rather than the traces model employed here.

6 Adequacy and Application

Whilst the processes corresponding to our activity graphs defined in Section 4 and illustrated in the ticket machine example in Section 5 are entirely correct
they cannot in practise be used we come to analysing the system using a model checker. The interleaving of the process corresponding to each state with itself causes the system to diverge. This interleaving is, however, necessary if, for instance, we wish to allow the user of the ticket machine to be able to insert two coins before they press either the red or the green button.

To eliminate divergence we must, in Petri net terms, put an upper limit on the number of tokens allowed in any place; that is, for each action state and activity state we must only allow a finite number of events corresponding to an incoming line to occur before an event corresponding to an outgoing transition occurs.

We extend the definition of State to incorporate the variables isDynamic and dynamicMultiplicity as defined in the official documentation [17]. The Boolean isDynamic is True for action states and activity states in which the state's actions may be executed concurrently. If this Boolean is True then the integer dynamicMultiplicity limits the number of parallel executions of the actions of the state. If the Boolean isDynamic is False, then dynamicMultiplicity is ignored.

We define the functions inComing and outGoing which take a Practical-State and return respectively the set of events corresponding to the incoming non-looped lines and the set of events corresponding to the lines within the non-looped outgoing transitions. Given these definitions, we define the function constraint which returns the process which limits the number of parallel executions of each state.

\[\forall\text{ state : PracticalState }\bullet\]
\[\text{inComing state } = \varepsilon_{\text{line}}(\text{state.lines}) \setminus \alpha_{\text{loops state}}\]
\[\text{outGoing state } = \{tr : \text{state.transitions } \bullet \varepsilon_{\text{line}}(tr.line)\} \setminus \alpha_{\text{loops state}}\]
\[\text{constraint state } =\]
\[\text{let}\]
\[\text{Con}(n) =\]
\[\text{(n < state.dynamicMultiplicity) }\&\]
\[\square \text{ in : inComing state } \bullet \text{ in } \rightarrow \text{Con}(n + 1)\]
\[\square \text{ out : outGoing state } \bullet \text{ out } \rightarrow \text{Con}(n - 1)\]
\[\text{within}\]
\[\text{Con}(0)\]
Using these definitions, the process that, for practical purposes, we should use to model the behaviour of each state is given below.

\[
\rho_{\text{practical state}} : \text{PracticalState} \rightarrow \text{Env} \rightarrow \text{Process}
\]

\[
\rho_{\text{practical state}} = (\lambda \text{state} : \text{State} \bullet (\lambda \text{env} : \text{Env} \bullet \\
\text{if (isDynamic = False) then } \rho_{\text{state state env}} \\
\text{else } \rho_{\text{state state env}} \mid \left( \text{inComing state} \cup \text{outGoing state} \right) \mid \text{constraint state})
\]

Hence it follows that, for practical purposes, our semantic function should be

\[
\text{semanticsP} : \text{ActivityLabel} \rightarrow \text{EnvP} \rightarrow \text{Process}
\]

\[
\forall \text{env} : \text{EnvP}; \ \text{actL} : \text{ActivityLabel} \bullet \\
\text{semanticsP actL env} = \\
(\parallel \text{stateP : statesP} \sim (\text{env actL}) \bullet \\
[\alpha_{\text{stateP env}} \mid \rho_{\text{practical state stateP env}}) \\
\bigcup \{ \{ \text{stp : statesP} \sim (\text{env actL}) \bullet \varepsilon_{\text{line}} \} \} )
\]

where a EnvP maps activity labels onto “practical” graphs and where practical graphs are a collection of PracticalStates.

\[
\text{EnvP} \equiv \text{ActivityLabel} \rightarrow \text{GraphP}
\]

\[
\text{GraphP} \equiv \text{statesP} \parallel \mathcal{F} \text{PracticalState}
\]

7 Discussion

The Unified Modeling Language has many strengths. For instance, its graphical nature makes it easily readable by domain experts without a formal mathematical training; as such it provides a common ground for discussing the behaviour of the system to be developed. In addition, the combination of notations allows the user to reason at different levels of rigour and abstraction and from both behavioural and state-based perspectives.

However, as \[8\] observe, before UML can earn its place as a de facto standard for modelling object-oriented systems it must have a precisely defined semantics, and not just the precisely defined syntax that it has at present.

In this paper we have presented a formal behavioural semantics for activity graphs. We have illustrated, using a simple example, how this semantic model may be used to verify that the behaviour of the final class model description of a system is consistent with the behaviour of activity graphs constructed during its development process.

In addition, we have considered the practicalities of integrating such a semantics into a modelling tool; our syntax corresponds closely to the Rose interchange format \[20\], and we have built constraints into our practical model which will prevent the system from diverging.
A considerable amount of work remains to be done both in this particular area and in the wider scope of UML. In the semantics presented in this paper we have built guards into our model, but have not gone into detail as to how they should be used. At present we have adopted the most simplistic approach in which each guard is a constant value: null or True or False. An interesting area of research would be to consider how to model guards which are dependent on state variables and the effect that this would have on activity graphs with loops and multiple threads.

In addition, using the behavioural semantics presented here to verify consistency relies on the assumption that we can translate the final class model to an abstract data type and hence to its process equivalent. However the example we presented was a simple one with a single class. In theory a single class example is sufficient; [11] shows that formal descriptions of models with multiple classes and associations can be constructed by promoting the data types that model the individual classes. The process of reasoning about these descriptions can then be simplified: [25] explains in some detail how refinement distributes through promotion. However, we need to consider larger, more complex examples to check that this method is scalable and practicable.

Finally, it would be interesting to see if we could link the work presented here with that of [10]. Such a link would facilitate the automatic verification of consistency between activity graphs and the final java code.

Whilst these issues still need to be addressed, it is our hope that the work presented in this paper, combined with the work done by many others in this area, will lead to the automatic verification by modelling tools of the preservation within the final class description of both the static and behavioural properties captured in UML diagrams.

Acknowledgements. We would like to thank Jim Woodcock, Perdita Stevens and Geraint Jones for their helpful and insightful comments.

References


Structuring Real-Time Object-Z Specifications

Graeme Smith\(^1\) and Ian Hayes\(^2\)

\(^1\) Software Verification Research Centre
\(^2\) School of Computer Science and Electrical Engineering
University of Queensland, Australia

Abstract. This paper presents a means of structuring specifications in real-time Object-Z: an integration of Object-Z with the timed refinement calculus. Incremental modification of classes using inheritance and composition of classes to form multi-component systems are examined. Two approaches to the latter are considered: using Object-Z’s notion of object instantiation and introducing a parallel composition operator similar to those found in process algebras. The parallel composition operator approach is both more concise and allows more general modelling of concurrency. Its incorporation into the existing semantics of real-time Object-Z is presented.

1 Introduction

Object-Z [12] is an extension of Z [14] to facilitate specification in an object-oriented style. The major extension in Object-Z is the class schema which captures the object-oriented notion of a class by encapsulating a single state schema, and its associated initial state schema, with all the operation schemas which may change its variables. Classes may be incrementally specified using Object-Z’s notion of inheritance which enables definitions from one class (the inherited class) to be implicitly included in another class (the inheriting class). The enhanced structuring provided by object-oriented constructs, such as classes, and techniques, such as inheritance, significantly improve the clarity of large specifications.

In an earlier paper [13], we showed how Object-Z could be extended to model systems with continuous variables and real-time constraints. The approach was to provide a semantic basis for combining Object-Z and the real-time notation of the timed refinement calculus [9,5]. This notation allows a system to be specified by constraints over time intervals on which properties hold. However, the integrated approach, referred to as real-time Object-Z, did not utilise the structuring techniques of Object-Z provided by classes and inheritance. Hence, as presented, it is not suitable for large-scale specifications, nor for specifications comprising several components such as those of concurrent or distributed systems.

In this paper, we present an overview of real-time Object-Z (Section 2) and provide extensions to utilise Object-Z’s structuring techniques. In particular, we show how inheritance can be used to incrementally modify existing class specifications (Section 3), and how different classes can be composed to form
multi-component systems (Section 4). Two approaches to the latter are considered: using the object instantiation technique of Object-Z, and introducing a parallel composition operator similar to those found in process algebras. The parallel composition operator approach is both more concise and allows more general modelling of concurrency. It is also easily incorporated into the existing semantics of real-time Object-Z (Section 5).

2 Real-Time Object-Z

Real-time Object-Z \cite{13} is an integration of the timed refinement calculus \cite{9,5} with Object-Z \cite{12}. It differs from other approaches to specifying continuous and real-time systems in Object-Z since

- it uses only standard notation from Object-Z and the timed refinement calculus (the approaches of Friesen \cite{4} and Mahony and Dong \cite{7} introduce additional notation into schemas of Object-Z classes),
- it maintains Object-Z’s specification style (the approach of Periyasamy and Alagar \cite{10} requires each object to be specified by two classes: one for its functionality and one for its real-time properties), and
- it models the passing of time implicitly (the approach of Dong, et al. \cite{1} requires an explicit \textit{Tick} operation in each class).

2.1 Timed Refinement Calculus

The timed refinement calculus is a Z-based notation for the specification and refinement of real-time systems. It has been extended with a simple set-theoretic notation for concisely expressing time intervals \cite{6} and operators for accessing interval endpoints. We adopt a simplified subset of the notation based on that of Fidge, et al. \cite{3} which provides a minimal set of operators outside those of standard set theory.

Absolute time, $T$, is modelled by real numbers and, in this paper, we will assume has the units seconds. Observable variables of a system are modelled as total functions from the time domain to a type representing the set of all values the variable may assume. A system is specified by constraints on the time intervals over which properties hold. For example, the following expresses that an observable variable $v : T \rightarrow \mathbb{R}$ becomes equal to a differentiable (denoted by the function symbol $\sim \cite{2}$) observable variable $u : T \sim \mathbb{R}$ within 0.1 seconds whenever $u > 10$.

$\langle u > 10 \rangle \subseteq (\delta = 0.1) ; (v = u)$

The brackets $\langle \rangle$ are used to specify a set of time intervals$^1$. The left-hand side of the above predicate denotes the set of all time intervals where, for all times $t$ in the intervals, $u(t)$ is greater than 10.

$^1$ We adopt here a simpler notation than the brackets $\frac{1}{2} \frac{3}{4}$ used by Fidge et al. \cite{3} and our previous paper \cite{13}.
In general, the property in the brackets is any first-order predicate in which total functions from the time domain to some type \( X \) may be treated as values of type \( X \). The elision of explicit references to the time domain of these functions results in specifications which are more concise and readable.

The right-hand side of the above expression comprises two sets of intervals. The first uses the reserved symbol \( \delta \) which denotes the duration of an interval. Hence, this set contains all those intervals with duration 0.1 seconds. Other reserved symbols are \( \alpha \) and \( \omega \) denoting an interval’s start and end times respectively.

The second set denotes all intervals in which (for all times in the intervals) \( v \) equals \( u \). It is combined with the first set of intervals using the concatenation operator ‘\( ; \)’. This operator forms a set of intervals by joining intervals from one set to those of another whenever their end points meet. (One endpoint must be closed and the other open [3]). Hence, the right-hand side of the predicate specifies all those intervals where after 0.1 seconds, \( v \) equals \( u \).

The entire predicate, therefore, states (using \( \subseteq \)) that all intervals where \( u \) is greater than 10, are also intervals where, after 0.1 seconds, \( v \) equals \( u \).

### 2.2 Integration with Object-Z

The semantic integration of the timed refinement calculus with Object-Z was presented in our previous paper [13]. In this section, we provide an overview of the approach including two new extensions to the syntax.

Classes in the integrated notation comprise two parts separated by a horizontal line. The part above the line is essentially the standard Object-Z local definitions and schemas. The part below the line is further constraints on the class specified in the timed refinement calculus notation. The latter is divided into an assumption and effect part as in the timed refinement calculus [5]. All state variables \( x : X \) in the Object-Z part above the line are interpreted as timed trace variables \( x : T \to X \) in the timed trace part below the line.

Although all real-time properties could be specified in the timed trace part of the class, we also allow local constants and state variables of type \( T \) and include, in every class, an implicit state variable \( \tau : T \) denoting the current time. This is captured by an implicit constraint \( \forall t : T \cdot \tau(t) = t \) in the timed traced part of the class.

As an example, consider specifying a speedometer which calculates the speed of a vehicle by detecting the rotation of one of its wheels; the speed is calculated by dividing the wheel circumference by the time taken for a single rotation.

We assume a maximum speed of 60 metres per second (216 km/hr).

\[
\text{MaxSpeed} == 60 \quad \text{metres per second}
\]

The speed output by the speedometer is a natural number between 0 and \( \text{MaxSpeed} \).

\[
\text{Speed} == 0..\text{MaxSpeed} \quad \text{metres per second}
\]
A system specified by a class in the integrated approach is a digital system and, therefore, changes to state variables only occur at discrete points in time. However, it may interact with continuously changing variables in its environment. These variables are specified as (possibly differentiable) functions of time. For example, the speedometer’s environment includes a continuous variable representing the angle of the wheel in radians from some fixed position. This can be specified as follows.

\[\text{wheel\_angle}\ : \ T \sim \mathbb{R} \quad \text{radians}\]

Since this definition gives the values of the wheel’s angle over all time, it need not be treated as a modifiable state component and can appear as a local constant in the class. The “?” decoration on the name indicates that it is an environmental variable that acts as an input to the specified system. Similarly, environmental variables decorated with “!” act as “outputs” from the system.

Our first extension to the syntax allows such outputs to be declared as state variables (rather than constants) to indicate that they only change value when an operation, whose \(\Delta\)-list they appear in, occurs. Our second extension to the syntax allows operation names to appear as Boolean variables in the timed trace part of the class. The variable representing an operation is true in all intervals during which the operation is occurring. Examples of these features and the other features of a real-time Object-Z class are provided by the following specification of the speedometer.

\[
\begin{align*}
\text{Speedometer}_0 &\quad \text{wheel\_circum} = 1.5 \quad \text{metres} \\
&\quad \text{wheel\_angle}\ : \ T \sim \mathbb{R} \\
&\quad \text{last\_calculation} : T \\
&\quad \text{speed!} : \text{Speed} \\
\end{align*}
\]

\[
\begin{align*}
\text{INIT} &\quad \text{last\_calculation} < \tau - 2 \ast \text{wheel\_circum} \\
&\quad \text{speed!} = 0 \\
\end{align*}
\]

\[
\begin{align*}
\text{CalculateSpeed} &\quad \Delta(\text{last\_calculation}, \text{speed!}) \\
&\quad \text{wheel\_angle}\!(\tau) \mod 2\pi = 0 \\
&\quad \forall t : (\tau \ldots t') \bullet \text{wheel\_angle}\!(t) \mod 2\pi \neq 0 \\
&\quad \text{last\_calculation}' = \tau \\
&\quad \text{speed!!} = \text{wheel\_circum}/(\tau - \text{last\_calculation}) \pm 0.5 \\
\end{align*}
\]

\[
\begin{align*}
&\langle \Delta \text{wheel\_angle}\!\mod 2\pi = 0 \rangle \cup \langle \text{wheel\_angle}\!\mod 2\pi \neq 0 \rangle \\
&\langle \text{true} \rangle \cup \langle \text{CalculateSpeed} \rangle \cup \langle \text{true} \rangle \\
\end{align*}
\]
The speedometer calculates the speed ($speed$!) from the wheel circumference ($wheel\_circum = 1.5$ metres) and the wheel angle ($wheel\_angle$?) which implicitly records the number of whole revolutions of the wheel. To do this it keeps track of the time of the last speed calculation in a state variable $last\_calculation$. Initially, this variable is set to a time more than $2 \times wheel\_circum$ seconds before the current time $\tau$. This ensures that the first speed calculation, when the wheel starts rotating, will be zero (since the calculated speed is a natural number with units metres per second and a wheel rotation time of more than $2 \times wheel\_circum$ corresponds to a speed of less than 0.5 metres per second). Ensuring the first speed calculation is zero is necessary because the wheel may not undergo a full rotation before it occurs.

The operation $CalculateSpeed$ calculates the speed to the nearest natural number based on the wheel circumference and the time since the last calculation. It is enabled each time the wheel passes the point corresponding to a multiple of $2\pi$ radians. The first two predicates of the operation ensure that the wheel angle $mod\ 2\pi$ is $0$ only for the first time instant of the operation. This prevents the wheel completing an entire rotation before $CalculateSpeed$ has finished executing. (Note that intervals of real numbers can be specified using combinations of the brackets $[$ for closed intervals and $( )$ for open intervals.)

This latter constraint is feasible since the class has an assumption predicate which limits the rate of change of $wheel\_angle$? ($\frac{dv}{dt}$ denotes the derivative of a differentiable variable $v$ [2]). This assumption also ensures that the speed calculated by the final predicate of $CalculateSpeed$ is less than or equal to $MaxSpeed$. (Note that $\langle true \rangle$ denotes the set of all possible intervals.)

To ensure that $CalculateSpeed$ occurs every time the wheel passes the point corresponding to 0 radians, the class also has an effect predicate which states that $CalculateSpeed$ is a sub-interval of any interval where the wheel angle $mod\ 2\pi$ is $0$, and then becomes non-zero.

Note that operations in real-time Object-Z do not have input and output parameters: all communication is performed through environmental variables such as $wheel\_angle$? and $speed$!. This restriction enables a straightforward definition of refinement as shown in Section 5.

### 3 Inheritance

The speedometer specification of Section 2 works as we would expect when the wheel of the vehicle is rotating. If it stops rotating, however, the $CalculateSpeed$ operation does not occur and so the speed output by the class is that which was last calculated.

To overcome this problems we could add an operation which detects that the wheel is no longer rotating and sets the output speed to 0. Adding an operation can be done in standard Object-Z using inheritance [12]. When an Object-Z class inherits another it implicitly includes its constants, state schema, initial state schema and operations (and may extend these definitions or add to them). We extend the notion of inheritance to also implicitly include assumption and
effect predicates in the timed trace part of the inherited class. Hence, the desired modification to the speedometer can be specified as follows.

\[
\begin{align*}
\text{Speedometer} & \\
\text{Speedometer}_0 & \\
\text{TimeOut} & \\
\Delta (\text{last\_calculation}, \text{speed!}) & \\
\tau - \text{last\_calculation} & > 2 \times \text{wheel\_circum} \\
\text{speed!} & = 0 \\
\text{true} & \\
\langle \delta \geq 3 \times \text{wheel\_circum} \rangle & \subseteq \langle \text{true} \rangle ; \langle \text{TimeOut} \lor \text{CalculateSpeed} \rangle ; \langle \text{true} \rangle
\end{align*}
\]

The operation \text{TimeOut} is enabled when the time since the last calculation is greater than \(2 \times \text{wheel\_circum}\). This corresponds to a speed of less than half a metre per second (1.8 km/hr). The additional effect predicate ensures that \text{TimeOut} does occur before the time since the last calculation is greater than \(3 \times \text{wheel\_circum}\).

Semantically, inheritance in real-time Object-Z is the same as inheritance in standard Object-Z with the addition of conjoining of assumption predicates and conjoining of effect predicates from the inherited and inheriting classes. The variables and operations in the inherited assumption and effect predicates will be renamed to reflect any renaming in the inherited class [12].

4 Composition

To specify systems of concurrent, interacting objects, we need to be able to compose different classes. For example, consider specifying a cruise control system which is required to keep a car travelling at a desired speed set by the driver [8]. At any time, the driver can resume control of the car by applying the brake.

The system comprises three main components: a speedometer, a controller which accepts input from the driver, and a throttle which controls the car’s speed. It is illustrated in Figure 1. (Arrows indicate the direction of information flow.)

In Section 4.1, we specify the classes for the controller and throttle in such a way that they can interact with each other and the speedometer of Section 3 as shown in Figure 1. In Section 4.2, we look at composing the components using standard Object-Z composition and by introducing a parallel composition operator. The semantics of this operator is provided in Section 5.

A fundamental difference between our specification and that of Mahony and Dong using TCOZ [8] is our use of continuous variables for modelling the wheel angle and throttle inputs from the environment. TCOZ, based on timed CSP, can only model discrete events corresponding to reading these variables and cannot model the variables themselves.
4.1 Component Classes

The classes for the controller and throttle components are specified using real-time Object-Z as described in Section 2.

Controller. The controller operates in two modes: \textit{rest} when the speed of the car is being controlled by the driver, and \textit{set point} when a desired speed has been set by the driver and this speed is being maintained by the cruise control system.

\[
\text{Mode} ::= \text{rest} \mid \text{set point}
\]

Initially, the controller is in \textit{rest} mode and is changed to \textit{set point} when the driver presses a button indicating that he or she wants the car to maintain its current speed. The controller reverts to \textit{rest} mode when the brake is applied. While in \textit{set point} mode, the controller provides the throttle component with a desired value of the throttle setting. This is initially the current throttle setting and is updated periodically. The updated values of this setting are calculated from four parameters

- the current speed of the car,
- the desired speed (set by the driver),
- the speed of the car at the last calculation, and
- the current value of the throttle.

We abstractly specify this calculation by the function \textit{desired_throttle},

\[
desired\textunderscore throttle : \text{Speed} \times \text{Speed} \times \text{Speed} \times \mathbb{R} \rightarrow \mathbb{R}
\]

The controller class is specified as follows.
The timed refinement calculus predicate states that, when in `set_point` mode, the `Control` operation occurs in every 0.2 second interval. Since the duration of this operation is less than 0.1 seconds (as specified by its second predicate), this ensures that the operation occurs repeatedly with a period of less than 0.3 seconds (see Figure 2).
The throttle incrementally adjusts its output to reach a desired target output. This target depends on the mode of the controller. If its mode is rest the target output is equal to the input from the accelerator, otherwise it is equal to the control input from the controller. The throttle class is specified as follows.

\[
\text{Throttle} \\
\hspace{1cm} \text{throttle}_\text{adjust} : \mathbb{R} \\
\hspace{1cm} \text{mode} : \mathbb{T} \rightarrow \text{Mode} \\
\hspace{1cm} \text{accelerator} : \mathbb{T} \rightarrow \mathbb{R} \\
\hspace{1cm} \text{control} : \mathbb{T} \rightarrow \mathbb{R} \\
\]

\[
\text{throttle}! : \mathbb{T} \rightarrow \mathbb{R} \\
\]

\[
\text{Init} \\
\hspace{1cm} \text{throttle}! = 0 \\
\]

\[
\text{UpdateThrottle} \\
\hspace{1cm} \Delta(\text{throttle}!) \\
\hspace{2cm} \tau' - \tau < 0.1 \\
\hspace{2cm} \exists t : \mathbb{R} \bullet \\
\hspace{3cm} (\text{mode}?(\tau) = \text{rest} \Rightarrow t \in \text{accelerator}?(\exists \tau \exists \tau' \exists t)) \land \\
\hspace{3cm} (\text{mode}?(\tau) = \text{set}\_\text{point} \Rightarrow t = \text{control}?(\tau)) \land \\
\hspace{3cm} (\text{throttle}! < t - 0.5 \times \text{throttle}_\text{adjust} \Rightarrow \\
\hspace{4cm} \text{throttle}! = \text{throttle}! + \text{throttle}_\text{adjust}) \land \\
\hspace{3cm} (\text{throttle}! > t + 0.5 \times \text{throttle}_\text{adjust} \Rightarrow \\
\hspace{4cm} \text{throttle}! = \text{throttle}! - \text{throttle}_\text{adjust}) \land \\
\hspace{3cm} (\text{throttle}! \in t \pm 0.5 \times \text{throttle}_\text{adjust} \Rightarrow \text{throttle}! = \text{throttle}!) \\
\]

\[
\text{true} \\
\]

\[
\{\delta = 0.1\} \subseteq \{\text{true}\} ; \langle \text{UpdateThrottle} \rangle ; \{\text{true}\} \\
\]

The timed refinement calculus predicate states that the \text{UpdateThrottle} operation occurs repeatedly with a period of less than 0.2 seconds.
4.2 Composing the Components

When composing the components of the cruise control system we need to ensure

– that the corresponding inputs and outputs (e.g., speed! of Speedometer and speed? of Controller) are identified and equated, and
– that operations which use inputs, do not do so at a time when another component is updating the corresponding output.

The second condition is necessary since the value of a variable updated by an operation is undefined for the duration of the operation. This may be, for example, because in an implementation the value is stored as a sequence of bytes which are updated one at a time. At any time during the operation, some of the bytes may be updated and others not.

In this section we consider two approaches to composing real-time Object-Z classes: object instantiation and parallel composition.

Object instantiation. In standard Object-Z, systems are composed from instances of classes called objects [12]. Given an object $a$, we can refer to a variable or constant $x$ of the object’s class by the notation $a.x$. Similarly, we can refer to the initial condition or an operation $Op$ of the object’s class by $a.INIT$ and $a.Op$ respectively. Adopting this approach, we might specify the cruise control system as follows.

\[
\begin{align*}
\text{CruiseControl} : & \quad \text{Speedometer} \\
& \quad \text{Controller} \\
& \quad \text{Throttle} \\
\text{s.speed}! &= \text{c.speed}! \\
\text{c.mode}! &= \text{t.mode}! \\
\text{c.control}! &= \text{t.control}! \\
\text{t.throttle}! &= \text{c.throttle}! \\
\text{INIT} : & \quad \text{s.INIT} \land \text{c.INIT} \land \text{t.INIT} \\
\text{CalculateSpeed} \equiv & \quad \text{s.CalculateSpeed} \\
\text{TimeOut} \equiv & \quad \text{s.TimeOut} \\
\text{Set} \equiv & \quad \text{c.Set} \\
\text{Control} \equiv & \quad \text{c.Control} \\
\text{Brake} \equiv & \quad \text{c.Brake} \\
\text{UpdateThrottle} \equiv & \quad \text{t.UpdateThrottle}
\end{align*}
\]

The system \text{CruiseControl} comprises an object of each component class and explicitly equates their corresponding inputs and outputs. The initial condition and operations of \text{CruiseControl} are constructed explicitly from those of the component classes.
classes. We assume that the real-time predicates of the component classes are implicitly maintained for each of the objects. We also assume a common $\tau$ variable for each object and CruiseControl.

The condition that inputs are not used when they are being updated holds automatically in this case since the semantics of real-time Object-Z does not allow operations within a single class to overlap in time [13] (see Appendix A). This means, however, that our system as specified exhibits no concurrency. Concurrency can be specified explicitly. For example, the concurrent occurrence of operations CalculateSpeed and Brake could be specified by adding an additional operation to CruiseControl of the form

$$\text{CalculateSpeed} \& \text{Brake} \equiv \text{CalculateSpeed} \land \text{Brake}$$

However, this is an undesirable approach for two reasons.

1. Explicitly stating all combinations of operations which can occur concurrently may become unwieldy for large systems comprising many components. Indeed, even the explicit identification of corresponding inputs and outputs and construction of individual operations can be verbose when using object instantiation.

2. The conjoined operations must have the same start and finish times. While this allows us to specify synchronising events, it does not allow us to specify events which partially overlap.

**Parallel composition.** To overcome the problems with specifying concurrency, we introduce a parallel composition operator “$\|\$” for classes. The idea of this operator is that it allows each component to satisfy its specified behaviour over time synchronising with other components on environmental variables with common basenames (i.e., apart from the “?” and “!”) and on common-named operations. Since operations may in general overlap, if we require that two (or more) operations should be mutually exclusive in time, this needs to be explicitly specified.

One way to do this is to specify monitor classes which are composed with the components. A monitor class has a set of “dummy” operations (i.e., operations which perform no function). These operations comprise a subset of the total operations of the specified system which are not allowed to overlap in time. Due to the semantics of real-time Object-Z classes, these operations do not overlap within the monitor class. Due to synchronisation with the common-named operations of the component classes, these operations also cannot overlap in time.

For example, since the Controller operations Set and Control utilise the output speed! of Speedometer, they should not occur at times when the Speedometer operation CalculateSpeed, which changes speed!, occurs. The required monitor class is as follows.
Similarly, since the Set and Control operations of Controller utilise the output throttle! of Throttle, we need a monitor class with the operations Set, Control and UpdateThrottle. This class also ensures that the mode! and control! outputs of Controller are not updated when used by Throttle.

By having two separate monitor classes we do not preclude the possibility of CalculateSpeed and UpdateThrottle occurring concurrently. The cruise control system is specified as follows.

\[
\text{CruiseControl} = \text{Speedometer} \parallel \text{Monitor}_SC \parallel \text{Controller} \\
\parallel \text{Monitor}_CT \parallel \text{Throttle}
\]

The definition is both concise (due to implicit modelling of concurrency) and allows for more general modelling of concurrency (by allowing partially overlapping, as well as synchronising, operations). The semantics of the parallel operator is discussed in Section 5.

5 Semantics of Parallel Composition

Appendix A gives the semantics of a real-time Object-Z class as developed in our previous work [13]. A class \( C \) is represented by a set of real-time histories. These consist of the signature of the class, i.e., its sets of input, output and local variables and set of operations, together with traces of the variables and the occurrences of operations. A trace is the value of a variable over time. It is represented by a total function from time to value.

\[
\text{Trace} == \mathbb{T} \rightarrow \text{Value}
\]

An operation is represented by an identifier corresponding to its name.

\[
\text{Operation} == \text{Ident}
\]

From this semantics, we extract a model of a class which we use to define refinement and parallel composition. This model separates the class’s signature from its set of traces and operation occurrences. A signature of a class is specified as:
and a history as:

\[
RTHistory
\]

\[
\begin{align*}
\text{trace} &: \text{Ident} \Rightarrow \text{Trace} \\
\text{occurs} &: \text{Operation} \Rightarrow \mathbb{P}I
\end{align*}
\]

where \(I\) is the set of all time intervals.

A class is then modelled as follows.

\[
\begin{align*}
\text{RTClass} &\quad \text{sig} : \text{Signature} \\
\text{histories} &\quad : \mathbb{P} RTHistory
\end{align*}
\]

\[
\forall h : \text{histories} \quad \bullet
\]

\[
\begin{align*}
\text{dom}\ h.\text{trace} &= \text{sig} . \text{inputs} \cup \text{sig} . \text{outputs} \cup \text{sig} . \text{locals} \\
\text{dom}\ h.\text{occurs} &= \text{sig} . \text{opids}
\end{align*}
\]

Since the only communication mechanism between classes is environmental variables, and environmental inputs cannot be constrained by a class [13], all information about when operations can be refused by a particular class are captured by its histories. Hence, a class, \(C\), is refined by a class, \(D\), if \(C\) and \(D\) have the same signatures and the histories of \(C\) contain the histories of \(D\).

\[
\sqsubseteq : \text{RTClass} \leftrightarrow \text{RTClass}
\]

\[
C \subseteq D \Leftrightarrow C.\text{sig} = D.\text{sig} \land D.\text{histories} \subseteq C.\text{histories}
\]

When two classes are composed, the signature of the resulting composite class has those inputs of either class which are not also an output of the other class. That is, inputs which have the same name as an output in the other class are semantically identified with this output and no longer appear as an input to the composite class. This models the output’s value being communicated to the input.

The outputs, local variables and operations of a composite class are those from either class. The outputs and operations which are common to the component classes must satisfy the constraints of both classes within the composite class. (We assume that the local variables of the component classes are distinct. Our semantics could be made more general by adding some form of renaming to ensure this, thus removing the need for this assumption.)

The binary operator \(\text{comp}\) composes the signatures \(S\) and \(T\) of two classes.
The \texttt{comp} operator is commutative and associative.

The history of a component class can be derived from a history of a composite class by restricting the history of the composite class to the component class's signature.

The binary operator \texttt{restrict} extracts, from a real-time history $h$, a history corresponding to the signature $S$ of a component class.

The parallel combination of two classes, $C \parallel D$, has a signature which is the composition of the signatures of $C$ and $D$. Each of the histories of the parallel combination, if restricted to the signature of $C$ (respectively $D$), is a trace of $C$ ($D$).

Parallel composition of classes is commutative and associative. Furthermore, it is monotonic with respect to refinement in both its arguments.

### Conclusion

In this paper, we have shown how Object-Z's class and inheritance constructs can be used to structure specifications in real-time Object-Z: an integration of Object-Z with the timed refinement calculus. In particular, for composing Object-Z classes to form multi-component systems, we have introduced a parallel composition operator similar to those found in process algebras. This operator provides a means of composition which is both concise (due to implicit modelling of concurrency) and allows more general modelling of concurrency (by allowing partially overlapping, as well as synchronising, operations). The existing semantics of real-time Object-Z was extended to accommodate the parallel composition operator in such a way that the operator is commutative, associative and monotonic with respect to refinement.
Acknowledgements. This work is funded by Australian Research Council Large Grant A49801500, A Unified Formalism for Concurrent Real-Time Software Development.

References

A Semantics of Real-Time Classes

To provide a semantics for our integrated notation, we show how to map the standard Object-Z semantics to timed traces. Smith [11, §2.3] gives a history model for an Object-Z class in terms of sequences of states and operations. We introduce that semantics and then show how to relate it to timed traces.

A.1 Histories

Let \( \text{Ident} \) denote the set of all identifiers, and \( \text{Value} \) the set of all values of any type. A state is an assignment of values to a set of identifiers representing its attributes. It can be defined by a finite partial function from identifiers to values:

\[
\text{State} \equiv \text{Ident} \mapsto \text{Value}.
\]

An operation can be defined as an identifier corresponding to the operation’s name (since operations do not have input and output parameters in real-time Object-Z):

\[
\text{Operation} \equiv \text{Ident}.
\]

The history of an object consists of (possibly infinite) sequences of states and operations. The sequence of states is non-empty as there must be at least an initial state. The set of attributes of every state in the sequence comprises the state variables of the object’s class\(^2\) and hence must be the same. If the sequence of operations is finite, then the sequence of states has the initial state plus an element corresponding to the final state of every operation. Hence the sequence of states is one longer than the sequence of operations. If the sequence of operations is infinite, then so is the sequence of states.

These conditions are captured by the following schema where \( \text{seq}_\infty X \equiv \text{seq} X \cup (\mathbb{N}_1 \to X) \). For state variables corresponding to environmental outputs, \( \text{attributes} \) contains their names without the “!” decoration. The fact that they are environmental outputs is captured in \( \text{TraceHistory} \) (Section A.3).

\[
\begin{align*}
\text{History} & \quad \text{states} : \text{seq}_\infty \text{State} \\
& \quad \text{ops} : \text{seq}_\infty \text{Operation} \\
& \quad \text{attributes} : \mathbb{F} \text{Ident} \\
& \quad \text{opids} : \mathbb{F} \text{Operation}
\end{align*}
\]

\[
\begin{align*}
\text{states} \neq \emptyset \\
\forall i : \text{dom} \text{states} \bullet \text{dom} \text{states} i = \text{attributes} \\
\text{ran} \text{ops} \subseteq \text{opids} \\
\forall i : \mathbb{N}_1 \bullet i \in \text{dom} \text{ops} \iff i + 1 \in \text{dom} \text{states}
\end{align*}
\]

\(^2\) In Smith’s model [11] the attributes of states include, as well as state variables, all constants the object’s class can refer to. Here we take an alternative view that the values of such constants are parameters to the semantics.
A.2 Start and Finish Times

To map histories to timed traces, we extend the standard definition of an Object-Z history given above. The first extension is to allow for the start and finish times of each operation. The variable \( \text{start} \) denotes the sequence of start times of operations, and the variable \( \text{finish} \) denotes a sequence, with indices starting from 0, of finish times. We use \( \text{finish}(0) \) to represent the time at which the initialisation completed, and if the sequence of operations is finite we add an extra start time, with value \( \infty \), representing that after the last operation, the state is stable forever.

\[
\text{TimedHistory} \\
\text{History} \\
\text{start} : \text{seq}_\infty \mathbb{T} \\
\text{finish} : \mathbb{N} \to \mathbb{T} \\
\forall i : \mathbb{N} \cdot i \in \text{dom} \text{ops} \Leftrightarrow \{i, i + 1\} \subseteq \text{dom} \text{start} \\
\text{dom} \text{start} \neq \mathbb{N} \Rightarrow \text{last} \text{(start)} = \infty \\
\text{dom} \text{finish} = \{0\} \cup \text{dom} \text{ops} \\
\forall i : \text{dom} \text{ops} \cdot \text{start}(i) \leq \text{finish}(i) \\
\forall i : \text{dom} \text{finish}; j : \text{dom} \text{start} \cdot i < j \Rightarrow \text{finish}(i) \leq \text{start}(j)
\]

A.3 Timed Traces

The next extension is to add timed traces of variables. The timed trace of a variable is a mapping from time to the value of the variable at that time.

\[
\text{Trace} == \mathbb{T} \to \text{Value} \\
\text{We add a timed trace for every environmental variable and each state variable of the class. The names of environmental variables appear in the semantics without their “?!” or “!” decorations. This information is captured instead by three sets of identifiers: inputs for environmental inputs, output for environmental outputs, and locals for local state variables. The local state variables are a subset of the attributes, the remainder of the attributes being environmental outputs. The conditions under which a timed trace corresponds to an Object-Z history is defined below.}
\]

\[
\text{TraceHistory} \\
\text{TimedHistory} \\
\text{inputs, outputs, locals} : \text{F Ident} \\
\text{trace} : \text{Ident} \to \text{Trace} \\
\langle \text{inputs, outputs, locals} \rangle \text{ partitions(} \text{dom} \text{trace} \rangle \\
\text{locals} \subseteq \text{attributes} \\
\text{attributes} \subseteq \text{locals} \cup \text{outputs} \\
\forall i : \text{dom} \text{states}; id : \text{attributes} \cdot \\
\langle \text{trace id} \rangle \forall \text{finish}(i - 1) \ldots \text{start}(i) \exists = \{\text{states i}(id)\}
\]
A trace of a state attribute is stable with value \( states(i) \) from the finish time of an operation, \( finish(i - 1) \), until the start time of the next operation, \( start(i) \). The initial state, \( states(1) \), is stable from the finish time of the initialisation, \( finish(0) \), and, if the sequence of states is finite, the final state after the last operation is stable until the start time of the end-of-time event, i.e., infinity. Note that the value of the state trace is only determined for the stable periods between operations; it may be any value during the execution time of an operation.

### A.4 Operation Intervals

We extend the definition further to include the intervals in which particular operations occur. This allows the timed trace part of a class to refer to times when a particular operation is occurring.

Intervals are contiguous sets of times:

\[
\mathcal{I} : \mathcal{P}(\mathbb{R})
\]

\[
\mathcal{I} = \{ I : \mathbb{R} \mid [\inf(I) \ldots \sup(I)] \subseteq I \}
\]

where \( \inf(I) \) and \( \sup(I) \) stand for the infimum (greatest lower bound) and supremum (lowest upper bound), respectively, of the set \( I \).

For each operation in the Object-Z history, the set of time intervals over which it occurs is just the set of intervals between its start and finish times. From the constraints on start and finish times, no two of these intervals can overlap by more than just a single point of time.

\[
\text{RealTimeHistory}
\]

\[
\text{TraceHistory}
\]

\[
occurs : \text{Operation} \Rightarrow \mathcal{P} \mathcal{I}
\]

\[
occurs = (\lambda \text{op} : \text{opids} \bullet
\{ i : \text{dom} \text{ops} \mid \text{ops}(i) = \text{op} \bullet [\text{start}(i) \ldots \text{finish}(i)]\})
\]

The function \( occurs \) is derived from the \( \text{ops} \) sequence.

### A.5 Class Histories

An Object-Z class defines a possible set of histories for objects of that class. Smith [11] gives a function \( H \) which given a class returns a set of possible histories of that class. We extend this function to map the Object-Z part of a class in our notation to a set of real-time histories, i.e., histories extended as in the previous sections.

\[
H : \text{Class} \Rightarrow \mathcal{P} \text{RealTimeHistory}
\]

The details of \( \text{Class} \) and the mapping from a class to a set of real-time histories are the same as those given by Smith [11][3], except that in operation specifications

\[3\] The type we refer to as \( \text{Class} \) is called \( \text{ClassStruct} \) in Smith.
references to \( \tau \) and \( \tau' \) correspond to the start and finish times, respectively, of the operation. In addition, the semantics need to allow direct references to the environmental variables; such references treat an environmental variable as an explicit trace. The formalisation of these additional relationships within the framework used by Smith [11] is straightforward and we do not give the details here.

A.6 Timed Trace Predicates

To give the semantics of a class in our notation, we also need to consider the timed trace part of the class. A timed trace predicate defines a set of real-time histories that satisfy the predicate. Let \( \text{TimedTracePred} \) denote the set of all timed trace predicates.

\[
\text{traces} : \text{TimedTracePred} \rightarrow \mathcal{P} \text{RealTimeHistory}
\]

A real-time history, \( h \), satisfies a timed trace predicate if the predicate is true when we replace any reference to a trace variable, \( v \), by the corresponding value, \( h.trace(v) \). To allow operations in such predicates to represent Boolean values which are true in the sets of intervals in which they occur, we also need to replace any reference to an operation, \( op \), by true in intervals in the set, \( h.\text{occurs}(op) \), and false elsewhere.

An interval expression, such as \( \langle P \rangle \), where \( P \) is a predicate, is interpreted as the set of intervals such that \( P \) holds at all points in the interval:

\[
\{ \phi : \mathbb{I} \mid \exists \alpha, \omega, \delta : \mathbb{T} \cdot \alpha = \inf(\phi) \land \omega = \sup(\phi) \land \delta = \omega - \alpha \land \forall \tau : \phi \cdot P[v(\tau)/v] \}
\]

where \( v \) stands for the vector of all timed trace variables, i.e., \( \text{dom} trace \).

A class in our notation consists of the standard class components (augmented with environmental variables) and two real-time predicates specifying respectively the assumptions the class makes about environmental variables and the effect the class is to achieve on environmental variables.

\[
\text{RealTimeClass}
\]

\[
\begin{align*}
\text{class} : \text{Class} \\
\text{assumption, effect} : \text{TimedTracePred}
\end{align*}
\]

The possible real-time histories of such a class \( C \) consist of those histories that, if the assumption holds, also satisfy the effect and are real-time histories of the corresponding Object-Z part of the class.

\[
\mathcal{R} : \text{RealTimeClass} \rightarrow \mathcal{P} \text{RealTimeHistory}
\]

\[
\mathcal{R}(C) = \{ h : \text{RealTimeHistory} \mid h \in \text{traces}(C.\text{assumption}) \Rightarrow \\
\quad h \in \text{traces}(C.\text{effect}) \land \mathcal{H}(C.\text{class}) \}
\]
ISpec:
Towards Practical and Sound
Interface Specifications

Hans B.M. Jonkers
Philips Research Laboratories Eindhoven,
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
hans.jonkers@philips.com

Abstract. This paper introduces the ISpec approach to interface specification. ISpec supports the development of interface specifications at various levels of formality and detail in a way compatible with object-oriented modelling techniques (UML). The incremental nature of the levels and the underlying formal framework of ISpec allow informal interface specifications to be made formal in steps. The body of the paper consists of a discussion of the main characteristics of ISpec, which reflect the important decisions taken in the design of ISpec. The idea of component-based specifications and specification plug-ins for constructing heterogeneous specifications is discussed and a small example showing the various levels of specification supported by ISpec is presented.

1 Introduction

1.1 The Role of Interfaces

Interfaces play a role of increasing importance in modern software development. One reason for this is the current trend towards the use of open systems. The SEI definition of open system [15] clearly indicates why interfaces, and in particular interface specifications, are essential in open systems:

“An open system is a collection of interacting software, hardware, and human components
– designed to satisfy stated needs
– with interface specifications of its components that are
  • fully defined
  • available to the public
  • maintained according to group consensus
– in which the implementations of the components conform to the interface specifications”

Another reason for the increasing importance of interfaces is the growing use of component-based software development techniques [19]. This trend is supported by the availability of standard component models such as COM, CORBA and Java Beans. Components use interfaces to make their context-dependencies explicit. This use of interfaces as a decoupling mechanism can be seen as the crux of component-based software development.

© Springer-Verlag Berlin Heidelberg 2000
1.2 Objectives of ISpec

ISpec is an interface specification approach developed by Philips Research with the aim of providing a systematic approach to interface specification that

1. supports *multiple levels of formality and detail*;
2. fits in with main-stream *object-oriented modelling techniques* (UML);
3. has a solid footing in *formal techniques*;
4. provides a basis for *systematic testing*.

These objectives are based on our past and recent experience with the use of formal and informal specification techniques in several product divisions of Philips; see e.g. [5,13]. As many formal methodists have learned the hard way, formal techniques are considered a problem rather than a solution by the average software developer in industry. We therefore aim at an approach that can also be used informally and with varying degrees of detail, rather than an all-or-nothing formal approach (objective 1).

Despite the aversion to formal techniques displayed by many industrial software developers, there is a general feeling that better techniques for controlling the complexity of software are required. The growing popularity of object-oriented modelling techniques, and in particular UML, can be seen as a sign of this. We consider it important to fit in with this trend (objective 2), not just because it is opportune to do so but above all because the idea of object-oriented modelling and the formal-methods idea of ‘abstract modelling’ are closely related (see Section 2.4).

The unbridled use of informal techniques can easily lead to unstructured and inconsistent specifications. This can be avoided by choosing an appropriate formal framework as the basis of the approach and structuring specifications in such a way that the various parts of a specification, whether informal or formal, can be related unambiguously to parts of that framework (objective 3). Hence, *if necessary*, an ISpec interface specification can be made ‘completely’ formal. We put ‘completely’ between quotes because there are always aspects of interfaces that can only be expressed informally.

Another reason why it is important to have a formal basis is to be able to provide semantic tool support. The intention is to support the automatic generation of black-box tests from an ISpec specification (objective 4). The price to be paid for this is that the specification, or at least part of it, has to be made formal. Though this price may be high, there are potentially large benefits to be gained in the testing phase. We do not consider the practical use of proof tools a viable option yet, though there could be special situations in which it is.

ISpec focusses on the specification of interfaces as encountered in component technologies such as COM and CORBA. Compared with classical APIs, such as Win32, OpenGL, etc. which are monolithic and static, these interfaces are typically small and dynamic. Furthermore, we generally have to deal with a group of mutually related interfaces, a so-called *interface suite*, rather than a single interface. This focus on component interfaces is not really a restriction since a classical API can be seen as a special case of an interface suite (with one element).
2 Main Features of ISpec

2.1 Contractual View of Interface Specifications

The idea of using a contractual paradigm in software development has been advocated by many researchers, in particular by Meyer [12]. The contractual paradigm is fundamental to the ISpec approach. An interface specification in ISpec is viewed as an unsigned multi-party contract between providers of services and users of services, as will be further explained below. It contains, among other things:

- An identification of the services, interfaces and parties involved in the contract. This serves primarily to introduce names and types for the services, interfaces and parties, leaving the entities themselves as yet undefined.
- A definition of the relevant terms and concepts. This can be seen as an information model defining the necessary vocabulary for the contract. In practice, the definitions may be drawn from a more comprehensive domain model.
- A definition of the rights and duties of each party. In specifications these rights and duties are formulated using terms such as: requires and provides, relies and guarantees, assumes and ensures, precondition and postcondition, etc.

The contract is multi-party because several parties may be involved in the contract, and not just one or two. This is due to the fact that component interfaces, unlike monolithic APIs, are generally small and dedicated to a particular aspect of a service. The overall service is defined by a collection of interfaces, also called an interface suite. Because of mutual dependencies, the interfaces in a suite cannot be separately specified. Since each interface has at least a provider and a user, this leads to multiple parties in the contract. An example of an interface suite is the collection of ‘connection point’ interfaces from Microsoft COM, defining a generic notification service. The interfaces and parties, and their main dependencies, are indicated in the UML class diagram in Figure 1.

The contract is unsigned in the sense that the parties identified in an interface specification are abstract, i.e., they are roles and not physical entities. In ISpec specifications these parties are represented as abstract object classes. In reality, the roles are played by concrete parties that implement (their part of) the contract. These concrete parties are usually defined as object classes in programming languages such as Java and C++. In order to determine whether an interface specification as a whole has been correctly implemented, we must first associate a concrete party with each abstract party identified in the contract. That is, the contract must be signed; see Figure 2.

Note that a concrete party can play several roles, as a provider as well as a user of interfaces, and that it can be involved in several contracts. Note also that the provider and the user of an interface are treated in the same way in ISpec in the sense that both may be subject to specification constraints, and not just the provider of the interface.
2.2 Incremental Levels of Detail and Formality

One of the explicit objectives of ISpec is to support interface specifications at different levels of detail and formality. In order to enable a smooth transition from one level to the next more detailed or formal level, these levels have been chosen so that they are \textit{incremental}. This implies that a level $n + 1$ specification is obtained from a level $n$ specification by \textit{adding} new sections to the specification. The existing sections retain their function, though their contents may be adjusted to the newly added detail. For example, adding a UML class diagram to an informal specification can enable an informal requirement to be made more precise by phrasing it in terms of the entities and attributes occurring in the diagram.
In ISpec six specification levels are identified, as characterised below:

1. **IDL level**: Defines the *signature* of the interfaces in terms of some IDL (Interface Definition Language). Though most IDLs have (very) limited ways of expressing semantic attributes of interfaces, they basically define the syntax of interfaces only.

2. **Summary level**: Adds *operation summaries* to the interfaces, informally and succinctly describing the effect of the operations in the interfaces. This style of specification corresponds more or less to the current state of practice.

3. **Model level**: Adds an *object-oriented model* to the specification, typically in UML, with abstract object classes representing the parties in the contract. The effect of the operations in an interface is described informally in terms of the attributes of the abstract class providing the interface.

4. **Action level**: Adds *action clauses* to the operations, defining the effect of the operations in a pseudo-algorithmic way. This corresponds to an operational style of specification similar to pseudo-code.

5. **Pre & Post level**: Adds *preconditions* and *postconditions* and other declarative elements such as invariants to the operation specifications. The action clauses are still used at this level, but they are usually more abstract than at the action level (see Section 2.5).

6. **Formal level**: Adds *formal text* and turns the interface specification into a complete formal specification. This level of formality is not normally used in practice, unless formal verification or automatic test generation is required.

The above gives a rough description of each level only. A simple example demonstrating the above levels will be presented in Section 4.

### 2.3 Template-Based Approach

Rather than providing a concrete syntax for interface specifications, ISpec defines a *document structure* which is essentially a tree structure of *templates*, such as templates for specifying data types, classes, operations, etc. These templates are similar to the kinds of templates used in object-oriented methods such as Catalysis [17]. The levels of detail discussed in Section 2.2 correspond to incremental levels in the tree structure.

The ISpec document structure identifies *logical components* of specification documents. The corresponding *physical* documents will generally have a project-dependent structure and will contain additional components not defined by ISpec. These documents may appear in various representations such as MS Word or Rational Rose format. A part of the logical document structure is shown in Figure 3 using UML class diagrams as a kind of abstract syntax.

Informal and formal text occurs at the leaves of the document tree structure only. This raises the immediate question what language to use for the formal text and what the semantics of a specification structured like this is. This issue will be discussed in Section 3.
2.4 Abstract Modelling as a Basis

Specifying a system by means of an abstract model, also called abstract modelling or model-oriented specification, amounts to defining an ‘abstract implementation’ of the system. A concrete implementation is said to conform to such a specification if its observable behaviour is consistent with that of the abstract implementation.

The abstract model differs from a real implementation in the use of programming variables with abstract types such as sets, relations, functions, etc., and the use of declarative constructs such as pre- and postconditions rather than code to specify operations. One of the first formal methods supporting model-oriented specification was VDM [10].

ISpec uses abstract modelling as the basis of interface specifications. The first reason for this choice is that it constitutes a good compromise between providing a high level of abstraction and providing a high level of intuition. For software developers it is generally easier to think in terms of constructive models than in terms of non-constructive properties, even though, or maybe because, the use of models may introduce implementation bias in specifications. The second reason is that the technique fits in quite well with object-oriented modelling techniques, making it easier to bridge the gap between formal techniques and UML.

An abstract model is defined in an ISpec interface specification by representing each party in the contract by an abstract object class and associating each interface type with the object class representing the provider of the interface. This implies, among other things, that each object class may have zero, one, or more interfaces and that in a real implementation two different abstract ob-
ject classes may correspond to the same concrete class, playing two roles at the same time. For each abstract object class an internal representation is chosen in terms of abstract state variables, and the required and provided behaviour of the parties is specified in terms of the abstract state variables.

The approach described above is very similar to the way interface behaviour would be modelled in UML, but there are some differences. The first is that the current version of ISpec does not use UML’s assertion language OCL, mainly because OCL lacks formality and expressivity. Instead, the current version of ISpec uses the assertion language and type mechanism of Z [18], in a way similar to Object-Z [14]. The intention is to make ISpec independent of the assertion language used; see Section 3.

The second difference has to do with the ‘impressionistic’ way UML is used in practice. UML models are typically defined by drawing lots of diagrams highlighting various aspects of the model without making the model itself explicit; see Figure 4. ISpec is based on an ‘expressionistic’ way of modelling in the sense that the UML diagrams are used as an expression of an explicit model, thus making it a lot easier to guarantee consistency. This raises the question exactly what that model is, an issue that will be addressed in Section 2.6. The above is not to say that the impressionistic approach has no value; both styles of modelling are useful but they serve different purposes (see Section 2.7).

![Fig. 4. Impressionistic versus Expressionistic Modelling](image)

### 2.5 Mixed Declarative/Operational Style

In specifying the effect of operations ISpec uses an extended form of the pre- and postcondition technique based on the extensions described in [11]. An operation specification is split into effect specifications specifying the effect of the operation under different circumstances defined by preconditions. An effect specification can be seen as a mini-contract between the caller and the callee of an operation. The structure of an effect specification and its interpretation as a mini-contract is indicated in Figure 5.
The part of the effect specification that is different from a classical pre- and postcondition specification is the action clause. The action clause specifies a collection of allowed state transitions in an operational way, using abstract and often non-deterministic algorithmic constructs. This collection of state transitions acts as a kind of upper limit for the nondeterministic behaviour of the operation. The pre- and postcondition clauses are used to further constrain this set of state transitions in a declarative way.

The combined use of the action clause and the pre- and postcondition clauses allows an operation to be specified in a mixed declarative/operational style, which is important for a number of reasons. First of all, certain aspects of operations are inherently operational and very hard to specify declaratively. This applies for example to the call-backs performed by an operation. Call-backs are a phenomenon frequently encountered in component-based software. Secondly, some operations can simply be specified more naturally in an abstract algorithmic way than in a purely declarative way. This applies particularly to control-like operations. Finally, the action clause can be generalised in such a way that it allows the specification of non-atomic operations, such as blocking operations and operations with internal interaction points.

The ratio between declarative and operational content in an operation specification can in principle be chosen arbitrarily. One extreme is that the action clause is only used to indicate which variables may be modified by the operation, and the precondition and postcondition are used to define the effect of the operation. This corresponds to the classical use of pre- and postconditions as in VDM-SL [10] or the Larch behavioural interface specification languages [8]. The other extreme is that the action clause is a piece of pure program code and that the precondition and postcondition are omitted. The example below is somewhere in the middle of the spectrum, specifying an operation that increments a counter and notifies the objects in the set subscribers after every 100 ticks. Since in this case the operation specification consists of a single effect specification, the effect specification is identified with the operation specification:

```plaintext
operation tick()
pre true
action modify counter
  ; if counter = 0 mod 100 then for s : subscribers do s.notify()
post counter = counter' + 1
```
Here the for-clause iterates over the elements of a set in an unspecified order. We note that problems with re-entrancy of call-backs, such as \texttt{s.notify()} modifying the value of \texttt{counter}, can be avoided in ISpec in a simple and natural way. ISpec uses the \textit{closed world assumption} in the sense that (abstract) state variables introduced in an interface specification can be modified by means of the mechanisms identified in that specification only. Any call-back interfaces used by the operations being specified are part of the interface specification and any possible effect of their operations on the state variables should be specified as part of the interface specification. Clearly we do not want \texttt{s.notify()} to modify the state variable \texttt{counter} of the notifying object, which can be specified in the specification of the call-back interface containing the \texttt{notify()} operation.

### 2.6 Transition Systems as Semantic Model

One of the main objectives of ISpec is to support interface specifications at different levels of formality. In order to maintain consistency between the informal and semi-formal parts of a specification on the one hand, and the formal parts on the other hand, it is essential that both are based on the same semantic model. This means that the semantic model must be both intuitive and formal, which is not an obvious combination. ISpec uses the transition system model \cite{16} as its underlying semantic model, which is believed to satisfy these requirements. This will be further explained and motivated below.

Transition systems have a long-standing reputation as semantic models for concurrent systems. In its basic form a transition system consists of a collection of states, an initial state, a state transition relation defining which state transitions \textit{may} occur, and a set of progress rules defining when state transitions \textit{will} occur. The progress rules can be formulated in various ways, such as very abstractly, based on some notion of \textit{fairness}, or very concretely, based on some notion of real-time \textit{triggering}.

The basic transition system model is very simple and easy to explain. Besides that, it has other advantages, such as:

- it fits in with traditional finite state black-box testing techniques \cite{1};
- it can be used to unify the concepts of callable operation and autonomous transition, leading to a uniform treatment of both;
- it helps in suppressing ‘control-bias’ in specifications by the absence of ‘loci of control’.

The only problem is that the basic transition system model is too austere and unstructured. For specifications of component interfaces we need an object-oriented semantic framework. Fortunately, creating such a framework is mainly a matter of extending the transition system model without affecting the essentials of the model. A short reconstruction of the enhanced transition system model used in ISpec is given below. The enhanced model is obtained by taking the basic transition system model and successively adding the following concepts:
- **Classes**: introduce collections of objects and provide a natural way of structuring states, transitions, and progress rules. State is aggregated in the instance variables of objects, transitions are aggregated in the operations of objects, and progress rules are aggregated in the activities of objects.

- **Interfaces**: introduce information hiding by defining the external access mechanisms of objects. ISpec interfaces contain operations only and no instance variables, so instance variables are always hidden.

- **Observability**: defines which part of the behaviour of the model is externally observable. This is required to define conformance. Observable behaviour is not the same as behaviour observable at the interfaces, because behaviour may also be observable by other means, such as connected hardware represented by certain classes in the model.

- **Factories**: are special classes providing constructors for objects. ISpec classes, like component classes in COM, do not have static methods. Constructors are considered normal operations of objects, creating a chicken-and-egg problem that is solved by factories. Factories are special in only one sense: they do not have explicit constructors.

The concept of abstract model discussed in Section 2.4 should not be confused with that of transition system. Abstract models are syntactic entities, like programs, while transition systems are semantic entities. The link between the two is that abstract models corresponding to ISpec interface specifications describe transition systems. Due to the presence of declarative constructs in abstract models, the transition system described by an abstract model need not be unique. The meaning of an ISpec interface specification, therefore, is defined as the collection of all transition systems described by the abstract model.

Note that the above requires an adjustment of the notion of conformance. An implementation conforms to an ISpec interface specification if its observable behaviour is consistent with the observable behaviour of at least one transition system described by the specification; see Figure 6.

![Fig. 6. Conformance to an ISpec Specification](image-url)
2.7 Separation of Modelling Concerns

ISpec is essentially an approach to developing an interface specification document. Such a document is the result of an interface engineering activity and is usually released at the time the interfaces are released to customers. Besides a contractual function, this document also has an introductory and explanatory function. These two functions require different perspectives in the presentation of the interfaces and are dealt with in separate sections of the document referred to as the contractual model and the conceptual model.

The purpose of the conceptual model is, among other things, to

- introduce the important terms and concepts necessary to explain the functionality associated with the interfaces;
- explain the purpose and functionality of the interfaces;
- provide the link with the interface requirements;
- introduce the outlines of the abstract model.

In the conceptual model the primary concern is providing the proper intuition, rather than accuracy and completeness. The impressionistic style of modelling discussed in Section 2.4 is the most appropriate in providing that intuition, using UML diagrams showing various aspects of the interface functionality and deliberately ‘forgetting’ certain details of the abstract model.

Besides separating the conceptual aspects of interface specifications from the contractual aspects, it also makes sense to separate the technology-independent aspects from the technology-dependent aspects. By ‘technology’ we mean the programming language and/or component technology in which the interfaces are supposed to be used. The contractual model is therefore split into two parts:

- a technology-independent part referred to as the specification model, specifying the interfaces in terms of a neutral Java-like class model;
- a technology-dependent part referred to as the technical model, mapping the technology-independent specification to the component technology and/or programming language used. It defines the representation of the interfaces in terms of the technology used and the relation between the technology-independent and the technology-dependent concepts associated with the interfaces.

Note that the technical model can be defined by a generic mapping, requiring only one definition that can be used for all interfaces defined in a project. This is similar to the way some automatic Java to COM interface mappings are defined at the programming language level.

The main advantages of the technology abstraction used in the specification model are:

- Clarity: Specifying interfaces in terms of technology-oriented concepts clutters up interface specifications, which is now avoided.
- Reuse: The same technology-independent specification can be used for various incarnations of the interfaces in, for example, COM, CORBA, Java or C++.
The main document structure of an ISpec interface specification is indicated in Figure 7, showing the separation into conceptual, specification and technical models, as well as other parts of the document structure.

![Diagram showing ISpec Main Document Structure](image)

**Fig. 7. ISpec Main Document Structure**

### 3 Component-Based Specifications

ISpec is meant to be an *open* approach to the development of interface specifications. This is reflected by, for example, the absence of bias towards a particular component technology or programming language. There is one question, though, that seems inevitable: which specification language should be used for the formal parts of an ISpec interface specification? It is one of the longer term goals of ISpec to turn this question into a non-question and provide openness with respect to the specification language used as well. The key to solving this problem lies in the basic idea of component technology and interfaces itself.

Component technologies with a binary interface standard such as COM are open with respect to the programming language used to implement components. Components can be implemented in any language and which language is used is irrelevant to the users of components. The interfaces shield the details of the implementation language from the users of the components. We can use the same idea at the specification level, as will be explained below.

The issue which specification language to use becomes important at the ‘leaves’ of the ISpec document structure, at those points where an assertion, expression, etc. is expected. Examples of such leaves are the precondition, action and postcondition clauses discussed in Section 2.5. Ideally we would like to be able to ‘plug in’ descriptions written in different specification languages at these points;
see Figure 8. Viewing the plug-ins as components and using the analogy with component technology, it follows that this requires the definition of interfaces that shield the details of the specification language used in the plug-ins.

![Diagram of Interface Specification Document]

Fig. 8. The Idea of Specification Plug-ins

What should the interfaces of specification plug-ins look like? First of all, the interface of a plug-in should be bidirectional since it provides information to as well as requires information from the environment into which it is inserted. Since plug-ins are essentially language constructs, such bidirectional interfaces are most conveniently represented by sets of synthesized and inherited attributes as in attribute grammars.

The attributes in a plug-in interface can be divided into syntactic and semantic attributes. The values of syntactic attributes are syntactic objects such as identifiers, types, etc. The values of semantic attributes are semantic objects such as states, valuations (mappings from names to values), etc.

Each type of plug-in has its own interface specification, defining the names and types of the synthesized and inherited attributes occurring in the interface as well as possible constraints on the values of the attributes. At a point in the document structure at which a plug-in of a particular type is expected, such as a 'precondition' or an 'action clause', any plug-in inserted at that point should satisfy the interface specification associated with that type.

The component-based specification set-up sketched above leads to a fully compositional approach to the semantics of a multi-language ISpec interface specification. The semantics is defined in terms of the structure of the specification and the semantics of the individual plug-ins occurring in the specification. The semantic definition refers to the attributes in the interfaces of plug-ins only and is thereby fully independent of the specification languages used in the plug-
The semantics of each individual plug-in, on the other hand, is of course determined by the language the plug-in is written in.

The approach described above is still tentative and subject of ongoing research. The current version of ISpec uses a blend of COLD and Z as its formal basis. Essential to the overall approach is the use of a *uniform semantic model* in terms of which expressions from different specification languages can be interpreted and in terms of which the attributes of plug-in interfaces can be defined. The role of this model can be compared with that of the binary standard in COM, although it is of a completely different, semantic, nature. The transition system model discussed in Section 2.6 is the obvious candidate for this model.

4 A Simple Example

4.1 Iterator Interface

In this section we will give a simple example of an interface specification, demonstrating the levels of detail and formality discussed in Section 2.2. We will restrict ourselves to the specification model and (part of) the technical model; see Section 2.7. The example is the specification of the iterator interface IIterator. The service provided through this interface is that of traversing the elements of a finite collection of objects. We distinguish three parties in the contract: IteratorFactory, Iterator and IteratorUser, whose roles are indicated in the UML class diagram shown in Figure 9.

`Fig. 9. Iterator Interface and the Parties Involved`

4.2 IDL Level

The technology-independent IDL-level specification of the iterator interface and a possible technology-dependent representation of the interface in Microsoft COM IDL are indicated in Figure 10. The former is part of the ‘specification model’ and the latter is part of the ‘technical model’ of the iterator interface. The technical model should, among other things, define what the precise relation between the two representations of the iterator interface is. For a partial answer, resolving the question mark in Figure 10, see [2], Chapter 14. The technical model will be ignored in the rest of the example.

The IDL ‘specification’ of IIterator raises many questions such as

- Should the collection of objects be traversed in a particular order?
- Are duplicates in the collection allowed?
What happens when all objects in the collection have been traversed?
What is the initial state of an iterator?
How do iterators come into existence?
What if the collection of objects is modified during the iteration?

It is clear that these questions can only be answered by adding more semantic detail to the specification.

4.3 Summary Level

The summary level introduces a short informal description for each operation. We will not give the complete specification, but restrict ourselves to the operation specifications indicated in Figure 11.

Note that the initial state of an iterator, though not specified in the operation summaries, could be specified in another part of the interface specification.

4.4 Model Level

At the model level a UML model is added; see Figure 12. Note that the UML model includes a factory for iterators. It has a single operation that is not part of any interface but is used only to model the creation of iterators and specify their
required initial state. Whoever creates an iterator should meet the constraints specified for the \texttt{create} operation.

The UML model allows the operations to be described in more accurate terms; see Figure 13. The UML ‘role name’ \texttt{elems} in the UML model is interpreted as a sequence of objects in the Z sense. The notations \texttt{elems(i)} and \#\texttt{elems} are used as in Z to denote the \texttt{i}-th element of \texttt{elems} and the length of \texttt{elems}, respectively (sequence indices start at 1).

\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig12}
\caption{Iterator Model}
\end{figure}

\begin{figure}
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Signature} & \textbf{Signature} \\
\texttt{Iterator create()} & \texttt{Object next()} \\
\textbf{Summary} & \textbf{Summary} \\
Returns a new & Increments \texttt{index} and \\
iterator whose & returns \texttt{elems(index)} \\
\texttt{index} is 0. & if \texttt{index} is less than \\
& the length of \texttt{elems}. \\
\hline
\texttt{Bool atEnd()} & \texttt{void reset()} \\
\textbf{Summary} & \textbf{Summary} \\
Returns true if \texttt{index} & Sets the value \\
is equal to the length & of \texttt{index} to 0. \\
of \texttt{elems}, otherwise & \texttt{false}. \\
false. & \\
\hline
\end{tabular}
\caption{Model Level Specification}
\end{figure}

Note that the summary of \texttt{create} does not specify what the value of \texttt{elems} is for the newly created object. This is deliberate, because the value of \texttt{elems} is determined by the creator of the iterator.

\subsection{4.5 Action Level}

At the action level the effect of operations is specified in ‘structured English’ using constructs such as ‘Let …’, ‘Set …’, ‘Return …’, ‘If … then … else …’, ‘While … do …’, etc. Of course, in this simple example there is not much algorithmic detail and no control structures are used; see Figure 14. Bullets and indentation are used to make action clauses more readable. The bullets in action clauses amount to sequential composition and indentation is typically used in control structures to reduce the number of parentheses.

\subsection{4.6 Pre & Post Level}

The pre & post level introduces pre- and postconditions while reducing the role of action clauses. The action clauses are typically used to specify the raw side-effects of operations only; see Figure 15. Bullets and indentation are used in pre-
and postconditions in the same way as in action clauses to improve readability, except that a bullet in a pre- or postcondition amounts to conjunction. The standard keyword `result` is used to denote the result returned by an operation. Quotes are used in postconditions to refer to the old values of variables.

4.7 Formal Level

In turning an ISpec specification into a formal specification we should distinguish between the structure of an ISpec specification and the contents of that structure, i.e. between the ‘branches’ and the ‘leaves’ of the tree structure defined by the specification. The structure of an ISpec specification is itself already formal. Starting from a pre & post level specification such as the one in Figure 15 there should be no reason to change that structure. The contents of the structure, on the other hand, may be informal or semi-formal. Some parts are meant to stay
informal, such as the summaries of operations, but other parts have to be made formal to turn the specification as a whole into a formal specification with a well-defined semantics.

As discussed in Section 3, our intention is to ultimately support different formal languages at the leaves of an ISpec specification. For now, we will restrict ourselves to the COLD/Z mix used in the prototype ISpec implementation. The transformation to a formal specification is not very interesting in this simple case, but for completeness we give the formal specifications of the operations in ASCII ISpec syntax anyway; see Figure 16. These specifications assume that (in another part of the interface specification) the instance variables of objects of type \textit{Iterator} have been formally declared like this:

\begin{verbatim}
  elems : seq Object
  index : Nat
\end{verbatim}

\begin{verbatim}
operation Iterator create()
  summary
    Returns a new iterator.
  pre  true
  action let it == new Iterator
  post it.index = 0
    & result   = it

operation Object next()
  summary
    Returns the next object in the collection.
  pre  index < card elems
  action modify index
  post  index = index' + 1
    & result = elems(index)

operation Bool atEnd()
  summary
    Returns true if all objects in the collection have been traversed, otherwise false.
  pre  true
  action skip
  post result = (index = card elems)

operation void reset()
  summary
    Resets the iterator to its initial state.
  pre  true
  action modify index
  post  index = 0
\end{verbatim}

Fig. 16. Formal Level Specification

5 Related Work

ISpec is an eclectic approach. It can be seen as an attempt to put together a number of well-known and proven ideas and techniques to obtain a practical and consistent framework for interface specification. Some of the key inputs to ISpec and related approaches are mentioned below.

ISpec is permeated with the contractual view to interface specification. This view is closely related to the idea of ‘design by contract’ [12], with two main differences. The first is that ISpec does not focus on design but on the contract itself. An ISpec specification is a contract; see Section 2.1. The second is that
ISpec contracts are inherently multi-party and not just contracts between a caller and a callee (called mini-contracts in ISpec), making them very similar to (design) patterns (cf. [9]).

The multiple levels of formality and detail in ISpec are inspired by the multiple levels of abstraction provided by wide-spectrum languages such as VDM-SL [10] and COLD [4]. For its expression and assertion language, the current version of ISpec uses a combination of language elements from COLD and Z [18].

The separation of modelling concerns as incorporated in ISpec goes back to the ‘modelling perspectives’ introduced in [3]. Following the terminology used in [6], the conceptual and specification perspectives correspond to the conceptual and specification models of ISpec; see Section 2.7. The implementation perspective corresponds partly to the technical model of ISpec because the latter deals only with the representation of interfaces in the implementation technology used.

The approach that comes closest to the model-oriented way interfaces are specified in ISpec is probably Catalysis [17], though Catalysis is a complete software development method, which ISpec is certainly not. The notion of collaboration framework in Catalysis corresponds more or less to an interface specification in ISpec, and Catalysis ‘types’ correspond to the abstract object classes associated with interfaces in ISpec. The main difference is that in ISpec the collaborations follow from the specification of the types, while in Catalysis they lead a life of their own. (Referring to the terminology introduced in Section 2.4, Catalysis uses an impressionistic style of modelling collaborations while ISpec uses an expressionistic style.)

6 Conclusion

In this paper we have given a survey of the main features of ISpec. Several more detailed technical features have not been discussed. They include features dealing with aspects of interface specifications such as exception handling, object lifetime, asynchronous behaviour, non-atomic operations and multi-threading.

The ISpec approach, as indicated by the title of this paper, is intended to be both practical and sound. The practicality of ISpec is being validated in software development projects at Philips, where the approach is used in combination with standard CASE tools. The focus in these projects is on the more informal usage levels of ISpec. This is in line with the idea of the approach to provide an easy entry to the development of better and more rigorous interface specifications. An ISpec course is part of the regular technical training program at Philips.

The formal part of ISpec is currently being used in experiments with automatic black-box test generation techniques. This is done by compiling sufficiently constrained formal ISpec specifications to transition systems encoded in $\mu$CRL [7] and using the $\mu$CRL tool-set to derive test traces. These traces are then used to test implementations of the interfaces against their corresponding transition systems.

As to the ‘soundness’ of ISpec, further research is necessary to develop the uniform semantic model that can act as the basis of a language-independent ver-
sion of the approach. Other issues that deserve attention are a better integration of ISpec and CASE tools, support for timing aspects in interface specifications, more guidance with respect to the design of interfaces, and CSpec, the complementary approach to component specification. A number of these issues are currently being addressed in a recently started research project at the Eindhoven University of Technology in cooperation with Philips Research.

References

Cooperation of Formal Methods in an Engineering Based Software Development Process

Yamine Ait-Ameur

LISI/ENSMA BP 109 Site du Futuroscope. 86960 FUTUROSCOPE CEDEX. FRANCE,
yamine@ensma.fr Tel : +33 5 49 49 80 66, Fax: +33 5 49 49 80 64

Abstract. This paper presents a global development approach for using different formal techniques in a common software development. The underlying methodology is based on the identification, formalization and verification of the properties, of the system to be developed, expressed in the requirements. The approach we suggest consists in identifying a main formal technique to support the whole system specification at an abstract level and one or several secondary techniques that prove efficient for parts of the development. This approach has been put into practice in different and distant application domains. The considered application domain covered by this paper is human-computer interaction software.

Keywords: formal techniques, cooperation of techniques, development methodology, human-computer interaction.

1 Introduction

It is well accepted that the development of safe and secure software requires the use of rigorous and mathematically based development techniques. These techniques shall ensure that the developed software satisfy a set of suited properties expressed in the requirements. The use of formal techniques appears as a solution for the specification, development, validation and verification of software. The processes resulting from these different activities shall be be mastered and controlled by the developer.

In order to organize the use of formal techniques, it is necessary to have a methodology or a model that shows when and how one or several formal techniques need to be jointly applied. By defining a methodology or a model, we mean the definition of a global approach, independent from any particular application, that express methodological rules which show when and how one or several formal techniques can be used.

Formal techniques shall allow the capability to perform formal reasoning in order to prove that the developed programs satisfy all the properties expressed in the requirements. Moreover, establishing properties on programs requires the definition of formal systems, supported by formal techniques, with formal semantics and a proof system. These systems allow to express programs, specifications and properties and to prove that specifications and programs satisfy
these properties. However, each formal technique is usually applicable and/or efficient in part of the development to be achieved only.

Our claim is that for each program development there is a need to use a formal technique in each part of this development where this technique is efficient. Proceeding this way leads to:

- the use of several formal techniques in a common program development,
- study how the verification of properties, in a given formal technique, can be preserved in another formal technique.

The last point shows that the problem of integrating several formal techniques is related to the preservation of the set of properties of a given program from a technique to another one and from a development step to another one. Indeed, properties are the basis of the choice of a given formal technique allowing to prove and to preserve them during the program development.

The work presented below proposes a methodology which allows to use several formal techniques in order to prove program properties and to preserve them during the development process. It does not define any new formal technique but it suggests to use existing formal techniques jointly and to make them to cooperate. This methodology has been applied in several application domains like: non functional properties of programs [4] [5] [6], human-computer interaction [13][12][10], computer aided design databases [49][14][8][52] etc.

This paper is structured as follows. Next section gives an overview and a chosen classification of formal techniques applied in software development and related tools. Section 3 gives the motivations for proposing such a development model to integrate formal techniques. Then, section 4 exposes the details of the suggested development methodology. It shows criteria to select techniques and a sequence of steps to be applied while developing software according to the proposed approach. In section 5, we show how this development model can be applied for different development approaches, including informal and formal ones. Finally, in section 6, this approach is put into practice in the human-computer interaction software development area and a conclusion is given in section 7.

2 Formal Techniques

A set of formal techniques allowing verification, validation, development and maintenance of software have been suggested during the last thirty years. They support different phases of the software life cycle.

Formal techniques can be distinguished with different points of view: defined goal (development, or verification), underlying proof system (logic based or type control based), associated formal semantics (state based semantics, algebraic semantics), implementation of the proof system (incremental proof system, fully automated proof system), approach (a priori or a posteriori) etc. These different points of view make difficult the choice of one or several formal techniques.
Therefore, according to these points of view, several formal techniques taxonomies have been suggested in the literature \[30\] \[51\]. In the rest of this section, we give a review of formal techniques based on the nature of the proof system: incremental or fully automatic. More details about these classifications, formal techniques and tools can be found in \[7\].

### 2.1 Incremental Development Techniques

Incremental development techniques handle operations that allow to express and to refine a given specification into a program, by ensuring that a set of properties is preserved and verified. Examples of such operations are refinements, transformation rules, tactics, abstract data type implementation, proof steps etc. The development is built incrementally by combining these operations. Two main categories of incremental techniques can be distinguished:

- **Model oriented formal techniques** based on the definition of a set of variables defining a state. Logical formulas are used to describe a state and properties on this state. A proof system and a set of refinement rules are associated to this description. Several techniques belong to this category VDM \[17\], Z \[54\], B \[2\] etc. They are distinguished thanks to their semantics (Hoare Logic, Dijkstra’s logic etc.). These techniques are a posteriori techniques i.e. the developer exhibits the refinement and then he proves its correctness.

- **Transformational or synthesis techniques**, are a priori techniques, which usually use algebraic specifications \[57\], are based on a small set of operations, already proved to be correct, that allow to transform a given specification into another one. The sequence of operations applications defines a formal transformational development. Approaches like the Fold/Unfold transformation system of Burstall and Darlington \[20\], program synthesis of Manna and Waldinger \[44\] were the first approaches to appear, then other ones like \[41\] \[25\] \[38\] \[47\] \[22\] have been proposed.

### 2.2 Model Checking Techniques

The second type of formal techniques concerns the ones based on automatic property verification and program generation. Like for incremental development techniques, the distinction between model checking techniques is also based on the kind of formal semantics supported by these techniques:

- **The first category** is related to the techniques where the model is expressed by a transition system showing how state variables evolve. A decision procedure, based on a transition system traversal, allows to verify a set of properties, usually expressed by temporal logic formulas. These techniques face the state explosion problem but abstraction techniques makes it possible to reduce the number of states. Examples of such techniques and systems are described in \[23\] \[34\] etc.

- **Algebraic semantics** gave rise to model checking techniques through rewrite systems \[28\] which use term algebras as models. Moreover, these techniques support validation thanks to test generation and verification by property rewriting.
The last category consists in building particular interpreters or analyzers which allow the verification of particular properties. This category of techniques gathers static analyzers like abstract interpretation \[26\] or dynamic analyzers for tests or formal debugging for example.

Finally, notice that there exist several behavioral formalisms related either to incremental development techniques or to model checking techniques.

### 2.3 Tools

Like for techniques, several tools have been developed and can be classified from different points of view: implementation of the proofs (semi-automatic, fully automatic), tool capabilities (animation, test, proof, documentation etc.), generic or technique dedicated tools etc. We have chosen to classify the techniques according to the last point of view.

- First, technique dedicated tools are the ones which support a particular formal technique. Examples are the Atelier B \[55\] for B, the VDM-SL \[3\] language for VDM, SMV \[23\] for CTL* logic, LUSTRE/LESAR for the LUSTRE language, LP \[19\], the PLUSS \[16\] language for algebraic specifications, REVE or ORME \[42\] for rewrite systems, etc.

- The second category of tools supports higher order logics or higher order type systems. They are generic tools and they can be “instantiated” to support particular formal techniques. Several tools have been developed in this category, among them COQ \[37\], DEVA \[56\] \[21\], NUPRL \[24\], NqThm \[18\], ISABELLE \[48\], LF \[35\], PVS \[45\], HOL \[32\] etc.

### 3 Motivation for the Integration of Formal Techniques

The previous brief review of formal techniques and tools shows the existence of several different techniques and tools. Moreover, these techniques may be put into practice at different levels of the software life cycle. It becomes crucial to define a set of methodological rules allowing to use one or more formal technique according to the problem to be solved and not according to the deep knowledge of one formal technique a developer may possess. In other words, there is a need of definition of choice criteria for formal techniques, identification of the development phases where formal techniques are used and how different techniques can be simultaneously used in a common development. Moreover, according to the previous descriptions, we discover:

- the presence of several different techniques and tools,
- that formal techniques are designed for different purposes: verification, development, analysis, validation etc.
- that the techniques are self contained and are usually not designed to cooperate with other techniques. This is due to different semantics, different type systems, different proof systems, different supported theories, etc.
- that few methodological rules to use these techniques are defined. They are frequently limited to the user manual of the technique or to some case studies.
Since formal techniques usually cover only a part of the development of a software, it becomes urgent to think to the possibility to define a cooperation model where formal techniques will be capable to cooperate in the same software development process, by restricting the use of a formal technique to each step where it proves efficient. This is our main claim for defining the following cooperative development model.

4 Our Approach

For the rest of our description, we will use the word “object program” to mean either a program, a specification or a formal model.

Our approach is a general one. It is capable to support different development phases, and particularly:

- development, if the studied object programs are specifications,
- property verification, if the studied object programs are models,
- validation, if the studied object programs are programs or executable specifications,
- maintenance and program understanding, if the studied object programs are programs or specifications

4.1 Selecting a Formal Technique

Selecting a formal technique is an important step of our approach, based on:

- the application domain of the problem to be solved,
- formal knowledge of the application domain. By formal knowledge, we mean all the formal notations and techniques used in the application domain. For example, finite elements analysis, interval analysis, formal models for mechanics, physics, electronic, signal processing, etc ...
- development environment and particularly the existing informal notations like OMT, existing software, existing compilers used by the developers,
- expertise of a developer on a particular formal technique, or a particular programming language,
- and finally, the more important, properties of the software product to be developed.

In fact, the selection of a technique shall be motivated by the nature of the problem to be solved, by the application domain and by the properties expressed in the requirements. It is not acceptable to enforce the application of a given technique nor to extend it, if it is not adapted to the current problem. To summarize, a formal technique is chosen if it proves to be the most efficient according to the previous points.
4.2 Main and Secondary Formal Techniques

The suggested general model, to integrate formal techniques, identifies two types of formal techniques, each of them is selected according to the previous selection criteria.

- The main formal technique (MT) is the one used to express the problem and its specifications. It is assumed to be the technique where the maximum of properties can be expressed and checked.
- The secondary formal technique (ST) is a technique selected in order to verify properties of the whole system or of a subset of the whole system. There may exist several secondary techniques put into practice during one program development. Moreover, the use of a secondary technique requires the translation, of the whole or of a subset of the system and of the properties to be checked, from their expressions in the main technique into their corresponding expressions in the secondary technique. This translation shall preserve properties of the system from one technique to another.

4.3 A General Model to Integrate Formal Techniques

In the following description, figures use continuous arrows to show that the carried operations are formally expressed, and dashed arrows to show that the carried operations are either formally or informally expressed. The methodology to integrate different formal techniques can be described as follows:

1. Starting from the requirements, identify all the properties and their possible formalizations that the final system shall fulfill. Requirement engineering techniques from the literature are taken into account at this level [29] [39] [46] [36] [40]. We do not address this last topic in our methodology.

2. According to the selection criteria identified previously, select a formal technique to represent a formal expression of the whole system to be developed and of the whole or of a subset of the identified properties. The chosen technique is considered as the main formal technique (MT).

3. Let OP_p be the object program representing the studied system and P_p the set of properties of OP_p to be checked. Check the properties P_p expressed at step 2:

   3.1. either in the main formal technique according to figure 1,
   3.2. or in a selected secondary technique (ST) according to figure 2. In this case, the object program OP_p and the set or a subset of properties P_p are re-expressed by another object program OP_s and another set of properties P_s. The verification of the properties P_s in ST must ensure that the properties P_p of OP_p are preserved by the translation from one technique to another. When P_s are proved in ST, the translation of the P_s properties from the secondary technique ST into the main technique shall ensure that the properties P_p are proved.

4. If development, refinement or transformation operations on OP_p are possible, then proceed to the development. We obtain a new OP'_p and a new set of
properties $P'_p$. The properties of the original $OP_p$ shall be preserved by the refinement, so, the $P'_p \Rightarrow P_p$ formula shall be checked. The development operation is achieved:

4.1. either in the main technique according to figure 3. The properties are preserved in the main technique,

4.2. or in the selected secondary technique according to figure 4. In this case $P'_s \Rightarrow P_s$ is formally established and the translation from one formal technique to another one must preserve $(P'_s \Rightarrow P_s) \Rightarrow (P'_p \Rightarrow P_p)$.

5. The sequence of development of step 4 is repeated until the satisfaction of the expressed needs and the obtention of the suited system.

The previous steps describe a methodology allowing to develop a system satisfying a set of properties formalized from the requirements expression. In the opposite of classical formal development models, this method encourages the use of several different formal techniques provided that the properties are preserved from one technique to another. Preservation of properties is the basis of method integration. This point is discussed in next section.
5 Integration of Formal Techniques: Three Derived Development Models

As stated previously, the development model we proposed makes it possible to use several formal techniques for the development of a given system. The proposed method is based on property verification by the most efficient technique and on property preservation from one technique to another one and between development levels. It uses formal techniques only.
Starting from figure 4, it is possible to derive three different development models where techniques can cooperate. These three models are defined by relaxing the continuous arrows or by strengthening the dashed arrows of the suggested development model. The following development models are obtained.

a- The “informal model” is obtained when all the lines are dashed. It means that all the steps of development and proof verifications are informally performed.

b- The “formal model” is obtained when all the arrows are continuous. It means that translations are formally described and that properties are formally preserved. This model is the complete formal model with integration of techniques. However, it is not a trivial task to scale up this model to complex development problems.

c- The “engineering model” is the one obtained by figure 4. This model considers that several steps of a development can be formally performed, but it accepts that the links between some formal development pieces remain informally realized.

The informal model is out of the scope of our work on formal techniques. We did not apply it but we show that it can be derived from our methodology.

The engineering model approach is close to the ones applied in engineering. As example, civil engineers use several formal techniques, like finite elements, formal aerodynamic or thermic models etc and they usually do not formally relate them but they exploit the results of each model in their global system. The engineering model is inspired from this approach. Obviously, ideally, the developer tries to reach the formal model where all the steps and the integration of techniques are formally stated.

Finally, as we stated above the engineering model and the formal model have been applied in several real case studies using different formal techniques [7].

6 Experimentation for Human-Computer Interaction Software Development

The previous methodology has been put into practice in several areas of software development. We will review an application of the suggested development model and particularly the engineering model for the development of HCI software using the B formal technique as the main technique and algebraic specifications as secondary technique.

6.1 The Human-Computer Interaction Application Domain

It becomes crucial to give the same attention to the development of the human-computer interface (HCI) of an application as the one given to the development of the functional core of the same application, particularly for critical systems. Indeed, control panels in nuclear reactors, pilot interfaces in planes an so on require a secure and safe HCI allowing the implementation of all the critical user tasks.
According to our development model of section 4.3, the first step consists in an application domain analysis. Like for classical software engineering, several notations and models have been suggested for the development of software in the human-computer interaction area. We have classified these notations into two categories:

- **user notations:** are issued from psychologists and ergonomists. They deal with tasks and tasks analysis and express the external specifications and the user needs. By task, we mean the sequence of basic operations of the human-computer interface allowing to reach a particular goal. Examples of such notations are MAD [53], UAN [50], etc.

- **developers notations:** deal with software development. The human-computer interaction area is equipped with several architecture models like PAC [27], ARCH [15], etc. and several toolkits like X/Motif, Tcl-TK etc. and several model based systems, allowing interface description and code generation like UiM/X [31], FUSE [43], $H^4$ [33] etc.

For all the application domains we investigated, and particularly for human-computer interaction, our approach does not consist to propose a new formal technique. It suggests to keep all the knowhow of HCI developers and to plug formal techniques according to our methodology, described in section 4.3, in order to improve the quality of HCI software development and product and to formally check HCI properties.

By HCI properties, the literature, of this application domain, means:

- **visibility:** relates to the feedback and information perception by the user,

- **reachability:** deals with what can be done with the user interface and how it can be done,

- **reliability:** concerns the way the interface works with the underlying system.

Proceeding this way requires a deep study of the application domain on the one hand and a good understanding of formal techniques for development and verification of software on the other hand. These two points are the two required characteristics for a real transfer of formal techniques from toy examples to real world applications.

Let us consider a subset of a study we conducted for the development of an air traffic controller human-computer interface.

### 6.2 The Case Study

The very restricted version of the case study we describe consists in a set of windows (that may represent planes). Each window can be created (a plane entering the control area), resized (focus on the characteristics of a plane), moved (to move a plane), destroyed (a plane exiting the control area) and so on. Moreover, a mouse is defined in order to perform these operations by direct manipulation.
6.3 The Main Technique: B

First of all, we do not review, in this paper, the B technique and its associated 
generalized substitutions semantics. Full details are given in [2], and we assume 
that the B developments given in this paper are understandable by the reader.

B formal technique is a model oriented technique based on the description, 
in an abstract machine, of a set of state variables, invariant properties on these 
variables and a set of operations which modify the values of the state variables. 
Moreover, abstract machines are combined by programming in the large ope-
rators and can be refined from an abstract level representing specifications to 
the programming language concrete level. The Atelier B [55] tool, we have used 
in our experiments, allows an automatic proof obligation generation and proof 
obligation checking.

According to the selection criteria given in section 4.1, the choice of B as 
main technique is motivated by:

- its capability to support either programs, specifications, proofs and deve-
lopments,
- the possibility to design and to control the development incrementally,
- supporting modular decomposition,
- the availability of a tool,
- and finally, may be the most important, the adequacy of the B specification 
  decomposition with the informal developer notations used in HCI development 
  context. Indeed, the ARCH model, we have used, decomposes the whole ap-
lication into a set of modules that are positioned on an arch going from the 
functional core on the left to the toolkit on the right with intermediate modules 
in between.

6.4 Specification

As an illustration, we have chosen to show part of the specification of the mouse 
management. A mouse is defined by attributes: \( x \) and \( y \) positions on the screen, 
a state showing that it is either \textit{up}, \textit{down} or \textit{clicked} and a Boolean attesting that 
the mouse is actually created by the system. In B, these attributes are represented 
by the functions \( \text{x\_pos\_mouse} \), \( \text{y\_pos\_mouse} \), \( \text{mouse\_state} \) and \( \text{mouse\_creation} \). 
The mouse is created in a set whose number of elements shall be less or equal to \( 1 \).

Two operations related to the mouse management are described:

- \textit{create\_mouse} allows to create a mouse of type \textit{MOUSE} and adds it to the 
  set \textit{mouse} at a predefined coordinate with an \textit{up} state. Its B specification is 
given by:

\[
\begin{align*}
\text{mm} & \leftarrow \text{create\_mouse} = \\
& \text{PRE} \\
& \quad \text{mouse} \neq \text{MOUSE} \\
& \text{THEN} \\
& \quad \text{ANY} \\
& \quad \text{tt}
\end{align*}
\]
WHERE
\[ tt \in \text{MOUSE} - \text{mouse} \]
THEN
\[ mm := tt \parallel\]
\[ \text{mouse} := \text{mouse} \cup \{tt\} \parallel\]
\[ x_{\text{pos}}(\text{mm}) := x_{\text{mouse\_default}} \parallel\]
\[ y_{\text{pos}}(\text{mm}) := y_{\text{mouse\_default}} \parallel\]
\[ \text{mouse\_state}(tt) := \text{up} \parallel\]
\[ \text{mouse\_creation}(tt) := \text{TRUE} \]
END
END

- \text{move}\_\text{mouse\_with\_drag} allows to move a mouse to a new position with a state at \textit{down}. This operation requires that the new \(xx\) and \(yy\) coordinates are in the screen limits and that the mouse state is \textit{down}. Its B specification is given by:

\[
\text{move}\_\text{mouse\_with\_drag}(mm, xx, yy) = \\
\text{PRE} \\
\quad mm \in \text{mouse} \land \\
\quad xx \in \text{NAT} \land xx \in 1..\text{max}\_\text{mouse\_wide} \land \\
\quad yy \in \text{NAT} \land yy \in 1..\text{max}\_\text{mouse\_high} \land \\
\quad \text{mouse\_state}(mm) = \text{down} \land \\
\quad \text{mouse\_creation}(mm) = \text{TRUE} \\
\text{THEN} \\
\quad x_{\text{pos}}(\text{mm}) := xx \parallel \\
\quad y_{\text{pos}}(\text{mm}) := yy \\
\text{END}
\]

Safety properties are expressed by an invariant property. The \textit{INVARIANT} B clause of the previous specifications contains the following property:

\[ \forall mm \in \text{mouse}, (x_{\text{pos}}(mm)) \in 1..\text{max}\_\text{mouse\_wide} \land \\
\quad yy_{\text{pos}}(mm) \in 1..\text{max}\_\text{mouse\_high}) \]

It states that each window (plane) of the screen is reachable (meaning that each plane in the control space area is reachable by the air traffic controller). This is one of the safety properties related to reachability and reliability properties, in human-computer interaction classification, we have checked.

6.5 Refinement

Refinement from specifications to code is conducted in two steps.

Refinement of Operations. The two previous operations are refined to:

- \textit{create\_mouse} by introducing the \textit{begin end} bloc, the local variable \textit{VAR}, the sequence and the \textit{if then else} conditional substitutions.
\[
\text{create\_mouse =}
\begin{align*}
&\text{BEGIN} \\
&\text{VAR} \\
&\hspace{1em} tt \text{ IN } tt \in \text{MOUSE} - \text{mouse} \\
&\hspace{1em} IF \hspace{1em} (\text{mouse} \neq \text{MOUSE}) \\
&\hspace{1em} THEN \\
&\hspace{2em} \text{mouse} := \text{mouse} \cup \{tt\}; \\
&\hspace{2em} x_{\text{pos\_mouse}}(mm) := x_{\text{mouse\_default}}; \\
&\hspace{2em} y_{\text{pos\_mouse}}(mm) := y_{\text{mouse\_default}}; \\
&\hspace{2em} \text{mouse\_state}(tt) := \text{up}; \\
&\hspace{2em} \text{mouse\_creation}(tt) := \text{TRUE}; \\
&\hspace{1em} \text{mm} := tt \\
&\text{END} \\
&\text{END}
\end{align*}
\]

- \text{move\_mouse\_with\_drag} by introducing the \text{begin end} bloc, the sequence and the \text{if then else} conditional substitutions,

\[
\text{move\_mouse\_with\_drag(mm,xx,yy) =}
\begin{align*}
&\text{BEGIN} \\
&\hspace{1em} IF \hspace{1em} \text{mm} \in \text{mouse} \land \\
&\hspace{2em} xx \in \text{NAT} \land xx \in 1..\text{max\_mouse\_wide} \land \\
&\hspace{2em} yy \in \text{NAT} \land yy \in 1..\text{max\_mouse\_high} \land \\
&\hspace{2em} \text{mouse\_state}(mm) = \text{down} \land \\
&\hspace{2em} \text{mouse\_creation}(mm) = \text{TRUE} \\
&\hspace{1em} THEN \\
&\hspace{2em} x_{\text{pos\_mouse}}(mm) := xx; \\
&\hspace{2em} y_{\text{pos\_mouse}}(mm) := yy \\
&\hspace{1em} END \\
&\text{END}
\end{align*}
\]

For the refinement of operations, all the proof obligations were automatically generated and checked by the automatic prover of Atelier B. At this stage, we reach the programming language level, except that data are still abstract.

\textbf{Refinement of Data.} When the previous refinement is achieved, two problems arise:

- the code reached by the previous refinement does not allow reusability of the code of the existing toolkits. Indeed, HCI software development is based on pre-existing toolkits implementing basic operations to manage mouse, windows, interactions and so on. It is not acceptable, for HCI software developers to design a new toolkit, at each development,

- the implementation of abstract data types describing sets and selectors by the corresponding pre-existing implementation abstract machines of Atelier B, generates a set of \textit{unreadable} proof obligations, making the implementation
phase impossible in B. To solve these two problems, we had to use two secondary techniques:

- reverse engineering techniques in order to synthesize abstract machines representing toolkits i.e. B specifications to represent toolkit software. We have practically described a set of abstract machines to represent, in B, a subset of the Tcl-Tk toolkit. This process is facilitated by the nature of the software describing this toolkit: a set of modules with attributes and functions. The synthesis of an abstract machine followed the existing module decomposition. Nevertheless, the difficulty we faced is the extraction of the relevant properties to be expressed in the specifications. This methodological point shall be solved by HCI software developers.

Therefore, it became possible to refine operations of our problem specifications to reach Tcl-Tk code through a call to the B operations contained in the B abstract machines which abstract Tcl-Tk operations,

- algebraic techniques to implement abstract data types (ADT’s) by concrete data types (CDT’s). We have used the algebraic refinement of ADT’s in order to implement sets by lists and selectors by introducing records. Moreover, thanks to the formal algebraic refinement technique, we formally ensured abstract data types implementation correctness. This work consisted in defining representation functions for the \textit{empty} and \textit{add} sets constructors by the \textit{nil} and \textit{cons} lists constructors.

6.6 Domain Oriented Conclusion

In the context of human-computer interaction software development, the use of the B technique gave several interesting results. This approach handles all the development steps from specification to code generation, property verification and supports an incremental controlled development. Several studies following the proposed development model validated this approach at a large scale \cite{12,13,11,10,9}.

The proposed approach does not suggest any new technique nor the extension of an existing technique. Moreover, it takes into account the existing notations for application domains (architecture model ARCH) and reuses existing software (Tcl-Tk toolkit). It shows that formal techniques are plugged to existing development models and notations and are put into practice at every place where formal verification is required.

Moreover, the developed approach gave rise to several perspectives like:

- considering other notations of the human-computer interaction application domain. Indeed, it is necessary to take into account user notations and tasks analysis models describing the HCI external behavior,

- combining formal validation techniques in order to be capable to test B specifications for implementing user tasks. A current work using the EXPRESS formal specification language is being conducted. It consists in translating B specifications into EXPRESS data models. In EXPRESS all the B operations are expressed by state transformations (from the starting state to the final state). Each state is an EXPRESS entity with attributes and logical constraints. An
operation is full defined by two state entities (start and final). Instances of state entities describe test cases. It is planned to publish this work later [9].

- combining model checking techniques encoding temporal logics with incremental development techniques in order to prove liveness properties representing most of the users tasks and reachability properties in human-computer interaction,

- and finally, define a reverse engineering methodology in order to get libraries of specifications (to represent existing toolkits specifications) with the relevant operators and properties.

6.7 Use of the Integration Model

The suggested solution for studying the development of HCI software follows the development model, precisely the engineering model, we proposed in section 4.3. Properties like the one of section 6.3 have been identified, formalized, proved and preserved. The development followed:

- step 4.1. for refinement of operations and property, verification and preservation, expressed in the unique B main technique according to figure 5.

- step 4.2. for reverse engineering of toolkit specifications and implementation of ADT’s by CDT’s using algebraic formal techniques as secondary techniques. The translation of ADT’s specifications from B specifications to algebraic specifications has not been formalized. This step is assumed to be handled by the
developer, and its formalization is not free from a development cost point of view. However, if required, its formalization is possible. In this case, we are using the *engineering development model* according to figure 6.

![Diagram](image)

**Fig. 6.** Development and verification of part of the properties in the secondary technique.

The *engineering development model* proved useful for integrating different formal software development techniques. It showed that it is possible to use different formal techniques jointly avoiding heavy proof steps to formally establish how these techniques cooperate. However, if these proofs are required, then the cooperation shall be formalized according to the complete formal model derived in section 5.

## 7 Global Conclusion

This paper has shown a development model where one or several formal techniques can be integrated in order to support formal development and verification of programs. The underlying methodology showed that:

- it supports complete or incomplete formal developments thanks to the formal and to the engineering models respectively,
property based reasoning allows to guide software development process and to select a formal technique,
properties of specifications are preserved form one step to another one even if the development is not completely formalized,
several techniques can be put into practice in a common software development. Each technique is chosen with respect to the properties to be checked and to its efficiency to handle these verifications and to solve the current problem.
The development model we have suggested shows that even if an universal or global formal technique does not exist, the use of several techniques, in an organized manner using a precise methodology and the application domain knowhow, leads to the encoding of non trivial and complex program developments.
However, notice that the choice of techniques and the order of application of these techniques remain one of the hardest tasks of the software developer and shall take into account all the expertise, notations and knowhow related to the application domain.
Finally, obviously the main perspective related to the definition of our development model consists in formalizing the greatest number of links or bridges between formal techniques. Regarding the different figures illustrating our methodology this perspective leads to transform all the dashed arrows of figure 4 into continuous arrows. Two approaches allowing to formalize links between techniques are identified. Either by,
- translating one formal technique into another one and prove the correctness of this translation, - or by translating the two formal techniques into a third one, mainly a higher order logic or a higher order type system. The third technique is used to prove equivalence.
Moreover, our claim is that the formalization of these links between techniques shall be done according to the application domain where the development model would be put into practice.

References


Abstract. In this paper we point out the potential of software engineering methods in developing control systems as well as in deriving the control software for them. The general strategy for such a development comprises two phases. An initial system-level specification models both the physical environment and its control software within the same state machine oriented view of the system. The control software can be then extracted from this specification and further refined via correctness-preserving steps. The design strategy is intended for the development of control system components and combines the UML statechart diagrams and the OO-action system formalism in a novel way. We show the applicability of the method using the Production Cell case study.

1 Introduction

Component-oriented design consists in assembling new systems from existing components, as well as in constructing components dedicated to the assembly. The software component term has been coined about three decades ago as a logical solution for the software crisis [18], by analogy with plug-and-play inter-operability available to producers and customers in other engineering fields. Thus, in 1968, McIlroy pointed out that mature industrial engineering builds on components and so should software engineering [18]. Component-oriented programming is still not very spread, but the experience of three decades is now used to ground it as a fundamental software construction method [25].

A component is essentially a black box implementing a set of services provided that a required set of context dependencies are available. A component is thus an abstraction over the space of services of a system. The services that a system has to perform are partitioned into groups, each group of services being implemented by a component. Components interact with each other by using the provided services, and thus, an important issue regarding a component-based system consists in the interactions among the participating components. Rather than designing systems using existing components, in this paper we focus on correctly developing the components and on designing their interaction patterns in the context of control systems.

A control system component usually consists of hardware and software parts that collaborate for achieving a set of services. Good integration techniques for constructing control systems are consequently required from this very basic level.
However, in the context of control systems there are also some other important integration issues to handle.

Control systems are typically embedded into complex assemblies. On one hand, the inherent complexity of a control system requires some scalable engineering methods for modelling purposes. Graphical and diagrammatic specification methods, which yet often lack a formal basis, have turned out valuable in this respect. On the other hand, the correct functioning of the control system with its complicated interrupt handlers is critical, requiring a formal approach to ensure dependability. Thus, in order to model control systems, formal and informal design methods have to be integrated.

In this paper we use the UML (Unified Modeling Language) [7] as the informal specification method. UML becomes the ‘de facto’ standard method for developing object-oriented software, providing a collection of diagrams for specifying, developing, and maintaining software. We focus on UML statechart diagrams, used to describe the behaviour of active objects, as active objects are a suitable model for control system components. For the formal specifications we use the OO-action systems framework [6], an object-based approach to the design of parallel and distributed systems. OO-action systems have their roots in other state-based formalisms like action systems [3] and UNITY [11]. We chose this formalism as it allows us to concentrate on the collaborative behaviour of the system components first and develop the details of the system later, when the overall functionality is specified. In this way we obtain a level of abstraction that is so high that it is easy to reason about the behaviour of the system and yet detailed enough so that the systems can be developed all the way to their implementations within the same framework. The use of these two specification frameworks, informal and formal, for the development of control systems poses a couple of integration questions.

First, we need to address the integration of a diagrammatic specification method with a formalism. This problem has already been investigated, indicating that formalisms are in general not scalable nor easy to use, thus requiring a manageable complement [20]. Integrating diagrams with a formalism has been shown as viable, for instance in [19]. Moreover, as UML is widely used for software construction, defining a formal semantics for its different diagrams became quite important [13]. This is usually achieved via different formalisms and addresses a diverse range of diagrams, see for instance [1,21,9]. There has also been work towards using different techniques, formal and informal, to model different aspects of a system design and to show that these techniques provide a consistent view of the system under construction [8,26]. In this paper we follow the latter approach and use the UML statechart diagrams to give an informal and basic specification for the control system under construction. The formal specification, written using the OO-action systems formalism is usually difficult to initiate. Here it is created from the statechart diagrams. At this stage, both specifications provide the same view upon the control system under development. The formal specification is then improved via a number of refinement steps.
Second, the UML statechart diagrams are an event-based method of specification, while the OO-action systems are a state-based formalism. For using both of them in the development process of a control system, we have to find a suitable way of integrating them. In this paper we concentrate on this second aspect of integration. The ability of OO-action systems in modelling events and event-based mechanisms is based on a common feature shared by them and the UML statechart diagrams. Namely, both frameworks are models for active objects, i.e. objects having an autonomous behaviour. Given this basic property, the two frameworks can be integrated and used together for specifying control systems components. The initial development stage is based on UML statechart diagrams, that are then translated to an OO-action systems specification.

OO-action systems are intended to be stepwise developed from a high-level specification to an implementation via correctness-preserving refinement steps relying on a solid mathematical foundation, the refinement calculus for action systems [3]. Rather than embody all the requirements in the initial OO-action system specification, we introduce the requirements in successive refinement steps. Although refinement is used normally as a way of verifying the correctness of an implementation w.r.t. a specification, it is used here as a way of structuring the requirements such that they are easier to validate. This technique was used before in [10].

Hence, we make the following contributions:

– We develop a construction method for the components of control systems.
– We point out the potential of software engineering methods in developing control systems components.
– We argue for the suitability of integrating event-based UML statechart diagrams and state-based OO-action systems formalism.

To make the presentation concrete, we will throughout this paper apply our method on a part of the Production Cell example, a well-known case study [22] on control systems that has been used as a sort of a benchmark for formal methods.

We proceed as follows. In Section 2 we present our view on the development of control systems. The UML statechart diagrams form the focus of section 3. Section 4 describes the formal approach on control systems, represented by OO-action systems. In Section 5 we present our case study on the Production Cell, concentrating on two of its control system components. Section 6 is dedicated to refining the specification of a control system component, by emphasising its internal decomposition. We conclude in Section 7 with some final remarks.

2 The Development of Control Systems

A control system is composed from a physical environment called plant, that is usually coordinated by a software system, called controller. The controller is informed upon the state of the plant using observer devices called sensors and commands the update of the state of the plant using devices that perform
Developing Control Systems Components

Developing Control Systems Components

the update, called actuators. The sensors and the actuators are specific control entities. They are also used by the plant, that observes the updates made to its state by the actuators and modifies accordingly the values displayed by the sensors. This pattern of communication between the plant and the controller, using sensors and actuators, is illustrated in Fig. 1 and was also used in [23]. It is specific to a control system, and so is the logical decomposition of a control system into entities like plant, controller, sensors, and actuators.

As we intend to build a control system from components, the strategy for developing it comprises two main phases. Initially, the control system components are identified. At this stage, the main purpose is to capture the essential functions and properties of the overall system into the participating components and also to determine the communication patterns between the components. Following this phase, the focus moves from the system as a whole to each component of the system. The specification of each component has to be stepwise refined until it reaches a structure as depicted in Fig. 1. The process as a whole has to be provably correct and according to the initial specification of the control system to develop.

For expressing this rather general strategy we work in this paper with two specification frameworks: the UML statechart diagrams and the OO-action systems. The former is used in the first phase of the system to ease the specification of a control system with its complicated reactive structure. The latter is also used in the first phase for making the statechart diagrams specification precise, but its potential is fully exploited in the second phase of the development strategy. An OO-action system specification can be verified to fulfil certain goals and also stepwise refined, hence ensuring the correctness of the developed system.

Therefore, in this paper we focus on the development of control system components. The following section presents the initial specification phase for such components using the informal, event-based approach of the UML statechart diagrams.
3 Graphical Specification of Control Systems Components

The Unified Modeling Language (UML) [7] is a powerful and expressive diagrammatic notation used for specifying object-oriented software systems. UML is focused on the conceptual description of a system and its basic building blocks include diagrams, used to describe different views or aspects of a system. A diagram is essentially a projection of the system. The elements of the system under development may appear in all or only in few diagrams. For instance, use case diagrams describe a functional view of the system under development, while the class diagrams present the problem domain of the system, emphasising its object-oriented structure. In this paper, we focus on statecharts diagrams, that show the internal behaviour of class instances (objects), revealing an event-driven view of the system.

Statecharts were introduced by Harel [14] and further developed since [5, 24]. Statechart diagrams, the UML’s approach [7] to Harel’s statecharts show the state machines of objects. A state machine is a behaviour that specifies the sequences of states an object goes through during its lifetime, in response to events, together with its responses to those events. A state is a condition or a situation during the life of an object. During a state, the object satisfies some condition, performs some activity, or just waits for some event. An event is a specification of a significant occurrence, that in a state machine can trigger some transition. Events can be asynchronous (like signals) or synchronous (like method calls). A transition is fired if, besides the occurrence of its significant event, a guard condition holds. While the object is in a transition between states, some action can also take place.

Statechart diagrams, useful in describing objects whom behaviour depend on their past, prove useful as models for control system components. A statechart diagram is intended here to model the behaviour of one component. However, due to the event-based approach of this specification method, a collection of different statechart diagrams will model the behaviour of several components. The communication between them takes place by sending and receiving events. Even though control systems combine hardware and software parts, at this component-oriented level we apply a software engineering technique (UML) to specify them. Later on, the hardware parts will be detached from each component.

The question to be answered at this level refers to the suitability of the statechart diagrams in describing the components interaction with each other. In general, a component implements some public interfaces that produce some result when certain inputs are provided. This mechanism is not hindered by the event-based approach of the statechart diagrams method, but modelled in the latter by sending and receiving events. A received event can embed the input parameters. Upon receiving it, a transition can be fired, possibly some action can take place, the entity might go through a set of states, and eventually some other event is sent, embedding the result parameter. The question posed is thus technically answered. However, an interesting aspect can be observed in this context. The component-orientation is an engineering strategy based on
Developing Control Systems Components 161

encapsulation, since each component is seen only via the interfaces (services) that it is able to provide. A statechart diagram, on the other hand, is breaking this encapsulation by essentially showing the internal sequence of states that component goes through, in order to provide different services.

The explanation of this duality stands in the fact that these two sides of a component represent two phases of the engineering process. The components of interest are discovered in the first phase, based on the services they should provide. Following this phase, the development of each component starts by determining its statechart diagram. This graphical representation is the skeleton of the component. The approach we take for constructing control systems is component-based, but in this paper we focus on the development of their components. The statecharts are useful as they allow us to focus on one component at a time while still being able to model interactions via sending and receiving events.

As an example, consider the Production cell that is a control system whose task is to forge blanks. However, the blanks are supposed to be fed in by some component, transported by another component, pressed by another one, transported again and finally removed from the system by yet another component. For clarity, we focus in this paper on the pressing component, called press and on its interaction with the transporting component, called robot. The robot, besides transporting blanks and interacting with the press, is also interacting with other components. We will show these interactions, but do not develop them later.

By establishing the components, the system is specified in ‘breadth’. Next, we can focus on each component of interest, thus specifying the system in ‘depth’. The press and the robot component have the statechart diagrams as in Fig. 2 and Fig. 3 respectively. These diagrams are based on the following specification [22]: ‘The task of the press is to forge metal blanks. The press consists of two horizontal plates, with the lower plate being movable along a vertical axis. The press operates by pressing the lower plate against the upper plate. The press has three positions. In the lower position, the press is unloaded by arm 2 (of the robot), while in the middle position it is loaded by arm 1 (of the robot). The operation of the press is coordinated with the robot arms as follows: (1) Open the press in its lower position and wait until arm 2 has retrieved the metal plate and left the press. (2) Move the lower plate to the middle position and wait until arm 1 has loaded and left the press. (3) Close the press, i.e. forge the metal plate. This process is carried out cyclically. The robot has two orthogonal arms. Each arm can retract and extend horizontally. The end of each arm is fitted with an electromagnet that allows the arm to pick up metal plates. The robot’s task consists in taking metal blanks to the press and transporting forged plates from the press. The order of the rotation operations the robot has to perform is as follows. (...) The robot rotates counterclockwise until arm 2 points towards the press. Arm 2 is extended until it reaches the press, then it picks up a forged metal plate and retracts. (...) The robot rotates counterclockwise until arm 1 points towards the press. Arm 1 extends, deposits the blank in the press and retracts again. Finally, the robot rotates clockwise towards its original position,
and the cycle starts again.’ The complete specification of the Production Cell case study is given in [22].

![Statechart Diagram for the Press Component](image)

The diagrams in Fig. 2 and Fig. 3 are rather simple statechart diagrams. The initial state of each component is shown by an arrow coming from a filled circle and the activity performed within some states is preceded by the key word ‘do/’. The states of the robot containing some activity are the states in which it interacts with the press. The rest of the states are only mentioned. Since the robot and the press are supposed to synchronise when exchanging blanks, the robot advances to an interaction state only when it received the event PressReady. In these interactions, the press is passive, so it advances to its new states only when it receives an event acknowledging that the blank has been processed (Received or Delivered) and the correspondent robot arm is safely retracted (RobotSafe).

![Statechart Diagram for the Robot Component](image)

Statechart diagrams are an intuitive model for reactive components in a control system. These components usually work cyclically, expect events to whom they react and send events to other components. Thereby, the diagrams above
Developing Control Systems Components 163

are a simple means for expressing collaborations in a control system, at an informal level. The statechart diagrams can also express more complex behaviour in terms of more details of the interactions or of the internal behaviour of the components. Yet, on one hand, when concentrating on components and the basics of their interactions, such a pattern of specification as illustrated in Fig. 2 and Fig. 3 suffices. On the other hand, in order to specify the components precisely and systematically, the statechart diagrams (as their statecharts predecessors [14]) are too informal, as revealed in many sources [15,16,17]. Moreover, they are not equipped with refinement techniques, thereby one cannot ensure the consistency of simple skeleton diagrams like the ones above with other, more complex diagrams of the same component. One solution consists in using the statechart diagrams together with a formalism that has these features built-in. Such a formalism should also be compatible with the statechart diagrams. We present in the following section one such formal framework, the OO-action systems.

4 Formal Specification of Control System Components

In this section we describe the OO-action systems formalism, show how it relates to the component-based design of control systems, and also describe the common ground that enables its use together with the statechart diagrams for specifying control systems components.

An OO-action system consists of a finite set of classes, each class specifying the behaviour of objects (class instances) that are to be created and executed in parallel. Let $\text{Attr}$ be a fixed set of attributes, each attribute being associated with a nonempty set of values, the type set of the attribute. Let further $\text{CName}$ be a fixed set of class names and $\text{OName}$ a set of valid names for objects. We also consider a fixed set of object variables $\text{OVar}$ assumed to be disjoint from $\text{Attr}$. The valid values for object variables are the names for objects in $\text{OName}$.

A class is a pair $<c, \mathcal{C}>$, where $c \in \text{CName}$ is the name of the class and $\mathcal{C}$ is its body, i.e., a collection of data (attributes and object variables), services (methods) and behaviour (actions to be executed by the instances of the class):

$$\mathcal{C} = [\langle \text{attr} \ x := x_0 \rangle \ \langle \text{obj} \ n := n_0 \rangle \ \langle \text{meth} \ m_1 = M_1 \ ; \cdots \ ; m_h = M_h \rangle \ \langle \text{do A od} \rangle$$

The class body consists of an action $A$ and of three declaration sections. The section attr declares the local attributes $x$, used only by the instance of the class itself. The section obj of object variables $n$ declares a special kind of attributes local to an instance of the class. They contain names of objects used for calling methods of other objects. The attributes and the object variables are initialised to the values $x_0$ and $n_0$, respectively. We assume the lists $x$ and $n$ pairwise disjoint. The section meth of methods $m_i = M_i$ describes procedures
of an instance of the class. They can be called by actions of the object itself or by actions of another instance of possibly another class. Methods consist of a header $m_i$ and a body $M_i$, the bodies being actions. The action specified in the `do ... od` loop of a class is a description of the autonomous behaviour that an instance of this class has at run time. An action can refer to the attributes and object variables declared within the class or can contain method calls of the form $n.m$ to methods declared in another class or itself. The actions are defined by a grammar presented in detail in [6]. An action can model statements that always deadlock, stuttering statements, multiple assignments ($=$), conditional, sequential ($;$) and non-deterministic ($\{\}$) composition of actions, method calls or guarded commands of the form $b \rightarrow A$, where $b$ is a predicate and the action $A$ is executed only when its guard $b$ evaluates to true; in this case the action is said to be enabled. All these actions have a well defined semantics [6], in terms of the weakest precondition predicate transformers in the style of Dijkstra [12].

An OO-action system $\mathcal{O}$ consists of a finite set of classes

$$\mathcal{O} = \{ < c_1, C_1 >, ..., < c_n, C_n > \}$$

such that actions in each $C_i$ or bodies of methods declared in each $C_i$ do not contain constructor methods referring to class names not used by classes in $\mathcal{O}$. Every object has a local state. An object, when created, chooses enabled actions and executes them. If there is more than one enabled action in any given state, the choice between them is non-deterministic. Actions are taken to be atomic. Because of this, actions within an object operating on disjoint sets of attributes and object variables can be executed in parallel. Moreover, enabled actions of different objects can be executed in parallel as they are operating on disjoint sets of attributes. Computationally, an OO-action system is a parallel composition of many objects, each of them representing an instance of a class [6].

The interaction between objects is by calling methods of other objects. Method calls are usually prefixed by an object variable. If such a prefix is missing, the called method should be declared in the calling object. Calls to methods of other objects are possible only if the calling object has a reference, i.e. an object variable, to the called object; this models the encapsulation mechanism. The meaning of a call on a method is determined by the substitution principle: let $A$ be an action in an object $a_1$ calling a method $m$ in some object $a_2$. Following the call, the body of the method is substituted for each method call. Hence, the action $A$ becomes $A[a_2.M/n.m]$ where $n$ is the object variable in $a_1$ containing the reference to $a_2$ and $M$ is the body of the method $m$ in the object $a_2$. The action $A$ is then executed jointly by the objects $a_1$ and $a_2$ in an atomic fashion. Here we assumed that the method $m$ is parameterless. For a similar principle including parameters, the reader is referred elsewhere [3].

Components. In this framework of OO-action systems we define the notion of component as follows. Let $\text{Comp}$ be a fixed set of component names. A component is a class-based specification using an OO-action system $\mathcal{O}$ as in (2). The name $\mathcal{O} \in \text{Comp}$. The interaction between components is modelled via calls to
Developing Control Systems Components

methods declared in a class body of the same or some other component (OO-action system). In the latter case, holding a reference only to the name of the component is enough, provided that the names of the class methods in a component are unique. We extend an initial requirement and consider as valid values for object variables the names for objects in $OName$ and also the names for components in $Comp$.

Therefore, we can specify interactions among components using the OO-action systems formalism. By doing so we get a number of advantages, like an object-oriented approach to components and a programming language-like specification for them. These features provide an easier grasp for programmers. Moreover, by using this approach we get another important benefit: we can focus on the design of one component at a time. This independent development of components is an essential feature of the component-based paradigm. The methods of the classes in a component as described above model the interfaces provided to the exterior by that component. The object variables, holding the names of the components of interest, model the dependencies required by the component from its context. Thus, the naming mechanism offers a simple method of communication between components.

Refinement. An important feature coming with this formalism consists in the possibility of stepwise refining the component specifications. The refinement as a process is seen as transforming a specification $O$ into a specification $O'$ when $O$ is abstract and non-deterministic and $O'$ is more concrete, more deterministic, and preserves the functionality of $O$. A particular method of refinement consists in adding on top of one specification new attributes and methods in a way that preserves the old behaviour but makes the new one also possible. This type of refinement is usually referred to as the ‘superposition’ refinement and is discussed formally elsewhere [4,6]. In this paper we take a rather informal view of it, focusing on the engineering process as a whole. We will thereby check whether by adding new data or services the old behaviour is still satisfied. The following section presents an example of this strategy, by refining the interaction mechanisms between the press and the robot component, as described above. In section 6 we show that by using OO-action systems mechanisms like superposition refinement and parallel composition some other type of refinement to the specification of an individual component can yet be performed.

Integration with Statechart Diagrams. In order to refine the interaction patterns between components, an initial OO-action systems specification is required. Like all the formal frameworks, the OO-action systems formalism lacks an intuitive and easy-to-use method for constructing an initial specification. In this respect, the integration with a diagrammatic method of specification would bring this advantage for the OO-action systems framework.

The UML statechart diagrams can be seen as the skeleton of a component and translated into an OO-action systems specification. This operation embeds an important result, namely the integration of an event-based framework with a state-based one. The availability of this process is due to the fact that both
the statechart diagrams and the OO-action systems are models for *active objects*. Thus, the entities modelled by these two methods have a local state and autonomous behaviour. This autonomous behaviour is modelled in the case of OO-action systems with the action loop in the class bodies, while in the case of statechart diagrams with the transition of an entity from a state to another. The latter mechanism can be naturally modelled by a *state* attribute of a class in an OO-action systems specification. The type set of this attribute is formed by all the states that the object can have in the statechart diagram and its value is modified by actions in the action loop of the body of the class. The modifying actions model the transitions from state to state. If these transitions are triggered by some events, the latter are modelled by predicates or method calls, depending on whether the triggering event is local to the component or not, respectively.

In the latter case, the method has a boolean result parameter that is function of the state of that component. The component requiring this triggering event for firing its transitions will keep an object variable to that component name, for being able to call the method of the latter. These principles are applied in the following section on the statechart diagrams from Fig. 2 and Fig. 3.

5  Case Study

In this section we apply the translation principles presented in the previous section on the *Production Cell* example. We also apply the superposition refinement principle presented in Section 4 for making the translated specification precise.

The statechart diagrams in Fig. 2 and Fig. 3 are translated into the components \( \text{Press} = \{ < p, Pr > \} \) and \( \text{Robot} = \{ < r, Ro > \} \). In the component \( \text{Press} \) the class body \( Pr \) has a single attribute \( \text{state} \) that models the current state of the press and similarly, the component \( \text{Robot} \) has the class body \( Ro \) where the single attribute \( \text{state} \) models the current state of the robot.

\[
Pr = \begin{array}{l}
\text{attr} \quad \text{state} : = \text{Receiving} \\
\text{obj} \quad \text{rr} : \text{Robot} \\
\text{meth} \quad \text{Ready} = \ldots
\end{array}
\begin{array}{l}
\text{do} \quad \text{state} = \text{Receiving} \land \text{Received} \land \text{rr.Safe} \rightarrow \text{state} : = \text{Pressing} \\
\quad \text{state} = \text{Pressing} \rightarrow \text{state} : = \text{MoveToDeliver} \\
\quad \text{state} = \text{MoveToDeliver} \rightarrow \text{state} : = \text{Delivering} \\
\quad \text{state} = \text{Delivering} \land \text{Delivered} \land \text{rr.Safe} \rightarrow \text{state} : = \text{MoveToReceive} \\
\quad \text{state} = \text{MoveToReceive} \rightarrow \text{state} : = \text{Receiving} \\
\text{od}
\end{array}
\]

Both components require some information from each other in order to function, modelled in Fig. 2 and Fig. 3 with the events \( \text{RobotSafe} \) and \( \text{PressReady} \), respectively. We model this requirement in the OO-action systems specification with two methods, \( \text{Safe} \) and \( \text{Ready} \), declared in the \( \text{Robot} \) and \( \text{Press} \) components, respectively. These methods return a boolean result parameter that is
function of the respective component state. In order to access these methods, the Press component maintains an object variable \( rr \) to the Robot component and similarly, the Robot component maintains an object variable \( pp \) to the Press component. The events Received and Delivered in Fig. 2 are modelled with two predicates with the same names, while the activities LoadPress and UnloadPress in Fig. 3 are modelled with some actions denoted by the same names:

\[
Ro = \begin{array}{l}
|\text{attr} state := \text{Loading} \\
|\text{obj} pp : \text{Press} \\
|\text{meth Safe} = ...
\end{array}
\]

\[
\begin{array}{l}
\text{do} state = \text{Loading} \rightarrow \text{LoadPress ; state} := \text{MoveToFeed} \\
\quad state = \text{MoveToFeed} \rightarrow state := \text{Feeding} \\
\quad state = \text{Feeding} \rightarrow state := \text{MoveToUnload} \\
\quad state = \text{MoveToUnload} \land pp.\text{Ready} \rightarrow state := \text{Unloading} \\
\quad state = \text{Unloading} \rightarrow \text{UnloadPress ; state} := \text{MoveToDeposit} \\
\quad state = \text{MoveToDeposit} \rightarrow state := \text{Depositing} \\
\quad state = \text{Depositing} \rightarrow state := \text{MoveToLoad} \\
\quad state = \text{MoveToLoad} \land pp.\text{Ready} \rightarrow state := \text{Loading}
\end{array}
\]

The specifications above embed the same information as illustrated by Fig. 2 and Fig. 3. However, in order to further develop the system, this information is too abstract. We therefore proceed by specifying the methods Ready and Safe, the predicates Received and Delivered as well as the abstract actions LoadPress and UnloadPress. The basic mechanism for this consists in enlarging the state of the components, i.e., in adding new attributes to each class body. Adding attributes and making thereby the specifications of the methods, predicates and actions more concrete respects the principles of the superposition refinement, and hence we correctly develop the control systems components specification.

The method Safe of the Robot models the fact that the robot arms are no more inside the press. It becomes apparent that we need two attributes for the robot, \( arm_1 \) and \( arm_2 \) with values from the type set \{ext, retr\}, standing for extended and retracted respectively. Expressing the actions LoadPress, UnloadPress requires some extra attributes since we have to model not only the extension and the retracting of the robot arms but also the picking up and the putting down of blanks from and to the press. As these two actions are performed with different robot arms we specify yet two more attributes, namely \( mg_1 \) and \( mg_2 \) modelling the electromagnets that the arms of the robot are fitted with. These attributes have two values \{on, off\} and the respective methods and actions of the robot have the following specification:

\[
\begin{align*}
\text{Safe} &= (arm_1 = \text{retr} \land arm_2 = \text{retr}) \\
\text{LoadPress} &= (arm_1 := \text{ext} ; mg_1 := \text{off} ; arm_1 := \text{retr}) \\
\text{UnloadPress} &= (arm_2 := \text{ext} ; mg_2 := \text{on} ; arm_2 := \text{retr})
\end{align*}
\]
For the press, we first need to determine the method \textit{Ready}. This method describes the proper interaction position of the press plate along its vertical axis and the fact that the blank has not been yet processed. For specifying this we need two attributes, \textit{pos} and \textit{blank}. The former attribute denotes the position on the vertical axis and has the type set \{\textit{mid}, \textit{down}, \textit{top}\}. The latter attribute stands for the existence of a blank in the press and is a boolean attribute with two values \{\textit{F}, \textit{T}\}. The method \textit{Ready} and the predicates \textit{Received}, \textit{Delivered} can now be specified:

\[
\begin{align*}
\text{Ready} &= (\text{pos} = \text{mid} \land \neg \text{blank}) \lor (\text{pos} = \text{down} \land \text{blank}) \\
\text{Received} &= (\text{blank}), \quad \text{Delivered} = (\neg \text{blank})
\end{align*}
\]

The update of the attribute \textit{blank} has to be done accordingly with the picking up or dropping of the blank by the robot arms. Thus, the attribute \textit{blank} must be set to value \textit{T} after the action \textit{LoadPress} has taken place and to the value \textit{F} after the action \textit{UnloadPress} has taken place. As the attribute is local to the press component, the robot has to be able to access a method for manipulating it, \textit{SetBlank}. This method is declared in the class body \textit{Pr}.

The specification of the example illustrates the component-based manner used for constructing it. The attributes \textit{state} of the components are used to express the cyclic computation of the control system components as well as for synchronising the components with each other. These attributes also provide an intuitive transition from the statechart diagrams specification to the OO-action systems specification.

Besides the translation and specification of the elements in the diagrams, there might be some other issues to be specified in the OO-action systems specification, too. These are usually due to the internal behaviour of the components, used for implementing the public methods.

For instance, we still have to introduce an explicit update for the attribute \textit{pos}, modelling the internal movement of the press plate along its vertical axis. The robot and press components have now the following specification:

\[
\begin{align*}
\text{Pr} &= [\text{attr} \quad \text{state} := \text{Receiving} \land \text{blank} := \text{F} \land \text{pos} := \text{mid} \\
& \quad \text{obj} \quad \text{rr} := \text{Robot} \\
& \quad \text{meth} \quad \text{Ready} := (\text{pos} = \text{mid} \land \neg \text{blank}) \lor (\text{pos} = \text{down} \land \text{blank}); \quad \text{SetBlank}(\text{val}) := (\text{blank} := \text{val}) \\
& \quad \text{do} \quad \text{state} := \text{Receiving} \land \text{blank} \land \text{rr}.\text{Safe} \rightarrow \text{state} := \text{Pressing} \\
& \quad \quad \text{state} := \text{Pressing} \rightarrow \text{pos} := \text{top} \land \text{state} := \text{MoveToDeliver} \\
& \quad \quad \text{state} := \text{MoveToDeliver} \rightarrow \text{pos} := \text{down} \land \text{state} := \text{Delivering} \\
& \quad \quad \text{state} := \text{Delivering} \land \neg \text{blank} \land \text{rr}.\text{Safe} \rightarrow \text{state} := \text{MoveToReceive} \\
& \quad \quad \text{state} := \text{MoveToReceive} \rightarrow \text{pos} := \text{mid} \land \text{state} := \text{Receiving} \\
& \quad \text{od}
\end{align*}
\]

This component specification concludes the first phase of the development. The interactions between components are fully specified and this complete specification...
tion respects the principles of the superposition refinement. That is, the functionality of the components specification is obviously preserved by the introduction of the attributes and of the method SetBlank.

\[
R_0 = \langle \left[ \begin{array}{c}
\text{attr} \quad \text{state} := \text{Loading} ; \text{arm}_1, \text{arm}_2 := \text{retr}, \text{retr} ; \text{mg}_1, \text{mg}_2 := \text{on}, \text{off} \\
\text{obj} \quad \text{pp} : \text{Press} \\
\text{meth} \quad \text{Safe} = (\text{arm}_1 = \text{retr} \land \text{arm}_2 = \text{retr}) \\
\text{do} \quad \text{state} := \text{Loading} \rightarrow \text{arm}_1 := \text{ext} ; \text{mg}_1 := \text{off} ; \text{arm}_1 := \text{retr} \\
\text{pp.SetBlank}(T) ; \text{state} := \text{MoveToFeed} \\
\mid \text{state} := \text{MoveToFeed} \rightarrow \text{state} := \text{Feeding} \\
\mid \text{state} := \text{Feeding} \rightarrow \text{state} := \text{MoveToUnload} \\
\mid \text{state} := \text{MoveToUnload} \land \text{pp.Ready} \rightarrow \text{state} := \text{Unloading} \\
\mid \text{state} := \text{Unloading} \rightarrow \text{arm}_2 := \text{ext} ; \text{mg}_2 := \text{on} ; \text{arm}_2 := \text{retr} ; \\
\text{pp.SetBlank}(F) ; \text{state} := \text{MoveToDeposit} \\
\mid \text{state} := \text{MoveToDeposit} \rightarrow \text{state} := \text{Depositing} \\
\mid \text{state} := \text{Depositing} \rightarrow \text{state} := \text{MoveToLoad} \\
\mid \text{state} := \text{MoveToLoad} \land \text{pp.Ready} \rightarrow \text{state} := \text{Loading} \\
\text{od} \\
\rangle
\]

Other operations can be performed at this level of the specification as well. We can, for example, establish some invariant properties that the components have to respect and then formally check that these properties are respected by the autonomous behaviour of the components. In this way the components specifications deduced from the statechart diagrams specifications are provably correct and according to the requirements of the control system. For instance, an invariant for the press component can be that the plate of the press is moved only within the interval \([\text{down}, \text{top}]\). The formal verification of such a property was performed for instance in [2], using essentially the weakest precondition predicate transformer semantics of actions.

Starting from a specification that agrees with the requirements of the control system to develop, the individual components can be further refined to model the physical plant, the software controller, sensors, and actuators. This forms the object of the following section.

6 Refining Control System Components

In this section we show how can the superposition refinement principle be used together with a parallel composition rule for OO-action systems for refining a component specification towards the structure illustrated in Fig. 1. The number and type of the specific control system entities are discovered during this phase of the component development.

Separating the Plant and the Controller. A control system component consists in a physical plant that is coordinated by a software controller. The evolution of the plant can be discrete or continuous, but when it reaches specific
states, some control decisions have to be taken. The plant and the controller are modelled by a class each. The plant class contains the transitions to the new states. Before each such transition, a controller method is called, taking care of the required decisions to be taken. If no such decision is necessary, then the respective controller method body is modelled with a stuttering action \textit{skip}. The controller class has only methods, thought of as 'interrupt procedures' [23].

In order to ensure a correct component development, the splitting of the existing specification into a two-class specification has to represent a component refinement. This operation is achieved using an intermediary step. The existent component specification is first refined into a similar class that has the actions and methods differently organised. First, a number of methods are added, as many as many actions the initial specification has. Second, each action of the form $\text{state} = \text{act1} \land A \rightarrow B ; \text{state} := \text{act2}$ is replaced with $\text{state} = \text{act1} \rightarrow \text{act2} ; \text{state} := \text{act2}$, where the method $\text{act2} = (\text{state} = \text{act1} \land A \rightarrow B ; \text{state} := \text{act2})$. Obviously, the effect of the new action is the same as the effect of the old action. Only some redundancy has been added. Since adding methods and keeping the old functionality agrees with the superposition refinement step, the process so far is a refinement. Third, for each attribute that is read in the body of the methods we add a method $\text{GetAttr} = (\text{attr})$ and replace each reading of the attribute with a call to this method. Again, this process represents a refinement step. For instance, the intermediate class body between the press specification from the previous section and the desired specification containing a plant and a controller is like this:

\[
Pr' = [\begin{array}{l}
\text{attr state} : = \text{Receiving} ; \text{blank} : = F ; \text{pos} : = \text{mid}
\text{obj rr : Robot}
\text{meth SetBlank(val) = (blank} : = \text{val}) ; \text{GetBlank = (blank)} ;
\text{Ready} = (\text{pos} = \text{mid} \land \neg \text{p.GetBlank}) \lor (\text{pos} = \text{down} \land \text{p.GetBlank}) ;
\text{Pressing} = (\text{state} = \text{Receiving} \land \text{p.GetBlank} \land \text{rr.Safe} \rightarrow \text{state} := \text{Pressing}) ;
\text{MoveToDeliver} = (\text{state} = \text{Pressing} \rightarrow \text{pos} = \text{top} ; \text{state} := \text{MoveToDeliver}) ;
\text{Delivering} = (\text{state} = \text{MoveToDeliver} \rightarrow \text{pos} = \text{down} ; \text{state} := \text{Delivering}) ;
\text{MoveToReceive} = (\text{state} = \text{Delivering} \land \neg \text{p.GetBlank} \land \text{rr.Safe} \rightarrow \\
\text{state} := \text{MoveToReceive}) ;
\text{Receiving} = (\text{state} = \text{MoveToReceive} \rightarrow \text{pos} = \text{mid} ; \text{state} := \text{Receiving})
\text{do}
\text{state} = \text{Receiving} \rightarrow \text{Pressing} ; \text{state} := \text{Pressing}
\text{state} = \text{Pressing} \rightarrow \text{MoveToDeliver} ; \text{state} := \text{MoveToDeliver}
\text{state} = \text{MoveToDeliver} \rightarrow \text{Delivering} ; \text{state} := \text{Delivering}
\text{state} = \text{Delivering} \rightarrow \text{MoveToReceive} ; \text{state} := \text{MoveToReceive}
\text{state} = \text{MoveToReceive} \rightarrow \text{Receiving} ; \text{state} := \text{Receiving}
\text{od}
\end{array}]
\]

Having this intermediate class, a specific rule of OO-action systems can be applied, namely the parallel decomposition. Given two classes, their parallel composition consists in the union of their attributes, object variables, methods, and actions, such that if one of the classes has a reference (object variable) to another, this object variable appears no longer in the composition and the methods
of the latter class are called in the former without the respective prefix. Other
classes not participating to the composition may have references to one of the
composing classes. Such references are replaced with a reference to composition
of these classes. This operation is formally described elsewhere [6,3].

The reverse operation can be applied on the intermediate class and hence
split it into two classes: a plant and a controller. The controller class contains
all the controlling methods introduced previously and also the methods used to
signal triggering events to other components. All the attributes that are modified
in the controller methods are also declared in the controller class. The attributes
that are only read in these methods are declared in the plant class, together
with their SetAttr and GetAttr methods. Since the state attribute is used both
in the plant actions and in the controller methods, it is duplicated in each class.
Its role is to help the proper coordination of the plant with the controller. It is
important to note, however, that the two ‘states’ are different attributes. The
press will always contain a reference to the controller for calling its methods.
The controller will keep a reference to the plant only if it needs some attributes
that are declared in this class.

The parallel decomposition of a class into several classes that have the same
effect as the initial class is only a structuring technique. Semantically, the initial
and the final specification represent the same component. Hence, the overall
process of splitting of a component specification into a plant and a controller
is a correct refinement step, since the intermediate step was a superposition
refinement. Therefore, the process of control system component development
remains correct.

For instance, the press component is now specified as follows: Press = {<
prp, PrP >, < prc, PrC >}. Since the attribute blank is read by the controller,
there is also a reference p : prp from the controller to the plant and a method
GetBlank declared in the press plant.

\[
\begin{align*}
PrP &= [\textbf{attr} \quad \text{state} = \text{Receiving} ; \text{blank} = F \\
& \textbf{obj} \quad \text{p} : \text{prc} \\
& \textbf{meth} \quad \text{SetBlank}(\text{val}) = (\text{blank} : = \text{val}) ; \text{GetBlank} = (\text{blank}) \\
& \text{do} \quad \text{state} = \text{Receiving} \rightarrow \text{p}.\text{Pressing} ; \text{state} = \text{Pressing} \\
& \text{state} = \text{Pressing} \rightarrow \text{p}.\text{MoveToDeliver} ; \text{state} = \text{MoveToDeliver} \\
& \text{state} = \text{MoveToDeliver} \rightarrow \text{p}.\text{Delivering} ; \text{state} = \text{Delivering} \\
& \text{state} = \text{Delivering} \rightarrow \text{p}.\text{MoveToReceive} ; \text{state} = \text{MoveToReceive} \\
& \text{state} = \text{MoveToReceive} \rightarrow \text{p}.\text{Receiving} ; \text{state} = \text{Receiving} \\
& \text{od} \\
\end{align*}
\]
\[PrC = \begin{array}{l}
| \text{attr} \state = \text{Receiving} & \text{pos} = \text{mid} \\
| \text{obj} \rr : \text{Robot} & p : \text{prp}\\
| \text{meth} \text{Ready} = (\text{pos} = \text{mid} \land \neg p.\text{GetBlank}) \lor (\text{pos} = \text{down} \land p.\text{GetBlank})\\
| \text{Pressing} = (\state = \text{Receiving} \land p.\text{GetBlank} \land \rr.\text{Safe} \rightarrow \state = \text{Pressing})\\
| \text{MoveToDeliver} = (\state = \text{Pressing} \rightarrow \text{pos} = \text{top} \land \state = \text{MoveToDeliver})\\
| \text{Delivering} = (\state = \text{MoveToDeliver} \rightarrow \text{pos} = \text{down} \land \state = \text{Delivering})\\
| \text{MoveToReceive} = (\state = \text{Delivering} \land \neg p.\text{GetBlank} \land \rr.\text{Safe} \rightarrow \state = \text{MoveToReceive})\\
| \text{Receiving} = (\state = \text{MoveToReceive} \rightarrow \text{pos} = \text{mid} \land \state = \text{Receiving})
\end{array} \]

The robot component can be refined using the same pattern into a robot plant and a robot controller. The attributes of the robot are only modified, so they are declared all in the controller. In the following we skip the specification of the robot component for brevity.

**Specifying Sensors and Actuators.** The development of a control system is initiated by splitting the overall system into several components. This step is usually not difficult, as a group of related services to be performed by the system are likely to be implemented by the same component. A control system normally contains also some typical entities like sensors and actuators. However, it is not obvious from the beginning how many of these entities are necessary and likewise their types are not easy to determine. It is more natural instead to discover the sensors and the actuators on the way, as they are needed. We adopt this approach here and determine the sensors and the actuators in the last refinement step, from the attributes of the components and their distribution in the plant or in the controller.

The separation into a plant and a controller implied also a splitting of the attributes in the two entities. However, the attributes of the plant and of the controller model usually more than just properties of these entities, as their values are read by the other entities. They would thus model sensors and actuators, used within one component for synchronising the physical plant with the software controller and also used indirectly for modelling the interaction between components, via the boolean methods modelling triggering events.

To conform with a control system specification as depicted in Fig. 1 we have to establish which attributes are sensors and which are actuators. The strategy for this is to check what is the information read by the controller and to transform it into sensors, then to check what is the information read by the plant and transform it into actuators and finally to make all updates of the sensors within the plant and all updates of the actuators within the controller as shown in the specification below.

For instance the `blank` attribute is read in the press controller (via the method `GetBlank`), so it should model a sensor that is supposed to sense whether there is something in the press of the consistence of a blank. When a blank is put into the press or taken from the press, the sensor should reflect these changes. The other press attribute `pos` models an actuator, defined and updated by the controller.
By applying the same splitting strategy as for separating the plant and the controller we get a final specification of a control system component according to Fig. 1. This specification is the result of a correct development process based on refinement techniques. A component will consist in a collection of sensors, actuators, plant and controller classes. For instance, the press component has become: Press = \{< \text{pp}, \text{PrP} >, < \text{prc}, \text{PrC} >, < \text{blank}, \text{blank} >, < \text{ALEvel}, \text{ALevel} >, < \text{SLevel}, \text{SLevel} >\}.

Blank = \{\text{attr } \text{val} : = \text{F} \text{meth } \text{SetBlank}(v) = (\text{val} : = v) ; \text{GetBlank} = (\text{val})\}.

\text{SLevel} = \{\text{attr } \text{val} : = \text{mid} \text{meth } \text{Set}(v) = (\text{val} : = v) ; \text{Get} = (\text{val})\}.

\text{ALevel} = \{\text{attr } \text{val} : = \text{mid} \text{meth } \text{Set}(v) = (\text{val} : = v) ; \text{Get} = (\text{val})\}.

\text{PrP} = \{\text{attr } \text{state} : = \text{Receiving} \text{obj } p : \text{prc} ; \text{slev} : \text{SLevel} ; \text{alev} : \text{ALevel} \text{do } \text{state} = \text{Receiving} \rightarrow p.\text{Pressing} ; \text{state} : = \text{Pressing} \text{state} = \text{Pressing} \rightarrow p.\text{MoveToDeliver} ; (\text{alev}.\text{Get} = \text{top} \rightarrow \text{slev}.\text{Set}(\text{top}) ; \text{state} : = \text{MoveToDeliver} \text{state} = \text{MoveToDeliver} \rightarrow p.\text{Delivering} ; (\text{alev}.\text{Get} = \text{down} \rightarrow \text{slev}.\text{Set}(\text{down}) ; \text{state} : = \text{Delivering} \text{state} = \text{Delivering} \rightarrow p.\text{MoveToReceive} ; \text{state} : = \text{MoveToReceive} \text{state} = \text{MoveToReceive} \rightarrow p.\text{Receiving} ; (\text{alev}.\text{Get} = \text{mid} \rightarrow \text{slev}.\text{Set}(\text{mid}) ; \text{state} : = \text{Receiving} \text{od} \}\.

\text{PrC} = \{\text{attr } \text{state} : = \text{Receiving} \text{obj } \text{rr} : \text{Robot} ; \text{bl} : \text{Blank} ; \text{spos} : \text{SLevel} ; \text{apos} : \text{ALevel} \text{meth } \text{Ready} = (\text{apos}.\text{Get} = \text{mid} \land \neg \text{bl}.\text{GetBlank}) \lor (\text{apos}.\text{Get} = \text{down} \land \text{bl}.\text{GetBlank}) ; \text{Pressing} = (\text{state} = \text{Receiving} \land \text{bl}.\text{GetBlank} \land \text{rr}.\text{Safe} \rightarrow \text{state} : = \text{Pressing}) ; \text{MoveToDeliver} = (\text{state} = \text{Pressing} \rightarrow \text{apos}.\text{Set}(\text{top}) ; \text{state} : = \text{MoveToDeliver}) ; \text{Delivering} = (\text{state} = \text{MoveToDeliver} \rightarrow \text{apos}.\text{Set}(\text{down}) ; \text{state} : = \text{Delivering}) ; \text{MoveToReceive} = (\text{state} = \text{Delivering} \land \neg \text{bl}.\text{GetBlank} \land \text{rr}.\text{Safe} \rightarrow \text{state} : = \text{MoveToReceive}) ; \text{Receiving} = (\text{state} = \text{MoveToReceive} \rightarrow \text{apos}.\text{Set}(\text{mid}) ; \text{state} : = \text{Receiving}) \}\.

One can notice the suitability of OO-action systems in modelling component interfaces. Thus, the robot component maintains a reference \text{pp} to the press component and uses among other methods the method \text{SetBlank} declared in the press component. \text{Within} the press component, this method is declared first in the class body \text{Pr}, then in \text{Pr}' \text{, then in PrP and finally in the class body Blank. From the robot component’s perspective, this is however irrelevant. The interaction works as long as the name of the method remains unchanged.}
Concluding the control systems development, we have determined the sensors, the actuators, the cyclic evolution of the physical plant and the control software following completely software engineering techniques. The controller methods get activated when corresponding sensor values change, as a result of some actuators actions. Moreover, the state attributes become obsolete and can be more or less replaced by up-to-date information received via the sensors.

7 Conclusions

In this paper we have proposed a development method for control system components. As a novelty, our method combines UML statechart diagrams and OO-action systems, based on their compatibility. The method shows the power of software engineering techniques in developing systems where it is not clear from the beginning which parts will be implemented in hardware and which in software.

A particular role is played by the refinement techniques that help identifying different hardware or software entities. It seems that the principles of refinement and its different approaches as data, algorithmic, or superposition refinement go beyond the context of formal methods for software construction where they were introduced.

Using the two specification techniques for the development of components, a number of integration questions appear. The most important result here is that event-based statechart diagrams and state-based OO-action systems can integrate rather naturally, as they both model active objects. Consequently, we can take advantage of this integration result and work towards the development of a formal semantics for UML statechart diagrams in terms of OO-action systems. In this context it seems interesting to define the notion of refinement for statechart diagrams. However, this is left for future research.

Acknowledgement. This work is supported by the Academy of Finland.

References

Specification and Analysis of Automata-Based Designs

Jeremy Bryans\textsuperscript{1}, Lynne Blair\textsuperscript{2}, Howard Bowman\textsuperscript{1}, and John Derrick\textsuperscript{1}

\textsuperscript{1} Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent, CT2 7NF, UK,
\textsuperscript{2} Computing Department, Faculty of Applied Sciences, Lancaster University, Lancaster, LA1 4YR, UK

Abstract. One of the results of research into formal system specification has been the large number of notations which have been developed. Of these notations, automata have emerged as a promising vehicle for the specification, and particularly the analysis, of systems. This is especially so when the systems under consideration include timing requirements, and timed automata model such systems as a finite set of states with timed transitions between them. However, not all specifications involve deterministic timing, and stochastic automata can be used in these circumstances.

In this paper we consider both timed and stochastic automata, and demonstrate how they can be used in the same design. We will also consider what analysis of the specification can then be performed. In particular, we will describe how to translate stochastic to timed automata, and look at two approaches to model checking the stochastic components of an integrated design.

Keywords: Timed automata, stochastic automata, model checking.

1 Introduction

One of the results of research into formal system specification has been the large number of notations which have been developed. There are now many notations which can be used to specify and design systems. Potential problems with this are that specifiers working on the same system may be familiar with different notations, and that different notations may be better suited for different parts of the same design.

Even within notations there can be variants, and in this paper we will confine ourselves to automata. We will demonstrate how different automata notations can be used in the same design, and how analysis of the specification can be performed according to the particular notation used. This means that designers need not be restricted to a monolithic notation, and that the most convenient notation can be chosen to describe each component within the design.

In this paper we will focus on timed automata with deadlines and stochastic automata. Timed automata are now well established as a specification notation, and there has been extensive work on analysis techniques for them, and in
particular model checking algorithms. Stochastic automata are a relatively new extension to timed automata, where the emphasis has been shifted from deterministic timing to timings picked from a probabilistic distribution, thus enabling a new range of systems to be specified.

The structure of the paper is as follows. In Section 2 we present the automata-based notations that we will use throughout the paper, and in particular timed automata with deadlines and stochastic automata. Section 3 presents an example using these notations which models cars arriving at a port wishing to board a ferry. Section 4 looks at possible ways to analyse such a specification, and compares them with each other.

Specifically we are interested in timed vs stochastic analysis. For the former there is a wide range of techniques available and therefore we concentrate on how we can integrate the stochastic components into this analysis. To this end we show how a stochastic automata can be translated into a timed automata with deadlines, as this allows the integrated specification to be interpreted within a single simpler notation.

However this clearly involves a loss of some stochastic information, and to perform stochastic analysis we look at two approaches to model checking the stochastic components of an integrated design. Finally in Section 5 we draw some conclusions, and mention ongoing and possible future work.

2 Notations

For the purposes of this paper, we choose automata as a “base” notation, and we will use the timed automata with deadlines (TAD) of [5], and the stochastic automata (SA) of [10] as necessary. Although different versions of timed automata exist, we chosen TAD over the others because of the ease of translating from SA to TAD (see Section 4.1.) Both TAD and SA are extensions of ordinary automata, and we give definitions for them now.

Definition 1 In this paper, a TAD is:

- A discrete labelled transition system \((\mathcal{U}, \rightarrow, A)\) where
  - \(\mathcal{U}\) is a finite set of discrete states
  - \(A\) is a finite set of actions
  - \(\rightarrow \subseteq \mathcal{U} \times A \times \mathcal{U}\) is an untimed transition relation
- A set \(X = \{x_1, \ldots, x_n\}\) of non-negative real valued variables called clocks.
- A labelling function \(h\) mapping untimed transitions into timed transitions:
  \[ h(u, a, u') = (u, (a, g, d, r), u') \]  
  where
  - \(g\) and \(d\) are the guard and deadline of the transition. Guards and deadlines are predicates \(p\) defined by the following grammar:
    \[ p ::= x \# w \mid p \land p \mid p \lor p \]
    where \(x \in X\), \(w \in \mathbb{R}_{\geq 0}\) and \(\# \in \{\leq, <, >, \geq\}\).
- \(r\) is the set of clocks which are reset to zero when the transition takes place.
A transition may occur only when the guard is true, and must occur if the deadline is true. (The definition of the grammar for defining the guard and deadline predicate is slightly modified from the one found in [5].)

The clocks in TAD always begin counting at zero and count upwards. This is in contrast to the clocks in SA, which are set to some value in \( \mathbb{R}_{\geq 0} \) according to their probability distribution function, and count downwards.

As an example, consider the TAD depicted in Figure 1. From state \( u_0 \), the action \( a \) may occur provided the clock \( x_1 \) is greater than 2, and must occur if it is equal to 4. When it does occur, the clock \( x_2 \) is reset and the automaton moves to state \( u_1 \). From here, the action \( b \) may occur provided clock \( x_1 \) is in the range \( [6, 8] \) and clock \( x_2 \) is greater than 3. The deadline imposes no restriction, and when the action does occur no clocks are reset.

**Fig. 1.** A Timed Automaton with Deadlines

Stochastic automata are an extension of timed automata, in which the time at which actions occur may be a random variable. In this paper we use the stochastic automata defined in [10], which are presented below.

**Definition 2** A stochastic automaton is a structure \( (S, s_0, C, A, \rightarrow, \kappa, F) \) where:

- \( S \) is a set of locations with \( s_0 \in S \) being the initial location, \( C \) is the set of all clocks, and \( A \) is a set of actions.
- \( \rightarrow \subseteq S \times (A \times C) \times S \) is the set of edges. If \( s \) and \( s' \) are states, \( a \) is an action and \( C \) is a finite subset of \( C \), then we denote the edge \( (s, a, C, s') \in \rightarrow \) by \( s \xrightarrow{a,c} s' \) and we say that \( C \) is the trigger set of action \( a \). We use \( s \xrightarrow{a} s' \) as a shorthand notation for \( \exists C. s \xrightarrow{a,c} s' \). In this paper we will associate only a single clock with each action.
- \( \kappa : S \rightarrow \mathbb{P}_{fin}(C) \) is the clock setting function, and indicates which clocks are to be set in which states, where \( \mathbb{P}_{fin}(C) \) is the finite powerset of clocks.
- \( F : C \rightarrow (\mathbb{R} \rightarrow [0, 1]) \) assigns to each clock a distribution function such that, for any clock \( x \), \( F(x)(t) = 0 \) for \( t < 0 \); we write \( F_x \) for \( F(x) \) and thus \( F_x(t) \) states the probability that the value selected for the clock \( x \) is less than or equal to \( t \). Each clock \( x \in C \) is a random variable with distribution \( F_x \).
In this paper we will assume that clocks are only used on transitions emanating from the states in which they are set. We will also find it easier to refer to probability density functions (pdf’s), which are the derivatives of the distribution functions. We will use $P_x$ for the pdf of $F_x$.

As an example, of a stochastic automaton, consider Figure 2. This is written $((s_0, s_1), s_0, \{x, y\}, \{a, b\}, \triangleright, \kappa, \{P_x, P_y\})$ where $\triangleright = \{(s_0, a, \{x\}, s_1), (s_0, b, \{y\}, s_0)\}$, and the pdf’s for clocks $x$ and $y$ are

$$P_x(t) = 4 - 2t, \text{ if } t \in [1, 2] = 0, \text{ otherwise}$$

$$P_y(t) = 2t - 2, \text{ if } t \in [1, 2] = 0, \text{ otherwise}$$

as depicted. The horizontal axis measures time, and the vertical axis measures the probability of the clock being set to a value less than that time.

The SA starts in location $s_0$, and both clocks $x$ and $y$ are set according to the functions $F_x$ and $F_y$. If clock $x$ expires first, then action $a$ is triggered and the automaton moves to location $s_1$. This location has no outgoing transitions, and so nothing further happens. If clock $y$ expires first, then action $b$ is triggered and the automaton returns to state $s_0$. The clocks are reset according to their distributions, and the process is repeated.

In the following Section we will show how we can combine both stochastic and timed automata using a larger example.

3 Example - A Car Ferry

To illustrate these ideas, we will specify a system consisting of a number of cars at a port, trying to get on to a ferry (see Figure 3.) The cars enter the port at the traffic lights, and join the queue in the middle of the port. When they reach the front of the queue they move to the next free kiosk, where they are processed, and then they go on to join the ferry.

In this example, there are two parts of the model over which we do not have direct control. One is the arrival of the cars into the queue, and the other is the rate at which individual kiosk workers work. For both of these we use stochastic automata to model the inherent uncertainty.
We will consider that actions synchronise with other actions of the same name, and that in a parallel composition the intersection of the alphabets synchronise (as in [11].) For the car arrivals, we use the distribution shown in Figure 4. This can be thought of as modelling the behaviour resulting from a set of nearby traffic lights: If one car arrives it is quite likely that another will arrive very shortly afterwards, (between 5 and 10 seconds). If no car arrives in this time then the lights will turn red, and no car will be able to arrive until 30 seconds have passed. Whether or not this function is an accurate representation of the environment in which the system will have to operate can only be determined by observing the actual behaviour of the cars.

An individual kiosk is modelled as an SA, as shown in Figure 5. We model the kiosk as opening immediately, and twelve seconds after a kiosk opens, a car leaves the queue to be processed. This is modelled by the clock $c_2$, which is deterministically set to 12 seconds every time state $K_2$ is entered.
The processing takes between 30 and 60 seconds, and this continues until the kiosk is closed. This is modelled by the clock $c_3$, set to a value between 30 and 60 seconds according to the pdf:

$$P_x(t) = \begin{cases} \frac{t-30}{450}, & \text{if } t \in [30, 60] \\ 0, & \text{otherwise} \end{cases}$$

Here we are in fact modelling the impact on the queue (using the cararrives actions) rather than on the kiosk directly.

We include the state $K_4$ in order to be able to distinguish the state in which the first car has been processed (and the second one has left the queue.) We will make use of this later in the analysis of the kiosk.

The queue that the cars form is essentially passive. It does not instigate either the cararrives or the carleaves actions, and it therefore needs no time deadlines (as TAD) or clocks (as SA) and can be modelled as a simple automaton. This is shown in Figure 6 (where states 3 and 4 and the transitions between them have been elided).

Notice that quite general distributions are allowed in our stochastic automata. Here we have used combinations of uniform and triangular distributions, and in general arbitrary distributions are allowed.
4 Analysing the Integrated Specifications

In order to analyse a specification defined using a number of different notations, we have two possibilities. We can either re-interpret all the components within one notation (and then use whatever analysis that notation permits) or we can analyse the components of the specification. Here we briefly consider both approaches which are illustrated using the car ferry example.

4.1 Translating SA to TAD

In this Section we consider the first approach and show how to interpret stochastic automata in terms of timed automata with deadlines. The interpretation must preserve the behaviour of the SA within a TAD as far as possible, so for each SA we must be able to generate the TAD which is capable of exactly the same set of runs\(^1\) as the SA. However, since a TAD cannot represent probabilistic information, this translation will necessarily lose all probabilistic information.

To illustrate the ideas we begin by deriving timed automata with deadlines from the stochastic automata in the example, and then give the formal definition of the translation.

Consider the SA (Figure 4) that models the arrivals of cars at the car ferry. The clock \(c_1\) is set (as we may deduce from the pdf of clock \(c_1\)) to some value in \([5, 10] \cup [30, 35] \cup [55, 60] \cup [80, 85]\), and then proceeds to count down. The action \(\text{car arrives}\) occurs when this clock expires.

We derive an corresponding TAD (Figure 7) which must therefore be capable of performing the action \(\text{car arrives}\) at any time in the range \([5, 10] \cup [30, 35] \cup [55, 60] \cup [80, 85]\), and the action must be performed by (or at) time 85. We use \(x_1\) to correspond to clock \(c_1\), and set the guard (the permitted occurrence times) to \(x_1 \in [5, 10] \cup [30, 35] \cup [55, 60] \cup [80, 85]\) and the deadline to \(x_1 \geq 85\). Setting the deadline to greater than or equal to 85 means that if this state is entered when \(x_1 \geq 85\) the action must occur.

In the SA, clock \(c_1\) is reset every time the state \(A_1\) is entered, so in the TAD \(\tau\) (which is the set of clocks being reset to zero when the transition occurs) is set to \(\{x_1\}\).

\[\text{car arrives} \quad x_1 \in [5, 10] \cup [30, 35] \cup [55, 60] \cup [80, 85]\]
\[x_1 = 85\]
\[\tau : x_1\]

Fig. 7. The translation of the car arrivals into a TAD

\(^1\) A run is a (finite or infinite) sequence of timed actions.
This turns out to be an automaton with just one state, however, not all translations are this simple, for example the kiosk description (Figure 5) becomes the TAD in Figure 8.

![Diagram](image)

**Fig. 8.** The translation of a single kiosk into a TAD

Using the ideas illustrated in these examples we can formalise the full definition of the translation of stochastic automata to timed automata with deadlines as follows.

**Definition 3** Translating an SA into a TAD.

Let \((S, s_0, C, A, \rightarrow, \kappa, F)\) be a stochastic automaton. This automaton is mapped to the timed automaton \((Z, \rightarrow_T, A)\) where

- \(Z = S\)
- \(A = A\)
- \(\rightarrow_T\) is the transition relation \(\rightarrow\) with the clocks removed, i.e.
  \(\rightarrow_T \subseteq Z \times A \times Z\) where
  \(\rightarrow_T = \{ (z, a, z') \mid \exists C_a (s, a, C_a, s') \in \rightarrow \land s = z \land s' = z' \}\)
- The set \(X\) contains (non-negative real-valued) clock variables, labelled \(x_i\) and indexed as the SA variables.
- \(\forall i \in X \leftrightarrow c_i \in C\)
- \(h(s, a, s') = (s, (a, g, d, r), s')\) where \(C_a\) is the trigger set for action \(a\) and
  - \(g = (\bigwedge_{c_i \in C_a} x_i \geq \min(c_i)\)
    \(\land\)
    \(\bigvee_{c_i \in C_a} x_i \in \text{ran}(c_i)\)
  - \(d = \bigwedge_{c_i \in C_a} x_i \geq \max(c_i)\)
  - \(r = \kappa(s')\)

We are indebted to Pedro D’Argenio [9] for this definition, and it is discussed in more detail in [7].
With the stochastic components turned into timed automata with deadlines, temporal observations of the system can be made, for example to address questions such as “Is a particular throughput of cars possible?” or “With only one kiosk open, what is the minimum/maximum time before the queue overflows?”. To support this task, work has started on extending the LUSCETA tool [12]. It currently supports the creation and editing of timed automata and timed automata with deadlines. It also supports the composition of either type of automata providing all automata are of the same type (the composition rules for timed automata with deadlines are presented in [4]). However, the simulator currently only supports timed automata; work is still required to extend this to timed automata with deadlines.

This translation provides us with the ability to analyse the temporal properties of a specification that originally included stochastic information. However, this translation has the obvious drawback that the exact stochastic properties of the specification can no longer be investigated. We move on to consider this in the next Section.

4.2 Model Checking Stochastic Automata

The alternative to translating SA to TAD (and thereby forfeiting the stochastic information) is to keep the stochastic information by retaining the SA, and performing more complex analysis only on the stochastic components. Much of the work done in stochastic modelling and performance evaluation uses the assumption that the random times at which actions occur are drawn from exponential distributions. While this allows many performance evaluation results to be derived, in practice it is unrealistic to consider only exponential distributions, and it is necessary for arbitrary distributions to be considered.

The analysis technique we consider is model checking [8]. This has proved very successful in many applications, and applying it to stochastic systems opens up several new research issues.

In this Section we discuss two approaches to model checking stochastic automata. The first calculates exact answers for stochastic automata involving arbitrary distributions. However, the cost of this precision is the complexity of the algorithm and we also describe a further algorithm which uses a discretisation to reduce this complexity and is discussed in more detail in [6].

A Probabilistic Real Time Temporal Logic. The basic approach we take to model checking is to try to show that a temporal logic property is satisfied by a stochastic automaton description of the system. Here we use a simple probabilistic real-time temporal logic. The purpose of the logic is to express properties that we wish to check the stochastic automaton against and the logic we define allows us to check a range of such properties.

The syntax of our logic is

\[
\psi ::= \mathrm{tt} \mid \text{ap} \mid \neg \psi \mid \psi_1 \land \psi_2 \mid [\phi_1 \mathcal{U}_c \phi_2] \simeq p
\]

\[
\phi ::= \mathrm{tt} \mid \text{ap} \mid \neg \phi \mid \phi_1 \land \phi_2
\]
Here \( ap \) is an atomic proposition, \( c \in \mathbb{N} \) (natural numbers), \( p \in [0, 1] \) is a probability value and \( \sim \in \{<, >, \leq, \geq\} \). The temporal aspects are described by \( [\phi_1 U_{\sim} c \phi_2] \simeq p \) which is an “until” formula. In general we would associate sets of atomic propositions with states of automata; however here it will be sufficient to assume a single distinct proposition for each state, (effectively identifying states and propositions) so that each state \( A_x \) models the set of propositions \( \{A_x, \text{tt}\} \). Using this logic we can also define a number of derived operators, for details see [6].

To understand an “until” formula, it is simplest to begin with an untimed, non-probabilistic version. Intuitively, \( \phi_1 U \phi_2 \) reads as: \( \phi_1 \) holds until \( \phi_2 \) does. The subscript \( \sim c \) is the time restriction — eg. if \( \sim \) is \( \leq \) then \( \phi_2 \) must hold before (or at) time point \( c \). The addition \( \simeq p \) is a probability restriction — e.g. if \( \simeq \) is \( > \) then \( \phi_1 U_{\sim} \phi_2 \) must be true with probability greater than \( p \).

The until formulae can only be used at the top level — they cannot be nested. This is because the model checking algorithms we discuss can only evaluate until formulae from the initial state; this is a necessary restriction of our current approach.

With this syntax, an example of a valid formula that we can check against the stochastic automaton in Figure 5 would be \( [\text{tt}U_{\leq} 60 K_4] > 0.3 \). This states that the probability of reaching the state \( K_4 \) (and therefore having processed the first car) within 60 seconds is greater than 0.3.

It should be clear that since we do not allow the until formulae to be nested we can use the following recipe in order to model check a formula \( \psi \) of our logic against a stochastic automaton \( A \).

1. For each until subformula (i.e. of the form \( [\phi_1 U_{\sim} c \phi_2] \simeq p \)) in \( \psi \) perform an individual model check to ascertain whether
   \[ A \models [\phi_1 U_{\sim} c \phi_2] \simeq p \]

2. Replace each until formula in \( \psi \) by \( \text{tt} \) if its corresponding model check was successful, or \( \text{ff} \) (\( \neg \text{tt} \)) otherwise.

3. Replace each atomic proposition in \( \psi \) by \( \text{tt} \) or \( \text{ff} \) depending upon its value in the initial location of \( A \).

4. \( \psi \) is now a ground term, i.e. truth values combined by a propositional connective (\( \neg \) and \( \land \)). Thus, it can simply be evaluated to yield a truth value. The automaton is a model of \( \psi \) if this evaluation yields \( \text{tt} \), and is not otherwise.

We assume that when we wish to model check a property against an automaton, we are also given an adversary [3] to resolve the nondeterminism within the automaton. Without this adversary, enumerative analysis would not be possible; the provision of an adversary is a prerequisite of model checking. To understand the notion of an adversary here, we must explain in a little more detail our conceptual model of automata. We consider an automaton to operate within an environment, and for this environment (if unspecified) to be the most general environment possible, and to permit all behaviours of the automaton\(^2\). We can

\(^2\) This follows closely the CSP [11] notion of process and environment, where the environment, if unspecified, is taken to be the most nondeterministic process.
then think of an adversary as an environment which is deterministic with respect to the automaton, and therefore resolves all nondeterminism within it.

If, for example, we were checking a property of a single kiosk, we could choose the adversary to perform the open action immediately, and to close the kiosk again after three hours.

This recipe employs standard techniques apart from the individual checking that $A \models [\phi_1 U_c \phi_2] \simeq p$ and this is what our two algorithms address.

**The First Algorithm – Using Region Trees.** The first algorithm (called the region tree algorithm) has similarities to the region graph construction of [2], [1], [13]. In general for this algorithm, we must assume that the clock distribution functions are continuous within the range of the function, in order to ensure that the probability of two clocks expiring at the same time is zero.

The algorithm works by unfolding the automaton to construct a region tree, and at each stage in the unfolding using the temporal logic formula to construct a probabilistic region tree. The regions are formed using the notion of valuation equivalence. A valuation records the values of all the clocks in a particular state at a particular moment in time. The unique clock $a \in C$, which we add to the set of clocks, is used to facilitate the model checking. It keeps track of the total time elapsed in the execution of the stochastic automaton, but plays no part in the behaviour of the automaton.

**Definition 4** A valuation is a function $v : C \cup \{a\} \rightarrow R \cup \{\bot\}$ such that $v(x) = \bot$ or $v(x) \leq x_{\max}$, where $x_{\max}$ is the maximum value to which clock $x$ can be set. If $d \in R_{\geq 0}$, $v - d$ is defined by $\forall x \in C \cup \{a\}. (v - d)(x) \overset{\text{def}}{=} v(x) - d$. The function $\text{min}(v)$ returns the value of the smallest defined clock.

Since we assume that clocks are only used in the states in which they are set, there is no need to remember their value once the state has been exited. Only the clock $a$ maintains its value; the rest are set to $\bot$. At the initialisation of a stochastic automaton, clock $a$ is set to the time value of the temporal formula, and all other clocks are undefined. We define this initial valuation as $O(a)$, if $O(a) = n$.

We also need a notion of equivalence between the valuations, which we will use to construct a finite number of regions at each node within the probabilistic region tree.

**Definition 5** Two clock valuations $v$ and $v'$ are equivalent (denoted $v \equiv v'$) provided the following conditions hold:

- For each clock $x \in C \cup \{a\}$, either both $v(x)$ and $v'(x)$ are defined, or $v(x) = \bot$ and $v'(x) = \bot$.
- For every (defined) pair of clocks $x, y \in C \cup \{a\}. v(x) < v(y) \iff v'(x) < v'(y)$.

3 The range of a function $F_x$ is given by the set $\{ t \mid F_x(t) > 0 \}$. 
The same clocks are defined in each valuation, and the order of the values of the defined clocks is all that is important, since the actions are triggered by the first clock to expire. Therefore we only need to know whether one clock is greater than or less than another.

In building the region tree, each level of unfolding comprises two steps. First, the regions within the region tree are formed by distinguishing the equivalence classes at each node, then the nodes which can be reached given these equivalence classes are calculated using the SA.

The probabilistic region tree records the resolution of the nondeterministic choices and the probabilities at the final nodes represent the chances of taking the particular sequence of actions that end in that node.

At each iteration, we update the information we have on the probability of a path satisfying the formula. To do this, we define three new propositions, and each node of the probabilistic region tree is labelled with \( p \), \( f \) or \( u \): \( p \), if it has passed (it is the end of a path which models the bounded until formula \( \psi \)); \( f \), if it has failed (it is the end of a path which cannot model \( \psi \)), or \( u \), if it is undecided. We also have two global variables, \( \Sigma p \) and \( \Sigma f \), which keep running totals of the probabilities of the pass and fail paths.

The basic idea of the model checking algorithm is that we check the values of \( \Sigma p \) and \( \Sigma f \) at each stage, and if we cannot deduce from these the truth or falsity of the formula we are checking, we look more closely at the undecided nodes. That is, we extend the undecided paths by each possible subsequent action, label these new nodes \( p \), \( f \) or \( u \), and calculate their probabilities. We then add these probabilities to \( \Sigma p \) and \( \Sigma f \) and repeat.

To determine the probabilities on the arcs, we need to use probability density functions of the distribution functions, and integrate these in the order given by the valuation equivalence class. It is this integration that is the cause of the complexity in this region tree algorithm.

As an example, consider the formula mentioned earlier: \[ tt U_{<10} K_4 \succ 0.3 \], which states that the probability of the first car processed by the kiosk being processed within one minute from the kiosk opening is greater than 0.3. Even though the kiosk contains a deterministically set clock (\( c_2 \) is set to 12), we can analyse it using the region tree algorithm because no other clocks can expire at the same time.

An example of a nondeterministic region tree is shown in Figure 10. Consider first the SA in Figure 9. When the clock \( x \) fires, both transitions \( a \) and \( b \) are enabled, because both are governed by \( x \). This gives rise to the nondeterministic region tree in Figure 10, and if we are to model check such a region tree, the nondeterministic choice between \( a \) and \( b \) must be resolved by an adversary.

The region tree for this example is shown in Figure 11. Because there are no nondeterministic choices in this region tree, the probabilistic region tree will be structurally identical, the only difference being the labelling. For this reason, we do not present the probabilistic region tree.

Consider region \( K_1 \) first. Since we are interested in the behaviour of the kiosk after it opens, we have an adversary which makes the action open happen...
Fig. 9. A nondeterministic Stochastic Automaton

Fig. 10. A nondeterministic region tree

immediately. Thus the automaton moves to state $K_2$, where the clock $c_2$ is set. Since clock $c_2$ is deterministically set to 12, we only consider the valuation equivalence $c_2^2 < a^0$, and the region graph moves from region 0 to region 1. The superscript indicates that this is the first time the clock has been set.

The automaton moves to state $K_3$ when clock $c_2$ expires, this is represented by the transition from region 1 to region 2. The clock $a$ has not expired by this state, since it is greater than clock $c_2$, which has only just expired, but we have not yet reached state $K_4$, so in the probabilistic region tree we would label this state $u$ (undecided). In this state clock $c_3$ is set, and there are two valuation equivalences (where $a_1$ is the value of clock $a$ at the time of transition): $c_3 < a_1$ (represented by region 3) and $a_1 < c_3$ (represented by region 4). Both of these moves will move the automaton to state $K_4$ when clock $c_3$ expires, but in one instance (region 4 to region 6) it is too late, because clock $a$ has already expired, and so more than 60 seconds have passed. Region 6 is therefore labelled $f$ in the probabilistic region tree. In the other instance (region 3 to region 5) state $K_4$ has been reached within 60 seconds, and it is therefore labelled $p$.

To determine the exact probability of reaching region 5 (and any other regions labelled with $p$), we need to use the probability density functions associated with the clocks. In our example, since we know that the transition from state $K_2$ to state $K_3$ occurs at precisely 12 seconds, $a_1$ is 48, the problem reduces to solving the integration

$$
\int_0^{48} P_{c_3} dt
$$
which, since \( P_{c3}(t) = 0 \) when \( t \) is less than 30, is equal to

\[
\frac{1}{450} \int_{30}^{48} (t - 30) \, dt
\]

which evaluates to 0.36, and so the formula is true.

This method could easily be adapted to answer queries such as “What is the probability of reaching a certain state within a certain time?” and could return a precise answer.

When the time of occurrence of one event may be dependent on the time of occurrence of a large number of other events, all the probabilistic density functions must be considered in order to calculate the probability of occurrence of one event, and the integrals which result become very complex. In order to avoid this, we consider a second algorithm, which uses discretisation.

![Fig. 11. Diagram for region tree algorithm](image_url)

**The Second Algorithm – Approximations Using Discretisation.** This algorithm avoids the calculation of integrals that the region tree algorithm was forced to undertake. In order for the discretisation to be possible we need to make a number of assumptions. In particular, we assume that the range of the clock functions is made up of a finite number of left/right closed intervals; that is, we consider only functions \( F \) such that

\[
\{ t \mid F'(t) > 0 \} = \bigcup_{1 \leq j \leq n} [g_j, h_j]
\]

where \([g_j, h_j]\) is a left/right closed interval and \( n \) is the number of intervals in the derivative. For example, the distributions on the stochastic automata given in Figures 2 and 3 conform to this template. The template also allows deterministic timing since the upper and lower bounds of an interval may be of the formula \([\phi_0 \cup c < \phi_1]\) being satisfied at this point.\(^4\) To build the next snapshot, the algorithm picks out at each time point \( n\delta \) the transitions that the automaton is capable of during the next interval of length \( \delta \). Because \( \delta \) is less than the minimum of all the clock lower bounds, a maximum of one transition per path can occur in each interval. Recording all possible states of the automaton at each time point is therefore enough to record all the possible transitions.

\(^4\) We also require that \( \exists n, n\delta = c \), which ensures that one of the snapshots will be at exactly time \( c \).
A snapshot is built by deriving a matrix for each state \( s \) and time \( t \) (which is a rational number and calculated as \( n\delta \)), denoted \( \text{matrix}(s, t) \), and placing in this matrix a record of the probabilities of the various combinations of clock values in state \( s \) at time \( t \). Each matrix will have as many dimensions as its state has clocks.

Each entry in the matrix \( \text{matrix}(s, t) \) is the probability that at time point \( t \), the automaton is in state \( s \), and each clock is within a particular time range. Thus, the value \( \text{matrix}(s, t)[k_1 \ldots k_n] \) is the probability that at time point \( t \), the automaton is in state \( s \), and \( v(c_i) \in (\delta(k_i - 1), \delta k_i] \) for each clock \( c_i \).

The algorithm stops when either enough information has been gathered to determine the truth or falsity of the formula, or enough time has passed so that \( n\delta > c \), and allowing time to pass further will make no difference to the information we already have. In this case the result undecided is returned.

Consider again the formula \( [t \mathcal{U} \leq 60] K_4 \geq 0.3 \). We choose \( \delta \) to be 10. The first matrix to be constructed would be \( m(K_1, 0) \), but the state \( K_1 \) has no associated clocks, therefore the automaton moves immediately to state \( K_2 \), and \( \text{matrix}(K_2, 0) \) is constructed (see Figure 12).

This matrix tells us that the probability of clock \( c_2 \) being somewhere between the values 10 and 20 at time zero is 1.

There are two different procedures for updating a matrix (that is, to derive \( \text{matrix}(s, \delta(n + 1)) \)) from the matrices referring to time \( \delta(n) \), both of which correspond to different situations. The first corresponds to the situation within the stochastic automaton where time passes, but the state remains unchanged. In this case we must shift the clock configuration probabilities in the previous
matrix down by one index step (which corresponds to $\delta$ time passing) and add the result to the matrix we are updating.

This is the situation here, and $\text{matrix}(K_2, 10)$ is formed as in Figure 12.

The second procedure is applied when new states can be reached from the current state during the $\delta$ time passing, and involves determining the probability of entering these states. We do this by looking at all the probability values in the matrix where at least one of the indices has the lowest possible value (10, in this example). If this is the case then we know that at least one clock will expire during the ensuing $\delta$ timestep.

If only one index in the configuration has the value 10 then only one clock can expire, and only one state can be entered from this clock configuration, and so the matrix for that state is built.

This is the case from $\text{matrix}(K_2, d)$, and so we get $\text{matrix}(K_3, 2\delta)$, which tells us that the probability of being within state $K_3$ at time $2\delta$ with clock $c_3$ between values 30 and 40 is $\frac{1}{9}$, being within state $K_3$ at time $2\delta$ with clock $c_3$ between values 40 and 50 is $\frac{2}{9}$, and being within state $K_3$ at time $2\delta$ with clock $c_3$ between values 50 and 60 is $\frac{5}{9}$.

If more than one index has the value 10, then we simply do not explore that configuration any further, and the configuration probability is added to error. In the example we are considering, this possibility does not occur.

In our example, the matrices $\text{matrix}(K_3, 3\delta)$, $\text{matrix}(K_3, 4\delta)$, $\text{matrix}(K_3, 5\delta)$ and $\text{matrix}(K_3, 6\delta)$ can all be constructed simply by moving the clock configuration probabilities with the previous matrices.

The second way to update a matrix corresponds to a transition from one state to another within the automaton. For each matrix entry we calculate the clock configuration probability, multiply it by the probability of moving into this state at this time, and add it to the matrix entry we are updating. Thus, in the example, we get $\text{matrix}(K_4, 6\delta)$.

We have now reached the timepoint $6\delta$, which corresponds to 60 seconds, and so the sum of all the probability values in the matrix at this point ($\frac{1}{9} + \frac{2}{9} + \frac{5}{9} = \frac{1}{1}$) is a lower bound on the probability that state $K_4$ will have been reached by time 60. Thus, since we are interested in whether the probability is greater than 0.3, we can conclude that the formula is true. A smaller $\delta$ would produce a more accurate result, but we do not illustrate that here.

\section{Conclusions}

In this paper we have begun to tackle the problem of integrating various automaton based notations within a specification. Specifically, we have

- given a translation from stochastic automata to timed automata with deadlines and shown which properties are retained;
- presented two methods for model checking stochastic automata, the first of which builds regions from the automaton, and uses integration of the probability density functions and the second of which uses an approximation technique based on discretisation.
Translating stochastic automata to timed automata with deadlines means that, although the stochastic information is lost, we can analyse the composed specification for temporal properties to do with, for example, throughput within a certain time.

The model checking methods cannot consider the composed specification, and must be restricted to the individual components, although the effects of the environment may be represented by the adversary chosen.

The two model checking methods presented complement each other. The region method is best used when the size of the model to be explored is small, because the number of integrations to be performed goes up exponentially with the number of clocks. The discretisation method is more promising for larger models. It can produce upper and lower bounds on the probabilities, and is therefore best suited for queries such as “Does the probability of reaching a state $s$ by a time $t$ lie within the range $[a, b]$?”

We are currently seeking to implement the second algorithm, and to integrate it with the LUSCETA [12] tool. We would also like to consider how to model check more general stochastic automata, and in particular to allow clocks to be set and used in any state.

Acknowledgments. We are indebted to Pedro D’Argenio for supplying us with Definition 3 [9].

References


Structural Refinement in Object-Z / CSP

John Derrick\(^1\) and Graeme Smith\(^2\)

\(^1\) Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK.
\(^2\) Software Verification Research Centre, University of Queensland 4072, Australia

Abstract. State-based refinement relations have been developed for use on the Object-Z components in an integrated Object-Z / CSP specification. However this refinement methodology does not allow the structure of a specification to be changed in a refinement, whereas a full methodology would allow concurrency to be introduced during the development life-cycle. In this paper we tackle these concerns and discuss refinements of specifications written using Object-Z and CSP where we change the structure of the specification when performing the refinement. In particular, we develop a set of structural simulation rules which allow a single Object-Z component to be refined to a number of communicating or interleaved classes. We prove soundness of these rules and illustrate them with a small example.

1 Introduction

There has been an increasing amount of work recently on combining state based languages such as Z\(^{11}\) and Object-Z\(^{8}\) with process algebras such as CSP\(^{4}\) and CCS\(^{6}\). The motivation for the work is that these combinations of languages provide a suitable medium for the description of complex systems which involve aspects of both concurrency and non-trivial data structures. Indeed, there are many application areas for such an approach including the obvious area of distributed systems specification.

The combination we are interested in here is the use of Object-Z together with CSP. This combination of languages has been investigated by a number of researchers including Smith \(^7\), Fischer \(^2\) and Mahony and Dong \(^5\). In this paper we work with the integration described by Smith \(^7\) and Smith and Derrick \(^9,10\), although the concerns we address are relevant to all combinations of these two languages.

In the integration discussed here Object-Z is used to describe the components of a system, and these are then combined using CSP operators which describe how the components synchronise and communicate. For example, an elevator in a building might be described in this approach as (∥\(_n\).Name User\(_n\))∥LiftSys where User\(_n\) and LiftSys are given as Object-Z classes describing a user and the lift respectively, and the CSP operators || and || describe the interaction between these components. The combined notation benefits from reusing existing notations, being relatively simple and having a well-defined meaning given by a semantics based upon the failures-divergences semantics of CSP.
Of course as well as specifying systems we need a method of developing them, and there has been considerable work on refinement for both state-based languages and process algebras. This work has been applied to the combination of Object-Z and CSP by Smith and Derrick [9,10] who develop state-based refinement relations for use on the Object-Z components within an integrated specification. Because Object-Z classes have been given a CSP semantics in the combined language, the refinements are compositional so that when a single Object-Z component is refined then so is the overall Object-Z / CSP specification. For example, if we refine the single LiftSys component to LiftSys2 then application of the theory tells us that $(\|\|_n: Name User_n)\|\|LiftSys2$ will be a refinement of $(\|\|_n: Name User_n)\|\|LiftSys$.

However, the rules presented in [9,10] do not allow the structure of the specification to be changed in a refinement. That is, only single Object-Z classes can be refined individually and therefore the structure of the specification, and in particular the use of the CSP operators, has to be fixed at the initial level of abstraction. This is clearly undesirable and in this paper we provide a means to refine the very structure of a specification written in the integrated notation.

In particular, we develop refinement rules that allow us to refine a single Object-Z class into a number of communicating or interleaved classes. For example, we show how to refine the class LiftSys into a parallel composition of a class Lift, representing the lift, with a class Con, representing the lift controller. This approach therefore allows concurrency to be introduced during refinement as and when appropriate rather than having to fix the eventual structure of the implementation early in the development life-cycle. Similar concerns have been addressed by Fischer and Wehrheim [3], however there the emphasis was on using model checking in order to demonstrate refinement between specifications with different structures. Our work compliments this work nicely.

This paper is structured as follows. In Sections 2 and 3 we discuss the integration and refinement of specifications written using Object-Z and CSP. Then in Section 4 we look at refinements between specifications where we change their structure, and we develop rules for introducing the CSP parallel operator $\parallel$. These rules are illustrated using the elevator case study. In Section 5 we derive similar rules for the CSP interleaving operator $\mid\mid$. We conclude in Section 6.

2 Combining Object-Z and CSP

In this section we discuss the specification of systems described using a combination of Object-Z and CSP. In general the specification of a system described using these languages comprises three phases. The first phase involves specifying the components using Object-Z, the second involves modifying the class interfaces (e.g. using inheritance) so that they will communicate and synchronise as desired, and the third phase constructs the final system by combining the components using CSP operators.

Since all interaction of system components is specified via the CSP operators a restricted subset of Object-Z is used. In particular, there is no need for object
instantiation, polymorphism, object containment nor any need for the parallel or enrichment schema operators. Similarly not all CSP operators are required. For example, neither piping nor the sequential composition operator are needed. These restrictions help simplify the language and its semantic basis considerably.

### 2.1 Example - A Lift System

As a simple example we consider the classic lift case study [3]. The components of our system will be the users and the elevator system, and both are specified by Object-Z classes. To do so let \( \text{Name} \) denote the names of all possible users of the system, and suppose the available floors in a building are described as follows.

\[
\begin{align*}
\text{minFloor}, \text{maxFloor} & : \mathbb{N} \\
\text{minFloor} & < \text{maxFloor}
\end{align*}
\]

\( \text{Floor} = \text{minFloor} .. \text{maxFloor} \)

A single user is capable of one operation: to request a lift to a floor. The state variable \( \text{requested} \) is the set of all floors requested by the user.

\[
\begin{array}{l}
\text{User} \\
\hline
\text{name} : \text{Name} \\
\hline
\text{requested} : \mathcal{P} \text{Floor} \\
\hline
\text{INIT} \\
\text{requested} = \emptyset \\
\hline
\text{Request} \\
\Delta(\text{requested}) \\
\text{r}! : \text{Floor} \\
\text{requested}' = \text{requested} \cup \{r\}
\end{array}
\]

Our initial specification also contains the Object-Z class \( \text{LiftSys} \) which describes an abstract view of the elevator system. The class consists of four operations: \( \text{Request} \) models requests for the lift being made by customers, \( \text{CloseD} \) and \( \text{OpenD} \) model the closing and opening of the lift doors respectively, and \( \text{Move} \) describes the movement of the lift inside the shaft. Which request is serviced next is non-deterministic – any valid request is potentially chosen.

\[
\text{Status} ::= \text{open} | \text{closed} | \text{stop}
\]
To specify the complete system we combine the components together in a way which captures their interaction. If we define \textit{User}_n to be the \textit{User} class with \textit{name} instantiated to \textit{n} [7], then this interaction is given by

\[
\text{Building} = (\parallel n.\text{name User}_n)\parallel \text{LiftSys}
\]

which describes a single lift with which a number of users can independently interact.

2.2 The Semantic Model

Combined Object-Z and CSP specifications are given a well-defined meaning by giving the Object-Z classes a failures-divergences semantics identical to that of a CSP process. In a failures-divergences semantics a process is modelled by a triple \((A, F, D)\) where \(A\) is its alphabet, \(F\) its failures and \(D\) its divergences. The failures of a process are pairs \((t, X)\) where \(t\) is a finite sequence of events that the process may undergo, and \(X\) is a set of events the process may refuse to perform after undergoing \(t\).

To give Object-Z classes a failures-divergences semantics the failures of a class are derived from its \textit{histories}, that is from the Object-Z semantic model of a class [10].

In doing so Object-Z operations are mapped to CSP events using the following function which turns an operation \textit{op} with assignment of values to its parameters \(p\) to the appropriate event:
\[
\text{event}(\langle op, p \rangle) = \text{op.}\beta(p)
\]

The meta-function \( \beta \) replaces each parameter name in \( p \) by its basename, i.e., it removes the ? or !. Thus the event corresponding to an operation \( \langle op, p \rangle \) is a communication event with the operation name \( op \) as the channel and an assignment of values to the basenames of the operation’s parameters as the value passed on that channel. For example, the event corresponding to a user requesting a floor \( f \) is \( \text{Request.}\{r,f\} \).

Since Object-Z does not allow hiding of operations (hiding is only possible at the CSP level), divergence is not possible within a component. Therefore a class is represented by its failures together with empty divergences.

As well as giving a well-defined meaning to a combined Object-Z / CSP specification, the semantics also allows a coherent theory of refinement to be developed. We discuss this in the next section.

3 Refinement in Object-Z and CSP

With our integrated semantics, refinement is based on CSP failures-divergences. Thus a specification \( C \) is a refinement of a specification \( A \) if

\[
\text{failures } C \subseteq \text{failures } A \quad \text{and} \quad \text{divergences } C \subseteq \text{divergences } A
\]

and if we are considering a single Object-Z component we need consider only the failures since its divergences will be empty as noted above.

However, calculating the failures of a system is not practical for anything other than small specifications. To make the verification of refinements tractable we can adapt state-based verification techniques for use in our combined notation, and in particular adapt the idea of upward and downward simulations used in Z [12]. This allows refinements to be verified at the specification level, rather than working explicitly in terms of failures, traces and refusals at the semantic level.

The use of simulations between Object-Z components in the integrated notation is described by Smith and Derrick in [9,10]. In a simulation, a retrieve relation \( Abs \) links the abstract state (\( AState \)) and the concrete state (\( CState \)), and, for example, the definition of a downward simulation is as follows.

**Definition 1 Downward simulation**

An Object-Z class \( C \) is a downward simulation of the class \( A \) if there is a retrieve relation \( Abs \) such that every abstract operation \( AOp \) is recast into a concrete operation \( COp \) and the following hold.

\[
\begin{align*}
\text{DS.1} & \quad \forall AState; CState \quad \text{Abs} \implies (\text{pre } AOp \iff \text{pre } COp) \\
\text{DS.2} & \quad \forall AState; CState; CState' \quad \text{Abs} \land COp \implies (\exists AState' \quad \text{Abs}' \land AOp) \\
\text{DS.3} & \quad \forall CInit \exists AInit \quad \text{Abs}
\end{align*}
\]
Not all refinements change the state space, those that do not are called operation refinements as opposed to data refinements and these can be verified with a retrieve relation which is the identity (thus simplifying the refinement rules).

The simulation rules allow a single Object-Z class to be refined by another. For example, we might refine the LiftSys component to LiftSys2. This new lift system is identical to the initial specification except that our Move operation is more deterministic and chooses the nearest requested floor instead of an arbitrary one.

LiftSys2

\[
\text{Move} \quad \Delta (req, pos, door) \\
\text{Door} : \text{Floor} \\
\quad \text{door} = \text{closed} \land \text{door}' = \text{stop} \\
\quad pos' \in \text{req} \land \neg (\exists p : \text{req} \bullet \text{pos} - p < | \text{pos} - \text{pos}' |) \\
\quad \text{req}' = \text{req} \setminus \{\text{pos}'\} \\
\quad \text{pos}' = \text{f!} 
\]

This refinement can be verified in the standard way using a downward simulation, and since simulations are together sound and complete with respect to CSP failures-divergences refinement, (\(\parallel_n: \text{Name User}_n\))\(\parallel\text{LiftSys2}\) is a refinement of (\(\parallel_n: \text{Name User}_n\))\(\parallel\text{LiftSys}\).

However, this refinement is between single Object-Z classes and simulations do not allow us to change the overall structure of the specification. To understand the problem consider the following example.

3.1 Example - Changing the Structure of the Lift Specification

In an implementation we wish to refine the lift system into two separate components (a lift and a controller) which reflect more accurately the underlying physical configuration. The Lift class will control the position and movement of the lift, whilst the controller Con will marshal the requests and determine the next floor that the lift should service.

The Lift class consists of a position (pos), a door and a current target, and the class Con keeps track of the current requests (req). The two classes communicate in order to determine the current position and the new target floor.

The Lift class is as follows.
The controller accepts requests and determines which floor the lift should go to next. Now, instead of being completely non-deterministic, the floor closest to the current position is chosen. To achieve this, the two classes communicate when performing the \texttt{SetTarget} operation. A similar communication takes place in the \texttt{Move} operation to determine which floor has been reached. We have also changed the granularity of \texttt{Move} from \texttt{LiftSys} by using \texttt{MoveOneFloor} to move the lift one floor at a time. Since Object-Z operations are blocked outside their preconditions, \texttt{MoveOneFloor} can only happen a finite number of times in a row.

Note also that neither class contains the complete temporal ordering of operations. This will be determined by the final synchronisation between the two classes.
It is then possible to show (e.g. by a calculation of failures-divergences) that LiftSys is refined by \((\text{Lift} \parallel \text{Con}) \setminus \{\text{SetTarget}, \text{MoveOneFloor}\}\). However, we cannot use simulations to verify the refinement. Furthermore, if we compare the LiftSys component with those given by Lift and Con we cannot even claim that individually the latter classes refine LiftSys. In particular,

- the classes are not conformal, i.e. neither Lift nor Con contain all the operations in LiftSys, yet they also contain additional operations such as SetTarget;
- the new operations have additional inputs and outputs, and
- the behaviour of the operations is different, e.g. the preconditions have been changed.

Yet clearly LiftSys is refined by \((\text{Lift} \parallel \text{Con}) \setminus \{\text{SetTarget}, \text{MoveOneFloor}\}\), and what we seek to do is to derive state-based techniques that allow us to verify refinements like these without having to expand the synchronisation between the two classes and then calculate their failures. The next section discusses how we can do this.

---

1 Throughout this paper we use the shorthand \(C \setminus \{Op\}\) to denote the hiding of all events corresponding to the operation \(Op\). In general, the names of these events will consist of, as well as the name of the operation, a value corresponding to the mapping of the operation’s parameters to their values.
4 Structural Refinement

In this section, we present simulation rules that allow us to prove refinements between Object-Z classes and CSP expressions involving more than one Object-Z class. Specifically, we look at rules for introducing the CSP parallel operator \( \parallel \).

In Smith and Derrick [9,10], simulation rules which correspond to failures-divergences refinement were presented for refining an Object-Z class to another Object-Z class. To build on this work, we show, in this section, how to construct an Object-Z class which is semantically equivalent to a CSP expression involving parallel composition and hiding. This constructed class, and hence the equivalent CSP expression, can be shown to be a refinement of another class using the existing simulation rules.

From the relationship between the schemas of the constructed class and those of the component classes of the CSP expression, we can also re-express the existing simulation rules in terms of schemas of the component classes.

Figure 1 illustrates the process. We wish to refine a class \( A \) into \( (D \parallel B) \setminus \{x_1, \ldots, x_n\} \). To do so we show that \( D \parallel B \) is failures-divergences equivalent to the Object-Z class \( E \) and that \( E \setminus \{x_1, \ldots, x_n\} \) is failures-divergences equivalent to the class \( C \), and then, from the existing simulation rules, we derive alternative simulation rules to show that \( C \) is a downward simulation of \( A \). The usefulness of the approach is that these alternative rules are expressed in terms of the original classes \( B \) and \( D \), thus these rules allow us to verify the refinement without constructing the semantically equivalent class \( C \).

![Diagram of Approaches to Refinement](image)

**Figure 1. Approaches to refinement**

In Section 4.1 we show how to construct the semantically equivalent class and in Section 4.2 we derive the simulation rules in terms of the component classes of the CSP expression. In Section 4.4 we prove the refinement of the Lift example from Section 3. The ideas are extended to the CSP interleaving operator \( || \) in Section 5.

4.1 Constructing an Equivalent Class

Our approach works for refinements of a class \( A \) into specifications of the form 
\[
(D \parallel B) \setminus \{x_1, \ldots, x_n\}
\]
where classes \( D \) and \( B \) and events \( x_1, \ldots, x_n \) are restricted as follows.
1. The variables declared in the state schema of class $D$ are distinct from those declared in the state schema of class $B$.

2. Any operations common to $D$ and $B$ (i.e. they have the same operation name) have parameters with identical basenames (i.e., apart from the '?'s and '!'s).

3. Each hidden event $x_i$, $i \in 1 \ldots n$, must, in one of the classes $D$ or $B$, occur a finite number of times immediately before a visible event $y$ corresponding to one particular operation and not at any other time. In such cases, the finite sequence of hidden events followed by the event $y$ represents an operation refinement of an event $y$ of the abstract specification.

4. When an operation name is shared by $D$ and $B$, an input in one of the operations with the same basename as an output in the other cannot be constrained more than the output. That is, given that $Op$ in $D$ has input $x?$ and predicate $p$ and $Op$ in $B$ has output $x!$ and predicate $q$, the following must hold.

$$\exists BState, BState' \bullet q \Rightarrow \exists DState; DState' \bullet p[x!/x?]$$

where $DState$ and $BState$ are the state schemas of classes $D$ and $B$ respectively, and $p[x!/x?]$ is the predicate $p$ with all free occurrences of $x?$ renamed to $x!$.

These restrictions are in fact entirely natural consequences of the events $x_1, \ldots, x_n$ acting as a communication medium between the two classes.

Restriction 1 allows us to derive simulation rules expressed as rules on the two separate classes. Restriction 2 says that operations common to $D$ and $B$ will communicate on common channels, and restriction 3 stops divergence due to infinite sequences of hidden events.

Restriction 4 allows us to construct an equivalent class by combining same-named operations with the Object-Z associative parallel composition operator $\parallel$ [8]. This operator conjoins its argument operations and renames any inputs in one operation for which there exists a common-named output in the other operation to an output. The common-named parameters are hence identified in the conjoined operation and exist as an output.

To see why the restriction is needed, consider the following same-named operations from classes $D$ and $B$.

\[
\begin{array}{c}
\text{Op} \\
x?: \mathbb{N} \\
x? \leq 5 \\
\end{array}
\quad
\begin{array}{c}
\text{Op} \\
x!: \mathbb{N} \\
x! \leq 10 \\
\end{array}
\]

When combined, the operations communicate via their parameters. The predicate of the operation from $D$, that with the input, places a stronger condition on the communicated value than the predicate of the operation from $B$ (thus restriction 4 is not satisfied). The result is that the combined operation can occur with the communicated value less than or equal to 5.

Now consider refining the operation in $B$ to the following.
This is possible since refinement allows conditions on outputs to be strengthened [10]. However, now the combined operation can never occur since there is no value of the communicated variable which satisfies both the operation in $D$ and the operation in $B$. Hence, despite the individual classes $D$ and $B$ being refined, the resulting equivalent class is not refined (since we have effectively increased the refusals for any trace after which $Op$ could have been performed). Restriction 4 prevents this situation from occurring.

We will now show how to construct an equivalent class $C$ for the CSP expression $(D \parallel B) \setminus \{x_1, \ldots, x_n\}$ by considering first parallel composition and then hiding.

**Parallel composition.** Consider classes $D$ and $B$ below where $i, j, k \in \mathbb{N}$ and $j \leq i$.

$$
\begin{array}{c}
D \\
DState \\
DINIT \\
Op_1 \\
\vdots \\
Op_i \\
\hline \\
B \\
BState \\
BINIT \\
Op_{i-j+1} \\
\vdots \\
Op_k \\
\end{array}
$$

When $j \neq 0$, the classes share the operation names $Op_{i-j+1} \ldots Op_i$.

The parallel composition of classes $D$ and $B$, $D \parallel B$, is semantically equivalent to the following class.

$$
\begin{array}{c}
E \\
DState \wedge BState \\
DINIT \wedge BINIT \\
Op_1 \\
\vdots \\
Op_k \\
\end{array}
$$

where for each $n : 1 \ldots i-j$, $Op_n$ is defined as in $D$, and for each $n : i+1 \ldots k$, $Op_n$ is defined as in $B$, and for each $n : i-j+1 \ldots i$, $Op_n$ is the associative parallel composition of the definition in $D$ with the definition in $B$, i.e. $D.Op_n \parallel B.Op_n$.

Therefore, for each $n : i-j+1 \ldots i$, due to the $DState$ and $BState$ declaring distinct variables (restriction 1), $Op_n$ in $E$ transforms those variables from $DState$ according to the operation $Op_n$ of $D$ and those variables in $BState$ according to the operation $Op_n$ of $B$. Furthermore, $Op_n$ in $E$ has parameters with
identical basenames to those in $Op_n$ of $D$ and $B$. Therefore, the alphabet of $E$ is the union of the alphabets of $D$ and $B$.

To see why the constructed class $E$ is equivalent to $D \parallel B$, consider deriving the failures of $E$ by the approach outlined in [10]. The failures of $E$ are all traces $s$ and refusal sets $X \cup Y$ where

- $s$ is a trace comprising events corresponding to operations $Op_1 \ldots Op_k$,
- $X$ and $Y$ are sets of events corresponding to operations in $D$ and $B$ respectively,
- after $s$, since $DState$ is only changed by events corresponding to operations of $D$ (due to $DState$ and $BState$ declaring distinct variables), $X$ includes only those events that can be refused by $D$ after undergoing trace $s$ restricted to the alphabet of $D$,
- similarly, $Y$ includes only those events that can be refused by $B$ after undergoing trace $s$ restricted to the alphabet of $B$.

Hence,

$$\text{failures}(E) = \{ s, X \cup Y \mid s \in \text{alphabet}(D) \cup \text{alphabet}(B) \wedge (s \triangleright \text{alphabet}(D), X) \in \text{failures}(D) \wedge (s \triangleright \text{alphabet}(B), Y) \in \text{failures}(B) \}$$

Since an Object-Z class has no divergences, this is equivalent to the failures of $(D \parallel B)$ as given by Hoare [4].

**Hiding.** Consider the class $E$ below where $Op_2$ occurs $m : \mathbb{N}$ times before each occurrence of $Op_3$ and not at any other time.

```
E
EState
EINIT
Op1
Op2
Op3
```

The CSP expression which hides $Op_2$ in $E$, i.e., $E \setminus \{Op_2\}$, is semantically equivalent to the following class$^2$. ‘$Op_3$’ denotes the name of the operation $Op_3$ in $E$, as opposed to its definition denoted simply by $Op_3$. The effect of ‘$Op_3$’ is to perform $Op2 \setminus (x_1, \ldots, x_n)$ $m$ times in sequence followed by the original operation $Op_3$. ($x_1, \ldots, x_n$ are the parameters of $Op_2$.)

```
C
EState
EINIT
Op1
‘Op3’ ≡ (\(i : \mathbb{N} \mid i < m \cdot Op2 \setminus (x_1, \ldots, x_n)\)) \(\triangleright\) Op3
```

$^2$ Note that the syntax for hiding operation parameters in Object-Z, $Op \setminus (x)$, is different to that of hiding events in CSP, $P \setminus \{x\}$. 
Consider deriving the failures of the constructed class $C$ by the approach outlined in [10]. The failures of $C$ are all traces $t$ and refusal sets $X$ where there exists a failure $(s, Y)$ of $E$ such that $s$ restricted to the events of $C$ is $t$, and $Y$ includes, as well as the events in $X$, all events corresponding to $Op_2$.

Hence,

$$\text{failures}(C) = \{ s \triangleright \text{alphabet}(C), X \mid (s, X \cup \{ Op_2 \}) \in \text{failures}(E) \}$$

Since an Object-Z class has no divergences, this is equivalent to the failures of $E \setminus \{ Op_2 \}$ as given by Hoare [4]. The definition can be extended for hiding of multiple operations in the obvious way. In fact, what we are using here is a restricted form of weak refinement [1] where we compare in the refinement an abstract operation with a concrete one preceded by a finite number of internal operations.

4.2 Deriving the Simulation Rules

Given a refinement $(D \parallel B) \setminus \{ x_1, \ldots, x_n \}$ of $A$, we could verify the refinement by constructing an equivalent class as outlined in Section 4.1 and using the simulation rules of Smith and Derrick [10]. However, it is preferable not to have to construct an equivalent class but to instead have rules which refer directly to the schemas of the component classes $D$ and $B$. We now show how we can derive these rules.

**Parallel composition.** We begin by considering the case where we have parallel composition only (and no hiding). For operation names occurring in only one component class, the operation given this name in the constructed class is identical to that in the component class in which it occurs. Hence, the simulation rules are unchanged.

For shared operations, however, the operation in the constructed class is the associative parallel composition of the operations in the component classes. In this case to verify the refinement we can use the downward simulation rules $\text{DS.1}$ and $\text{DS.2}$ which, for the communicating operations, require that:

- $\text{DS.1}' \forall AState; DState; BState \bullet \text{pre } AOp \iff \text{pre } (DOp \parallel BOp)$
- $\text{DS.2}' \forall AState; DState; BState; DState'; BState' \bullet$
  $$ (DOp \parallel BOp) \implies (\exists AState' \bullet AOp)$$

where $DState$ and $BState$, and $DOp$ and $BOp$ are the two component states and operations respectively.

These rules still involve an operation, $DOp \parallel BOp$, to be constructed from the two classes. However, we can by-pass the necessity to construct this operation as follows. Consider the following operations $DOp$ and $BOp$ ($p$ and $q$ are predicates).
\[ \text{DOp} \parallel \text{BOp} \]

\[
\begin{array}{l}
\Delta(x) \\
x, x': X \\
z?: Z \\
p
\end{array}
\]

\[
\begin{array}{l}
\Delta(y) \\
y, y': Y \\
z!: Z \\
q
\end{array}
\]

where \( x \) and \( y \) are the state variables of the two component classes and are distinct (by restriction 1).

The associative parallel composition of these operations is

\[ \text{DOp} \parallel \! \text{BOp} \]

\[
\begin{array}{l}
\Delta(x, y) \\
x, x': X \\
y, y': Y \\
z!: Z \\
p[z! / z?] \land q
\end{array}
\]

where \( p[z! / z?] \) is the predicate \( p \) with all free occurrences of \( z? \) renamed to \( z! \).

Hence we can simplify the precondition calculation as follows:

\[
\text{pre}(\text{DOp} \parallel \! \text{BOp}) \equiv \exists x': X, y': Y, z!: Z \bullet p[z! / z?] \land q
\]

\[
\equiv \exists x': X, y': Y, z!: Z, w!: Z \bullet p[w! / z?] \land q
\]

[by restriction 4]

\[
\equiv (\exists x': X, w!: Z \bullet p[w! / z?]) \land (\exists y': Y, z!: Z \bullet q)
\]

[since \( p[w! / z?] \) and \( q \) refer to distinct variables]

\[
\equiv \text{pre DOp}[w! / z?] \land \text{pre BOp}
\]

In addition, we have

\[ \text{DOp} \parallel \! \text{BOp} \equiv \text{DOp}[z! / z?] \land \text{BOp} \]

Extrapolating to the general case, we have the following.

\[
\text{pre}(\text{DOp} \parallel \! \text{BOp}) \equiv \text{pre DOp}[w_1! / z_1?, \ldots, w_n! / z_n?]
\]

\[
\land \text{pre BOp}[w_{n+1}! / z_{n+1}?, \ldots, w_{n+m}! / z_{n+m}?]
\]

\[ \text{DOp} \parallel \! \text{BOp} \equiv \text{DOp}[z_1! / z_1?, \ldots, z_n! / z_n?]
\]

\[
\land \text{BOp}[z_{n+1}! / z_{n+1}?, \ldots, z_{n+m}! / z_{n+m}?]
\]

Hence, the simulation rules can be re-expressed as follows.

**Definition 2** Parallel downward simulation

A CSP expression \( D \parallel B \) is an operation downward simulation of the Object-Z class \( A \) if \( D \) and \( B \) satisfy restrictions 1, 2 and 4 (above) and the following hold.
Hiding. When we have hiding (as well as parallel composition), for those particular operations which can be preceded by a finite sequence of hidden operations (as required by restriction 3), we replace the operation in the simulation rules with a sequential composition comprising the sequence of hidden operations and the operation. For example, if $O_{P3}$ can be preceded by $m : N$ occurrences of a hidden operation $O_{P2}$ (as in the example of Section 4.1), we replace $O_{P3}$ in the simulation rules by $(\forall i : N | i < m \cdot O_{P2} \setminus \{x_i, \ldots, x_n\})$.

To illustrate this, we consider two cases. The first when one of the hidden events $x_i$ occurs in just one of the classes $D$ and $B$ and has no parameters, and the second when it occurs in both classes and has parameters for communication. Our lift example illustrates both: MoveOneFloor only occurs in Lift and has no parameters whereas SetTarget occurs in Lift and Con and the parameters $f'$ and $pos!', f'$ and $pos?$ respectively, are used to communicate values.

Case 1. Suppose, without loss of generality, an event $x$ occurs in just one class $D$ in $(D \parallel B) \setminus \{x\}$ and can occur $m$ times in a row where $m \in S \subseteq N$. Let us denote this operation by $Dx$. Given $Dx$ has no parameters, we have to show a simulation between $A_{Op}$ and $Int_D \parallel (D_{Op} \parallel! B_{Op})$, where $Int_D$ is $\{ m : S \cdot (\forall i : N | i < m \cdot Dx) \}$.

Since the state spaces of $D$ and $B$ are disjoint this can be re-written as $(Int_D \parallel D_{Op}) \parallel! B_{Op}$, and thus the simulation rules for the operations require that:

\[
\text{PS.1 } \forall AState; DState; BState \bullet \quad \text{pre } A_{Op} \iff \text{pre } D_{Op}[w_1/?; \ldots, w_n/?] \\
\land \quad \text{pre } B_{Op}[w_{n+1}/?; \ldots, w_{n+m}/?] \\
\]

\[
\text{PS.2 } \forall AState; DState; BState; DState'; BState' \bullet \quad (Int_D \parallel D_{Op}[z_1/?; \ldots, z_n/?]) \\
\land \quad B_{Op}[z_{n+1}/?; \ldots, z_{n+m}/?] \\
\implies (\exists AState' \bullet A_{Op})
\]

Case 2. Suppose an event $x$ occurs in both classes (e.g. in order to perform a communication as in SetTarget) in $(D \parallel B) \setminus \{x\}$ and that it communicates over
channel $y$. Let $Dx$ denote the operation $x$ in class $D$ etc. Then we have to show a simulation between $AOp$ and $((Dx \parallel Bx) \setminus \{y\}) \parallel (DOp \parallel BOp)$.

Since the state spaces of $D$ and $B$ are disjoint and assuming $y$ does not occur as a parameter in $DOp$ and $BOp$, this can be re-written as $((Dx \parallel DOp) \parallel (Bx \parallel BOp)) \setminus \{y\}$. Since hiding distributes through pre, the simulation requirements become:

$$PS.1 \ \forall AState; DState; BState \bullet$$
$$\text{pre } AOp \iff \exists y! \bullet \text{pre}(Dx \parallel DOp[w_1!/z_1?, \ldots, w_n!/z_n?])$$
$$\land \text{pre}(Bx \parallel BOp[w_{n+1}!/z_{n+1}? \ldots, w_{n+m}!/z_{n+m}?])$$

$$PS.2 \ \forall AState; DState; BState; DState'; BState' \bullet$$
$$\exists y! \bullet ((Dx \parallel DOp[z_1!/z_1?, \ldots, z_n!/z_n?])$$
$$\land (Bx \parallel BOp[z_{n+1}!/z_{n+1}? \ldots, z_{n+m}!/z_{n+m}?])$$
$$\implies (\exists AState' \bullet AOp)$$

These rules can be generalised to multiple events and parameters, as well as to the case when the hidden event $x$ occurs more than once before its corresponding visible event.

### 4.3 Data Refinement

In this section we generalise the rules to cover data refinement. That is, we consider the case when the state space of $A$ is changed when refining this class to $(D \parallel B) \setminus \{x_1, \ldots, x_n\}$ and as noted in Section 3, a retrieve relation $Abs$ is used in these circumstances to verify the simulation rules. Our task here then is to determine how this impacts on the structural refinement rules given in Definition 2.

In taking a change of data into account we first note that the construction of the single class $C$ which is semantically identical to $(D \parallel B) \setminus \{x_1, \ldots, x_n\}$ is unchanged. Hence the impact of data refinement only occurs in using the simulation rules when verifying the refinement from $C$ to the original component class $A$. The rules for parallel composition are easily expressed as follows.

$$PS.1 \ \forall AState; DState; BState \bullet$$
$$Abs \implies (\text{pre } AOp \iff \text{pre } DOp[w_1!/z_1?, \ldots, w_n!/z_n?])$$
$$\land \text{pre } BOp[w_{n+1}!/z_{n+1}? \ldots, w_{n+m}!/z_{n+m}?])$$

$$PS.2 \ \forall AState; DState; BState; DState'; BState' \bullet$$
$$Abs \land DOp[z_1!/z_1?, \ldots, z_n!/z_n?]$$
$$\land BOp[z_{n+1}!/z_{n+1}? \ldots, z_{n+m}!/z_{n+m}?]$$
$$\implies (\exists AState' \bullet AOp \land Abs')$$

$$PS.3 \ \forall DInit \land BInit \bullet (\exists AInit \bullet Abs)$$
4.4 Example

Using these rules we can now show that LiftSys is refined by \((\text{Lift} \parallel \text{Con}) \setminus \{\text{SetTarget}, \text{MoveOneFloor}\}\). To do so we must check that the decomposition satisfies the restrictions 1–4 outlined at the start of Section 4.1, and then verify conditions PS.1-3.

Restrictions. Restriction 1 clearly holds. For restriction 2 we have to compare parameters in SetTarget, CloseD and Move between the two classes Lift and Con. In each case the basenames in the pairs of operations are the same (e.g. \(f\) in Move).

The hidden operations are SetTarget and MoveOneFloor. A Boolean variable in each class has been inserted to ensure that SetTarget happens once, but only once, before each (visible) CloseD operation. Restriction 3 therefore holds for SetTarget.

MoveOneFloor is slightly more complicated, because in effect the refinement has decomposed the Move in LiftSys into a finite number of MoveOneFloor operations followed by a Move in the classes Lift and Con. However, restriction 3 is still satisfied because MoveOneFloor can only happen a finite number of times (until the lift reaches its target), after which a Move must happen.

We also have to check the restrictions on the predicates given by 4. Thus, for example, for the Move operation we have to check:

\[
\exists \text{pos, target, door, pos}', \text{target}', \text{door}' \quad \quad \\
\text{door} = \text{closed} \land \text{door}' = \text{stop} \\
\text{pos} = \text{target} \land f \neq \text{pos} \\
\Rightarrow \\
\exists \text{req, close, req}', \text{close}' \quad \text{req} \neq \varnothing \land \text{req}' = \text{req} \setminus \{f!\}
\]

which, since no constraints are being placed on the input, is trivially satisfied.

Simulation rule PS.1. For each operation in LiftSys we have to show that either it is a standard refinement if it occurs in just one class, or show that PS.1 holds if it occurs in both classes.

For the former, Request and OpenD are identical in the refinement, and both operations appear in just one class.

For operations CloseD and Move we are going to have to demonstrate that PS.1 holds, remembering that we have to take into account the hidden operations SetTarget and MoveOneFloor in doing so.

Consider the Move operation. The MoveOneFloor hidden event can occur a finite number of times before it, thus according to the above we need to consider the effect of \(\bigcup m : (0..\text{maxFloor} - \text{minFloor}) \bullet (\exists i : \mathbb{N} \mid i < m \bullet \text{MoveOneFloor}) \parallel \text{Move}\). Although this sounds complicated, it is not difficult in practice. Simply looking at the behaviour of MoveOneFloor and Move together shows us that we have to verify (with Abs equating the variables req, pos and door of LiftSys with those of Lift and Con):
Abs \land \text{pre \textit{LiftSys}.\textit{Move} } \iff
\text{pre \textit{Lift}.\left( \bigwedge_{m \in \{0..\text{maxFloor} - \text{minFloor}\}} \left( \bigvee_{i \in \mathbb{N}, i < m} \bullet \text{\textit{MoveOneFloor}} \right) \right) \land \text{pre \textit{Con}.\textit{Move}}}

and this boils down to the trivial

\text{door} = \text{closed} \land \text{req} \neq \varnothing \iff \left( \text{door} = \text{closed} \right) \land \left( \text{req} \neq \varnothing \right)

**Simulation rule PS.2.** In a similar fashion we must show PS.2 holds for both \textit{CloseD} and \textit{Move}. For example, for \textit{CloseD} we need to show that

\text{Con.(\textit{SetTarget} \parallel \textit{CloseD}[\text{pos}/\text{pos}])} \land \text{\textit{Lift}.(\textit{SetTarget} \parallel \textit{CloseD}[f'/f])} \Rightarrow \exists \text{\textit{LiftSysState} \bullet \text{\textit{LiftSys}.\textit{CloseD}}}

This amounts to showing that

\begin{align*}
&f! \in \text{req} \land \neg \exists p : \text{req} \bullet \left( \begin{array}{l}
pos? - p < \mid pos? - f! \\
doors = \text{open} \land door' = \text{closed} \land f! = \text{target}' \land \text{pos}! = \text{pos}
\end{array} \right) \\
&\Rightarrow \exists \text{\textit{LiftSysState} \bullet req \neq \varnothing \land door = \text{open} \land door' = \text{closed}
\end{align*}

which again is clearly true.

**Simulation rule PS.3.** This amounts to showing that together the initialisations of \textit{Lift} and \textit{Con} imply the initialisation in \textit{LiftSys}.

Therefore \textit{LiftSys} is refined by \langle \text{\textit{Lift}.\textit{Con}} \rangle \setminus \{\textit{SetTarget}, \textit{MoveOneFloor}\}, and since the hidden events do not occur in \textit{User}_n, \langle \parallel_{n} \text{\textit{Name User}_n} \parallel \text{\textit{LiftSys}} \rangle is refined by \langle \parallel_{n} \text{\textit{Name User}_n} \parallel \langle \text{\textit{Lift}.\textit{Con}} \rangle \setminus \{\textit{SetTarget}, \textit{MoveOneFloor}\}\rangle.

5 Rules for Introducing Interleaving

We can also derive rules which allow us to refine a class \textit{A} into \textit{D} \parallel \textit{B}, by a similar derivation to the above. Given a CSP expression involving interleaving of the form \textit{D} \parallel \textit{B}, an equivalent class can be constructed following the approach for parallel composition except that the Object-Z choice operator [8], denoted [], is used in place of associative parallel composition to combine common-named operations. (Reasoning in terms of failures similar to that for parallel composition can be used to show why this is the case.) Because there is no communication between the components \textit{D} and \textit{B} there is no need to impose restriction 4 that was needed for the parallel composition operator.

The choice operator disjoins its arguments adding first to each a predicate stating that variables in the \Delta-list of the other operation which are not also in their \Delta-list remain unchanged. It also has a requirement that the combined operations have the same parameters.
Given the operations, \( DOp \) and \( BOp \) of Section 4.2, therefore, the operation in the equivalent constructed class is

\[
\begin{align*}
&\Delta(x, y) \\
x, x' : X \\
y, y' : Y \\
z?, z! : Z \\
(p \land y' = y) \\
\lor \\
(q \land x' = x)
\end{align*}
\]

Hence, we can simplify preconditions as follows.

\[
\text{pre}(DOp || BOp) \equiv \exists x' : X; y' : Y \cdot (p \land y' = y) \lor (q \land x' = x) \\
\equiv (\exists x' : X; y' : Y \cdot p \land y' = y) \\
\lor \\
(\exists x' : X; y' : Y \cdot q \land x' = x) \\
\equiv (\exists x' : X; p) \lor (\exists y' : Y \cdot q) \\
\text{[since } y' \text{ is not free in } p \text{ and } x' \text{ is not free in } q] \\
\equiv \text{pre } DOp \lor \text{pre } BOp
\]

In addition, we have

\[
DOp || BOp \equiv (DOp \land \lfloor y' = y \rfloor) \lor (BOp \land \lfloor x' = x \rfloor) \\
\equiv DOp \lor BOp
\]

since \( y \) is not in the \( \Delta \)-list of \( DOp \), and hence we implicitly have \( y' = y \) in \( DOp \) in the constructed class, and \( x \) is not in the \( \Delta \)-list of \( BOp \).

Hence, the simulation rules can be re-expressed as follows.

**Definition 3** Interleaving downward simulation

A CSP expression \( D ||| B \) is an operation downward simulation of the Object-Z class \( A \) if \( D \) and \( B \) satisfy restrictions 1-2 and the following hold.

**IS.1** \( \forall AState; DState; BState \cdot \text{pre } AOp \iff \text{pre } DOp \lor \text{pre } BOp \)

**IS.2** \( \forall AState; DState; BState; DState'; BState' \cdot DOp \lor BOp \implies (\exists AState' \cdot AOp) \)

**IS.3** \( \forall DInit \land BInit \cdot (\exists AInit \cdot \text{true}) \)

Given restriction 3, these rules can be modified for hiding in exactly the same way as the parallel composition rules. They also extend to data refinement in the obvious way.

### 6 Conclusions

The purpose of this paper has been to present some refinement rules for developments where we change the structure of a combined Object-Z / CSP specification.
The simulation rules we have developed are sound, since we have verified them with respect to failures-divergences refinement, but they are clearly not yet complete. That is, there will be further refinements of structure that cannot be verified by application of Definitions 2 and 3. One further avenue for this research is to extend these rules to a set of complete structural refinement rules.

One aspect that could be considered are the restrictions placed on the decomposition, and in particular, one could envisage relaxing the third restriction concerning when hidden events can occur. It is likely that a full generalisation would use the ideas of weak refinement [1] here to produce quite arbitrary combined classes.

This work could also be extended to other combinations of Object-Z and CSP, such as those described in [2,5].

References

Towards a Unified Development Methodology for Shared-Variable Parallel and Distributed Programs

Jürgen Dingel

Department of Computing and Information Science
Queen’s University, Kingston, Canada
dingel@cs.queensu.ca

Abstract. A formal framework for the design of distributed, message-passing programs from shared-variable parallel programs is presented. Based on a uniform semantic model for both paradigms and a trace-based refinement calculus, we show how a shared-variable parallel program can be refined into a distributed program. The calculus is used to introduce iteration, parallelism, and local channels, to replace access to shared variables by message-passing primitives, and to update the channels such that processes find the expected information on the expected channels at the right time. The methodology is illustrated with the development of a distributed implementation of an all-pair, shortest-paths algorithm.

1 Introduction

Despite their appealing performance-to-cost ratio and the increasing availability of tools, parallel computers still fail to have the expected impact on mainstream computing. A close look at the state of the art in parallel computing suggests at least two reasons:

1. Parallel programming is inherently complex. Compared to sequential programming, the developer additionally must deal with, for instance, interference, race conditions, process creation and termination, shared resources and consistency, synchronization and deadlock. Successful treatment of these issues requires knowledge about, for instance, the location, interconnection, and relative speeds of processors, and the location of and access to data. In parallel programs, efficiency in terms of explicit, fine-grained parallelism seems to exclude robustness, maintainability, and verifiability.

2. The paradigms, patterns and formal models of program execution for various parallel architectures differ substantially. This lack of commonality makes almost every task involving parallel programming very architecture-dependent. For instance, it is hard to move a program and verification efforts from one architecture to another. Typically, a program must be modified substantially to take full advantage of, or even to execute on, a different architecture. In short, parallel programs usually are not portable. The loss of portability in turn limits the expected lifetime of parallel implementations and their economic viability.
Proof systems, program transformation frameworks and refinement calculi have been suggested to overcome the first problem, see e.g., [11,14,3,2,15,1]. In particular, in previous work we have introduced a trace-based refinement calculus for shared-variable parallel programs [10]. The calculus supports the successive, formal derivation of shared-variable parallel programs from high-level specifications and has been used for the development of, for instance, versions of the Floyd-Warshall algorithm and the n-process tie-breaker algorithm for mutual exclusion that exhibit more parallelism than the standard textbook implementations [9].

To address the second problem, this paper extends our previous work mentioned above by presenting a uniform semantic model for shared-variable and message-passing concurrency and by extending the refinement calculus to include message-passing constructs. Modeling the two paradigms in the same semantic domain enables us to move easily between them. For instance, the reuse of an implementation using one paradigm for the development of another implementation using a different paradigm becomes possible. In particular, we show how a shared-variable parallel implementation of an algorithm can be refined into an equivalent distributed message-passing implementation. The treatment of liveness properties is currently subject to further work, but initial results are promising.

Sections 2 and 3 summarize the refinement calculus presented in [10]. Section 4 presents refinement rules for the introduction of iteration and parallelism. Section 5 shows how to model channels and the standard message-passing constructs. Section 6 presents a detailed example. Section 7 discusses the treatment of liveness properties and Section 8 concludes.

2 Syntax and Semantics of Programs

We describe the language used to represent abstract specifications and executable programs. For convenience, all elements of this language will be called programs.

**Programs.** Let \( \text{Var} \) denote the set of all program variables. An atomic statement has the form of Morgan’s specification statement \( V::[P,Q] \), where \( V \subseteq \text{Var} \) is a finite set of variables and \( P \) and \( Q \) are first-order predicates [12]. It describes a single atomic transition transforming a state satisfying \( P \) into a state satisfying \( Q \) by just changing the variables in \( V \). For instance, the statement \( \{x\}::[tt, x \geq 0] \) describes a random assignment to \( x \). An idling, or stuttering, step is expressed as \( \text{skip} \equiv \emptyset::[tt, tt] \) where \( tt \) denotes \( true \). To be able to refer to the value a variable held initially, that is, at the beginning of the transition, we reserve “hooked” variables \( \vec{x} \) in \( Q \). If a predicate does not contain hooked variables it is called unary. Otherwise it is called binary. In a statement \( V::[P,Q] \), \( P \) must be unary, whereas \( Q \) may be unary or binary. The semantics of atomic statements is captured by characteristic formulas.
Definition 1. Characteristic formula

The characteristic formula \( cf_{V::[P,Q]} \) of an atomic statement \( V::[P,Q] \) is given by

\[
 cf_{V::[P,Q]} \equiv \overrightarrow{P} \land Q \land \forall t \in \text{Var}\ V . \overrightarrow{t} = t
\]

where \( \overrightarrow{P} \) abbreviates the substitution of all free unhooked variables in \( P \) by their hooked counterpart and \( t \) is a metavariable that ranges over program variables.

We interpret a binary predicate \( Q \) over pairs of states \((s,s')\) where \( s \) assigns values to hooked variables and \( s' \) to the unhooked ones. Thus, \( (s,s') \models Q \) iff replacing the hooked variables in \( Q \) by their values in \( s \) and replacing the unhooked variables in \( Q \) by their values in \( s' \) makes \( Q \) true.

More complex programs are built using sequential and parallel composition, disjunction, iteration, and hiding using the grammar below. We assume that every program variable \( x \) has a domain \( \text{Dom}_x \) of values associated with it.

\[
 C ::= V::[P,Q] \mid C_1 ; C_2 \mid C_1 \lor C_2 \mid C_1 \parallel C_2 \mid C^* \mid C^\omega \mid \text{new } x = v \text{ in } C
\]

where \( v \in \text{Dom}_x \). The set of free variables \( \text{fv}(C) \) of a program \( C \) is defined inductively as usual with \( \text{fv}(V::[P,Q]) = V \cup \text{fv}(P) \cup \text{fv}(Q) \) as the base case.

Transition Traces. Let \( s,s',s_i \in \Sigma \) denote states, that is, mappings from the set of program variables \( \text{Var} \) to values. Transition traces\(^1\)

\[
(s_0,s'_0)(s_1,s'_1) \ldots (s_i,s'_i) \ldots
\]

have proven very useful for the definition of compositional models of shared-variable concurrency [13,7,4]. One such trace represents a possible finite or infinite “interactive” computation of a program in which state changes made by the program (from \( s_i \) to \( s'_i \)) are interleaved by state changes made by its environment (from \( s'_i \) to \( s_{i+1} \)). The meaning of a program is given by a set of transition traces closed under two conditions: stuttering and mumbling. Brookes has used these conditions to achieve full abstraction — a desirable property indicating that, informally, the semantics is at the right level of abstraction [4]. They correspond, respectively, to reflexivity and transitivity of the \( \rightarrow^* \) relation in a conventional operational semantics. Given a set \( T \) of traces, the closure under stuttering and mumbling \( T^\dagger \) is the smallest set which contains \( T \) and satisfies:

- Stuttering: if \( \alpha \beta \in T^\dagger \) then \( \alpha(s,s)\beta \in T^\dagger \) and
- Mumbling: if \( \alpha(s,s')(s',s')\beta \in T^\dagger \) then \( \alpha(s,s')\beta \in T^\dagger \).

We now define a few operations on traces and sets of traces. The concatenation \( T_1 ; T_2 \) and the infinite iteration operation \( T^\omega \) are defined as

\[
 T_1 ; T_2 = \{ \alpha \beta \mid \alpha \in T_1 \land \beta \in T_2 \}^\dagger
 T^\omega = \{ \alpha_0 \ldots \alpha_n \mid \forall i \geq 0. \alpha_i \in T \}^\dagger.
\]

\(^1\) Sometimes also called potential or partial computations or extended sequences.
$T^*$ denotes the smallest set containing $T$ and the empty trace, closed under stuttering, mumbling and concatenation, that is, $T^* = \bigcup_{n \in \mathbb{N}} T^n$ where $T^0 = \{\epsilon\}$ and $T^{n+1} = T; T^n$. Fair parallel composition is modeled by fair interleaving of sets of traces

$$T_1 \parallel T_2 = \bigcup \{ \alpha \parallel \beta \mid \alpha \in T_1 \land \beta \in T_2 \}$$

where $\alpha \parallel \beta$ is the set of all traces built by fairly interleaving $\alpha$ and $\beta$. One way to define $\alpha \parallel \beta$ formally can be found in [4].

\begin{align*}
\alpha \parallel \beta &= \{ \gamma \mid (\alpha, \beta, \gamma) \in \text{fairmerge} \} \\
\text{fairmerge} &= (L^* R^* L)^* \cup (L \cup R)^* A \\
L &= \{(s, s'), \epsilon, (s, s') \mid (s, s') \in (\Sigma \times \Sigma)\} \\
R &= \{((\epsilon, (s, s'), (s, s')) \mid (s, s') \in (\Sigma \times \Sigma)\} \\
A &= \{(\alpha, \epsilon, \alpha) \mid \alpha \in (\Sigma \times \Sigma)^\infty\} \\
&\cup \{(\epsilon, \beta, \beta) \mid \beta \in (\Sigma \times \Sigma)^\infty\}
\end{align*}

where concatenation and iteration are extended to sets and triples of traces in the obvious way: $AB = \{\alpha \beta \mid \alpha \in A \land \beta \in B\}$ and $(\alpha_1, \alpha_2, \alpha_3) (\beta_1, \beta_2, \beta_3) = (\alpha_1 \beta_1, \alpha_2 \beta_2, \alpha_3 \beta_3)$.

**Local Variables.** We use the notation $[s]x = v$ to denote the state that is like $s$ except that the value of $x$ is updated to $v$. Let $\alpha \equiv (s_0, s'_0)(s_1, s'_1)\ldots (s_i, s'_i)\ldots$ be a transition trace. The trace $(x = v)\alpha$ is like $\alpha$ except that $x$ is initialized to $v$ in the first state and that the value of $x$ is retained across points of possible interference. More precisely, $(x = v)\alpha$ is

$$(\{[s_0]x = v, s'_0]\} [s_1]x = s'_0(x)]\} [s_1]x = s'_0(x)]\} [s_1]x = s'_0(x)]\} \ldots$$

The trace $\alpha \setminus x$ on the other hand describes an interactive computation like $\alpha$ except that it never changes the value of $x$. That is, $\alpha \setminus x$ is

$$(s_0, [s'_0 | x = s_0(x)])(s_1, [s'_1 | x = s_1(x)]\ldots (s_i, [s'_i | x = s_i(x)]\ldots$$

**Definition 2. (Traces and executions)**

1. The semantic function $\mathcal{T}$ maps programs to closed sets of finite and infinite transition traces and is defined as

\begin{align*}
\mathcal{T}[V:_[P,Q]] &= \{(s, s') \mid s, s' \in \Sigma, (s, s') \models c_{V:_[P,Q]}\}^* \\
\mathcal{T}[C_1 ; C_2] &= \mathcal{T}[C_1]; \mathcal{T}[C_2] \\
\mathcal{T}[C_1 \lor C_2] &= \mathcal{T}[C_1] \cup \mathcal{T}[C_2] \\
\mathcal{T}[C_1 \land C_2] &= \mathcal{T}[C_1] \cap \mathcal{T}[C_2] \\
\mathcal{T}[\boxdot C] &= (\mathcal{T}[C])^* \\
\mathcal{T}[\text{new } x = v \text{ in } C] &= \{[\alpha \setminus x] \mid (x = v)\alpha \in \mathcal{T}[C]\}^*.
\end{align*}
2. A trace \((s_0, s'_0)(s_1, s'_1)\ldots\) is interference-free if \(s'_i = s_{i+1}\) for all \(i \geq 0\). The executions \(\mathcal{E}[C]\) of a program \(C\) are given by

\[
\mathcal{E}[C] = \{ \alpha \in \mathcal{T}[C] \mid \alpha \text{ is interference-free} \}.
\]

3. Let \(C_1 \subseteq_T T C_2\) and \(C_1 =_T C_2\) abbreviate \(T[C_1] \subseteq T[C_2]\) and \(T[C_1] = T[C_2]\) respectively. Similarly for \(\mathcal{E}\). □

The clause for \texttt{new} requires some explanation. The traces of \texttt{new} \(x = v\) in \(C\) do not change the value of \(x\) and are obtained by executing \(C\) under the assumption that \(x\) is set to \(v\) initially and that the environment cannot change the value of \(x\). Consequently, the value of a local variable cannot be changed outside its scope, nor can changes carried out inside the scope be observed on the outside. As an example, consider the following equivalence which follows directly from the definition of \texttt{new} and the closure conditions.

\[
\texttt{new } x = 1 \text{ in } x := x + 1; y := x + 1 \text{ end } =_T y := 2
\]

As we will see in Section 6, scope and locality make reasoning about parallel programs substantially more tractable. Knowing that a variable \(x\) is local, means that the environment cannot invalidate program properties involving \(x\) and that the environment cannot be influenced by changes to \(x\). Consider, for instance, the parallel composition

\[
\texttt{new } x = 1 \text{ in } C_1 \text{ end } || C_2.
\]

Since \(x\) is local, program \(C_1\) cannot change the value of any occurrence of \(x\) in \(C_2\), while \(C_2\) cannot change the value of any occurrence of \(x\) in \(C_1\).

The following proposition collects a few central properties of the semantics.

**Proposition 1. (Properties of \(T\))**

1. If \(C_1 \subseteq_T T C_2\), then \(C_1 \subseteq_T T C_2\).
2. Parallel composition is associative and commutative, that is, \([C_1 || C_2] || C_3 =_T [C_1] || [C_2 || C_3]\) and \([C_1] || C_2 =_T C_2 || [C_1]\).
3. Informally, let a context \(E\) be given by a program with a “hole”. Trace inclusion is a congruence, that is, \(C_1 \subseteq_T T C_2\) implies \(E[C_1] \subseteq_T E[C_2]\) for all \(E\). In other words, trace inclusion is preserved by all contexts.
4. Due to the closure conditions, the semantics is invariant under the addition of finite stuttering, e.g., \(C_1 ; \texttt{skip}^\ast ; C_2 =_T C_1 ; C_2\) and \(C || \texttt{skip}^\ast =_T C\). □

The proofs of full abstraction, associativity, commutativity and congruence can be found in [4]. Since parallel composition is commutative, we can abbreviate the \(n\)-ary parallel composition \(C_1 || \ldots || C_n\) by \(\{^1,\ldots,n\} C_i\).

**Encoding Standard Programming Language Constructs.** The standard constructs of a shared-variable parallel programming language are embedded into our setting through the following abbreviations. The \texttt{await} statement is
implemented using busy waiting. Let \( e \) be an expression whose value is defined if and only if the predicate \( \text{def}(e) \) holds. Moreover, let \( B \) be a boolean expression.

\[
\begin{align*}
\{B\} & \equiv \emptyset : [\text{def}(B) \land B, \text{def}(B) \land B] \\
\text{skip} & \equiv \{tt\} \\
x := e & \equiv \{x : [\text{def}(e), x = e]\} \\
\text{if} \ B \ \text{then} \ C_1 \ \text{else} \ C_2 & \equiv (\{B\} ; C_1) \lor (\{\neg B\} ; C_2) \\
\text{while} \ B \ \text{do} \ C & \equiv ((\{B\} ; C)^* ; \{\neg B\}) \lor ((\{B\} ; C)^{*}) \\
\text{await} \ B \ \text{then} \ x_1 := e_1 ; \ldots ; x_n := e_n & \equiv \{x_1 , \ldots , x_n : [B, Q] \lor \{\neg B\}^* \}
\end{align*}
\]

where \( Q \equiv x_1 = e_1 \land \ldots \land x_n = e_n \). Let \( C \) be a program in which \( i \) is a constant integer variable, that is, an integer variable that is only read and never assigned to. Also, let \( n \) be a natural number. Then, a for loop can be defined as

\[
\text{for } i = 1 \ \text{to} \ n \ \text{do} \ C \equiv C[1/i] ; \ldots ; C[n/i]
\]

where \( C[j/i] \) denotes the program that is obtained from \( C \) by replacing all free occurrences of \( i \) by \( j \). An array \( A \) with indices from 1 to \( n \) stands for a set of variables \( A[1] \) through \( A[n] \). Assignments to an array variable with a constant index \( i \) are defined by

\[
A[i] := e \equiv \{A[i] : [1 \leq i \leq n \land \text{def}(e), A[i] = e]\}.
\]

Note that the above assignment has no traces if the array index \( i \) is outside the array bounds or the expression \( e \) is undefined. A treatment of non-constant array indices can be found in [9]. Also note that our definition of the statement \( V : [P, Q] \) differs slightly from Morgan’s in [12]. There, the behaviour of \( V : [P, Q] \) in an initial state that does not satisfy \( P \) is completely arbitrary. Even non-termination is possible. In our setting, however, in initial states that do not satisfy \( P \) the statement \( V : [P, Q] \) exhibits no behaviour at all. Our encoding of the conditional statement, for instance, necessitates this deviation.

### 3 Refinement

A very natural notion of program approximation arises through transition trace inclusion. However, congruence implies that trace inclusion between two programs \( C_1 \) and \( C_2 \) implies that in all possible contexts the executions of \( C_1 \) are contained in those of \( C_2 \) in the same context. Thus, whenever we want to do refinement in a specific context, trace set inclusion typically is too strong, because it does not allow us to incorporate information about that particular context. Following Jones and Stirling [11,14] we will use assumption-guarantee reasoning\(^2\) to remedy this situation.

\(^2\) Sometimes also called rely-guarantee or assumption-commitment reasoning.
Definition 3. (Assumption-guarantee formulas)

1. Let \( \alpha \equiv (s_0, s'_0)(s_1, s'_1) \ldots \) be a transition trace, \( P \) be a unary predicate and \( \Gamma \) a set of unary predicates. We say that \( \alpha \) satisfies the assumptions \( P \) and \( \Gamma \), \( \alpha \models \text{assump}(P, \Gamma) \), for short, iff the first state satisfies \( P \) and all predicates in \( \Gamma \) are preserved across all gaps along \( \alpha \), that is,
   - \( s_0 \models P \), and
   - \( (s'_i, s_{i+1}) \models P' \Rightarrow P' \) for all \( P' \in \Gamma \) and for all \( 0 \leq i \leq \text{length}(\alpha) \)
   where \( \text{length}(\alpha) \) stands for the number of pairs in \( \alpha \) minus 1.

2. Let \( Q \) be a unary predicate and \( \Delta \) be a set of unary predicates. We say that \( \alpha \) satisfies the guarantees \( Q \) and \( \Delta \), \( \alpha \models \text{guar}(Q, \Delta) \), for short, iff the last state in \( \alpha \) (if it exists) satisfies \( Q \) and all predicates in \( \Delta \) are preserved across all transitions along \( \alpha \), that is,
   - \( \text{last}(\alpha) \models Q \), if \( \alpha \) is finite and
   - \( (s_i, s'_i) \models P' \Rightarrow P' \) for all \( P' \in \Delta \) and for all \( 0 \leq i \leq \text{length}(\alpha) \)
   where \( \text{last}(\alpha) \) denotes the last state of \( \alpha \), if \( \alpha \) is finite.

3. \( C \) guarantees \( Q \) and \( \Delta \) under assumptions \( P \) and \( \Gamma \), \([P, \Gamma] C [Q, \Delta]\) for short, iff for all \( \alpha \in T[C] \), \( \alpha \models \text{assump}(P, \Gamma) \) implies \( \alpha \models \text{guar}(Q, \Delta) \).

The following definition will allow us to express that the traces of a program are contained in those of another, if the environment satisfies certain assumptions and the changes to certain variables are ignored.

Definition 4. (Relativized trace inclusion)

Given programs \( C, C' \), a unary predicate \( P \), a set of unary predicates \( \Gamma \), and a program variable \( x \), the notation

\[
C \supseteq_T C' \ (P, \Gamma, x)
\]

expresses that for all traces \( \alpha \in T[C'] \) such that

- \( \alpha \models \text{assump}(P, \Gamma) \), and
- \( (x = v) \alpha \in T[C'] \) for some \( v \in \text{Dom}_x \),

there exists \( \beta \in T[C] \) such that

- \( \beta \models \text{assump}(P, \Gamma) \),
- \( (x = v) \beta \in T[C] \), and
- \( \alpha \setminus x = \beta \setminus x \).

Given a set of variables \( V \), let \( C \supseteq_T C' \ (P, \Gamma, V) \) be the obvious generalization.

Our definition of refinement combines assumption-guarantee reasoning and relativized trace inclusion.

Definition 5. (Refinement)

Given unary predicates \( P \) and \( Q \), sets of unary predicates \( \Gamma \) and \( \Delta \), and a set of program variables \( V \subseteq \text{Var} \), we say that \( C' \) refines \( C \) modulo \( V \) under the assumptions \( P \) and \( \Gamma \) and the guarantees \( Q \) and \( \Delta \), \([P, \Gamma] C \succ_V C' [Q, \Delta]\) for short, iff

\[
\text{...}
\]
1. no variable in $V$ occurs free in $C$, that is, $f_0(C) \cap V = \emptyset$, and
2. $C$ guarantees $\Delta$ under assumptions $P$ and $\Gamma$, that is,
   \[ [P, \Gamma] \quad C \quad [tt, \Delta], \quad \text{and} \]
3. $C'$ guarantees $Q$ and $\Delta$ under assumptions $P$ and $\Gamma$, that is,
   \[ [P, \Gamma] \quad C' \quad [Q, \Delta], \quad \text{and} \]
4. we have $C \supseteq_{\tau} C' \quad (P, \Gamma, V)$. \hfill \Box

Informally, the refinement $[P, \Gamma] \quad C \succ V \quad C' \quad [Q, \Delta]$ expresses that assuming an initial state that satisfies $P$, and a parallel context that preserves the predicates in $\Gamma$, then every transition of $C'$ can be matched by $C$ modulo the changes to variables in $V$, every transition of $C$ and $C'$ will preserve the predicates in $\Delta$, and whenever $C'$ and the parallel context terminate, they will do so in a state satisfying $Q$. Consider, for example,

\[ [x = 1, \{x = 1, x = 2\}] \quad x := x + 1 \succ \emptyset \quad x := 2 \quad [x = 2, \text{Preds} (\text{Var} \setminus \{x\})] \]

where $\text{Preds}(V)$ denotes the set of all unary predicates $P$ with free variables in $V$, that is, $f_0(P) \subseteq V$. If the initial state satisfies $x = 1$ and the parallel environment preserves the predicates $x = 1$ and $x = 2$, then every transition by $x := 2$ can be matched exactly by $x := x + 1$, the final state satisfies $x = 2$ and every transition by $x := 2$ and $x := x + 1$ preserves all predicates in $\text{Preds} (\text{Var} \setminus \{x\})$. For another example, consider

\[ [x = 1 \land y = 1 \land t = 0, \{x = 1, y = 1, t = 2, x = 3\}] \quad \text{skip} \quad x := 2 \cdot x + y \succ \{t\} \quad t := 2 \cdot x \quad x := t + y \quad [x = 3 \land y = 1, \text{Preds} (\text{Var} \setminus \{x, t\})]. \]

Here, the matching between transitions is not exact, but only modulo the value of variable $t$. Note that Proposition 1.4 implies $\text{skip} ; x := 2 \cdot x + y \equiv \tau \quad x := 2 \cdot x + y$.

The following proposition expresses that refinement is closed under the addition of equivalent and closed predicates to the assumptions and guarantees. Moreover, it captures the reflexivity and transitivity of refinement. Note that $\text{Preds}(\emptyset)$ is the set of all closed (constant) predicates.

**Proposition 2. (Properties of refinement)**

1. (Closure) We have $[P, \Gamma] \quad C \succ_{V} \quad C' \quad [Q, \Delta]$ iff
   \[ [P, \tau \cup \text{Preds}(\emptyset)] \quad C \succ_{V} \quad C' \quad [Q, \overline{\Delta} \cup \text{Preds}(\emptyset)] \]
   where $\tau \equiv \{ P' \mid P \in \Gamma, P \Leftrightarrow P' \}$ and similarly for $\overline{\Delta}$.
2. (Reflexivity) We have $[P, \Gamma] \quad C \succ_{\emptyset} \quad C \quad [Q, \Delta]$ iff $[P, \Gamma] \quad C \quad [Q, \Delta]$. 

3. (Transitivity) If $fs(C_1) \subseteq fs(C_2)$ and

$$[P, \Gamma_1] \ C_1 \triangleright V_1 \ C_2 [Q_1, \Delta_1]$$

and

$$[P, \Gamma_2] \ C_2 \triangleright V_2 \ C_3 [Q_2, \Delta_2],$$

then

$$[P, \Gamma_1 \cup \Gamma_2] \ C_1 \triangleright V_1 \cup V_2 \ C_3 [Q_2, \Delta_1 \cap \Delta_2].$$

Due to Proposition 2.1, equivalent and closed predicates will not be explicitly mentioned in the set of assumptions and guarantees of a refinement.

The next proposition expresses both trace inclusion and execution inclusion as special cases of refinement. The minimal amount of environment assumptions $Preds(\emptyset)$ gives rise to trace inclusion while the maximal amount of assumptions $Preds(Var)$ yields execution inclusion. All four lemmas follow directly from the definitions.

**Proposition 3. (Trace and execution inclusion via refinement)**

1. If $[tt, Preds(\emptyset)] \ C \triangleright \emptyset C' [Q, \Delta]$ for some $Q$ and $\Delta$, then $C \triangleright_T C'$.
2. If $C \triangleright_T C'$, then $[tt, Preds(\emptyset)] \ C \triangleright \emptyset \ [tt, Preds(\emptyset)]$.
3. If $[P, Preds(Var)] \ C \triangleright \emptyset C' [Q, \Delta]$ for some $\Delta$, then $\{P\} \ C \triangleright_E \{P\} \ C'$ and $\{P\} \ C' \{Q\}$ where $\{P\} \ C' \{Q\}$ is the standard Hoare-triple notation for partial correctness.
4. If $\{P\} \ C \triangleright_E \{P\} \ C'$ then $[P, Preds(Var)] \ C \triangleright \emptyset \ [tt, Preds(\emptyset)].$$

Refinement is governed by the syntax-directed rules in Figures 1 and 2 on pages 223 and 224. Soundness of the rules is proved in [9].

**General Refinement Methodology.** Let $C_1$ be a high-level specification of the implementation that is to be derived. $C_1$ can be viewed as an abstract statement of the computation to be performed. More precisely, $C_1$ defines the executions that all refinements, and thus also the final implementation, are allowed to exhibit. The refinement of $C_1$ then proceeds by finding a sequence of programs $C_2, \ldots, C_n$ and $P_i, \Gamma_i, Q_i,$ and $\Delta_i$ such that

$$[P_i, \Gamma_i] \ C_i \triangleright \emptyset C_{i+1} [Q_i, \Delta_i]$$

for all $1 \leq i < n$. Intuitively, for each refinement, $P_i$ and $\Gamma_i$ constitute the minimal assumptions necessary for the refinement to hold while $Q_i$ and $\Delta_i$ stand for the maximal guarantees the new component $C_{i+1}$ can give to its environment. Assumptions and guarantees arise during the use of the refinement calculus presented below. With transitivity (Proposition 2.3) and the weakening rule WEAK the above sequence then implies

$$[\wedge_i P_i \cup \Gamma_i] \ C_1 \triangleright \emptyset C_n [Q_n \cap \Delta_i]$$
ASSCOM If $A$ is an atomic statement and $\{P, Q\} \subseteq \Gamma$ and $(\bar{P} \land cf_A) \Rightarrow Q$ and $\Delta \subseteq \{Q \mid (\bar{P} \land Q \land cf_A \Rightarrow Q)\}$, then $[P, \Gamma] \ A [Q, \Delta]$.

ATOM If $A_1$ and $A_2$ are atomic statements and $f_4(A_1) \cap V = \emptyset$ and
1. $[P, \Gamma] \ A_1 [Q, \Delta]$, and $[P, \Gamma] \ A_2 [Q, \Delta]$, and
2. $(\exists x_1 \ldots x_n. \bar{P} \land cf_{A_2}) \Rightarrow (\exists x_1 \ldots x_n. \bar{P} \land cf_{A_1})$, then
   $$ [P, \Gamma] \ A_1 \blacktriangleright_{(x_1 \ldots x_n)} A_2 [Q, \Delta] $$

SEQ If $f_4(C_1) \cap V_2 = \emptyset$ and $f_4(C_2) \cap V_1 = \emptyset$, then
$$ [P, \Gamma] \ C_1 \triangleright_{V_1} C'_1 [Q_1, \Delta_1] \quad [Q_1, \Gamma] \ C_2 \triangleright_{V_2} C'_2 [Q_2, \Delta_2] \quad \Rightarrow \quad [P, \Gamma] \ C_1 \triangleright_{V_1 \cup V_2} C'_1 \triangleright_{V_2} C'_2 [Q_1 \cap \Delta_1, \Delta_2] $$

OR If $f_4(C_1) \cap V_2 = \emptyset$ and $f_4(C_2) \cap V_1 = \emptyset$, then
$$ [P, \Gamma] \ C_1 \triangleright_{V_1} C'_1 [Q_1, \Delta_1] \quad [Q_1, \Gamma] \ C_2 \triangleright_{V_2} C'_2 [Q_2, \Delta_2] \quad \Rightarrow \quad [P, \Gamma] \ C_1 \lor C_2 \triangleright_{V_1 \cup V_2} C'_1 \lor C'_2 [Q_1 \cap \Delta_1, \Delta_2] $$

STAR, OMEGA
$$ [I, \Gamma] \ C \triangleright_{V} C' \quad [I, \Delta] \quad \Rightarrow \quad [I, \Gamma] \ C \triangleright_{V} (C')^\ast \quad [I, \Delta] $$

PAR If $f_4(C_1) \cap V_2 = \emptyset$ and $f_4(C_2) \cap V_1 = \emptyset$ and $I_1 \subseteq \Delta_2$ and $I_2 \subseteq \Delta_1$, then
$$ [P_1, \Gamma_1] \ C_1 \triangleright_{V_1} C'_1 [Q_1, \Delta_1] \quad [P_2, \Gamma_2] \ C_2 \triangleright_{V_2} C'_2 [Q_2, \Delta_2] \quad \Rightarrow \quad [P_1 \land P_2, \Gamma_1 \cup \Gamma_2] \ C_1 \lor C_2 \triangleright_{V_1 \cup V_2} C'_1 \lor C'_2 [Q_1 \land Q_2, \Delta_1 \cap \Delta_2] $$

NEW If $I' = \{P[v/x] \mid P \in \Gamma, v \in Dom_x\}$, $\Delta' = \{P, P[v/x] \mid P[v/x] \in \Delta, v \in Dom_x\}$, then
$$ [P, \Gamma] \ C \triangleright_{V} C' \quad [Q, \Delta] \quad \Rightarrow \quad [P, \Gamma'] \ C \triangleright_{V} C' \quad [x \in V] $$

COND If $f_4(C_1) \cap V_2 = \emptyset$ and $f_4(C_2) \cap V_1 = \emptyset$ and $P \Rightarrow (B \Leftrightarrow B')$,
1. $[P \land B, \Gamma_1] \ C_1 \triangleright_{V_1} C'_1 [Q, \Delta_1]$, and
2. $[P \land \lnot B, \Gamma_2] \ C_2 \triangleright_{V_2} C'_2 [Q, \Delta_2]$, then
   $$ [P, \Gamma_1 \cup \Gamma_2 \cup \{P, B', \lnot B'\}] \ \begin{cases} \text{if } B \text{ then } C_1 \text{ else } C_2 \triangleright_{V_1 \cup V_2} \text{ if } B' \text{ then } C'_1 \text{ else } C'_2 \triangleright_{V_1 \cup V_2} \text{ if } B' \text{ then } C'_1 \text{ else } C'_2 \triangleright_{V_1 \cup V_2} \end{cases} $$

FOR If $i$ is a constant integer variable $C$ and $C'$ and
$$ [P[k - 1/i, \Gamma_1] \ C[k/i] \triangleright_{V_i} C'[k/i] \ [P[k/i, \Delta_i], $$
for all $1 \leq k \leq n$, then
$$ [P[1/i], \bigcup_{i=1}^n \Gamma_i] \quad \text{[for } i = 1 \text{ to } n \text{ do } C \triangleright_{V_i} C', \text{ for } i = 1 \text{ to } n \text{ do } C'] $$

$$ [P[n/i], \bigcap_{i=1}^n \Delta_i] $$

**Fig. 1.** Refinement rules
PAR-N  If for all \(1 \leq i \leq n\)
1. \([P_i, I_i] C_i \triangleright_V C'_i [Q_i, \Delta_i]\), and
2. \(I_i \subseteq \bigcap_{j=1, j \neq i}^{n} V_j\), and
3. \(\text{fs}(C_j) \cap V_i = \emptyset\) for all \(1 \leq j \leq n\) with \(j \neq i\), then
\[
\left[\bigwedge_{i=1}^{n} P_i, \bigcup_{i=1}^{n} I_i\right] \triangleright_{V_i} \bigcup_{i=1}^{n} C_i \triangleright_{V_i} \left[\bigwedge_{i=1}^{n} Q_i, \bigcap_{i=1}^{n} \Delta_i\right].
\]

NEW-INTRO  If \(I' = \{P[v'/x] \mid P \in I, v' \in \text{Dom}_x\}, \Delta' = \{P[P[v'/x] \mid P[v'/x] \in \Delta, v' \in \text{Dom}_x\}, then
\[
P, I \triangleright_{V \cup \{x\}} C \triangleright_{V} P, I' \triangleright_{V} C' \triangleright_{V} \left[Q, \Delta'\right].
\]

WEAK  If \(C'_1 =_{\tau} C_1, C_2 \subseteq_{\tau} C'_2, P \Rightarrow P', Q' \Rightarrow Q, I'' \subseteq I, \Delta \subseteq \Delta', \text{ and } V' \subseteq V, then
\[
P', I'' \triangleright_{V'} C'_2 [Q', \Delta']
\]
\[
P, I \triangleright_{V} C_2 [Q, \Delta]
\]

which yields \(\{\bigwedge_i P_i\}; C_i \geq_{\theta} \{\bigwedge_i P_i\}; C_n \) and \(\{\bigwedge_i P_i\} C_n \{Q_n\}\) by rule WEAK and Proposition 3.3. Thus, every execution \(\alpha\) of \(C_n\) that starts in a state satisfying \(\bigwedge_i P_i\) also is an execution of \(C_1\) and whenever \(\alpha\) is finite, the last state satisfies \(Q_n\). Note that the refinement methodology assumes that all \(C_i\) have a non-empty set of executions. Care must thus be taken to ensure that the initial program and all of its refinements have a non-empty set of executions. It can be shown that all programs that contain the standard programming language constructs only, have non-empty sets of executions [9]. Consequently, whenever the most refined program \(C_n\) only contains the standard constructs, the entire refinement is non-trivial.

4  Introducing Iteration and Parallelism

So far, the only rule that allows the introduction of a new construct across a refinement is NEW-INTRO. To make our calculus more versatile we need rules for the introduction of loops and parallel compositions. We start with a rule for introducing \texttt{for} loops.

FOR-INTRO. If \(i\) is a constant integer variable and
1. for all \(0 \leq k \leq n - 1\) we have
\[
[I[k/i], I_k] C \triangleright_{\emptyset} C'[k/i] [I[k+1/i], \Delta_k]
\]

and
2. $I[n/i] \Rightarrow Q$,

then

$$\left[I[0/i], \bigcup_{i=1}^{n} \Gamma_i\right] C^* ; \{Q\} \not\supset \emptyset \text{ for } i = 1 \text{ to } n \text{ do } C' \left[Q, \bigcap_{i=1}^{n} \Delta_i\right].$$

Informally, if $C$ can be refined to $C'[k/i]$ for each iteration $k$ using loop invariant $I[k/i]$ and assumptions $\Gamma_k$, and the desired postcondition $Q$ is implied by $I[n/i]$, then $C^* ; \{Q\}$ can be replaced by for $i = 1$ to $n$ do $C'$ under the assumptions $I[0/i]$ and $\bigcup_{i=1}^{n} \Gamma_i$. A similar rule can be given for the introduction of while loops [9].

The introduction of parallelism is more difficult, because, due to interference on shared variables, the parallel execution of processes may lead to unexpected results. We use the fact that the behaviour of some programs is unaffected by parallelism. For instance, the program $x := 1; x := 1$ is equivalent to the program $x := 1 \parallel x := 1$. We introduce the notion of robustness to generalize this property. Informally, the semantics of an $n$-fold parallel composition of a robust program is equivalent to its $n$-fold sequential composition.

**Definition 6. (Robust programs)**

A program $C$ is called robust if $C^* \supseteq \lbrack C \rbrack^n = \lbrack C^n \rbrack$ for all $n \geq 1$ where $C^n$ and $\lbrack C \rbrack^n$ denote the $n$-fold sequential composition and the $n$-fold parallel composition respectively, that is, $C^1 \equiv C$ and $C^{n+1} \equiv C; C^n$ and $\lbrack C \rbrack^n \equiv \lbrack C^n \rbrack$.

Besides $x := 1$, the programs $x := x + 1$ and $x := x + 1; x := x + 1$ are also robust, because we have chosen assignments to be atomic. The program $x := 1; x := x + 1$, however, is not. Atomic statements and finite loops over them are always robust.

**Proposition 4. (Sufficient conditions for robustness)**

1. Atomic statements $V : [P, Q]$ are robust.
2. If $C$ is robust, then $C^*$ and $C^m$ for all $m \geq 0$ are robust.

**Proof:** 1) is proved by induction while 2) follows from the associativity of parallel composition. □

Robustness allows us to define a rule for the introduction of parallelism.

**PAR-INTRO.** If

1. $C$ is robust and $[tt, Pred\{\emptyset\}] C \left[tt, \bigcap_{i=1}^{n} \Delta_i\right]$, and
2. $[P, \Gamma_i] C \not\supset \emptyset C_i \left[Q_i, \Delta_i\right]$ for all $1 \leq i \leq n$, and
3. $\Gamma_i \subseteq \bigcap_{j=1, j \neq i}^{n} \Delta_j$ for all $1 \leq i \leq n$, and
4. $(\forall 1 \leq i \leq n). Q_i \Rightarrow Q$,
then
\[ [P, \bigcup_i \Gamma_i] \cdot C^* \{ Q \} \succ_\emptyset \bigcap_{i=1}^n C_i \cdot \left[ Q, \bigcap_i \Delta_i \right] \]

Intuitively, robustness ensures that \( C^* \) can be refined into \( \bigcap_{i=1}^n C_i \). The second and third premise allow the refinement of \( \bigcap_{i=1}^n C_i \) into \( \bigcap_{i=1}^n C_i \) using rule PAR-N.

Note that FOR-INTRO and PAR-INTRO require the set of local variables \( V \) to be empty. While the rules could be generalized, the simpler versions suffice for our present purposes.

5 Modeling Message-Passing

In most of the literature on concurrency theory, shared-variable and message-passing concurrency are given sometimes very different semantic models, e.g., [5, 6,7,4]. Since we want to be able to move freely between the two paradigms, we need a uniform model that captures shared-variable and message-passing concurrency in the same semantic framework. Consider the two standard message-passing primitives \( c?x \) and \( c!e \) where \( c \) is a channel. The input statement \( c?x \) reads the next item off \( c \) and assigns it to \( x \). If \( c \) is empty, the statement blocks until \( c \) is non-empty. Thus, if \( c \) remains empty forever, the statement also blocks forever. The output statement \( c!e \) evaluates the expression \( e \) and appends the resulting value at the end of \( c \). It never blocks. The two primitives thus receive an asynchronous communication semantics. We fit these two constructs in our language by modeling a channel \( c \) as variable ranging over finite queues. More precisely,

\[ c?x \equiv \text{await } c \neq \emptyset \text{ then } x := \text{hd}(c); c := \text{tl}(c) \text{ end} \]

\[ c!e \equiv c := \text{enqueue}(c, e) \]

where \( c \) is a variable ranging over finite queues, \( \emptyset \) denotes the empty queue, \( \text{hd}(c) \) and \( \text{tl}(c) \) return the head and tail of \( c \) respectively and \( \text{enqueue}(c, e) \) returns a queue that is like \( c \) except that the value of \( e \) is appended at the end. This encoding gives rise to the following rules for introducing receive and send statements.

**INPUT-INTRO**

\[ [c = v : l, \{ c = v : l, c = l, x = v \}] \]

\[ x := v \succ_{\{ c \}} c?x \]

\[ [x = v \land c = l, \text{PredsVar} \setminus \{ x, c \}] \]

**OUTPUT-INTRO**

\[ [x = v \land c = l, \{ x = v, c = l, c = l : v \}] \]

\[ \text{skip} \succ_{\{ c \}} c!x \]

\[ [c = l : v, \text{PredsVar} \setminus \{ c \}] \]
Note how the assumptions of rule INPUT-INTRO imply that the input statement $c?x$ will never block forever.

To derive a distributed program from a shared-variable program, we need to

1. identify the non-local information for each parallel process,
2. introduce channels between producers and consumers of that information using NEW-INTRO, and
3. replace access to shared variables by access to the corresponding channels using INPUT-INTRO and OUTPUT-INTRO.

This methodology seems well suited for parallel algorithms with a limited form of parallelism that allows them to be implemented without explicit synchronizations. For instance, the Floyd-Warshall algorithm, the prefix-sum algorithm, and the all-pair, shortest-paths algorithm discussed below fall into this category. The environment of each parallel process in all of these algorithms can be shown to satisfy the assumptions required by rule INPUT-INTRO.

6 Example: All-Pair Shortest-Paths

Given an unweighted graph $G \equiv (V,E)$, the goal is to compute the length of a shortest path between any two vertices $v$ and $v'$ in $V$, $\text{dist}(v,v')$ for short. The length of a path is given by the number of vertices it contains minus 1. $G$ may be directed or undirected. Let $n$ be the number of vertices in $G$, that is, $n = |V|$. The shortest distances are to be stored in a two-dimensional array $D$.

The initial program $C_1$ allows an arbitrary but finite number of updates to the array $D$ before the final state satisfying $Q \equiv \forall x, x' \in V. D[x,x'] = \text{dist}(x,x')$ is established, that is,

$$C_1 \equiv \{D;[tt,tt]^* ; \{Q\}.$$  

We will begin by deriving a shared-variable implementation. Then, this solution is refined into a distributed implementation. The first part of the derivation is summarized in Figure 3 on page 228.

Refining $C_1$ into $C_4$. Given a vertex $v$, the first refinement $C_2$ considers the vertices reachable from $v$ in the order of increasing distance assuming the availability of the distance function $\text{dist}$. Formally, the refinement proceeds by showing

$$R_1 \equiv [tt, \text{Preds}(\{D\})] \{D;[tt,tt]^* \rightarrow V \parallel V' \text{ if } v = v' \text{ then } D[v,v']:=0 \text{ else } D[v,v']:=\text{nil} \forall x \in V.Q_x^0, \text{Preds}(\text{Var}\setminus\{D\})]$$
Fig. 3. Derivation of shared-variable solution to all-pair, shortest-paths problem
and

$$R_2 \equiv [\forall x \in V. Q^0_x, \text{Preds}(\{D\})]$$
$$\{D\}: [tt, tt]^* ; \{Q\}$$

$$\not\models \emptyset$$

for $k = 1$ to $n - 1$ do

$$[||_V^0 ||_V^\infty \text{if } \text{dist}(v, v'') = k \text{ then } D[v, v''] := k]$$

$$Q, \text{Preds(Var}\setminus \{D\})$$

where

$$Q_k^x \equiv \forall x' \in V. \text{dist}(x, x') \leq k \Rightarrow D[x, x'] = \text{dist}(x, x')$$

for all $0 \leq k \leq n$. The proof uses ASSCOM, ATOM, Proposition 4 and PAR-INTRO (twice) and FOR-INTRO. Refinement between $C_1$ and $C_2$ then follows from $R_1$, $R_2$ and

$$\{D\}: [tt, tt]^* \Rightarrow \{D\}: [tt, tt]^* ; \{D\}: [tt, tt]^*.$$ 

Refinement $C_3$ introduces the concept of a fringe. The fringe of a vertex $v$ with distance $k$, $\text{fringe}(k, v)$ for short, is defined to be the set of vertices that are reachable from $v$ through paths of length $k$. Formally,

$$\text{fringe}(k, v) = \{v' \mid \text{dist}(v, v') = k\}.$$ 

If $X = \text{fringe}(k, v)$ we say that $X$ is the $k$-fringe of $v$. An array of local variables $F$, where $F[k, v]$ holds the $k$-fringe of $v$, is introduced. In each iteration $k$, the $k$-fringe of $v$ is obtained by considering the $(k - 1)$-fringes of the immediate neighbours of $v$. Formally, we have the property

$$\text{fringe}(k, v) = \bigcup_{(v, v')}^F \{v'' \in \text{fringe}(k - 1, v') \mid D[v, v''] = \text{nil}\}.$$ (1)

Refinement between $C_2$ and $C_3$ is proved using the rules in Figures 1 and 2, and the rules FOR-INTRO and PAR-INTRO together with Proposition 4.
Refinement $C_4$ is obtained from $C_3$ by breaking down the computation of the $k$-fringe of $v$. More precisely, we use the refinement

$$\forall x \in V. F[k-1, v] = \text{fringe}(k-1, v), \Gamma$$

$$F[k, v] := \bigcup_{(v, v') \in E} \left\{ v'' \in \text{fringe}(k-1, v') \mid D[v, v''] = \text{nil} \right\}$$

new $t_1 = \emptyset$

new $f = \emptyset$ in

$E$

new $t_2 = \emptyset$ in

$\forall (v, v')$

$$\begin{cases} f := F[k-1, v']; \\ t_1 := t_1 \cup t_2; \end{cases}$$

$$\begin{cases} t_2 := t_2 \cup \{v''\}; \end{cases}$$

$$F[k, v] := t_1$$

end

$$[F[k, v] = \text{fringe}(k, v), \Delta]$$

where

$$\Gamma \equiv \text{Preds}(\{D[v, x], F[k-1, x], F[k, x] \mid x \in V\})$$

$$\Delta \equiv \{F[k, v] = \text{fringe}(k, v)\} \cup \text{Preds}(\{D[x, x'], F[k-1, x] \mid x, x' \in V\} \cup \{F[k, x] \mid x \in V \land x \neq v\}).$$

This refinement is derived using rules NEW-INTRO, SEQ, ATOM, and PAR-INTRO. Note that the correctness of this refinement crucially depends on the atomicity of the assignments to $t_1$ and $t_2$. Moreover, since $t_1$ and $t_2$ are local, no assumptions preventing harmful environment interference are necessary and reasoning is simplified considerably. Refinement between $C_3$ and $C_4$ follows using FOR, SEQ, ATOM, PAR-N, and NEW.

**Deriving a Distributed Implementation.** Suppose that every vertex $v$ only knows

- its immediate neighbours, that is, $E$ is not globally known or available,
- its own current fringe, that is, the array $F[k]$ is not globally available

and that all other information is considered non-local. We want to derive a program that is distributed in the sense that all non-local information that a vertex $v$ needs is communicated to $v$ explicitly through message passing. Let $C_v$ range over each of the processes of the top-most parallel composition in the for loop of $C_4$. $C_v$ accesses the fringes $F[k-1, v']$ of all vertices $v'$ that $v$ is adjacent to. This information is non-local to $C_v$ and thus needs to be communicated explicitly. To this end, we introduce a two dimensional array of local channels.
Each channel $c[v',v]$ will be used by vertex $v'$ to send its current fringe $F[k-1,v']$ to $v$. More precisely, the channels are subject to the following loop invariant: In iteration $k$, if $(v,v') \in E$ then $c[v',v]$ contains $\text{fringe}(k-1,v')$ (and nothing else). The resulting program $C_5$ is shown in Figure 4. The Rules INPUT-INTRO and OUTPUT-INTRO are essential for proving refinement.

Note that the number of send and receive actions in $C_5$ depends on the connectivity of the graph $G$. Only if $G$ is strongly connected do we get $n^3$ send actions. If $G$ has no edges, no messages are being sent. Moreover, while the computation of $\text{fringe}(k,v)$ in $C_4$ requires direct access to the array $F[k-1]$, the corresponding computation in $C_5$ does not. Consequently, the space requirements of $C_5$ could be reduced by removing the array $F$, initializing $c[v',v]$ directly with $\{v'\}$, wrapping the declaration of a new local variable $F_{k,v}$ around $C_v$, and replacing $F[k,v]$ by $F_{k,v}$ in $C_v$. Due to space limitations this step is not given.

Alternative Refinements. A different representation of Equation (1) gives rise to an alternative way of computing the $k$-fringe of $v$. The $k$-fringe of $v$ is now obtained by considering the immediate neighbours of all vertices in the $(k-1)$-fringe of $v$. Formally,

$$\text{fringe}(k,v) = \bigcup_{v''} F_{k-1,v} \{v'' \mid (v',v'') \in E \land D[v,v''] = \text{nil}\}.$$  

(2)
Suppose this property were used for an alternative refinement of \( C_2 \) into \( C'_3 \) and \( C'_4 \) where \( C'_3 \) computes the fringe using (2), and \( C'_4 \) breaks down this computation in the obvious fashion. We now argue that the distributed implementation that \( C'_4 \) would give rise to is less desirable than \( C_5 \) given above. Program \( C'_4 \) requires every vertex \( v \) to know the immediate neighbours \( v'' \) of every vertex \( v' \) in the fringe of \( v \). Unless \( v = v' \), this is non-local information and thus needs to be communicated via channels and message-passing. Consequently, in each iteration, every vertex \( v' \) in the graph would have to be prepared to send a list \( l \) of its immediate neighbours to \( v \). Compared to \( C_5 \) a distributed implementation based on \( C'_4 \) would thus have the following main disadvantages.

- Since vertex \( v' \) does not know the vertex \( v \) such that \( v' \) appears in the fringe of \( v \), \( v' \) has to be conservative and always send \( l \) to every vertex in the graph. Independent of the connectivity of the graph, the total number of messages sent (including the initialization) would thus always be \( n^3 \).
- Since not every vertex \( v' \) appears in the fringe of another vertex \( v \) in every iteration, some of these messages are redundant and will never be received. In case of the graph with no edges, none of the \( n^3 \) messages will ever be received. All of them are redundant.

We conclude that \( C_5 \) is superior to the distributed implementation that \( C'_4 \) would lead to.

### 7 Liveness

Since the channels are always non-empty when any parallel computation is initiated, the above example does not require an explicit proof of deadlock freedom. In this section, we sketch briefly the explicit modeling and verification of liveness properties. Consider, for instance, the standard producer/consumer example \( \text{Prod} \parallel \text{Cons} \) using the shared-variable notation where

\[
\text{Prod} \equiv \left[ \begin{array}{c}
\text{produce}(x); \\
\text{insert}(x, S)
\end{array} \right]^{\omega}
\quad \text{and} \quad
\text{Cons} \equiv \left[ \begin{array}{c}
\{S, y\}: [S \neq \emptyset, \text{remove}(S, y)]; \\
\text{consume}(y)
\end{array} \right]^{\omega}
\]

The producer \( \text{Prod} \) communicates its output to the consumer via a shared variable \( S \) ranging over sets. We want to replace the abstract statement \( \{S, y\}: [S \neq \emptyset, \text{remove}(S, y)] \) by an \( \text{await} \) statement \( \text{await } S \neq \emptyset \text{ then remove}(S, y) \). However, this refinement is sound in general only if it does not introduce any infinite blocking at the \( S \neq \emptyset \) condition. It is sufficient to show that \( \text{produce}(x) \) always terminates in the given context and never decreases the size of \( S \). In terms of our framework this can expressed as the refinement

\[
[P, I] \\
\text{Var:} [tt, |S|] \geq |\tilde{S}|^* \quad \text{at } 0 \text{ produce}(x) \\
[tt, \text{Preds}(\emptyset)].
\]
for some $P$ and $Γ$. To complete the proof, the environment of the producer must be shown to meet the assumptions $Γ$. In this case, the consumer never blocks forever and the above system $\text{Prod} \parallel \text{Cons}$ is equivalent to $\text{Prod} \parallel \text{Cons}'$ where

$$\text{Cons}' \equiv \left[ \begin{array}{c}
\text{await } S \neq \emptyset \text{ then remove}(S, y); \\
\text{consume}(y)
\end{array} \right]$$

This shared-variable implementation can then be further refined into a distributed solution in which the producer and consumer communicate via a common channel $c$ rather than a shared variable. In [9], these ideas are generalized into a refinement rule for the introduction of $\text{await}$ statements.

8 Conclusion, Future Work, and Related Work

A unified semantic model for fair shared-variable and message-passing concurrency is given. The refinement calculus first presented in [10] is extended with rules for the introduction of iteration, parallelism and message-passing. We show how a distributed program can be derived from a shared-variable program. Alternative implementations can be explored and compared. The calculus scales to finer-grained concurrency with non-atomic assignments or expressions [9]. Apart from the all-pair, shortest-paths algorithm the methodology has also been applied to the prefix-sum algorithm [9]. Safety and liveness properties can be expressed conveniently in our framework.

While our calculus allows the treatment of certain linear process topologies, it fails for circular process topologies required by, for instance, Dijkstra’s token ring algorithm. Allowing environment assumptions to contain liveness properties would help, but also complicate the semantics. Instead, we are currently investigating the use of more non-compositional methods to deal with liveness [8]. Another focus of future work is data reification. We conjecture that our calculus can be extended appropriately.

This paper extends the calculus in [10] with channels and rules for the introduction of iteration and message-passing. The ideas are presented in more detail in [9]. The proof systems in [11,14] both use assumption-guarantee reasoning to achieve compositionality. Whereas Jones employs logical formulas to specify the behaviour of the program and its environment, Stirling uses sets of predicates like we do. The work in [15] augments Jones’ work with an explicit notion of refinement. Back’s refinement calculus for Action Systems [3] also models refinement explicitly. However, his calculus is not syntax-directed. All of the mentioned approaches differ from ours at least in that they lack uniform support for shared variables and message passing.

References

Construction of Finite Labelled Transition Systems from B Abstract Systems *

Didier Bert and Francis Cave

Laboratoire Logiciels, Systèmes, Réseaux - LSR-IMAG - Grenoble, France
{Didier.Bert, Francis.Cave}@imag.fr

Abstract. In this paper, we investigate how to represent the behaviour of B abstract systems by finite labelled transition systems (LTS). We choose to decompose the state of an abstract system in several disjunctive predicates. These predicates provide the basis for defining a set of states which are the nodes of the LTS, while the events are the transitions. We have carried out a connection between the B environment (Atelier B) and the Cæsar/Aldebaran Development Package (CADP) which is able to deal with LTS. We illustrate the method by developing the SCSI-2 (Small Computer Systems Interface) input-output system. Finally, we discuss about the outcomes of this method and about its applicability.

1 Introduction

Abstract systems were introduced in 1996 by J.-R. Abrial [2] in the framework of the B method. This proposal was very much influenced by the work on Action Systems [4]. Abstract systems have a state like abstract machines, but the operations are replaced by events. Events are not invoked as the operations, but they can be enabled if their guard is true. Then, one of the enabled events may be fired and the state of the system is changed according to the event action. Abstract systems can be refined and dynamic constraints can be specified to express various properties not expressible as invariant [3].

In this paper, we are interested in representing the behaviour of abstract systems and to extract information from this representation. The aim of the study is to investigate how the behaviours can be interpreted by finite labelled transition systems (LTS). Moreover, we develop the first specification steps of a simplified version of the SCSI-2 [17] (Small Computer Systems Interface) input-output system. This system was designed to manage the command and data flow between a controller and peripherals through a shared bus. This case study illustrates several points of the approach. Our method, already proposed by other people [12], consists in the decomposition of the state of abstract systems in several cases, by the way of disjunctive predicates. These predicates provide the basis for defining the set of states that are the nodes of the LTS, while the events

---

* This work was partly supported by INRIA Rhône-Alpes, through the action VERDON (VERification et test de systèmes réactifs critiques comportant des DONnées), URL: http://www.inrialpes.fr/vasy/verdon/.

are the transitions. We show that this representation can be used to prove some properties by model-checking or by calculation on the graph. However the bottleneck of the approach is the computation of the transitions which “abstract” the behaviour of the system. This computation is achieved by proving formulas. Because this process is not decidable, the computation requires human interaction or, easier, the undecidability is encompassed in the abstraction. That means, for instance, that if a transition is not automatically proved (nor disproved), then it is assumed to hold; so the construction of the LTS can be fully automatic. Then the resulting LTS is processed by an environment called CADP\(^1\). This environment (Cesar/Akdebaran Development Package) \([10]\) is a toolbox for protocol engineering. It offers a wide functionality, from interactive simulation to the most recent formal verification techniques based on model-checkers for various temporal logics and \(\mu\)-calculus.

Section 2 gives useful definitions and introduces the SCSI-2 example. Section 3 presents various ways to build finite LTS from abstract systems, then it defines our method of decomposition. Section 4 develops the example and provides insights about the applications of the method. Finally, Section 5 details functionalities of CADP and the algorithm to build effectively states and transitions as input for the toolbox. In the conclusion, we indicate what could be some other research directions inside this technology.

2 Abstract Systems and Labelled Transition Systems

2.1 Abstract Systems in \(\mathbb{B}\)

The internal state of abstract systems is composed of a list of variables \(X = (x_1, x_2, \ldots, x_k)\). These variables take their values in a set \(D = D_1 \times \ldots \times D_k\) and must satisfy an invariant \(I\). The semantic interpretation of a state over \(X\) is defined by a mapping \(\sigma\) that associates with each variable \(x_i \in X\) a value \(v_i \in D_i\)

We denote by \(\sigma(x_i)\) the current value of \(x_i\) in the state \(\sigma\). We notice that \(\sigma\) can be considered as a generalized substitution (B-Book \([1]\), page 265) and it will be used as such in this paper. The state space of an abstract system is then the set of values of the variables which satisfy the invariant. When a machine or a system is not deterministic, it can be useful to consider that a variable is associated with a set of values. In such a more general framework, the interpretation of the variables is done in a set \(D\) of the form: \(D = \mathcal{P}(D_1) \times \ldots \times \mathcal{P}(D_k)\). There is an obvious correspondence between the interpretation of variables in values sets, and sets of states in the first interpretation. Alternatively and equivalently, states may be characterized by predicates. All of these interpretations will be used in this paper and made explicit if necessary, according to the contexts.

The generalized substitution \(U\) of the initialization determines a subset of values which constitute the initial states of the abstract system. The dynamic part of an abstract system consists in a list of events, that is to say, declarations

\(^1\) URL: http://www.inrialpes.fr/vasy/cadp/.
of the form: “$E = E$”, where $E$ is the name of the event and $E$ is the definition (the body). In this paper, the event bodies have the two following forms:

```plaintext
SELECT P THEN A END
ANY z WHERE P THEN A END
```

In the events above, we say that the guard (denoted by $\text{grd}$) is the feasibility condition [1] of the generalized substitution of the event body and the action is the $A$ part. So, we have:

- $\text{grd}(\text{SELECT } P \text{ THEN } A \text{ END}) = P \land \text{grd}(A)$
- $\text{grd}(\text{ANY } z \text{ WHERE } P \text{ THEN } A \text{ END}) = \exists z : (P \land \text{grd}(A))$

In many cases, the action $A$ is an always feasible substitution (simple substitution, etc.). Then the guard of the events reduces to $P$ for the first form or $\exists z \cdot P$ for the second one. In this paper, we assume that the termination condition of the events is true. If the list of events of a system is $E_1, E_2, \ldots, E_m$, then a system is deadlock free if and only if (iff) the disjunction of the guards is true when the invariant holds [3]: $I \Rightarrow \bigvee_{i=1}^{m} \text{grd}(E_i)$. An event $E_i$ is enabled in a state $\sigma$ iff $[\sigma] \text{grd}(E_i)$ holds. For deadlock free abstract systems, in any valid state, there are always enabled events. Finally, we use the conjugate weakest precondition:

- $\langle S \rangle P = \neg [S] \neg P$

This construction [6] computes the weakest precondition which asserts that it is possible for $S$ to establish $P$.

### 2.2 Example of an Abstract System

We want to specify some characteristics of an input-output system based on the bus SCSI-2. The aim is to control the requests of the accesses to the bus. Here, we consider that the peripherals are only disks. The controller and the disks share the bus in which messages are transmitted from the controller to the disks or from the disks to the controller. At a high level of the specification, the SCSI-2 system is seen as centralized. The bus and the arbitration algorithm to take control of the bus, will be introduced further by refining the system. Each disk gets a buffer (a queue) where commands are pushed on until they are processed. This first specification says only that the controller can send a command to a certain disk $jj$ iff the buffer of this disk is not full. On the other side, a disk can consume a request (and can send the result to the controller), iff it actually gets something to do, that is to say, if its buffer is not empty. For the sake of simplicity, the size of the buffers is only represented, not their content. This is expressed by two events:

- $\text{ctr.cmd}$ the controller sends a command to a disk $jj$
- $\text{disk.rec}$ the disk $jj$ consumes a request from its buffer

The useful declarations are:
The abstract system is defined as shown in Fig. 1.

2.3 Labelled Transition Systems

A labelled transition system \( T \) is defined by \( (N, M, L, R) \) where:
- \( N \) is a set of nodes or states,
- \( M \) is the set of initial states and \( M \subseteq N \),
- \( L \) is a set of labels,
- \( R \in \mathcal{P}(N \times L \times N) \) is the transition relation.

A transition \( (q_j, l, q_k) \in R \) is also denoted by \( q_j \xrightarrow{l} q_k \). Any labelled transition relation \( R \) can be reduced to a binary relation on states (by forgetting the labels) \( R \in \mathcal{P}(N \times N) \), by:

\[
R = \{ (q, q') | \exists l \cdot (l \in L \wedge q \xrightarrow{l} q' \in R) \}
\]

The transitive closure is denoted by \( R^+ \), the reflexive and transitive closure by \( R^* \) and the inverse by \( R^{-1} \). A state \( q \in N \) is reachable iff \( q \in R^*[M] \), that is to
say, \( q \) belongs to the image\(^2\) of the initial states by the reflexive and transitive closure of \( R \). If \( R \subseteq N_1 \times N_2 \) then we denote by \( \text{pre}[R] \in \mathcal{P}(N_2) \rightarrow \mathcal{P}(N_1) \) the “predecessor” function on sets of states and by \( \text{post}[R] \in \mathcal{P}(N_1) \rightarrow \mathcal{P}(N_2) \) the “successor” function. Let \( Q \) be a subset of \( N \), then \( \overline{Q} \) is the complement of \( Q \) with respect to \( N \). The following definitions are usual on binary relations [18, 15], where \( R \subseteq N_1 \times N_2 \), \( Q_1 \subseteq N_1 \) and \( Q_2 \subseteq N_2 
:
\begin{align*}
\text{pre}[R](Q_2) &= \{ q \mid q \in N_1 \land \exists q' \cdot (q' \in Q_2 \land (q, q') \in R) \} \\
\text{post}[R](Q_1) &= \{ q' \mid q' \in N_2 \land \exists q \cdot (q \in Q_1 \land (q, q') \in R) \} \\
\overline{\text{pre}[R]}(Q_2) &= \text{pre}[R](\overline{Q_2}) \\
\overline{\text{post}[R]}(Q_1) &= \text{post}[R](\overline{Q_1})
\end{align*}

If \( R \) is total, then \( \overline{\text{pre}[R]} \subseteq \text{pre}[R] \) and if \( R \) is a function, then \( \overline{\text{pre}[R]} = \text{pre}[R] \).

For labelled transition systems, the labels can be considered as references to the relations they label. So, the following definitions will be used, where \( \mathcal{R} \subseteq (N \times L \times N) \) and \( Q \subseteq N 
:
\begin{align*}
\text{pre}[\mathcal{I}](Q) &= \{ q \mid q \in N \land \exists q' \cdot (q' \in Q \land (q \overset{l}{\rightarrow} q') \in \mathcal{R}) \} \\
\text{post}[\mathcal{I}](Q) &= \{ q' \mid q' \in N \land \exists q \cdot (q \in Q \land (q \overset{l}{\rightarrow} q') \in \mathcal{R}) \}
\end{align*}

3 Principles for the Construction of Finite LTS

In this section, we study several ways to build finite labelled transition systems from B abstract systems. To illustrate the various cases, the example of the specification of SCSI-2 (Fig. 1) will be taken.

3.1 Enumeration of the States

The simplest way to define a labelled transition system from a B abstract system is to enumerate the states and the transitions. Given an abstract system \( \mathcal{S} \) with list of variables \( X = (x_1, \ldots, x_k) \), invariant \( I \), initialisation \( U \), list of events \( (E_i = E_i) \) for \( i \in 1..m \), then the initial states are the assignments \( \sigma_0 : \)

\[
\sigma_0 = \{ x_j \mapsto v_j \mid (U)(x_j = v_j) \} \quad \forall j \in 1..k
\]

For each state \( \sigma_p \) which satisfy the invariant \( I \) and each event \( E_i \), such that this event is enabled in the state \( \sigma_p \), the successors of \( \sigma_p \) by the transition labelled by \( E_i \) are the states the values of which are possibly obtained after the generalized substitution \( E_i \):

\[
\sigma_{p+1} = \{ x_j \mapsto v'_j \mid [\sigma_p](\langle E_i \rangle (x_j = v'_j)) \} \quad \forall j \in 1..k
\]

From the definitions of the B-Book [1] (page 296), that means that, if \( (x_j \mapsto v_j) \in \sigma_p \) then

\[
(v_j, v'_j) \in \text{rel}_x(E_i)
\]

\(^2\)In the paper, we use systematically the B set notations.
where \( \text{rel}_{E_i} \) is the binary relation which relates the values of \( x \) before and after the substitution \( E_i \). Because of our assumption about termination, the link between the \( B \) events \( E_i = E_j \) and the transitions of the associated LTS is:

\[
\text{post}[E_i] = \text{rel}(E_i)
\]

from which, we deduce: \( \text{pre}[E_i] = \text{rel}(E_i)^{-1} \) and \( \text{str}[E_i] = \text{str}(E_i) \). Usually, we will be interested in the reachable states; these states can be built inductively from the initial states. As an application, the LTS obtained from the example of Fig. 1 when \( \text{maxi} = 2 \) and \( \text{DSK} = \{d_1, d_2\} \) is given in Fig. 2. It exactly contains \( 3^2 = 9 \) reachable states.

![Fig. 2. LTS of the enumerated states of the simple SCSI-2.](image)

### 3.2 Symbolic Evaluation and Set Constraints

This approach was taken in [16] by using a formalism of constraint logic programming. The idea is to deal with a symbolic representation of the states or more precisely, with a representation of states by a mapping between variables and set constraints. The language of set constraints can be supported by logical solvers, augmented by procedures for solving satisfaction of set constraints [13, 14].

Set constraints are built upon usual set connectors, as \( \in, \notin, =, \neq \), logical connectors \( \land, \lor, \forall, \exists \) and the cardinality constraint. They are extended to cartesian product to encompass operations of relations and functions. All the operators on sets must be interpreted as operators on set constraints. This algebra is not developed here. In the same way, integer values must be represented as constraints (not implemented in [16]). The computation of reachable states relies upon several operations. Let \( C_1 \) and \( C_2 \) be set constraints:

- \( C_1 \oplus C_2 \) is the union of both constraints;
– $C_1 \subseteq C_2$ is the inclusion of set constraints. This inclusion means that the state represented by the constraints $C_2$ satisfies the constraints expressed by $C_1$. This can be used to prove that a state satisfies the invariant;
– reduction (red) of constraints into a normal form with disjunctions ($\lor$) of set of constraints;
– $C_1 - \{x\}$ suppression of $x$ in the set of constraints $C_1$;

For building LTS, there must be a procedure for the comparison of sets of set constraints, and for deciding if set constraints are satisfiable. Finally, the generalized substitutions must be interpreted as operations on constraints. If the substitution $S$ is enabled in a state $C_p$, then the operation $\text{propagate}(C_p, S)$ compute a new set of constraints which is the propagation of $C_p$ through $S$. For example, we have for the simple substitution cases:

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\text{propagate}(C_p, S)$</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := v$</td>
<td>$[x := x'] (\text{red}(C_p \oplus x' = v) - {x})$</td>
<td>$C_p \oplus P$ satisfiable</td>
</tr>
<tr>
<td>skip</td>
<td>$C_p$</td>
<td>$P \subseteq C_p$</td>
</tr>
<tr>
<td>$P \mid S$</td>
<td>$\text{propagate}(C_p \oplus P, S)$</td>
<td></td>
</tr>
<tr>
<td>$P \Rightarrow S$</td>
<td>$\text{propagate}(C_p, S)$</td>
<td></td>
</tr>
<tr>
<td>$S \mid T$</td>
<td>$\text{propagate}(C_p, S) \lor \text{propagate}(C_p, T)$</td>
<td></td>
</tr>
</tbody>
</table>

One can notice that, if the state space of the actual system is finite, then, it is possible to generate enumerated states from constrained states (by enumeration). Because $\mathbb{B}$ is strongly based on set theory, this method allows the generation of transition systems up to set properties (commutativity and associativity of the choice of elements in the $\mathbb{B}$ sets). In our example, if we choose as above the cardinal of $\text{DSK}$ to be 2, then the generation of the whole LTS is reduced by the method of set constraints to only 6 states. This is represented in Fig. 3. In the states, the variables do not identify specific elements. They contain the information about the elements, independently of their name. Indeed, all the states contain the constraint: $jj1 \in \{d1, d2\} \land jj2 \in \{d1, d2\} \land jj1 \neq jj2$. For instance, in the second state of the figure below, which represents two states of the enumerated graph, if the event $\text{ctr\_cmd}$ is fired, then the interpretation of the “ANY” substitution chooses a new element as either the one already chosen $jj1$ or the other one $jj2$ different from $jj1$. After that, in any cases, from both states, a new $\text{ctr\_cmd}$ event leads to only one event where a buffer is full (2 messages) and the other contains one element.

### 3.3 Abstract Interpretation

Abstract interpretation [7] is a very general technique which relies upon evaluation of syntactic descriptions (programs, specifications, etc.) into non-standard domains. These domains are usually simpler than the original ones. The interest of abstract interpretation is that it is possible to check automatically, on the abstract domain, properties that hold on the effective domain but which are not decidable on it. Obviously, the abstraction must be chosen appropriately to
preserve the desired properties. Moreover, the result is the most often an approximation of the effective properties, not an exact calculus. Nevertheless, the method is more and more used in various areas (imperative programming, logic programming, concurrent and distributed processing, rewriting systems, etc.).

In the domain of transition systems, a lot of work has been done to deal with infinite systems and to extend capabilities of model-checking. One can cite [8,9,11,5] among others. A systematic study of the properties preserved by abstraction has been done in [15]. Here we want to apply the mechanisms presented in [12] to B abstract systems. With respect to this work, we intend to use the B toolbox to validate the conditions that define the transitions rather than the PVS theorem prover.

An abstract LTS is similar to a concrete LTS, except that the set of nodes is a powerset of ordinary nodes. So, the nodes are structured as lattice, that is to say, are closed by the operations of union \( \cup \) and intersection \( \cap \). The top element (\( \top \)) represents all the nodes and the bottom element (\( \bot \)) represents the state of an empty set of nodes. Any LTS can be “lifted” into a powerset LTS, by taking the union of states and the unions on transitions. This powerset is ordered by the inclusion relation deduced from the lattice structure.

The abstraction relation between concrete and abstract labelled transition systems is characterized by a pair of monotonic functions \((\alpha, \gamma)\) from \(P(N)\) to \(P(N^A)\) also called Galois connections:

**Definition 1 (Galois connections of LTS).** Let \(T = (N,M,L,R)\) be a concrete LTS (Section 2.3) and \(T^A = (N^A,M^A,L^A,R^A)\) be an abstract LTS, then \((\alpha, \gamma)\) is a Galois connection between \(T\) and \(T^A\) iff the following conditions hold:

1. \(\alpha \in P(N) \rightarrow P(N^A)\) and \(\gamma \in P(N^A) \rightarrow P(N)\)
2. \(\alpha \circ \gamma \subseteq \text{id}_{P(N^A)}\) and \(\text{id}_{P(N)} \subseteq \gamma \circ \alpha\)
3. \(M \subseteq \gamma(M^A)\)
4. \(\forall l \in L, \forall q^A \in P(N^A), \text{post}[l](\gamma(q^A)) \subseteq \gamma(\text{post}[l](q^A))\)

The first two conditions are very general in the framework of abstraction [15]. The last two ones are specific to LTS. They express that for each concrete behaviour, there exists at least one abstract behaviour described by the abstract LTS. So, the abstract behaviours contain all the concrete behaviours. The mappings \(\alpha\) and \(\gamma\) can be deduced from each other by:

![Fig. 3. LTS of the constraint states of the simple SCSI-2.](image)
Proposition 1. For any connection \((\alpha, \gamma)\) from \(\mathbb{P}(N)\) to \(\mathbb{P}(N^A)\), we have:

1. \(\gamma = \lambda Y \cdot \bigcup \{ Z \in \mathbb{P}(N) \mid \alpha(Z) \subseteq Y \} \)
2. \(\alpha = \lambda Z \cdot \bigcap \{ Y \in \mathbb{P}(N^A) \mid Z \subseteq \gamma(Y) \} \)

As previously, we first illustrate the approach on the example of Fig. 1. In this abstraction\(^3\) of SCSI-2 (called \(SCSI-2^{A_1}\)), we consider a state \(mm\) which is the number of messages which are in the disk buffers. So, the variable of this abstract system is semantically defined as:

\[
mm = \sum jj \cdot (jj \in DSK \mid buf(jj))
\]

The next step is to compute an abstract LTS from \(SCSI-2^{A_1}\) where the state is \(mm\) and to compare it with the LTS presented in section 3.1. In this new LTS, the union of states (in the lattice of the states) could be considered as other states. They are useless here, so they are not drawn to simplify the figure.

![LTS of the first abstract SCSI-2](image)

Fig. 4. LTS of the first abstract SCSI-2.

The correspondence between the enumerated LTS in Fig. 2 and the abstract one in Fig. 4 can be defined completely on the states, through the abstraction and concretization functions. The abstraction conditions hold trivially with these functions. Because the states are finite, this can be listed as in the following table:

<table>
<thead>
<tr>
<th>(\nu \in \mathbb{P}(N))</th>
<th>(\alpha(\nu) \in \mathbb{P}(N^A))</th>
<th>(\mu \in \mathbb{P}(N^A))</th>
<th>(\gamma(\mu) \in \mathbb{P}(N))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>({1})</td>
<td>({A})</td>
<td>({A})</td>
<td>({1})</td>
</tr>
<tr>
<td>({2})</td>
<td>({B})</td>
<td>({B})</td>
<td>({2,3})</td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
</tr>
<tr>
<td>({1,2})</td>
<td>({A,B})</td>
<td>({A,B})</td>
<td>({1,2,3})</td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
</tr>
</tbody>
</table>

3.4 Yet Another Abstract Interpretation

Another simple abstraction of \(SCSI-2\), called \(SCSI-2^{A_2}\) is defined by a boolean variable:

\[
empty = \text{bool}(\forall jj \cdot (jj \in DSK \Rightarrow buf(jj) = 0))
\]

In that case, the abstract LTS contains two states and the transitions given in Fig. 5.

\(^3\) It is worth noticing the terminology problem. Here we want to take an abstract interpretation of an abstract system. We hope that the context allows the reader to understand what abstraction is meant.
If we consider the definition of the abstract variables with respect to the concrete ones, it determines an obvious relation. For example, we have the relations:

\[
\rho_1 = \{ \text{buf, mm} \mid \text{mm} = \sum_{j} (j \in \text{DSK} \mid \text{buf}(j) = 0) \}
\]

\[
\rho_2 = \{ \text{buf, empty} \mid \text{empty} = \text{bool}(\forall j \cdot (j \in \text{DSK} \Rightarrow \text{buf}(j) = 0)) \}
\]

So, given such a relation \( \rho \subseteq N \times N^A \), it is possible to build Galois connections, as expressed in Proposition 2 [15].

**Proposition 2.** Let \( \rho \) be a relation on \( N \times N^A \), then the pair \( (\text{post}[\rho], \text{pre}[\rho]) \) is a connection from \( \mathcal{P}(N) \) to \( \mathcal{P}(N^A) \).

### 3.5 A Simple Abstraction Scheme: Disjunctive Decomposition

In B, the state \( X \) is specified by a predicate (the invariant) \( I \). The principle of the abstraction that we call *disjunctive decomposition* is to find \( n \) predicates \( \phi_1, \ldots, \phi_n \) such that:

\[
I \Rightarrow \bigvee_{j=1}^{n} \phi_j
\]

In the following, we consider that each predicate \( I \land \phi_j \) is identified by a state \( q_j \). To consider the disjunctive decomposition as an abstraction, we have to define the abstraction framework, as sketched now.

The states can be combined by logical connectors, thus providing the lattice structure of the abstract system space. The nodes of this state space are denoted by boolean expressions of the form \( \beta(q_1, \ldots, q_n) \). The concretization function is then defined by:

\[
\gamma(\beta(q_1, \ldots, q_n)) = \{ X \mid [q_j := \phi_j](\beta(q_1, \ldots, q_n)) \}
\]

More specifically, we have \( \gamma(q_j) = \{ x \mid I \land \phi_j \} \). Following Proposition 1, the associated abstraction function could be:

\[
\alpha(Z) = \bigwedge \{ \beta(q_1, \ldots, q_n) \mid Z \subseteq \gamma(\beta(q_1, \ldots, q_n)) \}
\]
that is to say, if \( Z \) is characterized by the predicate \( \phi \):

\[
\alpha(\phi) = \bigwedge \{ \beta(q_1, \ldots, q_n) \mid \phi \Rightarrow [q_j := \phi_j((\beta(q_1, \ldots, q_n)) \}
\]

Rather than this complex definition, following [12], we use the approximation:

\[
\alpha'(\phi) = \bigwedge_{j=1}^{n} \{ q_j \mid \phi \Rightarrow \phi_j \}
\]

The pair \((\alpha', \gamma)\) is a Galois connection too. Moreover, the computation of the abstract transition relations can be restricted to the calculus on the states \( I \land \phi_j = q_j \).

Transitions are associated to the events of the \( B \) system. One says that the event \( \mathcal{E}_i \) is an abstract transition between the states \( q_j \) and \( q_k \) if the guard is true in the concretization of \( q_j \) and if it is possible to reach the concretization of \( q_k \):\[
q_j \xrightarrow{\mathcal{E}_i} q_k \Leftrightarrow \begin{cases} I \land \phi_j \Rightarrow \text{grd}(E_i) \\ I \land \phi_j \Rightarrow \langle E_i \rangle \phi_k \end{cases}
\]

The initial abstract state is the set of states which are possibly true after the initialization, that is:

\[
q_u \in M^A \Leftrightarrow \langle U \rangle (I \land \phi_u)
\]

The abstraction \( \text{SCSI-2}^A \) is a particular case of disjunctive decomposition with:

\[
\begin{align*}
A1 & \forall jj : (jj \in \text{DSK} \Rightarrow \text{buf}(jj) = 0) \\
A2 & \exists jj : (jj \in \text{DSK} \land \text{buf}(jj) \neq 0)
\end{align*}
\]

The condition \( \text{buf} \in \text{DSK} \Rightarrow \text{SIZE} \Rightarrow A1 \lor A2 \) holds. Moreover, this decomposition is a partition because: \( A1 \land A2 \Leftrightarrow \text{FALSE} \).

4 Construction of Abstract LTS and Applications

4.1 The SCSI-2 Example Continued

We are now interested in refinements of the SCSI-2 input-output system. The next refinement is to introduce a bus. This bus is either empty or it contains only one command. This command is a command sent from the controller to a disk \( uu \) or it is the end of processing a command by the disk \( vv \). So, the set of information transmitted by the bus is:

\[
\emptyset \quad \text{the bus is empty} \\
\{ \text{CMD} \mapsto uu \} \quad \text{a command is sent to the disk } uu \\
\{ \text{REC} \mapsto vv \} \quad \text{end of command processing from the disk } vv
\]

The command set on the bus consists of the enumerated set: \( \text{ACT} = \{ \text{CMD}, \text{REC} \} \). The variables are:
bus ∈ ACT ↔ DSK  the bus
\[ c_{2,dsk} ∈ DSK → \text{SIZE} \text{ information on the disks known by the controller} \]
\[ d_{2,buf} ∈ DSK → \text{SIZE} \text{ information known by the disks} \]

The gluing invariant means that the informations on both sides of the bus are equal if the bus is empty. Otherwise, there is a difference because of the delay between the sending and the reception of a message, via the bus. The expression \( f \leftarrow \{ x \mapsto v \} \) means that the function \( f \) is modified at the point \( x \) by the value \( v \) (function overriding).

\[
(bus = \emptyset \Rightarrow (c_{2,dsk} = buf \land d_{2,buf} = buf)) \land
\forall uu \cdot (uu ∈ DSK \land bus = \{ \text{CMD} \mapsto uu \} \Rightarrow
(c_{2,dsk} = buf \land (d_{2,buf} ≜ \{ uu \mapsto d_{2,buf}(uu) + 1 \}) = buf)) \land
\forall vv \cdot (vv ∈ DSK \land bus = \{ \text{REC} \mapsto vv \} \Rightarrow
((c_{2,dsk} ≜ \{ vv \mapsto c_{2,dsk}(vv) - 1 \}) = buf \land d_{2,buf} = buf))
\]

In this refinement, we consider four events (see Fig. 6):

- \( ctr_{cmd} \) the controller sends a command through the bus
- \( dsk_{cmd} \) a disk receives a command from the bus
- \( dsk_{rec} \) a disk sends the result of a command through the bus
- \( ctr_{rec} \) the controller receives the result of a command from the bus

As an application of the method of construction of a labelled transition system, we apply the method on the set of states:

\[
\begin{align*}
B_1 & \quad \text{bus} = \emptyset \\
B_2 & \quad \exists jj \cdot (jj ∈ DSK \land bus = \{ \text{CMD} \mapsto jj \}) \\
B_3 & \quad \exists jj \cdot (jj ∈ DSK \land bus = \{ \text{REC} \mapsto jj \})
\end{align*}
\]

The associated LTS contains three states. The events which are enabled in each state are computed from the conditions on the guards. For example for \( B_1 \), the following proof obligations are generated:

\[
\begin{align*}
1. & \quad I \land bus = \emptyset \Rightarrow \exists jj \cdot (jj ∈ DSK \land c_{2,dsk}(jj) < maxi \land bus = \emptyset) \\
2. & \quad I \land bus = \emptyset \Rightarrow \exists jj \cdot (jj ∈ DSK \land bus = \{ \text{CMD} \mapsto jj \}) \\
3. & \quad I \land bus = \emptyset \Rightarrow \exists jj \cdot (jj ∈ DSK \land d_{2,buf}(jj) > 0 \land bus = \emptyset) \\
4. & \quad I \land bus = \emptyset \Rightarrow \exists jj \cdot (jj ∈ DSK \land bus = \{ \text{REC} \mapsto jj \})
\end{align*}
\]

From these conditions, 1 and 3 are true. But 2 and 4 are not proved. That does not mean that the guard is effectively false. For that we have to prove:

\[
\begin{align*}
2'. & \quad I \land bus = \emptyset \Rightarrow \forall jj \cdot (jj ∈ DSK \Rightarrow bus \neq \{ \text{CMD} \mapsto jj \}) \\
4'. & \quad I \land bus = \emptyset \Rightarrow \forall jj \cdot (jj ∈ DSK \Rightarrow bus \neq \{ \text{REC} \mapsto jj \})
\end{align*}
\]

The successors of the state \( B_1 \) by the event \( ctr_{cmd} \) can be computed from the conditions below, where \( E_{ctr_{cmd}} \) is put for the generalized substitution “any \( jj \) where \( jj ∈ DSK \land c_{2,dsk}(jj) < maxi \land bus = \emptyset \) then \( bus := \{ \text{CMD} \mapsto jj \} \land c_{2,dsk}(jj) := c_{2,dsk}(jj) + 1 \) END”: 
The second formula holds, so one concludes that the transition: $B_1 \xrightarrow{ctr.cmd} B_2$ is possible in the abstract LTS. Here again, if the formula is not proved, one must try to prove the negation to be sure that the transition is not possible at all. From the complete calculus, the transition graph looks like the one in Fig. 7.

4.2 Disjunctive Decomposition and B Refinement

In a system refinement, as for a machine refinement, if the abstract system invariant is denoted by $I$, then the refined system invariant is $J$ where the
relation between the concrete state space on $y$ and the abstract state space on $x$ is given by: $I \land J$. Obviously, the new state space is the set of $y$ that satisfy $\exists x \cdot (I \land J)$.

When the abstract state has been decomposed into a set of disjunctive predicates $\bigvee_{j=1}^{n} \phi_j$ it is interesting to see if the refined state can be decomposed into images of the abstract system. The principle is easy; we have:

$I \land J \Rightarrow I \land \bigvee_{j=1}^{n} \phi_j \land J \Leftrightarrow \bigvee_{j=1}^{n} (I \land \phi_j \land J)$

From a theoretical point of view, the refined state is decomposed into $n$ subspaces defined by:

$\exists x \cdot (I \land \phi_j \land J)$

If the variables $x$ can be eliminated from this formula, then the expression provides a way to decompose the refined system.

Let us take the abstract SCSI-2 system with the second abstraction defined in Section 3.4. The calculus of A1 refined by the refinement above produces the case $A1'$ ($\text{empty} = \text{true}$):

$\{ \text{bus} = \emptyset \Rightarrow (c2_dsk = d2_buf \land \forall jj \cdot (jj \in \text{DSK} \Rightarrow c2_dsk(jj) = 0)) \land \\
\forall uu \cdot (uu \in \text{DSK} \Rightarrow \text{bus} \neq \{ \text{CMD} \mapsto uu \}) \land \\
\forall vv \cdot (vv \in \text{DSK} \land \text{bus} = \{ \text{REC} \mapsto vv \} \Rightarrow \\
(\forall jj \cdot (jj \in \text{DSK} \Rightarrow d2_buf(jj) = 0) \land (jj \neq vv \Rightarrow c2_dsk(jj) = 0)) \\
\land \ c2_dsk(vv) = 1) \}$

The state $A2'$ is the complement of $A1'$. The graph obtained by this abstraction is displayed in Fig. 8.

Another case of relationship between a refined and an abstract system is when the latter is decomposed into several states: $\phi_1, \phi_2, \ldots, \phi_p$. If the abstract states are denoted by $q_j$, $j = 1..n$ and the refined ones by $r_l$, $l = 1..p$, then the two LTS are themselves related by the state relation:

$\{ q_j, r_l \mid \exists x \cdot (I \land J \land \phi_l \Rightarrow \phi_j) \}$

### 4.3 Properties of Abstract Transition Systems

In this section, we consider the properties which can be proved by using the abstraction scheme defined by a disjunction of predicates and the representation
of an abstract system $S$ by a finite LTS $T$. From the definition of the initial states and the condition that all the abstract values are represented, then the concretization of the initial states of $T$ is included into the initial states of $S$. Then for any change of state $\sigma_j$ to $\sigma_k$ obtained by the event $E_i$ (where $\sigma_j$ and $\sigma_k$ satisfy the invariant $I$), there exists a transition $q_j \xrightarrow{E_i} q_k$ in $T$. So, the first result is [15]:

Proposition 3 (Invariant properties). Any property which is true on the reachable states of an LTS $T$ built from an abstract system $S$ by a disjunctive decomposition is an invariant property in $S$.

An improvement of the proposition above to prove invariant properties by model-checking on abstractions of the transition systems is to compute backward by approximation all the predecessors of a state which satisfy the desired property [5]. This technique builds a finite LTS which proves that the property holds in the actual system if the initial state(s) is (are) in the LTS. Obviously, the technique may not succeed, if the property does not hold, or if it is not inductive (so there is no possible finite decomposition). We have not yet experimented this method with the tools we develop around the B formalism.

In the refinement of abstract systems, new events can be introduced. These new events must refine the substitution skip. Moreover, to satisfy fairness conditions, the new events must not take control for ever, thus not preventing the “old” events to occur. Otherwise, the behaviour of the refined system should not be the same as the behaviour of the abstract system. In Abrial and Mussat’s paper [3], it is suggested that this proof can be done by the way of variant expressions which give a termination condition for the cycle of new events. Here we indicate another way connected to the construction of LTS.

Proposition 4 (Fairness of new events). In a LTS built from a refined system by a disjunctive decomposition, if there is no cycle of new events, then the refined system is fair, with respect to the old events.

The proof is obvious because if there is a cycle of the new events in the effective behaviour of the refined system, then there is also a cycle in the associated LTS, whatever the abstraction is.
In the example of the refined system of Fig. 8, the new events appear in cycles. So, we cannot conclude that the refinement is fair by this abstraction.

4.4 Development of the SCSI-2 System

The next development of the SCSI-2 system consists in finding an arbitration protocol to take control of the bus. In a simplified view of the SCSI-2 system, we can say that each peripheral gets a number which is its priority. The controller gets the highest priority. When the controller or a peripheral wants to transmit a message, then it asks for an access. All the peripherals and eventually the controller which ask for the access to the bus are competing all together. The device with the highest priority wins the control and can use the bus. To specify this arbitration protocol, we use a set of natural numbers identifying the devices $DEV$. In the minimal configuration, this is the interval $0 \ldots 7$, where $7$ is the number of the controller denoted $CTR$. So, $DSK \cup \{ CTR \} = DEV$ and $DSK \cap \{ CTR \} = \emptyset$. The set of peripherals which want to access the bus, is defined by:

$$wire \in \mathcal{P}(DEV)$$

When the set $wire$ is not empty, the winner of the arbitration is the number $kk$ which satisfies:

$$winner(kk) \iff bus = \emptyset \land kk = \max(wire)$$

We must keep the information that a controller decides to be engaged with a disk. For that, the new variable $dsk_{req} \in DEV$ contains the value of the disk number $jj$ if the controller is engaged with $jj$, otherwise, it contains the value of the controller. This is specified by the predicate:

$$ctr\_engaged \iff dsk\_req \in DSK$$

The new events are those where a peripheral or the controller asks for the access to the bus. They are:

- $ctr\_acc$ the controller asks for the access to the bus
- $dsk\_acc$ a disk asks for the access to the bus

The refinement of the events is given in Fig. 9.

We apply the method of disjunctive decomposition of the state of this refined system by considering the following cases:

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>$bus = \emptyset$</td>
</tr>
<tr>
<td>B2</td>
<td>$\exists jj \cdot (jj \in DSK \land bus = { CMD \mapsto jj })$ the bus automaton</td>
</tr>
<tr>
<td>B3</td>
<td>$\exists jj \cdot (jj \in DSK \land bus = { REC \mapsto jj })$</td>
</tr>
<tr>
<td>C1</td>
<td>$wire = \emptyset$</td>
</tr>
<tr>
<td>C2</td>
<td>$CTR \in wire$</td>
</tr>
<tr>
<td>C3</td>
<td>$\exists jj \cdot (jj \in DSK \land jj = \max(wire))$ that is: $winner(CTR)$</td>
</tr>
<tr>
<td>D1</td>
<td>$dsk_req = CTR \land \forall jj \cdot (jj \in DSK \Rightarrow jj \notin wire)$</td>
</tr>
<tr>
<td>D2</td>
<td>$dsk_req = CTR \land \exists jj \cdot (jj \in DSK \land jj \in wire)$</td>
</tr>
<tr>
<td>D3</td>
<td>$\exists jj \cdot (jj \in DSK \land dsk_req = jj)$</td>
</tr>
</tbody>
</table>
**Fig. 9.** Second refinement of the SCSI-2 system.

Conditions C describe the winning conditions by the examination of the content of wire. Conditions D indicate when the controller is engaged with a disk or not. Because the predicates range over various variables, we consider the decomposition obtained by monomials on the three groups of predicates. That means we have the states $B_1 \land C_1 \land D_1$, $B_2 \land C_1 \land D_1$, ..., $B_3 \land C_3 \land D_3$. Among these 27 states, only 10 are reachable from the initialization, which is characterized by $B_1 \land C_1 \land D_1$. The graph of the LTS is shown in Fig. 10.
On this graph, the relation with the LTS displayed at Fig. 7, corresponds to the homomorphic grouping $B_1 = \{1, 9, 12\}$, $B_2 = \{2, 6, 7, 10\}$ and $B_3 = \{5, 11, 14\}$. From a simple examination of the transitions, it is clear that the new events of the refinement ($\text{ctr}_{\text{acc}}$ and $\text{dsk}_{\text{acc}}$) cannot take the control for ever.

5 The CADP Tool

CADP [10] is a toolbox for protocol engineering. It offers many functionalities, from interactive simulation to the most recent formal verification techniques. It is dedicated to the efficient compilation, simulation, formal verification, and testing of descriptions written in the ISO language LOTOS. Besides the LOTOS language, the toolbox accepts descriptions as finite state automata, and networks of communicating finite state automata. By this way, we can transfer an LTS computed by analysis of an abstract system into the CADP environment. Among the other functionalities of CADP, there are two different tools for computing bisimulations, that achieve minimizations and comparisons of LTS; two different model-checkers for various temporal logic and $\mu$-calculus, and several verification algorithms. Finally, CADP offers a friendly user interface and a visual representation of the LTS. This is very useful for an easy presentation of the specifications to the non-specialist people.

The process of the construction of a LTS and its submission to CADP is the following. Given a $\mathcal{B}$ abstract system (or a refinement), the user provides a list of predicates $\phi_j$. From this list, a preprocessor can compute a list of formulas as defined in section 3.5. Actually, the preprocessor must dialog with the prover of the AtelierB. The first thing to do is to compute and to try to prove the conditions on the guards which determine what are the events enabled in each
state. If \( n \) is the number of states and \( m \) is the number of events, this leads to \( n \times m \) proofs. For those which are not proved automatically, the preprocessor computes the negation and submit the formulas to the prover. Here again, for the formulas which are proved automatically, then the decision that the event is not enabled in this state is guaranteed. For the other formulas, no decision can be taken. One possibility is to try to prove them (either the formula or its negation) interactively, or (as in [12]) the other possibility is to consider that the event is enabled. Let us remind that the principle is to compute an approximation of the effective LTS, without losing any possible transition. In a second phase, the preprocessor computes the formulas that characterize the transitions themselves, for the enabled events. The maximum number of generated formulas for determining the transitions which can occur is: \( 2 \times n^2 \times m \).

After this second phase, the LTS can be sent to the CADP toolbox and, optionally, functions of the toolbox can be applied (cycle discovery, reachable states, etc.).

6 Conclusion

In this paper, we have shown several methods to build a finite labelled transition system (LTS) from a B abstract system. Then, we have focus on a particular abstraction technique called “disjunctive decomposition”. We have shown several properties of this abstraction and we have applied it to a SCSI-2 [17] input-output case study.

A first outcome of such a representation is to obtain a graphical view of the behaviour of the system. Another one is to shown that some properties hold in the finite model without a complete specification of the elements of the proof by invariant and variant, as defined in [3]. At this stage of the work, it is not sure whether the approach will be useful for very large B systems. The main problem comes from the fact that we need to prove formulas to compute transitions. An important aspect of the further experiments with our tool is to measure the rate of automatic proofs in this kind of process. Probably some tactics can be tuned to improve the rate obtained by a naive use of the B prover. As explained in [12], the construction of an LTS can be fully automatic, but we wonder if the approximation obtained will not be too large, because this is depending on the strength of the prover. So, another outcome of the study is to appreciate if the B prover is clever enough to build a selective LTS.

Another research direction is to use the same machinery to prove inductive invariants by backward symbolic computation as shown in [5]. In that case, it is not necessary to give a decomposition of the global state. Instead, one gives the property to be proven about the system. Then, starting from the most abstract LTS, with one state and all the possible transitions, the tool computes what are the transitions which do not preserve the property. These transitions are removed and the state is split if necessary. At the end, if the initial states are in the automaton, then the tool guarantees that the property holds in the system.
References

\section*{Abstract.} In this paper we show, by a series of examples, how the $\mu$-chart formalism can be translated into Z. We give reasons for why this is an interesting and sensible thing to do and what it might be used for.

\section{Introduction}

In this paper we show, by a series of examples, how the $\mu$-chart formalism (as given in \cite{10}) can be translated into Z. We also discuss why this is a useful and interesting thing to do and give some examples of work that might be done in the future in this area which combines Z and $\mu$-charts.

It might seem obvious that we should simply express the denotational semantics for $\mu$-charts given in \cite{10} directly in Z and then do our proofs. After all, the semantics is given in set theory and so Z would be adequate for the task. However, our aim is to produce versions of $\mu$-charts that are recognisably Z models, \textit{i.e.} using the usual state and operation schema constructs and some schema calculus in natural ways—$\mu$-chart states and transitions appear as Z state and operation schemas respectively. Thus we aim for a Z model in the conventionally accepted form. We want this so that the Z model is readily understandable in its own right and also comparable with alternative models in Z, \textit{e.g.} those that are part of the outcome of Weber’s framework \cite{20} (which we will mention briefly again in Section 5.3).

To make the paper fairly self-contained we start by giving a brief introduction to the main conceptual apparatus of this paper: the reactive system modelling formalism of $\mu$-charts, the specification language Z and the Z proof tool Z/EVES. In each case, though, the references cited should be used by readers who wish to have a full and proper introduction to each topic.

We then explain our translation via a series of examples in Sections 2 and 3. In Section 4 we give an extended commentary on aspects of \cite{10} in which we seek to show how the concerns of that paper, which uses the technique of oracles, have been dealt with by our translation. We also, somewhat speculatively, consider extending the technique in \cite{10} in a way which appears to combine the semantics in that paper with the alternative semantics given in \cite{15}. Finally, we motivate the current work and look into the future.
1.1 \( \mu \)-Charts

\( \mu \)-Charts [10] are a visual representation used for the specification of reactive systems. They extend finite state transition diagrams by adding modularisation through hierarchical decomposition, \( i.e. \) allowing states to contain other \( \mu \)-charts, and allow the modelling of separate communicating processes by parallel composition, that is allowing two \( \mu \)-charts to communicate via broadcast signals. The communication between \( \mu \)-charts in a specification is modelled using instantaneous feedback of signals.

\( \mu \)-Charts are a variant of Statecharts [2] that exclude a number of syntactic concepts which cause semantic problems, \( e.g. \) inter-(hierarchy-)level transitions\(^1\). Also \( \mu \)-charts have been given a compositional semantics (and allow abstract specification by incorporating \( e.g. \) non-determinism) and refinement rules have also been given that describe a formal process of making an abstract or non-deterministic \( \mu \)-chart more concrete [14,15].

The compositional semantics for \( \mu \)-charts considered in this paper is that described in [10], though we do acknowledge there have since been alternatives to this proposed, as in [14] and [15]. The ramifications of these alternatives have not yet been fully investigated, though we make some comments on this in Sections 4, 5 and 6. Since we shall be introducing \( \mu \)-charts via several examples in the sequel, we will say no more about them here.

1.2 Z

The specification language Z is based upon typed set theory and first-order predicate calculus and includes the notion of schemas used to encapsulate mathematical objects and their properties by declaration and constraint. Z is a model-based notation usually used to represent abstract specifications of systems by describing observations of their state and some operations that can change that state (see [8], [21]).

To give an example of Z we specify a schema named \textit{State} with a single integer observation that is constrained to remain within a given range:

\[
\begin{align*}
\text{State} & \\
x & : \mathbb{N} \\
x & > 10 \\
x & \leq 20
\end{align*}
\]

\[
\begin{align*}
\text{InitState} & \\
x & = 11
\end{align*}
\]

Z state schemas are divided into declarations, \( i.e. \) observation names (or labels) and their types, and predicates constraining those observations. This division is represented graphically by a horizontal line. Two other principles of Z demonstrated here are the convention of initialisation and schema inclusion. Z specifications generally include an initialisation schema, \( e.g. \) \textit{InitState}, that specifies what happens at system startup. Schema inclusion, \( e.g. \) of \textit{State} included

\(^1\) The ZedCharts work [17] grew out of similar considerations and reservations.
in the declaration of $\text{InitState}$, is a notational shorthand for including the respective declaration and predicate of one schema into another (subject to certain consistency requirements between the declarations).

Operation schemas are different from state schemas in that they include two observations of the state. One, known as the unprimed copy, represents the values of the state observations before the operation and the other, the primed copy, represents the values of the observations after the operation.

\[
\begin{array}{c|c}
\text{Increment} & \text{Restart} \\
\hline
\Delta \text{State} & \Delta \text{State} \\
x < 20 & x \geq 20 \\
x' = x + 1 & x' = 11 \\
\end{array}
\]

$T(Op) \equiv \text{Increment} \lor \text{Restart}$

The schema inclusion of $\Delta \text{State}$ is another convention in Z where $\Delta$ schemas are recognised to be defined as including a primed and unprimed copy of the schema, e.g. $\Delta \text{State} \equiv [\text{State}, \text{State'}]$. Also schemas including $\Delta$-schemas are conventionally recognised to be operations that change the state. All the various forms of schema given above are examples of Z paragraphs.

An operation schema has a precondition which has explicit and implicit components, taken in conjunction, e.g. $\text{Increment}$ has the components $x < 20$, $x \leq 20$ and $x > 10$; the first explicit, the last two implicit. An operation is said to be total when its precondition is true for all possible values of the state. If a specification does not provide total operations then the reaction of the eventual implementation outside of the operation’s precondition will be undefined. This fact can be advantageous, remembering Z is used for abstract specification, when the specifier does not care to define the behaviour in some situations.

1.3 Z/EVES

Z/EVES [12] is a type checking and theorem proving tool for Z specifications. It is an interactive system in which Z specifications can be presented one paragraph at a time or in their entirety. Theorems can be defined and proofs attempted at any time. Z/EVES was developed by ORA [9] and is used here to prove properties about the Z translation of $\mu$-charts, and hence to prove properties of $\mu$-charts.

To give an example of using Z/EVES we evaluate the precondition of the schema $T(Op)$, from Section 1.2, to show that this operation is total when applied to any valid observation of the state, i.e. to prove that the predicate $\text{State} \Rightarrow \text{pre } T(Op)$ is true:
try State ⇒ pre T.Op;
invoke;
instantiate \( x' \leftarrow x + 1 \);
prove by reduce;
\( \rightarrow \) true

The try command (first line of input) allows the user to evaluate predicates about the current specification. In this case the predicate says that assuming the constraints on this state, the precondition (calculated by the Z prefix operator pre) of the operation schema \( T.Op \) is always true, i.e. the operation can be applied to any valid configuration of the state. The Z/EVES commands on lines 2-4 of the input instruct Z/EVES to carry out some evaluation on the given predicate, which is true as expected.

2 The Translation: An Example

In this section we concentrate on giving a series of examples which highlight the main features of our translation procedure which turns \( \mu \)-charts into Z. The presentation of the procedure in detail (and its formal justification) will be given in a later, fuller paper.

The \( \mu \)-chart given in Figure 1 was chosen as the simplest example that demonstrates the following concepts:

– several sequential automata;
– more than one level of hierarchical decomposition;
– instantaneous feedback of signals.

Some of the \( \mu \)-chart conventions used in Figure 1 include: states with a double lined border represent the default or start state for a \( \mu \)-chart, that is the state which is entered whenever the chart is entered; any state with a rectangular

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{mu-chart.png}
\caption{A simple \( \mu \)-chart}
\end{figure}


border is a decomposed state, that is it contains another \( \mu \)-chart, referred to as a slave, which is active (can make transitions) while the “master” chart is in the decomposed state; the labels on the transitions are made up of a trigger and output condition separated by a ‘/’. The trigger is a Boolean expression depending on presence (and/or absence) of signals and the signals in the output set are those which are output if this transition happens. If the output set is absent then the transition has no output associated with it (we also omit the ‘/’ in this case); finally the small rectangular box hanging from the bottom of the chart contains the set of signals that are to be instantaneously fed back as input to all \( \mu \)-charts and sub-charts within the chart to which the box is attached. The way in which this mechanism works will become clearer as we work through the examples below.

The Z translation is presented by giving a Z description (as a state schema) of each state in the example \( \mu \)-chart followed by a description of each transition as an operation schema. The correct combination of these transitions then describes the overall behaviour of the \( \mu \)-chart. We present this example in its entirety, despite some aspects appearing trivial, to allow the reader the greatest chance to follow the translation process. Note also that since we want a uniform process, i.e. one that applies to all charts in the same way, there may be pieces of Z produced that the accomplished Z-reader will consider clumsy or naive. However, again for reasons of clarity of process, we will resist the temptation to simplify and present the Z just as the process produces it.

Lastly we investigate the resulting Z specification using the tool Z/EVES.

### 2.1 Describing the States

The first task in the translation is to give a Z state schema, by convention of the same name, for each of the states in the \( \mu \)-chart. This includes those states that represent slaves (sub-\( \mu \)-charts) and those that represent atomic states. We first give values that will be used to name states and signals:

\[
\mu_{\text{State}} ::= a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \\
\text{Signal} ::= Sa \mid Sb \mid Sc \mid Sd \mid Sc \mid Sf \mid Sp
\]

Then we need a Z state which represents the whole chart together with its initialisation to the correct \( \mu \)-chart state:

\[
\begin{align*}
\text{Init}_U &::= u \\
\epsilon_U : \mu_{\text{State}} &::= a \\
U &::= u \\
\epsilon_U &= a
\end{align*}
\]

The next states to be described, \( A \) and \( H \), are atomic states and therefore are trivially described by the schemas \( A \) and \( H \): \(^2\)

\(^2\) Note that in the sequel we shall usually omit explicit mention of such outermost charts as \( U \) since their formalisation contributes nothing to our translation.
More interestingly, $B$ is a slave of $U$. Therefore, it introduces an observation of the slave $\mu$-chart’s state as well as defining what it means for $U$ to be in the state $B$, i.e. $c_U = b$. Each schema that models a $\mu$-chart (as opposed to an atomic state) also has an initialisation operation defined that specifies what state the slave is initialised to when it becomes active, that is when the master enters the corresponding state:

Now the three schemas $C$, $D$ and $E$ are given describing the states of the $\mu$-chart $B$. First, for the atomic states $C$ and $D$:

Notice that we don’t include the schema $B$ in $C$ or $D$ as may seem intuitive. If $B$ were included in $D$ then $\mu$-chart $U$ would be forced to be in state $B$ after any transition in the $\mu$-chart $B$ to state $D$—but this then blocks modelling in $\mu$-charts what are (semantically better behaved) counterparts for inter-level transitions in Statecharts. We want to allow such better behaved transitions because it allows modular design of $\mu$-charts and hence a more natural and tractable modelling of physical systems.

Schema $E$ models a $\mu$-chart and therefore has an initialisation schema:

Now schemas $F$ and $G$ are given for the two atomic states of $\mu$-chart $E$:

---

\(^3\) or—the $\mu$-chart $B$ is a slave of $U$. This view of $B$ as both a state and a $\mu$-chart (and indeed a schema) needs to be kept in mind, and when it is not clear from the context which we mean we make it explicit.

\(^4\) Note $\mu$-charts can allow two or more transitions to be triggered instantaneously if they are in different charts, including sub-charts.
Given all of the state descriptions we can now go on to describe the transitions between these states.

### 2.2 Describing the Transitions

One of the properties of $\mu$-charts that must be modelled is instantaneous feedback. The semantics of $\mu$-charts, as given in [10], uses a fixed-point construction to calculate, in an operational fashion, the overall input for a $\mu$-chart step. In the Z translation we use the fact that only one transition can happen per $\mu$-chart (in any hierarchy, i.e. one in $U$, one in $B$ and one in $E$) per step, and the nature of the schema calculus (the effects of schema inclusions), to give a declarative description of the possible inputs for a step.

The feed-back signals that are locally visible to each $\mu$-chart in the specification are given by the sets $l_U$, $l_B$ and $l_E$ defined below. The local sets can be read from any attached feed-back boxes in the graphical notation. The sets $f_U$, $f_B$ and $f_E$ determine the scope (due to the hierarchy) of feed-back signals in the $\mu$-chart and therefore give all the visible feed-back signals for each $\mu$-chart:

- $l_U = \{Sb, Sp\}$
- $l_B = \{\}$
- $l_E = \{\}$
- $f_U = l_U$
- $f_B = f_U \cup l_B$
- $f_E = f_B \cup l_E$

In this simple example the feed-back signals are common across all of the $\mu$-charts.

A schema, $Output$, that describes the allowable output information for a $\mu$-chart is also given:

- $o_U : \mathbb{P}\{Sb, Sp\}$
- $o_B : \mathbb{P}\{Sb, Sp\}$
- $o_E : \mathbb{P}\{Sp\}$
- $o! : \mathbb{P}\{Sb, Sp\}$

$$o! = \bigcup\{o_U, o_B, o_E\}$$

The schema $Output$ has an observation of the output signals generated by each $\mu$-chart in the specification. Each of these observations represents the set (possibly empty) of output signals from the given $\mu$-chart in a step and the union of these sets gives the overall output. The intersection of the output and the feed-back set gives the feed-back signals visible for the current step.
Now, with the states of the $\mu$-chart and the output/feed-back mechanism in place, an operation schema is given for each transition. The description of the transitions is organised to follow the hierarchy of the example $\mu$-chart.

**Transitions in $\mu$-Chart $U$:** Firstly an operation schema, $\delta_{ab}$, is given describing the transition between states $A$ and $B$ in $\mu$-chart $U$:

<table>
<thead>
<tr>
<th>$\delta_{ab}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
</tr>
<tr>
<td>$\text{Init}B'$</td>
</tr>
<tr>
<td>$\text{Output}$</td>
</tr>
<tr>
<td>$\text{input?} : \mathbb{P} \text{Signal}$</td>
</tr>
<tr>
<td>$Sa \in \text{input?} \cup (o! \cap f_U)$</td>
</tr>
<tr>
<td>$o_U = {}$</td>
</tr>
</tbody>
</table>

Due to the schema inclusions here, the precondition of $\delta_{ab}$ states that the current state of $U$ must be $A$, i.e. $c_U = a$, the overall output set is equal to the union of $o_U$, $o_B$ and $o_E$ and the signal $Sa$ must be an external input or a feedback signal for this transition to happen. Since the transition goes into a state which is also a slave $\mu$-chart the schema included has to be the initialisation schema for that slave.

Note the precondition of this transition is not yet determinable because the values of $o_B$ and $o_E$ are not known: it establishes a constraint on the allowable solutions. $\delta_{ab}$ stipulates that if the precondition is true then the $\mu$-chart $U$ will be in state $B$ after this transition and the $\mu$-chart $B$ will be in state $C$.

Similarly, the transition between states $B$ and $H$ is given by $\delta_{bh}$:

<table>
<thead>
<tr>
<th>$\delta_{bh}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
</tr>
<tr>
<td>$H'$</td>
</tr>
<tr>
<td>$\text{Output}$</td>
</tr>
<tr>
<td>$\text{input?} : \mathbb{P} \text{Signal}$</td>
</tr>
<tr>
<td>$Sb \in \text{input?} \cup (o! \cap f_U)$</td>
</tr>
<tr>
<td>$\lor Sp \in \text{input?} \cup (o! \cap f_U)$</td>
</tr>
<tr>
<td>$o_U = {}$</td>
</tr>
</tbody>
</table>

The precondition allows this transition only if the signals $Sb$ or $Sp$ are present in the input or feedback.

The $\mu$-chart semantics state that in the situation where no transition is triggered then the $\mu$-chart stays in its current state. To model this in Z we provide another operation schema. For the $\mu$-chart $U$ this schema is $\epsilon_U$:

5 Recall: $\text{Signal}$ is a set of all external, internal (i.e. feedback) and output signals that are available in the system being modelled.
The precondition of this schema states that no valid transition in the \( \mu \)-chart \( U \) is triggered. The postcondition states that the \( \mu \)-chart configuration remains unchanged. The output, given that the chart makes no transition, must clearly be empty.

There is no predicate involving \( H \) here (corresponding to the first and second involving \( A \) and \( B \) respectively) since if we are in state \( H \) then nothing happens unconditionally, i.e. whatever the input—so the condition for nothing happening is, vacuously, true. Contrast with the case for \( A \)—this predicate says that if we are in state \( A \) then the condition for nothing happening is that \( Sa \) is not in the input or feed-back.

**Transitions in \( \mu \)-Chart \( B \):** The transition descriptions for \( \mu \)-chart \( B \) are given by \( \delta_{cd}, \delta_{dd} \) and \( \delta_{de} \):

\[
\begin{align*}
\delta_{cd} & \\
B & \\
C & \\
D & \\
D' & \\
\text{input? : } & \mathbb{P} \text{ Signal} \\
\text{Output} & \\
Se \in & \text{input? } \cup (o! \cap f_B) \\
o_B = & \{\} \\
\end{align*}
\]

\[
\begin{align*}
\delta_{dd} & \\
B & \\
D & \\
D' & \\
\text{input? : } & \mathbb{P} \text{ Signal} \\
\text{Output} & \\
Se \in & \text{input? } \cup (o! \cap f_B) \\
o_B = & \{Sb\} \\
\end{align*}
\]

\[
\begin{align*}
\delta_{de} & \\
B & \\
D & \\
\text{InitE'} & \\
\text{input? : } & \mathbb{P} \text{ Signal} \\
\text{Output} & \\
Sd \in & \text{input? } \cup (o! \cap f_B) \\
o_B = & \{\} \\
\end{align*}
\]

Notice that the preconditions of these operations say that the transitions can only happen if the current \( \mu \)-chart is in the right state, e.g. \( c_B = c \) for \( \delta_{cd} \), and
any master $\mu$-chart is in the expected state or in other words the current $\mu$-chart is active, e.g. $c_U = b$ for $\delta_{cd}$.

As important (as we have said before) is the fact that the operation restricts only the after transition state of the $\mu$-chart in which the transition occurs and not its master’s state.

Next we give the $\epsilon$ schema and another auxiliary schema, $Inactive_B$, for the $\mu$-chart $B$:

$$\epsilon_B$$

$\epsilon_U, \epsilon_B, \epsilon_B' : \mu_{State}$

$Input? : \mathbb{P} Signal$

$Output$

$B$

$\neg (C \land Sc \in Input? \cup (\sigma \cap f_B))$

$\neg (D \land Sd \in Input? \cup (\sigma \cap f_B))$

$\neg (D \land Sc \in Input? \cup (\sigma \cap f_B))$

$o_B = \{}$

$c_B' = c_B$

$$Inactive_B$$

$\epsilon_U, \epsilon_B, \epsilon_B' : \mu_{State}$

$Output$

$\neg B$

$o_B = \{}$

$c_B' = c$

The operation schema $Inactive_B$ defines the action of $B$ when it is not active, i.e. its master, $U$, is not in state $B^6$. An inactive $\mu$-chart is to remain in its initial state and not react to any signals.

**Transitions in $\mu$-Chart $E$:** The transition descriptions for $\mu$-chart $E$ results in three schemas, $\delta_{fg}$, $\epsilon_E$ and $Inactive_E$:

$$\delta_{fg}$$

$B$

$E$

$F$

$G'$

$Input? : \mathbb{P} Signal$

$Output$

$Sf \in Input? \cup (\sigma \cap f_E)$

$o_E = \{Sp\}$

$\epsilon_E$.$\epsilon_E, \epsilon_E, \epsilon_E' : \mu_{State}$

$Output$

$B$

$E$

$\neg (F \land Sf \in Input? \cup (\sigma \cap f_E))$

$o_E = \{}$

$c_E' = c_E$

$$Inactive_E$$

$\epsilon_U, \epsilon_B, \epsilon_E, \epsilon_E' : \mu_{State}$

$Output$

$\neg B \lor \neg E$

$o_E = \{}$

$c_E' = f$

$^6$ There is no corresponding $Inactive_U$ since $U$ is not a sub-chart of any chart and so will never be inactive.
Notice, because a $\mu$-chart is inactive if its master is inactive, the schema $Inactive_E$ has the predicate $\neg B \lor \neg E$ that says the chart is considered inactive if either its master is not active, i.e. $\neg B$, or it is itself inactive, i.e. $\neg E$.

2.3 Describing the Step

Now all of the schemas needed to describe the step behaviour of the $\mu$-chart have been provided. The predicate of the step-modelling schema $Step$ conjoins the possible transitions for each $\mu$-chart and hides each $\mu$-chart’s component output observation. The schema $Step$ is a set of bindings that describes a transition function for the $\mu$-chart with respect to input:

$$
\exists o_U, o_B, o_E : \mathbb{P Signal} \bullet
\begin{align*}
&\delta_{ab} \lor \delta_{bh} \lor \epsilon_U \land \\
&\delta_{cd} \lor \delta_{de} \lor \delta_{dd} \lor \epsilon_B \lor Inactive_B \land \\
&\delta_{fg} \lor \epsilon_E \lor Inactive_E)
\end{align*}
$$

2.4 Using a Theorem Prover to Investigate the Z

This section gives some examples of exploration of the resulting Z specification using the tool Z/EVES.

The first two examples examine the behaviour of the $\mu$-chart when it is in its initial state and the signal $Sa$ is either present or absent:

$$
\begin{align*}
&\text{try } Step[e_U := a, e_B := c, e_E := f, input? := \{Sa\}]; \\
&\quad \rightarrow e'_U = b \land e'_B = c \land e'_E = f \land o! = \{} \\
&\text{try } Step[e_U := a, e_B := c, e_E := f, input? := \{Sp\}]; \\
&\quad \rightarrow e'_U = a \land e'_B = c \land e'_E = f \land o! = \{} \\
\end{align*}
$$

Both of these proofs result in the desired outcome. This method could be used in the same way to test all transition behaviours. For example the more interesting configuration of the $\mu$-chart where $U$ is in state $B$, $B$ is in state $D$, $E$ is inactive and $Se$ is input, can be checked as follows:

$$
\begin{align*}
&\text{try } Step[e_U := b, e_B := d, e_E := f, input? := \{Se\}]; \\
&\quad \rightarrow e'_U = b \land e'_B = d \land e'_E = f \land o! = \{Sh\}
\end{align*}
$$

Other ways of examining the $\mu$-chart include checking that there is some input signal set and configuration that allows a transition ending in another configuration valid of the system. Or, as valuably, no input and configuration exist that could result in a transition ending in an invalid configuration.
The first example gives the expected post-transition configuration and output and quantifies over the pre-configuration and input:

\[
\text{try } \exists \text{ input}? : \mathbb{P} \text{ Signal}; c_U, c_B, c_E : \mu_{\text{State}} \bullet \\
\text{Step}[c'_U := h, c'_B := c, c'_E := f, o! := \{\}]; \\
\rightarrow \text{true}
\]

That this is true can be interpreted to mean there do exist some configuration and some input that would allow a transition to the given configuration with the expected output, i.e. it allows us to test reachability.

The next example gives a contradictory configuration and resulting output, i.e. we expect that the system can never make a transition ending in this configuration with this output. The expected result is therefore false:

\[
\text{try } \exists \text{ input}? : \mathbb{P} \text{ Signal} \bullet \text{Step}[c'_U := h, c'_B := c, c'_E := f, o! := \{\text{Sp}\}]; \\
\rightarrow \text{false}
\]

A variation of this type of exploration is obtained by not quantifying over the starting configuration or giving the expected output. The resulting predicate simplifies to constraints on these observations rather than true or false. It is worth noting that as less fixed information is provided the more complicated the proof and resulting predicate become, e.g. there are likely to be numerous combinations of possible configurations and outputs on which a transition can give the expected configuration:

\[
\text{try } \exists \text{ input}? : \mathbb{P} \text{ Signal} \bullet \text{Step}[c'_U := h, c'_B := e, c'_E := g]; \\
\rightarrow c_U = b \land c_B = e \land c_E = g \land o! = \{\} \\
\lor c_U = b \land c_B = e \land c_E = f \land o! = \{\text{Sp}\}
\]

Other important properties of the specification may include the inability of the $\mu$-chart to get from one state to another in one step. For example we can prove that there is no input that will take this $\mu$-chart from state $A$ to state $H$ in one step:

\[
\text{try } \exists \text{ input}? : \mathbb{P} \text{ Signal} \bullet \text{Step}[c_U := a, c'_U := h]; \\
\rightarrow \text{false}
\]

We can also compose steps together to more rigorously check reachability of configurations. Due to the grammar used by Z/EVES \(^7\) this requires some definition of temporary schemas such as $\text{Step}_2$:

\(^7\) Z/EVES follows the grammar of Spivey [18] which does not class schema expressions as expressions simpliciter, hence the definition we require cannot even be expressed (though see the Acknowledgements). The grammar in the proposed standard [19] combines the two categories, so once Z/EVES supports the grammar in the new standard we would be able to straightforwardly define the function we desire.
Fig. 2. Example 1: Causality Problems

\[
\begin{align*}
\text{Step}_2 &\triangleq \text{Step}[i1/\text{input}? , o1/o!] \triangleright \text{Step}[i2/\text{input}? , o2/o!] \\
\text{try } \exists i_1, i_2 : \text{P Signal}; & \quad c_U, c_B, c_E : \mu_{\text{State}} \bullet \text{Step}_2[c'_U := h, c'_B := e, c'_E := g]; \\
& \quad \rightarrow o1 = \{\} \land o2 = \{Sp\}
\end{align*}
\]

More interestingly the two cases below check the reachability of a given configuration from the initial configuration of the \(\mu\)-chart. We need to again define temporary schemas that give compositions of Step with itself three times:

\[
\begin{align*}
\text{Step}_3 &\triangleq \text{Step}_2 \triangleright \text{Step}[i3/\text{input}? , o3/o!] \\
\text{try } \exists i_1, i_2, i_3, & \quad o1, o2, o3 : \text{P Signal}; \quad \bullet \text{Step}_3[c'_U := a, c'_B := e, c'_E := f]; \\
& \quad \rightarrow \text{true} \\
\text{try } \exists i_1, i_2, i_3, & \quad o1, o2, o3 : \text{P Signal}; \quad \bullet \text{Step}_3[c'_U := a, c'_B := e, c'_E := f]; \\
& \quad \rightarrow \text{true}
\end{align*}
\]

Given the above examples it is easily seen how we could invent different combinations to investigate more properties of the system. For example we might need to check there is at least one possible transition out of any valid configuration, i.e. the system is always ready to react.

3 Avoiding the Pitfalls: Getting the Right Semantics

This section gives some pathological examples taken from [10] which are given to show that the Z translation respects the semantics given for \(\mu\)-charts.

3.1 Causality Problems

The first example is given by the \(\mu\)-chart in Figure 2.

The resulting Z from the translation is:

\[
\begin{align*}
A \quad &\quad c_U : \mu_{\text{State}} \\
&\quad c_U = a \\
B \quad &\quad c_U : \mu_{\text{State}} \\
&\quad c_U = b
\end{align*}
\]
The schema $\delta_{ab}$ above demonstrates the first translation of a negated trigger expression. Negated trigger expressions are translated by a predicate, i.e. $Sa \notin \text{input} \cup (o! \cap f_U)$, that says the signal does not appear in the input (including feed-back). This example (and in fact the next example) demonstrates the causality problems that can be introduced by negated trigger expressions. For this example, when the chart is in its initial state, there is no solution when the signal $Sa$ is not input, and in this case the predicate of $\text{Step}$:

$$
\begin{align*}
\text{Step} & \\
\epsilon_U, \epsilon_U' : \mu_{\text{State}} & \\
\text{input}?, o! : \mathbb{P}\text{Signal} & \\
\delta_{ab} \lor \epsilon_U &
\end{align*}
$$

is false.

The predicate of the schema $\delta_{ab}$ is false in all cases which, as we would expect given the semantics for this $\mu$-chart, means this transition can never occur. If the signal $Sa$ is input the trigger condition on the transition is false causing the $\mu$-chart to stay in the same state and (the operation) $\epsilon_U$ to take place. If, when the chart is in state $A$, $Sa$ is not input then no transition is valid and the predicate of $\text{Step}$ is false. The semantics for $\mu$-charts given in [10] states that in the event of empty reactions (which means that the schema $\text{Step}$ is an empty set of bindings in the translated $Z$) the $\mu$-chart remains in its current configuration. Hence in both cases, $Sa$ present and $Sa$ absent, the $Z$ model constrains the after state of the transition to be $A$, i.e. the configuration of the $\mu$-chart remains the

---

8 The semantics given in [10] introduces the idea of oracle signals, which can be used to rule out solutions with such causality problems. See Section 4 for a fuller discussion of this.
same. That this accords with the definition in [10] is evidence of the correctness of our translation.

This can be easily illustrated using Z/EVES as follows:

\[
\text{try Step}\{\epsilon_U := a, \text{input}?: = \{\{Sa\}\}; \\
\quad \rightarrow \epsilon_U' = a \land \epsilon! = \{\}\} \\
\text{try Step}\{\epsilon_U := a, \text{input}?: = \{\}\}; \\
\quad \rightarrow \text{false}
\]

### 3.2 Causality Problems Continued

A more interesting example is that of Figure 3.

The behaviour of this $\mu$-chart is non-deterministic when neither $Sa$ nor $Sb$ is input. The correct configuration of the $\mu$-chart after a step with no input is either $U$ in state $A$ and $V$ in state $D$ or $U$ in $B$ and $V$ in $C$.

The translation gives:

\[
\begin{align*}
A & \quad B & \quad C \\
\epsilon_U : \mu_{\text{State}} & \quad \epsilon_U : \mu_{\text{State}} & \quad \epsilon_V : \mu_{\text{State}} \\
\epsilon_U = a & \quad \epsilon_U = b & \quad \epsilon_V = c \\
\end{align*}
\]

\[
\begin{align*}
D & & \quad l_U = \{Sa, Sb\} \\
\epsilon_V : \mu_{\text{State}} & & \quad f_U = l_U \\
\epsilon_V = d & & \quad l_V = \{Sa, Sb\} \\
\end{align*}
\]

\[
\begin{align*}
\text{Output} & \quad \delta_{ab} \\
o_U : \mathbb{P}\{Sb\} & \quad A \\
o_V : \mathbb{P}\{Sa\} & \quad B' \\
o! : \mathbb{P}\text{Signal} & \quad \text{input}?: \mathbb{P}\text{Signal} \\
o! = \bigcup\{o_U, o_V\} & \quad \text{Output} \\
& \quad Sa \notin \text{input}? \cup (o! \cap f_U) \\
& \quad o_U = \{Sb\}
\end{align*}
\]
Four Z/EVES tests are given for the translation to show that the behaviour of the Z respects the μ-chart meaning:

\[
\begin{align*}
\text{try Step}[c_U := a, c_V := c, \text{input?} := \{Sa\}] & ; \\
\quad & \rightarrow c'_U = a \land c'_V = d \land o! = \{Sa\} \\
\text{try Step}[c_U := a, c_V := c, \text{input?} := \{Sb\}] & ; \\
\quad & \rightarrow c'_U = b \land c'_V = c \land o! = \{Sb\} \\
\text{try Step}[c_U := a, c_V := c, \text{input?} := \{\}\] & ; \\
\quad & \rightarrow \text{if } c'_U = a \text{ then } c'_V = d \land o! = \{Sa\} \\
\quad & \text{else } c'_V = b \land c'_V = c \land o! = \{Sb\} \\
\text{try Step}[c_U := a, c_V := c, \text{input?} := \{Sa, Sb\}] & ; \\
\quad & \rightarrow c'_U = a \land c'_V = c \land o! = \{\}
\end{align*}
\]

This example also demonstrates how the translated Z deals with non-determinism: it is non-deterministic when neither of the signals Sa or Sb are input. The resulting Step schema allows one behaviour or the other. It is not possible for both to occur and Step makes no decision about which should occur.

4 Oracles

In [10] the authors introduce the idea of oracles in order to rule out certain unwanted solutions that their fixed-point construction would otherwise give. The unwanted solutions are those which lead to causality conflicts arising from the presence of negated triggers i.e. causality conflicts which arise when we allow the absence of signals to be acted on.

We give an example of how the oracle mechanism is used to rule out certain solutions. We then relate those solutions back to the ones we get in our Z model.
Fig. 4. Illustrating oracle use

for the same example in order to show one more aspect of the relationship between the two models.

4.1 Oracles and Causality

The example we will use is a simple variant of example 2 (which was given in Figure 3) where we take the negation off the trigger for the transition in $V$. (This also gives us an example of a case we have not yet explored, i.e. one in which both negated and un-negated triggers occur.)

The method that [10] uses to rule out certain solutions that violate some causality concerns is to first rename any negated trigger $\neg a$ as $\neg \tilde{a}$ and then conduct experiments to see what the reaction of the chart is when we make assumptions about feed-back involving these new signals, which are called oracles. So, the oracle signals represent signals that may be fed back during any of a number of instantaneous transitions and they were designed so that we can consider the reactions of the chart when signals are absent and when there are negated triggers in the chart. (We have seen examples of this in Sections 3.1 and 3.2.) The chart we are considering would, therefore, be re-written as in Figure 4.

Now, if we want to see what its reactions are when no (external) input is given there are two experiments to perform: firstly we assume that $\tilde{S}a$ is in the set of signals generated by the chart in this step (the fixed-point for this step), i.e. an experiment with $\tilde{S}a$ being in the fixed-point; secondly we assume that $\tilde{S}a$ is never output for any reason (and so does not appear in the fixed-point) which means we assume that $\neg \tilde{S}a$ is in the fixed-point. The reader can see that the resulting fixed point of the first experiment is $\{\tilde{S}a\}$ and for the second the fixed-point is $\{\neg \tilde{S}a, \tilde{S}b, S\alpha\}$.

The result of the first experiment is not allowed as a possible behaviour of the chart since when we assumed that $\tilde{S}a$ would appear in the fixed-point it never did. The result of the second experiment is not allowed as a solution either since a signal ($\tilde{S}a$) that we assumed would not appear in the fixed-point did. So, neither experimental solution is allowed since they each break certain causality constraints that it seems reasonable to impose.

The first constraint requires that if a signal can appear (because transitions that produce it as output can take place) then it does: this is self-fulfilment. The
second constraint requires that a signal does not appear as the output of any transition that takes place because of the non-appearance of that signal: this is consistency.

So, how is it that we disallow the same causally suspect solutions that the oracles mechanism does, but without using anything like that mechanism? The reason lies in the different ways the semantics are given. In our case, we rely on the usual semantics of set theory and give the semantics in a declarative fashion. In the case of [10] the semantics is given by a fixed-point construction, where each ‘iteration’ towards the fixed-point models the idea of one transition possibly leading to others once its output has been instantaneously fed back as input, this process continuing until no further transitions are triggered, when the fixed-point is reached.

In more detail: when computing the fixed-point, the absence of a signal has to be recorded in order for the causality test to be meaningfully performed. Otherwise, the reason that at one stage the signal is not in the feedback set (the so-far-computed fixed-point) might be either because it simply hasn’t appeared yet or because it must not appear. When we reach the fixed-point we need (for the causality test) to tell these two situations apart. Hence the oracles: \( \neg \tilde{a} \) appearing means that \( a \) must never appear; \( \neg \tilde{a} \) not appearing entails no such condition. 9

In our case, to get the same causality condition, we do not need the oracles. This is because our translation imposes global constraints on the possible solutions; constraints which are ‘in force’ permanently. That is, we declare what constraints a solution must satisfy and then rely on the semantics of Z to ensure that only acceptable sets are allowed as solutions, according to the usual meaning of the existential quantifier that appears outermost in our translation. Thus, if some transition is triggered only when \( a \) is absent, but some other transition adds \( a \) to the solution set then, since not both of these conditions can be satisfied, such a solution is ruled out. In the oracles case, since the solution set is essentially built iteratively (gradually computing the fixed-point) and since no constraints are passed between the iterations, the ‘global’ fact that \( a \) has been banned by the triggering of one transition must be recorded in order that the causality condition can be checked; simply leaving \( a \) out of the set will not do as it does not distinguish finely enough between two possibilities.

4.2 Extending the Use of Oracles

The semantics given in [15] differs, as we have mentioned a few times, from that in [10]. One way that this difference emerges is on the \( \mu \)-chart given in Figure 5.

In the absence of any external input, the semantics in [10] means that this chart does nothing. In contrast, the semantics in [15] means that chart is non-deterministic: it can either do nothing or both transitions can be triggered. Our translation process produces a description of the chart which agrees with the latter interpretation.

9 This relationship between declarative and procedural semantics closely mirrors the two ways of giving semantics to Prolog programs.
When we investigate the reasons for this we see that our constraints on solutions in the translation correspond to replacing not only negated triggers with negated oracle signals but also doing the same with un-negated trigger signals. Our conjecture (which we have shown holds for the examples in this paper) is that extending the use of oracles in this way would, in general, extend the solutions given by the semantics in [10] to those given by the semantics in [15] while still ruling out the solutions that offend certain sorts of causality scruples.

5 Uses

Having presented a translation of \(\mu\)-charts into Z we want in this section to motivate this activity, which we do by giving several positive results of such an activity. One overall result is that the translation has opened-up and made possible all sorts of links and comparisons with work on Z and, increasingly, work in the wider area of program development.

5.1 Alternative Views of the Solution

Giving two different views, via the different languages of \(\mu\)-charts and Z, of a solution to a problem can often give us another chance to see if we have the right model of a given system. Different aspects of the model may come to light, given the different expressions in the different languages, which might lead us to see mistakes in the model.

Some properties of the system being modelled might be more readily apparent in the Z view than in the chart view (and vice versa, of course).

5.2 Proof Instead of Model Checking

We have proof available for Z (via, for example, Z/EVES) which means that, via the translation, we can reason about the reactive model of our system. This forms an alternative to the sorts of model checking that staying with the \(\mu\)-chart model would give.

The importance of proof (or deduction) in contrast to model checking has recently been explored by Pnueli [11], where he makes the following points:
– deduction is based on induction whereas model checking (which he characterises as exploration) is based on computing a set of reachable states;
– deduction uses a more expressive language, leading to good ways of expressing, especially, parameterised systems;
– deduction-based methods have better scalability, e.g. as a system increases in size we may be able to rely on the same (or same size of) inductive proof as for the original, smaller system—so the amount of work does not increase.

Whatever the pros and cons of each method (and whether we accept Pnueli’s points or not), it is certainly the case that having both model-checking-based and deduction-based methods at our disposal to investigate systems is advantageous. This would clearly be the case when we come to proving invariants of systems.

5.3 Alternative Solutions to the Problem
Following Weber’s approach in [20], of specifying the problem from a Z perspective and a Statechart (for us μ-chart) perspective (and so having two different models of the problem, in contrast with the situation in Section 5.1 where we had two expressions of the same model), and then performing our translation on the μ-chart model, would allow us to directly compare the two models. We could also more directly check the consistency of the two models if they are both in Z.

6 Future Work

6.1 Commands
In line with the work in [15] and [16] we will add commands to the transitions. The idea of this is to allow the updating of values associated with the chart. For example, if the chart takes a transition labelled (along with the usual trigger and output signal set) \( a := a + 1 \) then the value of \( a \) is incremented.

The triggers can also be extended to allow Boolean combinations of expressions involving such values as \( a \), as well as the input signals as currently.

These extensions allow us to model such things as clocks (where the current time, for example, is represented by some value which is incremented by one sub-chart and used by others) or mechanisms which have to wait for certain external values to reach required levels before performing their tasks. Again, the external values (say the level in a tank of liquid) can be updated and used by sub-charts.

6.2 Correctness of the Translation
In this paper we have endeavoured to illustrate the translation process, show some of its properties and mention some motivation for it. It still remains for us to show that the translation is correct by showing that, for each construct of the charts, its semantics is the same as the semantics we get going from that chart construct via the translation to Z and the Z semantics. Given that we have a
compositional semantics for both formalisms this strategy seems to hold some hope, especially in the light of the transition semantics given in [15]. This is in a form that is closer to our Z form and which has already been shown to be equivalent to the original semantics in [15].

6.3 Refinement

[14] and [15] present a refinement calculus for $\mu$-charts based on a relationship of inclusion, i.e. (roughly) $S_2$ refines $S_1$ is defined by $\llbracket S_2 \rrbracket \subseteq \llbracket S_1 \rrbracket$. This is also the situation with our notion of refinement in [3], so investigating the relationships between these two notions of refinement would be interesting. In our work schemas are sets of bindings which represent state-and-signal combinations and in the $\mu$-charts work they are sets of state-and-input/output signal tuples, so there are close similarities there.

Since we are dealing with models for reactive systems we also plan to investigate the problems and solutions proposed in [1], which relate to the introduction of internal transitions, as often happens during a refinement step.

6.4 A Logic of $\mu$-Charts

We will develop a logic for $\mu$-charts, along the line of our logics for Z [4] [5] [6]. We expect to use the Z translation to at least suggest how this logic should look, if not use the translation to induce the $\mu$-chart logic from the Z logic.

6.5 Developing Implementations

Using the work on program development from Z, as in [3], and the translation given here, we will investigate program development from reactive specifications as given by $\mu$-charts.

Acknowledgements. Our thanks to Mark Saaltink, who has suggested to us an ingenious, though necessarily complicated, schema iterator which does the job we mentioned at the end of Section 2.4 [13].

Thanks also to Peter Scholz, for answering some of our basic questions about $\mu$-charts, and to the referees for doing such a thorough and helpful job. We especially thank the referee who suggested that it would be interesting to see whether the idea of our constraint semantics (as opposed to the original oracles semantics) generalises to other state machine semantics, e.g. Statecharts with their microsteps. Indeed it would, and that will form another future direction for our work.

Thanks finally to our colleagues on the ISuRF project [7]: Lindsay Groves, Ray Nickson and especially Mark Utting, and to the Foundation for Research, Science and Technology (FRST) for grant UOW805 and to the University of Waikato, which each provided funding which made this work possible.
References

9. The ORA web site is http://www.ora.on.ca.
Combining Operational Semantics, Logic Programming and Literate Programming in the Specification and Animation of the Verilog Hardware Description Language

Jonathan P. Bowen

South Bank University, Centre for Applied Formal Methods
School of Computing, Information Systems and Mathematics (SCISM)
Borough Road, London SE1 0AA, UK

Email: jonathan.bowen@sbu.ac.uk
URL: http://www.sbu.ac.uk/~bowenjp/

Abstract. An operational semantics for a significant subset of the Verilog Hardware Description Language (HDL) has been developed. An unusual aspect of the semantics is that it was formulated as a Prolog logic program. This allows the possibility of simulating the semantics. In addition, a literate programming style has been used, so the semantics can be processed by the LATEX document preparation system with minimal and fully automated preprocessing. Bringing together the paradigms of operational semantics, logic programming and literate programming in this manner has proved a great aid in a number of ways. It has helped improve the understanding of the semantics, in the formalization of semantic aspects left informal in the original mathematical formulation of the semantics, and in the maintenance of the formal semantics and its associated informal description.

Civilization advances by extending the number of important operations which we can perform without thinking about them.

Introduction to Mathematics (1911) ch. 5
Alfred North Whitehead (1861–1947)

1 Introduction

Traditional mathematics is written in a form designed for human assimilation together with pen and paper transformation. The most that could be hoped of a mathematical description in the past was that it might be typeset and printed (a time consuming process and of great expense). The advent of computers has meant that typesetting is now easy and cheap, even in the home. The ability to produce tools to aid transformation (either proof or calculational in style) is also possible, although this has proved difficult due to the pedantic and low-level nature of computers compared to abstract mathematical human

Work originally undertaken while on leave as a Visiting Research Fellow at the United Nations University International Institute for Software Technology (UNU/IIST), Macau.
thought. The presentation of proofs is problematic even for human readers [30]. As well as transformation, computers also allow a mathematical description to be “executed” in some cases [13]; of course, it is this with which the art of computer programming is chiefly concerned.

However, an important feature of a mathematical description, model or specification [6] is that it is not necessarily executable [17], even though some contend that it is helpful for it to be so [13]. An animation of a specification can help in understanding, just as formal reasoning can too, and in a complementary manner. Here we accept that a specification of a semantics is not normally directly executable. This is especially so if it is desirable to include non-deterministic aspects in the description; but if the non-determinism can be limited in a finite way, it is sometimes possible to model it successfully in a usefully executable manner.

Non-determinism is particularly helpful in the formulation of the meaning of parallel program threads since the exact ordering of interleaving of parallel execution of processes may not be known. This is important in the simulation of hardware since hardware systems are naturally parallel. Hardware may be thought of as an (often large) number of parallel and interconnected components, all constantly “executing” some (often simple) function.

In this paper we consider the combination of a mathematical description (semantics) and an executable program in a form that does not compromise either aspect more than is necessary. In addition, we use a style of documentation that allows this mathematical and executable description to be embedded and interleaved with an associated informal description. Having a single source for these three aspects greatly aids in their maintenance and consistency as all three are developed. The process of producing a formal specification is as important, if not more so, that the final specification itself in gaining understanding of the system involved [1].

1.1 Operational Semantics

There are a wide variety of semantics that have been developed to aid the mathematical description and specification of computer-based systems. These include denotational, algebraic and operational styles. Some effort has been made to demonstrate the relationship between these styles, especially with respect to the definition of different programming paradigms [24].

The style of operational semantics was originally formulated by Plotkin about two decades ago [34] and is still accepted as a useful semantic paradigm [33]. An operational semantics consists of a set of transition rules each of which may be applied if a condition is met. If more that one rule can be applied at a particular time, the specification is non-deterministic in that any of the applicable rules may be followed. The sequence of transitions may be terminated (normally indicated by a special program “$\epsilon$”).

The adoption of an operational approach is a natural one, especially for engineers used to the execution of an artifact in some form. The complete set of possible steps at any given time are considered. A potential disadvantage of this approach is that we may be overwhelmed with the mass of detail involved in the execution. However, it is possible to project just the steps of importance if necessary in any trace of the execution that may be of interest.
1.2 Logic Programming

Logic programming allows the specification of relations in a natural manner. With good judgement and an understanding of the procedural reading of logic programs, such a description can also be executed.

The Prolog language [11] is an early and still widely used general purpose logic programming language. It is an excellent rapid-prototyping language which can be applied in many areas, including digital hardware circuits [10]. In particular, it allows the execution of relations in a possibly non-deterministic manner, which can be especially useful in the modelling of parallel systems. Once the relations given in a specification are encoded in Prolog, the execution proceeds using a simple depth-first left to right search, depending on the order of the clauses used for the encoding. It is important to constrain any non-determinism in a finite way if termination is to be ensured.

Prolog includes a pseudo-clause called op(...) that allows the definition of infix, prefix and postfix operators with a specified precedence and also left or right associativity if required. This facility, although not extensively used by many Prolog programmers, allows a Prolog program to follow the form of an existing mathematical definition consisting of a set of relations in a fairly direct manner. The encoding of the relations themselves is typically of the same order of size as the original relations. Inevitably, most mathematical relations include constraints on numbers, sets, etc. Prolog provides simple integer arithmetic and support for lists that can be used to encode sets in a simple manner. Normally some extra clauses are required for encoding the constraints, but typically these can be of the same order of magnitude in size as the original relations themselves.

1.3 Literate Programming

The "literate programming" style of Knuth [28,29] is a useful aid to the maintenance of a formal description (be it a program or a more general mathematical formulation) as part of an informal document. Most programming languages expect input to be in the form of source program except where portions of the input file are explicitly marked as comments in some way.

In the literate style, the source file is considered as a document in its own right, containing portions of program (or mathematical description) only in explicitly marked sections. This allows documentation to be maintained in the same location (file) as the formal notation (e.g., as is normal using the Z notation [1,6]), aiding the goal of consistency between them during development.

Specifically, in the case of Prolog, text between /*...*/ is normally considered to be a comment. Instead we can view text between *...*/ as sections of Prolog program; the rest is the associated documentation. Here we choose the \TeX{} document preparation system [31] as the mechanism to maintain and print the document within which the mathematical description resides.

It is likely to be desirable to omit some support clauses required for the program from the final printed or displayed documentation. A convention of marking these clauses in a manner that is still recognized by the Prolog system as comment delimiters can be chosen (e.g., -*...*/). It may also be helpful to mark up some Prolog code within
comments as far as the Prolog system is concerned, but which should be included in formatted form in the documentation. Section delimiters such as */...*/ sections and */.../** sections to the $\texttt{begin}(...)...,\texttt{end}(...)$. Environment sections required by \LaTeX. The $-*/...*/$ sections need to be deleted or converted to \LaTeX comments by adding a ”%” to the start of all the lines in these parts of the file. The extra ”*/” at the start of the Prolog source file and ”*/*” at the end, required to satisfy the Prolog system, must be removed during the processing of the document. In addition, a number of the Prolog operators (functors) can be converted to more mathematical symbols (using \LaTeX macros) to aid readability.

For comparison, consider a small Prolog program fragment:

```
/*
par(S) :- seq(S).
par(S1//S2) :- seq(S1), par(S2).
*/
```

This is the form in which the semantics is maintained by the developer. The equivalent automatically generated \LaTeX document source for this fragment is the following rather unreadable sequence of instructions:

$$\begin{array}{lcl}
par(S) & \Leftarrow & seq(S). \\
par(S_1||S_2) & \Leftarrow & seq(S_1), \par(S_2).
\end{array}\$$

This ugly description need never normally be viewed. The matching formatted \LaTeX output is the following much more mathematical and readable form:

\[
\begin{array}{@{}l@{\Leftarrow}l@{}}
par(S) & seq(S). \\
par(S_1 || S_2) & seq(S_1), \par(S_2).
\end{array}\]

This is the description available for documentation purposes, intended to be read by humans.

### 1.4 Hardware Description Languages

Hardware Description Languages (HDLs) are used to specify electronic circuits, especially for digital systems. Two of the major HDLs in use are VHDL (Very High Speed Integrated Circuit (VHSIC) Hardware Description Language) and Verilog [14]. Both VHDL and Verilog have IEEE standards associated with them [25,26].

The formal semantics of VHDL has been studied quite extensively [8,12], but that of Verilog less so, even though VHDL is probably a more complex language than Verilog. Gordon has investigated the formal semantics of Verilog [15,16], but has considered a relatively small subset of the language. An OVI (Open Verilog International) \textit{Formal}
Verification Standards Working Group has been established with an aim to ensure interoperability among formal verification and other tools [35]. This is concentrating its effort on a synthesizable Verilog subset.

Combinational Verilog programs have been explored formally [37]. In addition, an operational semantics [34] for a fairly substantial subset of the language has been formulated [36]. Subsequently this semantics has been refined in a paper by He and Xu [20]. More recently, a Duration Calculus semantics for Verilog has been formulated [38].

An advantage of the availability of an operational semantics for a language is the increased understanding and the possibility of formal reasoning that this brings. A disadvantage is that the semantics cannot necessarily be directly animated. An operation semantics can allow non-determinism, which is advantageous in a specification, but potentially problematic in a simulator where one execution path must normally be selected.

In this paper, we consider a revised version of the mathematical operation semantics description in [20] that has been formulated using Prolog, with additional formalization of parts left informal in the original semantics. The semantics was developed using Knuth’s literate programming style as a single document that is available as a technical report containing the full semantic description [2]. Here we present parts of this description, concentrating especially on aspects omitted from the original mathematical description.

2 Verilog Operational Semantics

2.1 Dynamic Type-Checking

Prolog is essentially an untyped language. This is one of its powers in modelling mathematical relations, since much mathematics is used in a rather loosely typed manner, and is also one of its weaknesses, since type-checking can be used to help avoid errors in complex formal descriptions; hence its inclusion in most computer programming languages (used to produce formal descriptions that also have the convenient property that they are executable).

It is possible to use Prolog itself to do type-checking of structures dynamically in the form of assertions [23] during the Prolog program execution. Indeed, many type-checkers work in a similar way to Prolog interpreters, using resolution to infer types of expressions from other expressions in which the types are known.

In the case of Verilog, at the top level programs may be considered as a number of sequential programs running in a number of parallel threads:

\[
\text{par}(S) \leftarrow \text{seq}(S).
\]

\[
\text{par}(S_1 || S_2) \leftarrow \text{seq}(S_1), \text{par}(S_2).
\]

Note that we use the more mathematical reverse implication (\(\Leftarrow\)) instead of the Prolog “:-” in the clauses presented in this paper. “,” indicates conjunction within a clause and “;” indicates the end of a clause. Variables in Prolog are indicated by starting with a capital letter. We maintain this convention here, even if Greek letters such as \(\Sigma\) are sometimes used.
Sequential programs (in the subset of the Verilog language modelled here) consist of assignments, delays, programs guarded by events, standard conditional “if” statements and iterative “while” loops, and sequential composition of two or more sub-programs. Additionally, sequential programs may be terminated (indicated by the special program “ε”). We also allow macro definitions for sequential programs so that further derived constructs may be included (see later for some examples).

\[
\begin{align*}
\text{seq}(V = E) & \iff \text{variable}(V), \text{expr}(E). \\
\text{seq}(\#N) & \iff \text{integer}(N). \\
\text{seq}(G \cdot P) & \iff \text{guard}(G), \text{seq}(P). \\
\text{seq}(\text{if } EB \cdot P \text{ else } Q) & \iff \text{bool}(EB), \text{seq}(P), \text{seq}(Q). \\
\text{seq}(\text{while } EB \cdot P) & \iff \text{bool}(EB), \text{seq}(P). \\
\text{seq}(P; Q) & \iff \text{seq}(P), \text{seq}(Q). \\
\text{seq}(\epsilon) & \\
\text{seq}(X) & \iff (X \equiv P), \text{seq}(P).
\end{align*}
\]

The \text{variable}(...) clause to check for a variable name will be defined later. The built-in \text{integer}(...) clause is provided by Prolog systems as a check for an instantiated integer value. The \text{sans serif} font is used to indicate such standard built-in pseudo-clauses used in the rest of this paper.

Note also that the Prolog parser does not allow hidden infix operators. Hence the explicit used of “·” in some of the constructs above and later in this paper. Rather than coding a parser explicitly, we use the in-built parsing mechanism of Prolog. This means that we have to allow some deviation from the actual concrete syntax of Verilog, but it saves us the trouble of building a parser explicitly for the rapid-prototype. We simply define the Verilog program operators to be used with appropriate precedence, pre/in/post-fix, and associativity using Prolog’s \text{op}(...) pseudo-clause. If required, a separate pre-processor to take an actual Verilog program and convert it to a form acceptable directly by Prolog could be constructed.

Integer expressions follow the standard style in many programming languages. Here we allow integers, variables, addition and subtraction as an example:

\[
\begin{align*}
\text{expr}(N) & \iff \text{integer}(N). \\
\text{expr}(V) & \iff \text{variable}(V). \\
\text{expr}(E_1 + E_2) & \iff \text{expr}(E_1), \text{expr}(E_2). \\
\text{expr}(E_1 - E_2) & \iff \text{expr}(E_1), \text{expr}(E_2).
\end{align*}
\]

Boolean expressions are also required and we include some example operators below. Integer expressions are also allowed, with the assumption that non-zero values are considered “true”.
\texttt{bool(true).}
\texttt{bool(false).}
\texttt{bool(}\texttt{\!EB} \texttt{) \leftarrow bool(EB).}
\texttt{bool(E_1 = E_2) \leftarrow expr(E_1), expr(E_2).}
\texttt{bool(E_1 \neq E_2) \leftarrow expr(E_1), expr(E_2).}
\texttt{bool(E_1 > E_2) \leftarrow expr(E_1), expr(E_2).}
\texttt{bool(E_1 < E_2) \leftarrow expr(E_1), expr(E_2).}
\texttt{bool(EB) \leftarrow expr(EB).}

Event guards may involve a check for a change in a variable, or an explicit check for positive or negative edges (change) in a variable, or a disjunction of such events:

\texttt{guard(V) \leftarrow variable(V).}
\texttt{guard(posedge V) \leftarrow variable(V).}
\texttt{guard(negedge V) \leftarrow variable(V).}
\texttt{guard(G_1 \texttt{ or } G_2) \leftarrow guard(G_1), guard(G_2).}

The variable state is modelled as a table (set) of pairs of values consisting of a list the variable names V each associated with an integer value N:

\texttt{table(\langle\rangle).}
\texttt{table(\langle(\langle V = N \rangle | T)\rangle) \leftarrow variable(V), integer(N), table(T).}

We use \texttt{\langle...\rangle} for lists (normally written \texttt{[...]} in Prolog) for consistency with the operational semantics on which this is based [20]. Thus \texttt{\langle Head \texttt{ | } Tail\rangle} indicates a non-empty list with \texttt{Head} as the first element and \texttt{Tail} as the rest of the elements (if any). \texttt{\langle\rangle} denotes the empty list. Lists are normally used to model sets and other data structures in Prolog since they are conveniently built into the language.

The above definitions allow the type-checking of an instantiated system state for both sequential and parallel programs. The state for a sequential program consists of a tuple (modelled as a list in Prolog) with two components, the (sequential) program \texttt{S} to be executed and the state of the system variables \texttt{Σ}. The state for a parallel program includes two additional components, an extra state component for use in event detection \texttt{Σ_0} and an integer label \texttt{I} to indicate which parallel thread is being executed (zero for none) at any given time.

\texttt{seqtype((S, Σ)) \leftarrow seq(S), table(Σ).}
\texttt{partype((S, Σ, Σ_0, I)) \leftarrow par(S), table(Σ), table(Σ_0), integer(I).}

The \texttt{partype} clause can be used to dynamically check the state of the entire system at each step of execution, helping to avoid errors in the simulator.

2.2 Language Features

The meaning of each Verilog language feature is modelled as one or more transition rules, consisting of a condition under which the rule is valid (possibly \texttt{true} if no constraints apply), and then the transition itself. The transitions are defined use various sorts of arrow
relations, each defined on a tuple of state components before and after the transition. We often follow the convention of adding a prime “′” to names of after-state components (as in the standard Z notation style [1,6]). These rules all take the following form:

\[
\begin{align*}
\text{Condition} & \quad \text{State} \rightarrow \text{State}′.
\end{align*}
\]

All the various rules are composed together disjunctively (in the standard Prolog style) so that any rules that apply will be executed by the underlying Prolog program. They are also composed so that overall execution trace runs can be followed. For further details, see [2,5].

For the standard sequential features of the language, only the program and state variable components are required in the specification of transition rules. Procedural assignment overrides the state value associated with a variable \(V\) with the new value established by the expression \(E\). The program is then terminated (indicated by \(\epsilon\)).

\[
\Sigma′ = \Sigma \oplus (V = E)
\]

\[
\langle V = E, \Sigma \rangle \rightarrow \langle \epsilon, \Sigma' \rangle.
\]

The encoding required for the “\(\oplus\)” overriding operation will be explained later.

The delay operator \(#\) allows a delay of \(N\) time units (typically clock cycles in hardware). There are two possibilities. Firstly the delay may terminate successfully and unconditionally:

\[
\text{true} \quad \#N \rightarrow \epsilon.
\]

Secondly (and non-deterministically), the program may execute for some number of delay units less than the maximum specified:

\[
0 < N' < N \land T = N - N'
\]

\[
\#N \rightarrow \#N'.
\]

Note that Prolog cannot resolve arithmetic expressions to their values in parameters, unlike most other programming languages; hence the use of \(T\) above to hold the value of \(N - N'\). We also explicitly constrain the delay to be greater than zero to ensure that progress is made. Otherwise time may “stand still” in the simulation and the program may never terminate as a result. If this is not an issue, we could relax this constraint and allow zero delays if required. The calculation of \(N'\) is encoded (later) to return smaller values first so that maximal time progress is made if possible.

Events are another non-deterministic aspect of Verilog. There are three possibilities. Firstly the event may occur. This is determined from a change of state from \(\Sigma_1\) to \(\Sigma_2\) which must be provided to this clause:

\[
\langle \Sigma_1, \Sigma_2 \rangle \models \text{Event}
\]

\[
@\text{Event} \cdot S \rightarrow \langle \Sigma_1, \Sigma_2 \rangle \rightarrow S.
\]
Note that at the time of invocation of this clause, both $\Sigma_1$ and $\Sigma_2$ will be instantiated because of the way the clause is used by the rules for parallel composition later. This subtlety is not immediately obvious from the operational semantics, but in practice helps considerably with the implementation.

Of course, it is possible that the event does not occur, in which case there is no change of program state:

$$\langle \Sigma_1, \Sigma_2 \rangle \not\models \text{Event}$$

$$\@ \text{Event} \cdot S \not\rightarrow \@ \text{Event} \cdot S.$$  

Alternatively, time may advance, with no change in the program state:

$$\text{true}$$

$$\@ \text{Event} \cdot S \not\rightarrow \@ \text{Event} \cdot S.$$  

Note the prefix "_" used in the _T time Prolog variable (not to be confused with a Verilog variable) above. At this level we do not know (or even care) how many time units the program will advance. Thus _T could take on any value and is the only occurrence of this Prolog variable in the clause. The prefix "_" indicates to the Prolog system that we know this fact. Otherwise a warning message is issued by most Prolog systems when the program is loaded.

Verilog includes a fairly standard conditional "if" statement and "while" loop. For sequential composition of two programs, if the first program terminates successfully, we can proceed to the second one immediately. Alternatively, partial progression of the first program is possible. The transition may allow time to pass or may involve events. The transition rules for these constructs may be found in [2,5].

As well as sequential constructs, Verilog also includes parallel composition of sequential program components. Here we must deviate slightly from the operational semantics on which this is based since it uses informal "|| . . . ||" notation to indicate a set of indexed sequential programs running in parallel [20]. Thus we must use two or three clauses in place of each of the five original ones to cover the base case and one or more inductive cases. The associated transition rules must deal respectively with entering, remaining in, and exiting sequential program executions not involving time delays or events. These are referred to as "instantaneous sections." For details of these rules, see [2,5].

If no sequential program is engaged in an instantaneous section and all the parallel sequential program threads are willing to engage in an event transition, this may occur as a transition:

$$S \not\rightarrow \langle \Sigma_0, \Sigma \rangle \not\rightarrow S'$$

$$\langle S, \Sigma, \Sigma_0, 0 \rangle \not\rightarrow \langle S', \Sigma, \emptyset, 0 \rangle.$$  

$$\langle S_1, \Sigma, \Sigma_0, 0 \rangle \not\rightarrow \langle S'_1, \Sigma, \emptyset, 0 \rangle \wedge$$

$$\langle S_2, \Sigma, \Sigma_0, 0 \rangle \not\rightarrow \langle S'_2, \Sigma, \emptyset, 0 \rangle$$

$$\langle S_1 || S_2, \Sigma, \Sigma_0, 0 \rangle \not\rightarrow \langle S'_1 || S'_2, \Sigma, \emptyset, 0 \rangle.$$
Alternatively, if no sequential program is engaged in an instantaneous section and all the parallel sequential program threads are willing to engage in allowing time to progress by $T$ time units, then a timed transition can occur:

$$S \rightarrow^T S'$$

$$(S, \Sigma, \emptyset, 0) \rightarrow^T (S', \Sigma, \emptyset, 0).$$

$$(S_1, \Sigma, \emptyset, 0) \rightarrow^T (S'_1, \Sigma, \emptyset, 0) \land$$

$$(S_2, \Sigma, \emptyset, 0) \rightarrow^T (S'_2, \Sigma, \emptyset, 0)$$

$$(S_1 \parallel S_2, \Sigma, \emptyset, 0) \rightarrow^T (S'_1 \parallel S'_2, \Sigma, \emptyset, 0).$$

The above clauses, together with those for entering, remaining in and leaving an instantaneous section, cover the parallel transitions given in the original operational semantics [20].

### 2.3 Macro Definitions

The full set of transitions includes some additional rules left informal in the original semantics [2]. One such aspect is the addition of derived program constructs, based on existing program constructs. To allow this for sequential program components, we can add a macro expansion facility as an additional rule:

$$(X \triangleq P)$$

$$(X, \Sigma) \rightarrow (P, \Sigma).$$

Now we can include additional sequential program constructs as macro definitions. For instance, we can define a program block:

```verbatim
begin S end \triangleq S.
```

The “forever” construct of Verilog executes a program repeatedly in an infinite loop:

```verbatim
forever S \triangleq \text{while true } S.
```

The “for” loop is normally used for a known finite number of iterations, and can also be defined in terms of the while loop:

```verbatim
for (A ; C ; S) \triangleright A \triangleq A \text{ while C ; (P ; S).}
```

Other constructs include a “case ... endcase” statement which may be defined in terms of the conditional if statement. It includes a “default” sub-program for execution if none of the cases are true:

```verbatim
\text{case Sel \{N ; P ; Q\} endcase} \equiv

\text{if Sel = N ; P else (case Sel Q endcase).}

\text{case Sel \{default ; P\} endcase} \equiv P.
```
2.4 Constraint Clauses

The top part of a transition rule consists of constraints that need to be satisfied for the rule to be applied. In a mathematical operational semantics, this may be standard predicates requiring no further elaboration for the mathematically competent reader. However, to allow the transition rules to be executed on a computer, these constraints must be encoded in some form.

We adopt the standard Constraint Logic Programming (CLP) convention of enclosing constraint conditions within curly brackets “{...}” [32]. We must encode all the conditions we need for this particular set of rules using further clauses. Fortunately this proves to be relatively simple; the clauses needed are in fact no larger than the original rules themselves.

A true constraint is no constraint at all. We encode this by simply adding this fact to the set of constraint rules so Prolog will “execute” it successfully:

\{true\}.

On the other hand, the false constraint is impossible to satisfy. We can implicitly encode this by simply omitting it from the set of Prolog constraint rules. Alternatively, if we wish to be more explicit, we can use the built-in fail pseudo-clause that can never be satisfied:

\{false\} <= fail.

It is desirable to allow multiple constraints. In particular, the conjunction of conditions in rules is important. Here we can simply use the in-built conjunction of Prolog (the “,” operator):

\{H \land T\} <= \{H\}, \{T\}.

If required, disjunction may also be encoded directly in Prolog by simply using two clauses:

\{H \lor \lnot T\} <= \{H\}.
\{\lnot H \lor T\} <= \{T\}.

Delays require the non-deterministic generation of numerical values between minimum and maximum (known) time delays. We encode this as follows:

\{N_0 < N_1 < N_2\} <=
\{Min = N_0 + 1 \land Max = N_2 - 1 \land N_1 \in Min..Max\}.

The “..” operator gives all the possible values, if there are any, in ascending order from Min to Max.

We encode > and \geq to simply check the ordering of two integer values:

\{N_1 > N_2\} <= integer(N_1), integer(N_2), N_1 > N_2.
\{N_1 \geq N_2\} <= integer(N_1), integer(N_2), N_1 \geq N_2.
Set membership may be encoded using the standard \textit{member}(\ldots) clause, available in many Prolog systems. If this clause is not available, it is very easy to encode and is included later for completeness.

\[
\{ X \in S \} \iff \text{member}(X, S).
\]

The non-deterministic membership of a number range is encoded as a base case and an inductive case:

\[
\begin{align*}
\{ \text{Min} \in \text{Min..Max} \} & \iff \{ \text{Max} \geq \text{Min} \}. \\
\{ N \in \text{Min..Max} \} & \iff \{ \text{Max} > \text{Min} \land N_1 = \text{Min} + 1 \land N \in N_1..\text{Max} \}.
\end{align*}
\]

It is assumed that \text{Min} and \text{Max} are instantiated and all the possible values of \text{N} (if any) are returned (in ascending order as previously mentioned).

Equality is designed to handle numerical expressions. It is assumed that the right hand side \text{E} is known (instantiated) and the left hand side \text{N} is to be calculated:

\[
\{ N = E \} \iff \text{eval}(E, N, \emptyset).
\]

The \text{eval}(\ldots) clause will be defined later.

Checking for inequality is done using Prolog’s \textit{negation as failure} [9]:

\[
\{ X_1 \neq X_2 \} \iff \neg(X_1 = X_2).
\]

Prolog’s negation as failure must be used with care. Essentially if the (Prolog) variables are instantiated at the time of invocation, this form of negation can be used safely.

Procedural assignment allows the overriding of state with a new (Verilog) variable value. First the expression \text{E} on the right hand side is evaluated and then this value is used to override the original state to produce a new state for all the variables:

\[
\{ \Sigma' = \Sigma \oplus (V = E) \} \iff \text{eval}(E, N, \Sigma), \text{override}((V = N), \Sigma, \Sigma').
\]

The \textit{if} and \textit{while} constructs require the Boolean condition to be evaluated in an environment including all the values of the variables at that point. The positive and negative conditions need to be covered for each of the possible cases:

\[
\begin{align*}
\{ EB \cdot (\Sigma) \} & \iff \text{evalbool}(EB, \text{true}, \Sigma). \\
\{ \neg EB \cdot (\Sigma) \} & \iff \text{evalbool}(EB, \text{false}, \Sigma).
\end{align*}
\]

The detection of events must be encoded, and this is done as separate clauses (see later):

\[
\{ \Sigma \Sigma \models \text{Event} \} \iff \Sigma \Sigma \models \text{Event}.
\]

We can use negation as failure to encode the non-occurrence of events:

\[
\{ \Sigma \Sigma \not\models \text{Event} \} \iff \neg(\Sigma \Sigma \models \text{Event}).
\]

Any of the various types of transition may be included as conditions:
\{ S \langle \Sigma_0, \Sigma \rangle \rightarrow S' \} \iff \text{seq}(S), S \langle \Sigma_0, \Sigma \rangle \rightarrow S'.
\{ S \rightarrow S' \} \iff \text{seqtype}(S), S \rightarrow S'.
\{ S \rightarrow S' \} \iff \text{partype}(S), S \rightarrow S'.
\{ S \rightarrow S' \} \iff \text{partype}(S), S \rightarrow S'.
\
Note that we do dynamic type-checking above for safety. This is especially important (essential) when negation as failure is used, as in one of the cases above. It is not essential in the other cases.

Macro definition may be included as constraints:
\{ X \hat{=} P \} \iff (X \hat{=} P).

This completes the encoding required for the various constraints that can be included as conditions in transition rules.

### 2.5 Support Clauses

During the definition of the “\{\ldots\}” constraint clause, a number of additional Prolog clauses were used. These are defined in this section.

Firstly we need to be able to evaluate Boolean expression in an environment \( \Sigma \) containing the values for all the variables. Evaluation of \texttt{false} and \texttt{true} is easy:

\[
\begin{align*}
\text{evalbool}(\text{false}, \text{false}, \_\Sigma). \\
\text{evalbool}(\text{true}, \text{true}, \_\Sigma).
\end{align*}
\]

Boolean negation is straightforward:

\[
\begin{align*}
\text{evalbool}(\text{false}, \text{true}, \Sigma) & \iff \text{evalbool}(\text{false}, \Sigma) . \\
\text{evalbool}(\text{true}, \text{false}, \Sigma) & \iff \text{evalbool}(\text{true}, \Sigma).
\end{align*}
\]

We can use Prolog’s unification and built-in arithmetic inequality operator to encode equality of arithmetic expressions:

\[
\begin{align*}
\text{evalbool}(E_1 = E_2, \text{true}, \Sigma) & \iff \text{eval}(E_1, N_1, \Sigma), \text{eval}(E_2, N_2, \Sigma). \\
\text{evalbool}(E_1 = E_2, \text{false}, \Sigma) & \iff \text{eval}(E_1, N_1, \Sigma), \text{eval}(E_2, N_2, \Sigma), N_1 \neq N_2.
\end{align*}
\]

Negation may then be used to code inequality:

\[
\text{evalbool}(E_1 \neq E_2, B, \Sigma) \iff \text{evalbool}(! (E_1 = E_2), B, \Sigma).
\]

Arithmetic comparison operators can be encoded using Prolog’s built-in arithmetic support as required:
evalbool(E > E2, true, Σ) ⇐
    eval(E1, N1, Σ), eval(E2, N2, Σ), N1 > N2.

evalbool(E > E2, false, Σ) ⇐
    eval(E1, N1, Σ), eval(E2, N2, Σ), N1 ≤ N2.

evalbool(E < E2, B, Σ) ⇐ evalbool(E2 > E1, B, Σ).

We can easily allow integer expressions to be included as Boolean expressions by considering a value of zero as false and any other value as true:

evalbool(E, false, Σ) ⇐ eval(E, 0, Σ).

evalbool(E, true, Σ) ⇐ eval(E, V, Σ), V ≠ 0.

Further operators, such as Boolean conjunction and disjunction may be encoded as required.

As well as Boolean expressions, integer expressions are also needed. These may be a simple variable, in which case the value in the environment is used:

eval(V, N, Σ) ⇐ variable(V).

If the variable is not in the environment, we assume it has a value of zero:

eval(V, 0, Σ) ⇐ variable(V), ¬member(V = N, Σ).

The expression may also be a simple integer:

eval(N, N, Σ) ⇐ integer(N).

We can encode arithmetic operators such as addition and subtraction using the equivalent built-in Prolog operators:

eval(E1 + E2, N, Σ) ⇐
    eval(E1, N1, Σ), eval(E2, N2, Σ), N is N1 + N2.

eval(E1 - E2, N, Σ) ⇐
    eval(E1, N1, Σ), eval(E2, N2, Σ), N is N1 - N2.

Further operators may be added as required.

Events involving the change of state of variable may involve a positive edge (here assumed to be any increase in the value of a variable) or a negative edge (a decrease in the value of a variable):

⟨Σ1, Σ2⟩ |= posedge V ⇐
member((V = N1), Σ1), member((V = N2), Σ2), N2 > N1.

⟨Σ1, Σ2⟩ |= negedge V ⇐
member((V = N1), Σ1), member((V = N2), Σ2), N2 < N1.

Note that the variable V must have an actual value in Σ1 and Σ2, and it must have changed suitably, for an event to be deemed to have occurred.
An event involving a variable is either a positive or negative edge on that variable:

\[
\langle \Sigma_1, \Sigma_2 \rangle \models V \leftarrow \text{variable}(V), \langle \Sigma_1, \Sigma_2 \rangle \models \text{posedge} \ V \text{ or negedge} \ V.
\]

Events may involve the disjunction of several events on single variables:

\[
\langle \Sigma_1, \Sigma_2 \rangle \models \text{Event or Event} \leftarrow \langle \Sigma_1, \Sigma_2 \rangle \models \text{Event}.
\]

Variables consist of simple Prolog atoms (functors without parameters). However, certain reserved values and integer values should be avoided:

reserved(true).
reserved(false).
reserved(\(\epsilon\)).
reserved(\(\langle\rangle\)).

\[
\text{variable}(V) \leftarrow \text{atom}(V), \neg \text{reserved}(V), \neg \text{integer}(V).
\]

If the standard member(\(\ldots\)) clause is not available, it may be encoded as follows:

\[
\text{member}(X, \langle X | L \rangle),
\text{member}(X, \langle X | L \rangle) \leftarrow \text{member}(X, L).
\]

Finally, overriding of a variable’s value is required for Verilog’s procedural assignment:

\[
\text{override}(\langle V = N \rangle, \langle \rangle, \langle (V = N) \rangle),
\text{override}(\langle V = N \rangle, \langle (V = N) \rangle, \langle (V = N) \rangle, \langle (V = N) \rangle) \leftarrow \neg V_1 = V, \text{override}(\langle V = N \rangle, S_1, S_2).
\]

This concludes the support clauses needed for the semantics to allow the Prolog simulator to execute the transitions in a useful manner. These additional clauses also formalize and clarify the exact meaning of some aspects of the semantics that are left informal in the original mathematical formulation [20].

In execution, the top-level parallel transitions (both timed and untimed) are repeatedly applied to the system state from a desired initial program and state. Execution concludes successfully if and when the program reduces to a parallel set of \(\epsilon\) subprograms. The clauses are ordered carefully to return more “desirable” executions first, but the simulator will attempt to return alternative execution paths if these are possible according to the transition rules. Some example execution runs for simple programs are included in the next section.

3 Animation of the Semantics

We present a couple of example simulation runs of some simple Verilog programs using the operational semantics simulator partially presented in the last section. In normal
operation, Prolog will return possible answers to queries, in the order that they are found using its depth-first left to right search of the clauses. If an answer is found, the user has the option to request another answer or abort the search. If no answer can be found, the Prolog system simply responds with “No” and expects a fresh query from the user in response to its ?– prompt.

The simulation includes additional support clauses to aid in the running of Verilog programs. Rather than continuously prompting the user, possible transitions are simply output to the display, using Prolog’s built-in write(…) pseudo-clause. The simulator continues until all the possibilities are exhausted or a set number of time steps or transitions has been reached, in order to allow non-terminating programs to be aborted conveniently (using Prolog’s built-in abort clause).

?– run #3.
0 (#3, {}, {}, 0)
1 -<3>→ (ε, {}, {}, 0)
1 -<2>→ (#1, {}, {}, 0)
2 -<1>→ (ε, {}, {}, 0)
1 -<1>→ (#2, {}, {}, 0)
2 -<2>→ (ε, {}, {}, 0)
2 -<1>→ (#1, {}, {}, 0)
3 -<1>→ (ε, {}, {}, 0)
No

Fig. 1. A delay of 3 time units.

Consider a “#” delay statement, allowing a delay of 3 time units (e.g., clock cycles), as shown in Figure 1. The most “preferable” execution is to simply allow time to progress by 3 time units and then finish executing the program. However there are three other possibilities. Either time could progress by 2 units followed by 1 unit, or it could progress by 1 unit followed by 2 units, or it could progress by 1 unit in three steps. This program can proceed to termination in four different ways, as can be seen by the four occurrences of “ε” above.

Of course this non-determinism is not very interesting in the case of a single sequential program, but consider three parallel assignments, each delayed by a different amount (see Figure 2). The simulator attempts to progress time by the maximal possible amount and executes the assignments after the required delays. The non-determinism in the delay construct rule allows this to be done in a natural way, letting the Prolog system explore the possible paths for us, and resolving the non-determinism in the process. In this particular case there is only one possible execution path since the difference between the delays is only 1 time unit, the minimum allowable time delay.

Further examples of simulations can be found in [2,5]. The animation of such programs allows the semantics of Verilog to be explored in a simple manner, with the extra confidence that machine execution of a program brings compared to hand execution (e.g., as done in [20]). Once formulated, it is easy to add and remove rules to and from the Prolog simulator, and to modify them, to explore the consequences.
This exercise is considered a success in that the original operational semantics [20] has for non-deterministic aspects required. Tracking facilities built into the Prolog system itself, with little or no explicit encoding of potential execution paths can be explored in a convenient way by the search and backtracking facilities built into the Prolog system itself, with little or no explicit encoding for non-deterministic aspects required.

4 Conclusion

This exercise is considered a success in that the original operational semantics [20] has been very directly encoded in Prolog in a short period of time. Of course the rapid-prototype system produced is only suitable for very small Verilog programs, but it could be a useful didactic aid in the understanding of the semantics of Verilog programs which is otherwise only generally available in large informal documents such as the IEEE Standard [26] and textbooks [14] or in large software simulators that are necessarily deterministic for efficiency reasons. Alternatively, if the resources and demand were available, a much more robust simulator could be produced, based on the experience gained using this rapid-prototype.

Of course, the simulator presented here does not cover the whole of Verilog. In particular, modularization facilities are omitted, and these are important in larger Verilog programs. However, if further program constructs are considered worthwhile for inclusion, their operational semantics transition rules can be formulated and added to the simulator relatively easily.

Despite the constraint of executability, the Prolog simulator operational semantics for a subset of the Verilog Hardware Description Language (HDL) developed and partially presented here is pleasingly close to the original operation semantics on which it was based [2,20]. Much of the development time was spent on establishing the correct precedence and associativity of the operators and then ensuring the ordering of the encoded relations resulted in a preferred possible execution path being presented to the user first.

An advantage of the logic programming approach, over the functional programming approach for instance, is that different execution paths can be allowed in a natural and implicit manner. This is especially useful in the execution of parallel programs (as normally required in the simulation of physically parallel hardware) since the combinations of potential execution paths can be explored in a convenient way by the search and backtracking facilities built into the Prolog system itself, with little or no explicit encoding for non-deterministic aspects required.
The execution time for the simple examples shown here is essentially instantaneous, so the simulator is very usable for didactic examples. Of course the presence of non-determinism means that the full search space is potentially exponential, so returning all possible execution paths for larger examples could be unacceptably slow. In addition, the amount of information displayed to the user could be overwhelming. However, with careful ordering of the clauses, the more interesting of the possible execution paths can be returned first. If just a single execution path is required, the simulator could be made acceptably efficient for much larger examples. For longer traces, it would also be sensible to only display important state changes (for example, between executions of instantaneous sections). This facility could be easily added to the simulator presented here.

The architecture of the Prolog program modelling the operational semantics plays a vital role in its potential reusability. A two-level semantics for parallel threads and sequential programs (instantaneous sections) has been developed, and the success of the model demonstrates its overall compositionality [5]. It could be further enhanced by adding another level for open components, which could then help act as the basis for bisimulation and algebraic laws. Indeed, an interesting area for further exploration is the algebraic laws associated with Verilog [21]. The parallel aspects of Verilog mean that some rules associated with traditional sequential languages only apply in certain circumstances and additional laws are required for the parallel parts of the language. Using these laws, a compiler (written in Prolog, for example) from high-level Verilog programs to low-level digital hardware circuits could be produced.

Hardware compilation (and its verification) of a high-level executable description specifying the desired function of the system to a low-level description of the matching implementation components and their interconnections is an increasingly important technique in the production of digital hardware [4]. Ultimately, formalization of hardware/software co-design to help achieve a unified framework for computer system development is a goal worth pursuing [18,19]. It is possible to use a parallel programming language such as Occam in a unified manner for this purpose [27]. However, although attractive formally, unfortunately it is unlikely to be widely used in practice since Occam is not popular in industry.

Combining notations effectively and efficiently, be they informal, mathematical, non-executable or executable, is a way of adding extra assurance in the development of high-integrity systems by providing complementary ways of avoiding and removing errors [7]. It is a practice to be recommended in the industrial application of formal methods where failure could be costly in financial or even human terms [3,22].

Acknowledgements. I would like to thank Prof. He Jifeng at the United Nations University International Institute for Software Technology (UNU/IIST) in Macau for research collaboration over the years, and during my visit in particular, and Dr. Xu Qiwen (also at UNU/IIST) for providing helpful guidance on the operational semantics of Verilog. The research presented in this paper would not have been possible in the time-scale without the availability of their elegant and concise operational semantics [20].

References
Why Doesn’t Anyone Use Formal Methods?

Wolfram Schulte
Microsoft Research
One Microsoft Way, Redmond, WA, 98052-6399, USA
Phone: +1 425 703 4641, Email: schulte@microsoft.com
http://research.microsoft.com/users/schulte/

Of course, to say that nobody uses formal methods is an exaggeration, see the recent study by Dan Craigen: "Formal Methods Adoption: What’s Working? What’s Not!” (ORA Canada). Still, many things can be improved by adapting formal methods to the needs of its users (and not of researchers). This abstract highlights some aspects that we found important in our work with system designers within Microsoft.

Specifications should be easy to understand. The main purpose of writing specifications is to gain understanding about the problem domain and to come up with appropriate models. Whether this understanding is correct typically can not be checked by formal means at all. Only conceptual and experimental justification is possible. That’s why readability and – to less degree – executability is so important.

Specifications should be state oriented. Specifications have to deal with existing or proposed systems. And these systems – ranging from device drivers to Web applications – are typically state based. To “ignore” state does not help in developing reasonable, understandable models of these systems. This does not imply that particular logics are useless. Temporal logic, for example, is reasonable to specify liveness properties of state based systems.

Specifications should support abstraction, subtyping and modularization. Today systems are getting larger. They are getting more complex. Abstraction, subtyping and packaging play the central role in understanding these huge systems. Abstraction helps to ignore irrelevant detail. Subtyping helps to define similarities. Modularization helps to build manageable self-contained units. Although these features are exactly in the domain of formal methods, it seems as if formal methods have not contributed to design or build huge systems at all. Systems practitioners often raise the question: Do formal methods scale?

Specifications must live. We all believe in the value of specifications, but what happens when coding starts? Most code is not derived by any formal means from the specification. As a consequence, the specification very often becomes outdated and in most cases immediately losses its value. But even if we update the specification, how do we guarantee that the code behaves as described. There is only one answer: tools have to check that the specification and the implementation agree. One way to achieve this is to instrument the implementation with assertions derived from the model. Another might be to generate test cases from the model. And there are various other routes.

Specifications must be supported by tools. Current development practice is difficult to change. The only way to do it is to write tools with a high payoff.
The payoff can come in various forms: testing the design before implementing it, analysing the specification for properties like freedom from deadlock, etc. But it is simply not enough to have a notation and a process. Only when industrial strength tools are available, we are going to have a chance to influence the development process, too.

*Specification tools must be compatible and interoperable.* Nowadays, most systems are assembled from existing components rather than built from scratch. Specification methods and tools must be interoperable with existing environments. Emerging standards like COM, CORBA or the new .NET platform of Microsoft can help to integrate specifications into existing systems.

*Specification tools must be easily portable and available.* Many formal systems often only work in one environment, and very often this is Unix. Yet, companies typically do not choose their hardware based on the software tool they want to use. Simply porting tools is often not enough – modern tools should be integrated into the development environment. Specification documents should live in Word or HTML instead of TeX, specification sources should live in programming environments like Visual Studio, instead of just being command line applications.

*Specification building must be taught.* Building the right model for non-trivial systems is a challenging task. It takes a while to find the right abstractions and packaging. However it pays off. Experience shows that a week of training is needed to get started to write reasonable specifications. If you work for a company, maybe you can convince your company to go this way too?
How to Write a Healthiness Condition

Yifeng Chen
Oxford University Computing Laboratory
Parks Road, Oxford OX1 3QD, UK
yifeng.chen@comlab.ox.ac.uk

Abstract. This paper presents a technique called generic composition to provide a neat basis for different kinds of semantic compositions and various higher-order healthiness conditions appearing in a variety of semantic theories. The weak inverse of generic composition is defined. A completeness theorem shows that any predicate can be written in terms of generic composition and its weak inverse, and a number of algebraic laws are given to support reasoning.

1 Introduction

The modeling of programs as predicates coupled with the view of observation-oriented semantics simplifies the semantic interface, benefits from the expressive power of set theory and logic, and allows refinement to be expressed as implication. The use of predicate calculus is especially useful since variable hiding becomes as simple as existential quantifier applied to a predicate’s corresponding input and output variables.

Such predicate semantics corresponds directly to relational semantics [9,10,14] and has been shown to span sequential (imperative, logic and functional) and parallel (synchronous and asynchronous) programming paradigms. This paper helps unify those various applications of the theory by proposing a form of composition which unifies the forms of composition basic to those various paradigms.

In [14] a semantic model is a set of predicates satisfying some healthiness conditions. The semantic model $\text{Pred}$ without any healthiness condition is a complete lattice, which contains all predicates. Its top and bottom are predicate constants ‘false’ and ‘true’ respectively. Its lub (least upper bound) and glb (greatest lower bound) are conjunction and disjunction respectively. The ordering relation is refinement ordering $\sqsubseteq$, such that $A \sqsubseteq B$ if and only if $B \Rightarrow A$ is always true.

The notion of ‘healthiness condition’ was first introduced by Dijkstra in [7] for predicate-transformer semantics. In [14] a healthiness condition is written $h(\cdot)$ where $h$ is a higher-order predicate. A predicate $A$ is healthy with respect to this healthiness condition, if $h(A)$ is always true. A semantic model may satisfy several healthiness conditions. A model with more healthiness conditions is a subspace of a model with less ones. Let $\mathcal{S}$ be a semantic model. Imposing an additional healthiness condition $h(\cdot)$ on $\mathcal{S}$ generates a subset model $\mathcal{T} = \{A \in \mathcal{S} | h(A)\}$. Any subset model preserves the refinement ordering as its ordering.
relation; however, top, bottom, lub and glb of a subset model may differ from those of the original model.

If a healthiness condition $h(A)$ can be expressed as an algebraic equation $A = f(A)$, $f$ is then called a healthiness function. If it is also idempotent i.e. $f \circ f = f$, any unhealthy predicate $X$ will be mapped to a healthy one $f(X)$. A healthy predicate becomes exactly a fixed point of the function $f$.

If all healthiness conditions are written algebraically $A = f(A)$ and the corresponding healthiness functions are idempotent and commute with each other, the composition of those functions is again a monotonic and idempotent function called a grand healthiness function in which the order of the healthiness functions can be arbitrary.

Writing healthiness conditions in such a restricted way has following advantages:

1. such a healthiness function characterises a link between semantic models by transforming each unhealthy predicate to a healthy one in the more detailed model;
2. such a healthiness function maps a complete-lattice semantic model to a subset complete-lattice model;
3. the semantic definition (also a function) of a command (e.g. sequential composition) is closed under a model iff the command commutes with all healthiness functions;
4. according to a major theorem in [14] (Chapter 4), a healthiness function transforms the fixed point of a recursion program to that in a more detailed model if it satisfies a couple of simple conditions.

These advantages suggest that we should write healthiness conditions in this form whenever it is possible.

One would doubt whether this approach is general enough to cope with various healthiness conditions from very diversified semantic models. This is a reasonable concern, especially when those higher-order healthiness conditions are not as simple as conjunction $A = A \land H$ or disjunction $A = A \lor H$. Suppose we have $m$ healthiness conditions (in CSP $m = 8$ and in BSP $m > 10$) and $n$ commands (in CSP $n = 8$, in BSP $n = 13$); then $m \times m$ proofs are needed for the idempotence and commutativity of the healthiness functions and $m \times n$ are needed for the closure of commands. Many technical proofs are not trivial. Furthermore a semantic space may not even be a complete lattice. The finitary condition (finite non-determinism) is an example. It seems that such a healthiness condition cannot be written in terms of a monotonic and idempotent healthiness function.

In order to simplify the semantic studies on healthiness conditions, we propose a new unifying technique called generic composition. By writing all healthiness functions with generic compositions, we can reduce the proofs of idempotence and commutativity to some easy-to-check conditions. Although some semantic models are not complete lattices, we find it convenient to add either top or bottom to the space to ‘forge’ a complete lattice (see Section 5).
The weak inverse of generic composition is also defined. A completeness theorem shows that any predicate can be written in terms of generic composition and its weak inverse, thus showing the technique is general.

By using generic compositions, we have systematically defined (in terms of monotonic and idempotent fixed-point functions) higher-order healthiness conditions from a variety of models, such as Unifying Theories of Programming [14], CSP [11], CCS [15], Game Semantics [1], Z [22], probabilistic programming [8], safety and liveness properties [2] and BSP [18,5,6]. The technique has significantly simplified semantic studies on BSP [5].

[14] introduces a kind of parallel composition called parallel-by-merge. Its definition is based on sequential semantics. Two programs composed by a parallel composition are first simulated separately and their disjoint results are exported to a merge operation following them. However, as [18] noticed, the merge of BSP’s parallel composition may not be a valid merge (see [4,5]). Parallel-by-merge allows no interference between inputs in that its initial state is copied by both components. A Petri-net [20] counterexample in Section 5.1 shows that there can be ‘conflict’ between the inputs of the two components in a parallel composition. We propose a new kind of parallel composition called parallel-via-medium based on generic composition. It fixes BSP’s parallel composition [18] and leads to a more general parallel composition and simpler reasoning for it.

Section 2 reviews the formalism from [14] required here. Section 3 introduces generic composition and its weak inverse. Section 4 introduces generic-composition-based sequential and parallel compositions and higher-order healthiness conditions. More advanced forms of generic compositions are also introduced in this section. Section 5 shows how generic composition can be used to define parallel compositions and higher-order healthiness conditions for various semantic models.

2 Relational Semantics

In this section, we shall review some formalism of [14] required in this paper.

The fundamental concept in [14] is that of a binary relation. A binary relation is a pair \((\alpha, P)\) where \(\alpha\) is a set of variables called the alphabet and \(P\) is a predicate containing no free variables other than those in \(\alpha\). We abbreviate that pair to \(P\) if omitting its alphabet does not lead to confusion.

We may add a decoration to \(\alpha\) to create a set of new variables. For example, \(\alpha’\) is the set of dashed variables created from \(\alpha’ = \{v’ | v \in \alpha\}\). Such a syntactic convention [22] is convenient for expressing different observable aspects of a program variable. For example, if \(x\) is a program variable, \(x\) itself denotes the initial state of that program variable, while \(x’\) can denote its final state. We may also decorate \(\alpha\) in other ways than by using dash: \(\pi\), \(\alpha_0\), \(\alpha_1\), \(\alpha.0\), \(\alpha.1\), \(\pi.2\), etc.

The alphabet \(\alpha\) of a binary relation has two parts \(\alpha_{\text{in}}\) and \(\alpha’_{\text{out}}\) satisfying \(\alpha = \alpha_{\text{in}} \cup \alpha’_{\text{out}}\). \(\alpha_{\text{in}}\) is a set of undashed variables called the input alphabet, while \(\alpha’_{\text{out}}\) is a set of dashed variables called the output alphabet.
For example, if \( x \) and \( y \) are the only program variables, the semantics of program \( x := x + 1 \) becomes a relation

\[
(\{x, y, x', y'\}, x' = x + 1 \land y' = y).
\]

The refinement order between two relations \( A \) and \( B \) with alphabet \( \alpha \) is defined by \( A \sqsubseteq B \equiv \forall \alpha \cdot (B \Rightarrow A) \) where \( \alpha \) is a vector of all variables from \( \alpha \) in an alphabetical order. Using alphabetically ordered variable vectors instead of variable sets is convenient for substitution. For example, \( A[\beta_0/\beta] \) modifies \( A \) by substituting each variable \( x \) in \( \beta \) with another variable \( x_0 \). Adding a decoration to \( \beta \) creates a vector of new variables in the same alphabetical order. For example, vector \( \beta_0 \) has the same alphabetical order as \( \beta \). If \( x \) appears before \( y \) in \( \beta \), then variable \( x_0 \) must appear before \( y_0 \) in \( \beta_0 \).

Sequential composition between two relations is then defined by

\[
A; B \equiv \exists \alpha_0 \cdot A[\alpha_0/\alpha_{out}] \land B[\alpha_0/\alpha_{in}]
\]

where \( \alpha_0 \) is a vector of new variables (otherwise we use \( \alpha_1, \alpha_2 \), etc.).

Two special variables \( ok \) and \( ok' \) are introduced into the input and output alphabets respectively to represent termination/non-termination. \( ok \) represents the proper start of a computation, while \( ok' \) represents its successful termination.

A design [14] is a binary relation satisfying \( A = (ok \Rightarrow A) \) and the downwards-closure healthiness condition of \( ok' \): \( A[false/ok'] \Rightarrow A[true\!ok'] \). In general, a design can be written as \( b \vdash P \) where \( b \) is the weakest condition under which the computation terminates and \( P \) is a binary relation describing the constraints between the observable variables achieved on termination. Both \( b \) and \( P \) do not contain \( ok \) or \( ok' \).

A link between a semantic model \( S \) and its subset model \( T \subseteq S \) is a total function \( f:S \rightarrow S \), which ranges over all members of \( T \) i.e. \( \text{ran}\ f = T \). The subset model \( T \) satisfies not only all healthiness conditions of \( S \) but also an additional healthiness condition \( h(A) \equiv A \in \text{ran}\ f \).

The simplest form of parallelism that [14] (Chapter 7) tackles is called disjoint parallelism. If \( ok \) and \( ok' \) are the only free variables shared by \( b \vdash P \) and \( c \vdash Q \), their parallel composition becomes \( b \land c \vdash P \land Q \). That means chaos (or non-termination) from either side will lead to a chaotic parallel composition; however if both sides terminate successfully, the parallel composition also terminates and achieves both \( P \) and \( Q \) in conjunction.

Based on disjoint parallelism, a more general parallel composition called a parallel-by-merge can be defined [14]. In a parallel-by-merge composition, two components \( A \) and \( B \) are first simulated separately under disjoint alphabets by re-labeling. Then the results from both sides of the composition are combined and exported to a merge operation following them.

Suppose \( m \) is the only program variable shared by two programs \( A \) and \( B \). Their parallel composition is defined by

\[
A \parallel_M B \equiv (A[0.m'/m'] \parallel B[1.m'/m'])_{+m} ; M(ok, m, 0.m, 1.m, m', ok')
\]
where merge $M$ is a design, $0.m'$ and $1.m'$ are new dashed variables, and subscript $+m$ makes a copy of $m$: $P_{+m} \equiv P \land (m = m')$.

The following definition shows how complicated a healthiness condition can be. It is a healthiness condition for the trace record of CSP processes defined in [14]:

$$A = \left[\pi\right] A[s/\text{tr}, s \wedge (\text{tr}' - \text{tr})/\text{tr}']$$

where $\wedge$ denotes concatenation between sequences.

However the definition is not easy to understand. Actually it states that the behaviour of a reactive process does not depend on any behaviour before the start of this process. Its corresponding healthiness function is too complicated to allow straightforward proofs for its idempotence and commutativity with other healthiness functions.

## 3 Generic Composition

As motivation we observe that various compositional definitions and designs tend to have a similar pattern. For example,

- in relational semantics [19], the sequential composition of two binary relations $r$ and $s$ is defined by $x(r; s)z \equiv \exists y \cdot xry \land ysz$;
- in CSP ([11] Section 4.3), the parallel composition between two communicating processes is defined by $(((c!v \rightarrow P)\parallel(c?x \rightarrow Q(x))) \setminus C \equiv (P\parallel Q(v)) \setminus C$ where communications along channel $c$ are concealed;
- in CCS ([15] Section 2.2), two components $A$ and $B$ with port $c$ and $\overline{c}$ respectively can be combined and they communicate through the ports according to a rule $\frac{A \overset{\alpha}{\rightarrow} A', B \overset{\alpha'}{\rightarrow} B'}{A | B \overset{c}{\rightarrow} A' | B'}$ in which each communication becomes a silent action not directly observable from the outside.

In general, a system combining two components $A$ and $B$ demonstrates the behaviours of both $A$ and $B$, and communications between the two components are normally delivered through an interface hidden from the outside.

Generic composition provides a general form of such compositionality based on first-order logic.

### 3.1 Definition

A basic generic composition [6] is written as $A ; \beta B$ in which $\beta$ is called interface, and $A$ and $B$ can be any predicates. A generic composition hides the interface between its components by applying an existential quantifier on the interface.

**Definition 1**  $A ; \beta B \equiv \exists\beta_0 \cdot A[\beta_0/\beta] \land B[\beta_0/\beta]$.
In this definition, we suppose $\beta_0$ is a vector of new variables free in neither $A$ nor $B$ and the interface $\beta$ does not contain any overlined variable. The definition looks superficially like sequential composition. However it does not distinguish input and output variables; its interface is set explicitly; and there is no alphabetical restriction.

In this paper, we shall use $A = \exists_x A$ to denote the condition that predicate $A$ does not depend on variable $x$. For example, neither predicate $(y > 2)$ depends on $x$, nor does $(x = x \land y > 2)$. In the latter example $x$ is actually a free variable, although $(x = x \land y > 2)$ does not state anything interesting about $x$. Such variable independence is more general than syntactical restriction imposed by an alphabet ([14] Chapter 1). Interestingly $A = \forall_x A$ states the same healthiness condition.

Thus the alphabetical restriction can now be replaced by a series of healthiness conditions:

$$A = \exists_x A \quad (x \notin \alpha).$$

All these healthiness functions are idempotent and commutative, therefore they can be considered as a grand healthiness condition. Whenever we need an alphabet for a predicate, we can simply impose this healthiness condition on it.

The following propositions indicate some simple cases of basic generic composition and are proved by routine predicate manipulations.

**Proposition 1**  
$A \land B = A :_{\beta} B$.

**Proposition 2**  
$\exists \beta A = A :_{\beta} true$.

**Proposition 3**  
$A[\gamma / _{\beta}] = A :_{\beta} (\overline{\beta} = \gamma)$.

### 3.2 Weak Inverse

Most arithmetic operations have an inverse operation. For example $-$ is the inverse of $+$ in the sense that if $x + p = q$, we can restore $x$ by calculating $q - p$.

Not every operation has an inverse: sequential composition does not. In general $X$ cannot be restored from $X : P = Q$. Nevertheless it is possible to define sequential composition’s weak inverse $Q / P$, which is the weakest specification refining $Q$ if followed by $P$

$$Q / P \equiv [\overline{\beta} (Y : P \supseteq Q)].$$

This concept (Hoare and He [12]) is called the **weakest pre-specification**. A weak inverse of parallel composition called **weakest environment** was defined by Lai and Sanders [17]. It was applied to Valiant’s BSP [21] by Lecomber [18].

A similar weak inverse operation can be defined for generic composition. We call it the **weakest pre-predicate**. Given two predicates $Q$ and $P$, we define $Q / _{\beta} P$ to be the weakest predicate $X$ making $X :_{\beta} P \Rightarrow Q$ always true.

The following Galois connection provides a neat definition as a weak inverse in the sense of adjunctions.
Definition 2  \( Q \subseteq X \vdash P \) iff \( Q/P \subseteq X \).

Theorem 1  \( Q/P = \neg(\neg Q \vdash [P/\beta, \beta/\bar{\beta}]) \).

Now we can express some more primitive combinators; again the proofs are routine predicate manipulations.

Proposition 4  \( A \Rightarrow B = B/\emptyset A \).

Proposition 5  \( \neg A = false/\emptyset A \).

Proposition 6  \( A \lor B = B/\emptyset (false/\emptyset A) \).

Proposition 7  \( \forall \beta A = A/\beta true \).

Theorem 2 (Completeness) Any predicate can be written in terms of generic composition and its weak inverse using only the constant predicates true and false and predicate letters.

This is justified by the fact that by Propositions 1 to 7 all logic operators of predicate calculus can be eliminated from the formula of a predicate.

The completeness theorem shows the potential of generic composition for expressing and defining semantic operators, which we consider in section 4.

3.3 Laws

We shall use a convention \( x \Diamond y \) for disjoint sets: \( x \Diamond y \equiv (x \cap y = \emptyset) \). Only non-trivial proofs are given.

Law 1 (Associativity)  \( (A :_\beta B) :_\beta C = A :_\beta (B :_\beta C) \).

Assume that \( \alpha \Diamond \beta, \beta \Diamond \gamma, \gamma \Diamond \alpha, A = \exists \gamma A \) and \( C = \exists \alpha C \), then

Law 2 (Associativity’) \( (A :_{\alpha \cup \beta} B) :_{\beta \gamma} C = A :_{\alpha \cup \beta} (B :_{\beta \gamma} C) \).

This law reveals some interesting precondition of general associativity. Law 1 is a special case of Law 2 when \( \alpha = \gamma = \emptyset \).

Proof.

\[
\begin{align*}
A :_{\alpha \cup \beta} B :_{\beta \gamma} C & = \exists \beta_1 \gamma_1 \cdot (\exists \alpha_0 \beta_0 \cdot (A[\alpha_0/\alpha, \beta_0/\bar{\beta}] \land B[\alpha_0/\bar{\alpha}, \beta_0/\beta]) [\beta_1/\beta, \gamma_1/\gamma]) \\
& = \exists \beta_1 \gamma_1 \cdot \exists \alpha_0 \beta_0 \cdot (A[\alpha_0/\alpha, \beta_0/\beta] \land B[\beta_1/\beta, \beta_1/\gamma, \gamma_1/\gamma]) [\alpha_0/\alpha_0, \beta_0/\beta_0] \\
& = \exists \alpha_0 \beta_0 \cdot (A[\alpha_0/\alpha, \beta_0/\beta] \land (B :_{\beta \gamma} C)[\alpha_0/\bar{\alpha}, \beta_0/\beta]) \\
& = A :_{\alpha \cup \beta} (B :_{\beta \gamma} C) \qed
\end{align*}
\]
Law 3 (Distributivity of disjunction)

\[(A \lor B) : \beta C = (A : \beta C) \lor (B : \beta C)\]
\[A : \beta (B \lor C) = (A : \beta B) \lor (A : \beta C)\]

Law 4 (New interface)

\[A = A : \beta (\overline{B} = \beta)\]
\[A = (\overline{B} = \beta) : \beta A\]

\(\overline{B} = \beta\) is called the \(\beta\)-identity predicate. It forms a left and right zero of generic composition.

Law 5 (Conjunction elimination)

\[A \land (A : \beta B) = A : \beta (\overline{B} = \beta \land B)\]
\[(A : \beta B) \land B = (\overline{B} = \beta \land A) : \beta B\]

Law 6 (Disjunction elimination)

\[A \lor (A : \beta B) = A : \beta (\overline{B} = \beta \lor B)\]
\[(A : \beta B) \lor B = (\overline{B} = \beta \lor A) : \beta B\]

Assume \(A = \exists \gamma \exists \overline{B} A\) and \(B = \exists \gamma \exists \overline{B} B\), then

Law 7 (Interface re-labeling)

\[A : \beta B = A[\gamma/\beta] : \gamma B[\overline{B}/\overline{\beta}]\]

The interface of a generic composition can be replaced by some new variables.

Assume \(A = \exists \gamma A\) and \(B = \exists \overline{B} B\), then

Law 8 (Interface expansion)

\[A : \beta \exists \overline{B} B = A : \beta \exists \gamma \exists \overline{B} A : \beta \exists \overline{B} B\]

Assume \(\gamma \circ \beta\), then

Law 9 (Interface extension)

\[A : \beta \exists \overline{B} B = A : \beta \exists \gamma \exists \overline{B} (\gamma = \gamma \land \exists \overline{B} B)\]
\[\exists \gamma A : \beta B = (\overline{\gamma} = \gamma \land \exists B A) : \beta \exists \overline{B} B\]

Proof.

\[A : \beta \exists \overline{B} B = A : \beta \exists \gamma \exists \overline{B} B\]
\[= \exists \beta_0 \gamma_0 \cdot A[\beta_0/\beta, \gamma_0/\gamma] \land (\gamma_0 = \gamma) \land \exists \overline{B} B[\beta_0/\overline{B}]\]
\[= \exists \beta_0 \cdot A[\beta_0/\beta] \land \exists \overline{B} B[\beta_0/\overline{B}]\]
\[= A : \beta \exists \overline{B} B\]

The second part of this law can be proved similarly. \(\square\)

Assume \(\gamma \circ \beta\), \(B = \exists \gamma \exists \overline{B} B\) and \(C = \exists \beta \exists \overline{B} C\), then

Law 10 (Left/right interface commutativity)

\[(A : \beta B) : \gamma C = A : \beta \exists \gamma \exists \overline{B} (B \land C) = (A : \gamma C) : \beta B\]
\[C : \gamma (B : \beta A) = (C \land B) : \beta \exists \gamma \exists \overline{B} A = B : \beta (C : \gamma A)\]
This law describes a sufficient condition under which two interfaces $\beta$ and $\gamma$ will not interfere with each other, and therefore components $B$ and $C$ are actually placed in conjunction and can be connected to an existing component $A$ in either order. Thus components $B$ and $C$ can be considered to be in parallel. Parallel-via-medium in Section 4 is defined on this law.

**Proof.**

\[
\begin{aligned}
(A : \beta B) : \gamma C &= A : B, \gamma (\gamma = \gamma \wedge B) : (\gamma = \gamma \wedge B), C \\
&= A : B, \gamma (B \wedge C) \\
&= (A : \gamma C) : \beta B.
\end{aligned}
\]

The second part of this law can be proved similarly. \(\square\)

Assume $C = \exists \bar{\beta} | \bar{\beta} C$, then

**Law 11 (Commutativity with conjunction)**

\[
(A : \beta B) \wedge C = (A \wedge C) : \beta B = A : (B \wedge C).
\]

This is a special case of Law 10 when $\gamma = \emptyset$. If component $C$ has no interference with the interface of another generic composition in conjunction, it can be located anywhere in the composition.

Assume $A = \exists \gamma A$, $B = \exists \bar{\gamma} B$, $C = \exists \beta C$ and $D = \exists \bar{\beta} D$, then

**Law 12 (Interface merge)**

\[
(A : \beta B) \wedge (C : \gamma D) = (A \wedge C) : \beta, \gamma (B \wedge D).
\]

If the interfaces of two generic compositions in a conjunction have no interference with each other, they can be merged into a joint interface.

**Proof.**

1. If $\beta \cap \gamma = \emptyset$, then

\[
\begin{aligned}
(A : \beta B) \wedge (C : \gamma D) &= \exists \beta_0 \cdot A[\beta_0/\beta] \wedge B[\beta_0/\beta] \wedge (\exists \gamma_0 \cdot C[\gamma_0/\gamma] \wedge D[\gamma_0/\gamma]) \\
&= \exists \beta_0 \exists \gamma_0 \cdot A[\beta_0/\beta] \wedge B[\beta_0/\beta] \wedge C[\gamma_0/\gamma] \wedge D[\gamma_0/\gamma] \\
&= (A \wedge C) : \beta, \gamma (B \wedge D).
\end{aligned}
\]

2. If $\beta \cap \gamma \neq \emptyset = \emptyset$ and $\phi = \beta \backslash \phi$, then $A : \beta B = A : \phi B$ according to Law 8. \(\square\)

4 Defining Other Semantic Operators

4.1 Sequential Composition

A sequential computation is a predicate with input alphabet $\alpha_{in}$ and output alphabet $\alpha_{out}$. The input alphabet contains all observable program variables and the output alphabet contains all modifiable program variables. We suppose any modifiable program variable is also observable, i.e. $\alpha_{out} \subseteq \alpha_{in}$. Then sequential composition can be defined on generic composition.
Definition 3 $A; B \triangleq A \cdot \omega_{out} B[\alpha_{out}/\alpha_{out}]$.

Interestingly, it is also possible to define it in the reverse order. Note that the a generic composition $X : \beta Y$, any variable $x : \beta$ in $X$ corresponds to $\pi$ in $Y$.

Theorem 3 $A; B \triangleq B \cdot \omega_{out} A[\alpha_{out}/\alpha_{out}]$.

The sequential composition is a special kind of generic composition where the interface is set to be all output variables. Thus its properties can be easily deduced from the laws in section 3.3.

4.2 Parallel Composition via Medium

Parallel-via-medium composition is defined as follows:

Definition 4 $A \parallel_{\pi} M \parallel_{\pi} B \triangleq \omega_{\pi}(A[\pi.0/\pi] \land B[\pi.1/\pi]) : \pi.0 \lor \pi.1 M$.

Variable set $\pi$ explicitly sets the interface between $A$ and $B$. If components $A$ and $B$ do not depend on $\pi.0$ and $\pi.1$, $\pi$ will be substituted by two vectors of new variables $\pi.0$ and $\pi.1$ in $A$ and $B$, respectively. In case $A$ or $B$ depends on $\pi.0$ or $\pi.1$, law 7 can be used to re-label the interface with some new variables to avoid interference.

$M$ is called a medium between $A$ and $B$. A medium puts all information from $\pi.0$ and $\pi.1$ together and restores interface $\pi$. To make a parallel-via-medium composition commutative and associative, a medium needs to satisfy corresponding healthiness conditions.

The convention $I I^i \triangleq (\pi.i = \pi)$ is used in the following theorems. Thus we have $M = I I^0 \parallel_{\pi} I I^1$. We also assume that $A$, $B$ and $C$ do not depend on any variable in $\pi.0$ or $\pi.1$ $(i = 0, 1, \cdots)$.

Theorem 4 $A \parallel_{\pi} M \parallel_{\pi} B \equiv B \parallel_{\pi} M \parallel_{\pi} A$, iff $M$ is commutative in this sense

$$I I^0 \parallel_{\pi} I I^1 = I I^1 \parallel_{\pi} I I^0.$$  

Theorem 5 $(A \parallel_{\pi} M \parallel_{\pi} B) \parallel_{\pi} C \equiv A \parallel_{\pi} (B \parallel_{\pi} M \parallel_{\pi} C)$, iff $M$ is associative in this sense

$$(I I^0 \parallel_{\pi} I I^1) \parallel_{\pi} I I^2 = I I^0 \parallel_{\pi} (I I^1 \parallel_{\pi} I I^2).$$

Proof. Let $A_0 \triangleq [\pi.0/\pi]$, $B_0 \triangleq [\pi.1/\pi]$ and $C_0 \triangleq [\pi.2/\pi]$. If $M$ is associative then

$$(A \parallel_{\pi} M \parallel_{\pi} B) \parallel_{\pi} C$$

$$= (A_0 \land B_1 : \pi.0, \pi.1 I I^0 \parallel_{\pi} I I^1) \parallel_{\pi} C$$

$$= (A_0 \land B_1 : \pi.0, \pi.1 (I I^0 \parallel_{\pi} I I^1) \parallel_{\pi} (C_2 : \pi.2 I I^2))$$
Parallel-via-medium has one more parameter $\pi$. It may include either shared or unshared output variables, input variables or their mixture. However, if $\pi$ is the set of all shared output variables of both components, the composition becomes a parallel-by-merge composition [14].

4.3 Higher-Order Healthiness Conditions

A higher-order healthiness condition defined on generic composition is normally written in the form $A = A : \beta R$ and called generic healthiness condition. Its corresponding healthiness function is $\lambda X \cdot X : \beta R$.

Writing a healthiness condition in this form is motivated by [14] (Chapter 3) in which the condition $H3$ for the total-correctness of designs is defined by $A = A ; \parallel \bot$ where $\parallel \bot$ denotes the ‘skip’ command. Besides the simplicity of this condition, the proof of its idempotence is pleasingly straightforward $\parallel \bot = \parallel \bot$. As we discussed before in Section 1, the proofs of idempotence and commutativity can be otherwise extremely difficult for a real language [5]. This technique has provided a simple solution and a uniform way of writing and reasoning about higher-order healthiness conditions.

Most healthiness conditions cannot be defined in this form with a sequential composition. Fortunately, using generic composition, we can generalise this technique and tackle a variety of more complicated healthiness conditions (see Section 5).

In a real semantics, there can be a mixture of conjunctive, disjunctive and generic healthiness conditions. The following theorems study the conditions under which a healthiness function based on generic composition is idempotent and commute with other healthiness functions.

**Theorem 6** $\lambda X \cdot X : \beta R$ is an idempotent function iff $R : \beta R = R$.

**Theorem 7** If $R = \exists \gamma \exists \beta S$ and $S = \exists \beta \exists \beta S$, then $\lambda X \cdot X : \beta R$ and $\lambda X \cdot X : \gamma S$ commute.
Theorem 8  If $S = \exists \beta \exists S$, then $\lambda X : X : \beta R$ and $\lambda X : X : R \land S$ commute.

Theorem 9  If $S = S : \beta R$, then $\lambda X : X : \beta R$ and $\lambda X : X : R \lor S$ commute.

Theorem 10  If $S = S : \beta R$, then $\lambda X : X : \beta R$ and $\lambda X : X : \beta S$ commute.

In this definition, the binary conditional $A \triangleleft B$ is defined as $(b \land A) \lor (\neg b \land B)$.

4.4 Two-Dimensional Generic Composition

A parallel-via-medium composition in which the components on both sides are identical becomes a more general kind of composition called a two-dimensional generic composition.

Definition 5  The construct $A :: \beta B \triangleq \beta A \parallel \beta A$ is called a two-dimensional generic composition.

If we consider $A$ as a component, a two-dimensional generic composition with another component $B$ compares any two behaviours of $A$ by allowing only some of the observable variables (i.e. in $\beta$) to be different.

This composition enjoys some laws (e.g. associativity) similar to those of basic generic composition in Section 3.3 and Section 4.3.

A healthiness condition based on this composition is normally written in the form $A = A :: \beta R$. The significance is that the component $A$ in the definition appears twice; thus it makes it possible to compare any two values of the variables in $\beta$.

This technique is highly useful when we need to impose a healthiness condition on the range of some variable. A range is the set of the non-deterministic choices of a variable when other variables are fixed. For example, the range of $y$ in predicate $(y \geq x + 1)$ is $\{n \in \mathbb{N} | n \geq 5\}$ if $x$ is fixed on 4.

4.5 Multi-dimensional Generic Composition

We take one further step to allow up-to-infinitely-many copies of a component to appear in a predicate. Note that the following definition may contain an infinite number of variables.

Definition 6  $A :: \beta I B \triangleq \bigwedge_{i \in I} A[\beta_i / \beta] \cup \bigcup_{i \in I} \beta_i B$.

In the definition, $I$ is called an indexing set. Each index $i$ in $I$ corresponds to a copy of $A$ whose variables in $\beta$ are substituted by those in $\beta_i$. This composition enjoys some laws similar to those in Section 3.3 and Section 4.3.

If $I = \{0, 1\}$, this definition collapses to a two-dimensional generic composition; however if $I$ is an infinite set, interface $\{\beta_i | i \in I\}$ from the copies of $A$ will...
form a subset (cardinality up to that of $I$) of the all possible non-deterministic choices; if $I$ is the Cartesian product of the types of the variables from $\beta$, the interface can form any subset of the non-deterministic choices.

A healthiness condition based on this composition is normally written in the form $A = A \land \beta R$. The significance of this composition is that now we can impose healthiness restrictions on the cardinality or limit points of a variable’s range.

5 Applications

In this section we shall demonstrate the expressive power of generic composition by applying it to yield a variety of compositions and higher-order healthiness conditions.

5.1 Parallel Compositions

Conjunction. A (binary) parallel composition demonstrates the behaviour of two components at the same time and therefore becomes a conjunction if there is no interference between the two components.

Proposition 8 $A \land B = A \land B$.

Interference between the inputs. In a parallel-by-merge composition, the merge is composed sequentially after the two components. That means the initial states $x$ of the two components are simply the direct copies of the initial state of the parallel composition. No interference is allowed at the beginning of a parallel composition.

However this may not be general enough in some cases. For example, in the following Petri net, a token in $x$ can be received by either process $A$ or process $B$. It is not compatible with parallel-by-merge in that the number of tokens received by process $A$ (denoted by $x.0$) directly depends on the number of tokens received by process $B$ (denoted by $x.1$).
Fortunately we can define it as a parallel-via-medium composition with an interface containing both input and output variables.

**Definition 7** \( A \parallel B \triangleq A \parallel^M_{\{x,x'\}} B \) where
\[
M \triangleq (x = \overline{x}, 0 + \overline{x}, 1) \land (x' = \overline{x}', 0 + \overline{x}', 1).
\]

### 5.2 Healthiness Conditions Based on Basic Generic Composition

In this section, we will study some healthiness conditions appearing in various semantic models. Each healthiness condition is written in the form \( A = A ;_\beta R \), \( A = A ;_\beta R \) or \( A = A ;_\beta R \). The corresponding monotonic and idempotent healthiness function transforms any unhealthy predicate to a healthy one.

Recall that the range of a variable in a predicate is the set of all possible values of the variable when other variables are fixed.

**Healthiness 1** \( A = A ;_{\{x\}} (x \leq \overline{x}) \).

The ranges of \( x \) in the following healthy predicates can be simply compared by refinement ordering. For example \( (y + 1 > x) \Rightarrow (y + 1 \geq x) \Rightarrow (y + 2 \geq x) \Rightarrow (y + 2 \geq x) \). According to Theorem 6, the healthiness function \( \lambda X . X ;_{\{x\}} (x \leq \overline{x}) \) is idempotent if \( (x \leq \overline{x}) ;_{\{x\}} (x \leq \overline{x}) = (x \leq \overline{x}) \), which is obvious. Healthiness 1 defines a more general downwards-closure if the type of \( x \) is a partial ordering.

**Upwards-closure** can be expressed by replacing \( \leq \) with \( \geq \).

**Comparison of non-determinism.** A similar healthiness condition states that the value of \( y \) is always a possible non-deterministic choice of \( x \).

**Healthiness 2** \( A = A ;_{\{x\}} x \in \{\overline{x}, y\} \).

**Prefix independence.** The healthiness condition \( A = x A[s/tr, s^\wedge (tr' - tr)/tr'] \) for traces mentioned in section 2 now becomes as simple as follows:

**Healthiness 3** \( A = A ;_{(tr, tr')} (tr' - tr = \overline{tr} - \overline{tr}) \).

It is an easy exercise to check that the corresponding healthiness function is indeed monotonic and idempotent.
Healthy Designs. Another healthiness condition $A[\text{false}/ok'] \Rightarrow A[\text{true}/ok']$ for the designs mentioned in Section 2 is redefined:

**Healthiness 4**

$$A = A ;_{\{ok'\}} \ (ok' \Rightarrow ok').$$

If the range of $ok'$ in a predicate $A$ is $\{\text{false}\}$, the generic composition will transform the predicate by adding ‘true’ to the range. Thus a predicate is healthy in this sense iff it is either deterministic on ‘true’ or non-deterministic between ‘true’ and ‘false’.

Prefix closure of strategy. In game semantics [1] a strategy $S$ is a non-empty set of even-length $(2 \mid \text{len}(s))$ sequences satisfying a few healthiness conditions. One of them is the prefix-closure condition:

$$\text{sab} \in S \Rightarrow s \in S.$$  

The generic-composition way of writing this is:

**Healthiness 5**

$$A = A ;_{\{s\}} (2 \mid \text{len}(s) \land s \leq \text{len}).$$

The corresponding healthiness function is monotonic and idempotent and it transforms any unhealthy predicate to a healthy one by adding all even-length prefixes of each sequence.

5.3 Healthiness Conditions Based on Two-Dimensional Generic Composition

Sometimes the component $A$ needs to appear twice in $f(A)$. In that case, two-dimensional generic composition is needed.

Decoupled variables. For example, in distributed computing asynchronous communication exhibits latency. If there is no causal relation established by any communication between two observable variables, we call them **decoupled variables**. The following healthiness condition states that, if $x$ and $y$ are **decoupled** variables, their non-deterministic choices are not directly correlated. That means the range of joint non-deterministic choices of the two variables is exactly the Cartesian product of their separate ranges.

**Healthiness 6**

$$A = A ;_{\{x,y\}} (x \in \{\overline{x}.0, \overline{x}.1\} \land y \in \{\overline{y}.0, \overline{y}.1\}).$$

Let $\{x, y, z\}$ be the set of all free variables of $A$. The above healthiness condition is satisfied iff $A(x, z, y)$ can be decomposed into two predicates $A_1(x, z)$ and $A_2(z, y)$ in conjunction, i.e. $A(x, z, y) = A_1(x, z) \land A_2(z, y)$ where $x$ is not directly related to $y$. 
Determinism. In relational semantics [19], a variable is deterministic if its range is always a singleton set. The semantic space of deterministic relations is not a complete lattice because the glb operator (i.e. relation union) is not closed in the space. However adding the bottom ‘true’ to such a space will create a complete lattice, which can be expressed as:

**Healthiness 7** \[ A = A \lor A \vdash_{\{x\}} (\pi.0 \neq \pi.1). \]

The corresponding healthiness function transforms any deterministic predicate to itself and any non-deterministic predicate to the bottom. Although this definition has an equivalent form \[ A = A \vdash_{\{x\}} (x \in \{\pi.0, \pi.1\} \lor \pi.0 \neq \pi.1), \] Healthiness 7 looks simpler.

The range of \( x \) may be an empty set, a singleton set or the set of all possible values. Although it is not exactly a healthiness condition for pure determinism, their difference lies only at some ‘extreme’ points. This is a cheap price to pay, because what we obtain is a uniform way of writing its healthiness condition and the semantic space now becomes a complete lattice. This technique is called *Complettisation* and has been our standard technique to deal with non-complete-lattice semantic models [5].

Deterministic strategies. In game semantics [1], a player and its opponent act alternatively in a play. The player may play deterministically according to the pre-history: if both \( sab \) and \( sac \) are healthy (even-length) sequences of a strategy, they must be identical, i.e. \( b = c \). This healthiness condition is defined as follows:

**Healthiness 8** \[ A = A \lor A \vdash_{\{x\}} \exists abc (\pi.0 = tab \land \pi.1 = tac \land b \neq c). \]

The corresponding healthiness function transforms any deterministic strategy to itself and any non-deterministic one to the bottom predicate ‘true’. The *Innocence* condition in game semantics [1] is very similar to Healthiness 8. Other healthiness conditions in [1] including bracketing condition can be trivially defined as conjunctive conditions.

Convexity of probabilistic distributions. In probabilistic semantics [8], the value of a variable \( f \) is a probabilistic distribution function \( f : W \to [0, 1] \) where \( W \) is the set of all states. The following healthiness condition states that the distributions of variable \( f \) form a convex area in the space containing all possible distribution functions.

**Healthiness 9** \[ A = A \vdash_{(f)} (\exists \lambda : [0, 1] \cdot (f = \lambda \ast \mathcal{I}.0 + (1 - \lambda) \ast \mathcal{I}.1)). \]

The corresponding healthiness function transforms a predicate by adding intermediate linear combinations of every two distributions into the range.
5.4 Healthiness Conditions Based on Infinite-Dimensional Generic Composition

If infinitely many copies of component $A$ are allowed to appear in a healthiness condition, we will be able to reason about the cardinality of a range and the limit points (also called ‘cluster points’) in a range.

Closed sets and safety properties. Let the type of variable $x$ be $X$ where $(X, T)$ is a topological space. The following healthiness condition requires that the range of the variable is always a closed set.

Healthiness 10

$$A = A \land A : \{x\} \forall O : T \cdot (O \circ \Sigma^X \Rightarrow O = \emptyset)$$

where $\Sigma^X = \{\tau, i \mid i \in X\}$.

There are as many as cardinal$X$ copies of $A$’s appearing in this definition. Thus $\Sigma^X$ can be any non-empty subset of $x$’s range. The corresponding healthiness function of this condition transforms any predicate with not closed range to a predicate with the smallest closed range containing the original range.

In the semantic models based on temporal logic (e.g. TLA [16] and UNITY [3]), a reactive process is an infinite sequence of states, each of which has a type $T$ and hence the space of all processes is $X = T^\omega$. There is a natural topology on $X$ in which each basic open set is the set of all infinite sequences sharing a common prefix and each open set is the union of some basic open sets. It has been proved [2] that safety properties are exactly the closed sets of the topology, (pure) liveness properties are exactly the dense sets, and any property can be expressed as the intersection of a safety property and a liveness property.

Thus the set of all predicates satisfying Healthiness 10 on a sequence variable $x$ forms a semantic space for all safety properties. The corresponding healthiness function transforms any property to the strongest safety property implied by it.

Dense sets and liveness properties. A dense set intersects every non-empty open set. The following healthiness condition requires the range of variable $x$ to be a dense set:

Healthiness 11

$$A = A \land A : \{x\} \forall O : T \cdot (O \circ \Sigma^X \Rightarrow O = \emptyset)$$

There are as many as cardinal$X$ number of copies of $A$ appearing in the definition. The corresponding healthiness function (monotonic and idempotent) transforms any predicate with a dense range to itself and unhealthy predicate to the top ‘false’. Again we need the help of one extra point ‘false’ to make the space to become a complete lattice, because the intersection of dense sets may not be dense.

Another interesting way to write a healthiness condition for liveness is:

$$A = A : \{x\} \forall x \notin (\text{closure} \Sigma^X - \Sigma^X)$$
Its corresponding healthiness function together with that of Healthiness 10 decomposes any property into a pair of safety and liveness properties in conjunction. Although this function is idempotent, it is not monotonic and the corresponding semantic space is not a complete lattice.

**Cauchy-closure of probabilistic distributions.** Healthiness 9 requires the range of a variable to be a convex area. However the boundary points (i.e. limit points) may or may not be included in the range. The Cauchy-closure healthiness condition in [8] requires those boundary points to be included. To reason about limit points, we need an infinite-dimensional generic composition. Recall that the type of a variable \( f \) is a set of total functions \( \mathcal{F} \equiv W \rightarrow [0,1] \) (i.e. probabilistic distribution). If \( W \) is a finite set, \( \mathcal{F} \) is isomorphic to a \( n \)-dimensional Euclidean space where \( n \) is the cardinality of \( W \). Cauchy-closure is then defined by:

\[
\text{Healthiness 12 } A = A \uplus \{ f \}^\mathcal{F} \quad (\forall \varepsilon > 0 \exists i \cdot | f_i - f | < \varepsilon)
\]

In the definition, \( | \cdot | \) is a Euclidean distance measure associated with the Euclidean space. The corresponding healthiness function transforms any predicate by 'pasting' those boundary points to the range of variable \( f \).

If the state space is the set of real numbers (i.e. \( W = \mathbb{R} \)) and any \( f : \mathcal{F} \) is integrable and \( \int_{-\infty}^{+\infty} f(t)dt \leq 1 \), the distance is defined by \( | f | \equiv \left( \int_{-\infty}^{+\infty} f^2(t)dt \right)^{1/2} \) and Healthiness 12 is still applicable.

**Finitary condition of relations.** Another important healthiness condition of relational semantics [19] is called the **finitary condition**, which corresponds to the \( \bigvee \)-continuity condition of predicate-transformer semantics [7]. A variable is finitary if its range is either a finite set or the set of all possible values.

\[
\text{Healthiness 13 } A = A \lor A :\{ x \}^\mathcal{R} \quad (\text{cardinal } \Sigma_x^\mathcal{R} = \omega).
\]

There are up-to-\( \omega \) number of the copies of \( A \) appearing in the definition where \( \omega \) is the cardinality of \( \mathbb{N} \). Its corresponding healthiness function transforms a non-finitary predicate by pasting all other possible values to the range of \( x \).

## 6 Conclusions and Acknowledgement

Our objective is to aid semantic studies on real programming languages. The introduction of generic composition serves this purpose by unifying various semantic definitions. The definition of parallel composition and the presentation of its associativity become more general and simpler than parallel-by-merge. The technique also provides a unified form for defining healthiness conditions. The completeness theorem shows the evidence of its generality in theory. The examples from a variety of theories have shown its generality in practice.

This paper is part of the author’s D.Phil thesis. The author would like to thank his supervisor J. W. Sanders for the discussions and encouragement and the anonymous referees for their helpful comments.
References


Abstract. The aim of this paper is the introduction of preemption in a compositional model, called M-nets, which is based on Petri nets and hence provided with a concurrent semantics. We propose a way to model preemptible systems by extending the M-net model with priorities and the M-net algebra with a preemption operator. We show that these extensions can be seen as a high-level version of the well studied model of priority systems, and so, can be reduced to Petri nets (without priorities) which retain as much as possible of the original concurrency. As a consequence, Petri nets appear as a model powerful enough to deal with preemption in a compositional way and with a concurrent semantics.

Keywords. Petri nets, Preemption, Concurrency, Compositionality.

1 Introduction

Preemption relates to controlling the execution of the processes composing a concurrent system. Such processes are said preemptible if they can be suspended at any point of their execution.

Preemption is often addressed in reactive systems, for instance in synchronous models and languages [14,1]: for some of them, it is even an essential feature. In most cases, the underlying semantics is sequential, which is well suited to the modeling of systems in which the computation performed in response to an input coming from the environment is relatively simple. But when the structure of the computation becomes more important than the structure of the reaction, the sequential semantics may be not sufficient. A concurrent semantics is often more adapted to the modeling of heterogeneous architectures which combine software (distributed on several processors) and specialized hardware components. In particular, playing with the scheduling of operations often allows a better resource management.

Petri nets form an inherently asynchronous model in which concurrency can be represented explicitly. This model and some of its extensions [22,23] have been used for works on preemption, but in an unstructured way (non compositional).

This paper addresses the question of preemption in the context of compositional Petri nets. This naturally leads to consider the framework defined by the
Petri Box Calculus (PBC [3,2]). The proposed approach tends to be as conservative as possible with respect to the existing framework, with the goal to minimize the changes necessary in order to adapt the existing tools to the proposed model.

PBC is a process algebra with a syntactic domain of box-expressions and a corresponding semantic domain of boxes, a class of labelled 1-safe Petri nets provided with an algebraic structure. It has been introduced with the aim of modeling the semantics of concurrent systems and programming languages. In order to cope with the possibly huge size of the nets, higher level versions of PBC have been considered, and in particular an algebra of M-expressions (high-level equivalent of box-expressions [18,16]) and M-nets (high-level Petri net version of boxes [4]) which allow to represent large (possibly infinite) systems in a clear and compact way. The high- and low-level domains are related by an operation of unfolding which associates a box-expression to each M-expression and a box to each M-net.

The PBC framework also features a parallel programming language, $B(PN)^2$ [5], which can be seen as a “user friendly” syntax on the top of both, high- and low-level process algebras. It is implemented in PEP toolkit [13], allowing to simulate modeled systems and to verify their properties via model checking. Several contributions [5,4,20,12,19,15] provide applications to the PBC theory where box-expressions, M-nets and M-expressions are used as the semantical domain for $B(PN)^2$.

In this paper, the M-net model is extended by considering priority M-nets as pairs $(N, \rho)$ where $N$ is an M-net and $\rho$ a pairwise priority relation between its transitions. The M-net algebra is then enriched by a new operation, $\pi$, which allows to make preemptible any priority M-net and can be nested arbitrarily.

We are particularly interested in a sub-class of priority M-nets, called preemptible M-nets (P/M-nets), which fulfill some structural constraints. We show that the concurrent (step) semantics of P/M-nets is sound with respect to the semantics of preemption.

Moreover, applying results obtained in some related areas, we show for a large class of P/M-nets, that they can be transformed into 1-safe Petri nets (without priorities), retaining as much as possible of the concurrent semantics. The transformation leads to really huge nets which cannot be used in practice, nevertheless, this means that 1-safe Petri nets are expressive enough to model preemption in a compositional framework.

The rest of the paper is organized as follows. Section 2 gives some intuition about the aspects of M-nets which are important for our purpose. Section 3 discusses preemption and the impact of its introduction in the context of Petri nets. Section 4 introduces priority M-nets, defines operation $\pi$, and extends the usual M-net operations to such nets. Then, P/M-nets are introduced as a structurally restricted class of priority M-nets, and their concurrent semantics is shown sound with respect to the semantics of preemption. A detailed example with nested P/M-nets is given in section 5. Section 6 discusses some properties of P/M-nets and, in particular, their transformation to 1-safe Petri nets. The paper ends with some concluding remarks.
2 M-Net Model

2.1 Basic Definitions

Let $E$ be a set. A multi-set over $E$ is a function $\mu : E \to \mathbb{N}$, generally denoted with an extended set notation, e.g., $\{a, a, b\}$ for $\mu(a) = 2$, $\mu(b) = 1$ and $\mu(e) = 0$ for all $e \in E \setminus \{a, b\}$. $\mu$ is finite if so is its support set $E \setminus \mu^{-1}(0)$. We denote by $\mathcal{M}(E)$ (resp. $\mathcal{M}_f(E)$) the set of multi-sets (resp. finite multi-sets) over $E$, by $\oplus$ and $\ominus$ the sum and difference of multi-sets. We may also use the usual set notations such as $\subseteq$ or $\in$; for instance, $e \in \mu$ stands for $\mu(e) > 0$.

2.2 M-Nets

M-nets [4] form a class of high-level Petri nets provided with a set of operations giving them a structure of process algebra. We use here the M-net model defined in [8], and its asynchronous links extension from [17].

An M-net $N$ is a triple $(S, T, \iota)$, where $S$ is the set of places, $T$ is the set of transitions, $(T \times S) \cup (S \times T)$ is the set of arcs, and $\iota$ is the annotation function on places, transitions and arcs. The annotation of a place is of the form $\lambda.\tau$, where $\lambda$ is a label (entry $e$, exit $x$ or internal $i$) and $\tau$ is a type (a non-empty set of values from a fixed set $\text{Val}$). As usual, for each node (place or transition) $r \in S \cup T$, we denote by $\bullet r$ the set of nodes $\{r' \in S \cup T \mid \iota(r', r) \neq \emptyset\}$ and, similarly, $r \bullet = \{r' \in S \cup T \mid \iota(r, r') \neq \emptyset\}$.

Transitions annotations are of the form $\lambda.\gamma$ where $\lambda$ is a label (which can be hierarchical or for communications) and $\gamma$ is a guard (a finite set of predicates from a set $\text{Pr}$). Hierarchical labels are composed out of a single hierarchical action (e.g., $X$) indicating a future refinement (i.e., a substitution) by an M-net. Communications may be:

- **synchronous**, similar to CCS ones [21], e.g., between transitions labelled by synchronous communication actions such as $A(a_1, \ldots, a_n)$ or $\hat{A}(a'_1, \ldots, a'_n)$, where $A$ is a synchronous communication symbol, $\hat{A}$ is its conjugate and each $a_i$ and $a'_i$ is a value or a variable (belonging to a fixed set $\text{Var}$);

- **asynchronous**, e.g., between transitions labelled by asynchronous links such as $b^+(a_1)$ or $b^-(a_2)$, where $b$ is an asynchronous communication symbol and each $a_i$ is a value or a variable (ranging in type$(b) \subseteq \text{Val}$). The communication is done via a place $s_b$ of type $\tau(s_b) = \text{type}(b)$ which plays the role of a heap buffer. Link $b^+(a_1)$ means that $a_1$ can be sent to $s_b$ and $b^-(a_2)$ means that $a_2$ can be received from $s_b$;

- or possibly both at the same time.

Communication labels are then of the form $\lambda = \alpha.\beta$ where $\alpha$ is a finite multi-set of synchronous communication actions and $\beta$ is a finite multi-set of asynchronous links.

Arcs are inscribed by multi-sets of **structured annotations** representing multi-sets of values consumed or produced by a transition in a place. Structured annotations are variables, values or more complex structures allowing to cope with
place types generated by refinements [8,10]. As usual, in figures, arcs with an empty annotation will be omitted; moreover, annotation $\bullet$ on an arc is omitted most of time and singletons are often replaced by their unique element.

2.3 Dynamic Behavior and Concurrent Semantics

For each transition $t \in T$ we shall denote by $\var(t)$ the set of all the variables occurring in the annotations of $t$ and in the arcs coming to and from $t$. A binding for a transition $t$ is a substitution $\sigma : \var(t) \rightarrow Val$; it will be said enabling if it satisfies the guard, if it respects the types of the asynchronous links, and if the flow of tokens it implies respects the types of the places adjacent to $t$.

A marking of an M-net $(S,T,\iota)$ is a mapping $M : S \rightarrow M(Val)$ which associates to each place $s \in S$ a multi-set of values from $\tau(s)$. In particular, we shall distinguish the entry marking, denoted $M_e$, where, for each $s \in S$, $M_e(s) = \tau(s)$ if $\lambda(s) = e$ and the empty multi-set otherwise; the exit marking is defined similarly. The dynamic behavior of an M-net starts with its entry marking; it ends (if ever) with the exit marking.

The transition rule specifies the circumstances under which a marking $M'$ is reachable from a marking $M$. A transition $t$ is enabled at a marking $M$ (this is denoted $M[t]$) if there is an enabling binding $\sigma$ of $t$ such that $\forall s \in S : \iota(s,t)[\sigma] \subseteq M(s)$, i.e., there are enough tokens of each type to satisfy the required flow. The effect of an occurrence of $t$ is to remove from its input places all the tokens used for the enabling binding $\sigma$ and to add to its output places the tokens according to $\sigma$; this leads to a marking $M'$ such that $\forall s \in S : M'(s) = M(s) \ominus \iota(s,t)[\sigma] \oplus \iota(t,s)[\sigma]$.

The above transition rule defines the interleaving semantics of an M-net which consists in a set of occurrence sequences. This semantics can be generalized by introducing the step sequence semantics [9], which allows any number of transitions to occur simultaneously.

Given an M-net $N = (S,T,\iota)$, a multi-set $\delta$ of transitions is said concurrently enabled at a marking $M$ if there are enough tokens to allow the simultaneous firing of all the transitions in $\delta$. Such a $\delta$ is called a step. A step sequence of $N$ is a sequence $D = (\delta_1,\delta_2,\ldots)$ such that there are markings $M_1, M_2, \ldots$, where $M_1 = M_e$ and which satisfy $M_i[\delta_i]M_{i+1}$ for $i \geq 1$. The set of step sequences of $N$ is its step sequence semantics and is denoted by $\text{steps}(N)$. It is easy to see that $\text{steps}(N)$ is stable under linearisation: if $\delta$ belongs to a step sequence $D$ in $\text{steps}(N)$, then, replacing $\delta$ with any of its linearisation gives a step sequence which is also in $\text{steps}(N)$ (e.g., $\{t_1,t_2\}$ can be replaced by $\{t_1\}{t_2}$ or $\{t_2\}{t_1}$).

2.4 Unfolding

Let $N = (S,T,\iota)$ be an M-net. The unfolding of $N$ is the labelled Petri net $\mathcal{U}(N) = (\mathcal{U}(S),\mathcal{U}(T),W,\lambda)$, where $\mathcal{U}(S)$ is the set of places, $\mathcal{U}(T)$ the set of transitions, $W$ the weight function on arcs and $\lambda$ the labelling function on places and transitions, defined as follows:
- \( \mathcal{U}(S) = \{ (s, v) \mid s \in S \text{ and } v \in \tau(s) \} \), and \( \forall (s, v) \in \mathcal{U}(S) : \lambda((s, v)) = \lambda(s) \);
- \( \mathcal{U}(T) = \{ (t, \sigma) \mid t \in T \text{ and } \sigma \text{ is an enabling binding of } t \} \), and \( \forall (t, \sigma) \in \mathcal{U}(T) : \lambda((t, \sigma)) = \lambda(t) \);
- \( \mathcal{U}(T) = \{ (t, \sigma) \mid t \in T \text{ and } \sigma \text{ is an enabling binding of } t \} \), and \( \forall (t, \sigma) \in \mathcal{U}(T) : \lambda((t, \sigma)) = \lambda(t) \);
- \( W((s, v), (t, \sigma)) = \sum_{x \in \iota(s, t)} \iota(s, t)(x) \cdot x[\sigma](v) \), where \( x[\sigma](v) \) is the number of values \( v \) occurring in the structured annotation \( x \) evaluated under \( \sigma \); \( W((t, \sigma), (s, v)) \) is defined analogously.

If \( M \) is a marking of \( N \), the marking \( \mathcal{U}(M) \) of \( \mathcal{U}(N) \) is defined as follows: for every place \( (s, v) \in \mathcal{U}(S) \), \( \mathcal{U}(M)((s, v)) = M(s)(v) \), i.e., each low-level place \( (s, v) \in \mathcal{U}(S) \) contains as many tokens as the number of occurrences of \( v \) in the marking \( M \) of \( s \).

The unfolding can easily be extended to steps and step sequences, and one can observe that the step semantics obtained by unfolding the step semantics of an M-net \( N \) equals the step semantics obtained from \( \mathcal{U}(N) \).

**Theorem 1.** Let \( N \) be an M-net. Then, \( \mathcal{U} \text{steps}(N)) = \text{steps}(\mathcal{U}(N)) \).

**Proof.** By definition of the unfolding and by the analogous property for interleaving semantics [4].

### 2.5 Algebra of M-Nets

For compositionality, we are particularly interested in a sub-class of M-nets: we assume that each M-net has at least one entry and one exit place, that each transition has at least one input and one output place (T-restrictness property), and that there are neither arcs going to entry places nor from exit places. Such M-nets are said ex-good.

The algebra of ex-good M-nets comprises the operations listed below, where \( N_1, N_2 \) and \( N_3 \) are M-nets, \( \mathcal{X} \) is a hierarchical symbol, \( A \) is a synchronous communication symbol, \( b \) is an asynchronous link symbol and \( f \) is a renaming function on synchronous and asynchronous symbols. Detailed explanations and some examples of these operations are given in [4,10,17].

- \( N_1[\mathcal{X} \leftarrow N_2] \) refinement
- \( N_1 || N_2 \) parallel composition
- \( N_1; N_2 \) sequence
- \( N_1 \| N_2 \) choice
- \( [N_1 * N_2 * N_3] \) iteration

In the following the considered M-nets are ex-good, except if specified explicitly.

### 3 Preemption

In the following, we make a difference between programs (nets) and processes (actually, step sequences) which are their dynamic behaviors; each step sequence being a possible execution of its supporting net.
3.1 Abortion Versus Suspension

When dealing with preemption, two notions are usually separated: suspension which “freezes” an execution but keeps it alive for a possible restart, and abortion which kills an execution definitively. Our approach deals with both of them, but focusing on abortion. More precisely, we treat abortion as a suspension followed by some processing in order to remove all the tokens from the net which supports the execution, making it unable to evolve anymore. This solution is based on priorities which make the problem of suspension quite straightforward to solve: in order to suspend a given net, it is enough to enable one of its transitions which has the priority over all the others. Actually this is what we propose: during all the abortion stage, there is always such a transition which is enabled and thus freezes the rest of the net when it is being emptied.

3.2 Preemption and Time

Preemption is often associated to time, at least intuitively, because it is expected to have an immediate effect on a system. As far as Petri nets are concerned, “immediate” means that no program transition may fire, from the beginning until the end of the preemption. Of course, in some other contexts, introducing time together with preemption makes sense, but it is not the case for our purpose. This is the reason why this paper never deals with time or time related concepts.

3.3 Internal Versus External Abortion

From the point of view of an execution, there is a difference between an internal abortion (when the execution “decides” to give-up its current work) and an external abortion (when the execution is killed by its environment). In both cases, the execution is suspended (i.e., no further transition in the corresponding net is allowed to fire, except, possibly, some well identified transitions involved into the abortion itself), and its supporting net must be “emptied” (i.e., tokens must be removed from the net). All the tokens have to be removed: first, because they make alive the execution being killed, and moreover, because the net must be cleaned up for a possible future re-usage, as the support of another execution.

In the case of an internal abortion, however, the M-net must not be completely emptied because its environment is not aware of the abortion and so waits for its completion. Actually, the environment is expecting the exit marking of the net which supports the aborted execution. As a consequence, the case of internal abortion corresponds to an anticipated termination, which reflects the abortion of the execution, followed by its completion through the production of the exit marking. In a way, internal abortion is a simple mean by which an execution can terminate “cleanly”, regardless of its current state.

In the case of an external abortion, various behaviors may be acceptable since the environment is aware of the situation (by definition of external abortion, it is initiated by the environment). In our approach, we choose a solution where the aborted net fires a particular transition which warns its environment that
the abortion stage is over, but the exit marking is not reached. This way allows our construction to manage the abortion of nested executions quite elegantly: abortion is transmitted from the top to the bottom, from external executions to nested ones.

3.4 Modeling Preemption in Petri Nets

A preemptible Petri net should be able to run under two mutually exclusive modes: in “standard mode”, it processes its program; in “abortion mode”, it must stop its normal activity, empty and complete (reaching or not its exit marking). Abortion mode can interrupt standard mode, but not the reverse.

On the one hand, we can consider that, before any move, standard mode checks that abortion mode is disabled. The execution checks the absence of token in the net part which supports abortion mode. This is a zero test, which can be modeled by introducing inhibitor arcs or complementary places. In this point of view, standard mode is responsible for freezing itself when necessary. On the other hand, we can consider that abortion mode has the priority over normal mode: if, in the net, transitions $t_n$ for normal mode and $t_a$ for abortion mode are both enabled, $t_a$ should be always preferred. This point of view naturally leads to consider priorities between transitions. Here, standard mode is completely passive with respect to its freezing.

In this paper, we prefer the second point of view since it allows us to bring to the theory of priority systems as presented in [6]. Actually, M-nets extended with priorities and with a suitable definition of unfolding, lead directly to priority systems. This allows us to apply the main result from [6], which consists in transforming a priority system $(\Sigma, \rho)$ in a Petri net $\Sigma_p$ which is derived from $\Sigma$ in such a way that it preserves as much as possible of the concurrency in $\Sigma$ and does not violate the priority constraints specified by $\rho$.

4 Preemptible M-Nets

The purpose of this section is a definition of preemptible M-nets ($P/M$-nets for short), a class of composable M-nets provided with some additional information about priority between transitions and having some structural properties which ensure the soundness of their step semantics with respect to the semantics of preemption. For the definition of $P/M$-nets we proceed as follows: first, we consider an auxiliary and very powerful class of nets called priority M-nets, analogous to priority systems from [6], as M-nets equipped with a pairwise priority relation between their transitions. The transition rule of these nets takes into account the information about priority and so does the step semantics. Then, we extend M-net operations to priority M-nets, giving to them an algebraic structure, and we define a new operation for priority M-nets, called $\pi$, which serves to make preemptible any priority M-net. Finally, as the main definition of this section, we introduce $P/M$-nets as a sub-class of priority M-nets having interesting structural properties with respect to preemption.
4.1 Pairwise Priorities

Let $N = (S, T, i)$ be an M-net. A binary relation $\rho \subseteq T \times T$ is called a pairwise priority relation. Intuitively, $(t_1, t_2) \in \rho$ means that during an execution of $N$, the firing of transition $t_2$ is always preferred to $t_1$ when both are enabled; in other words, $t_1$ has a lower priority than $t_2$. We use standard mathematical notations, in particular, for $\rho \subseteq T \times T$, we denote:

$$\text{dom} \ (\rho) = \{ t_1 \in T \mid \exists t_2 \in T \text{ such that } (t_1, t_2) \in \rho \}$$

$$\text{cod} \ (\rho) = \{ t_2 \in T \mid \exists t_1 \in T \text{ such that } (t_1, t_2) \in \rho \}$$

4.2 Priority M-Nets

A priority M-net is a pair $P = (N, \rho)$ where $N = (S, T, i)$ is an M-net (possibly having some non T-restricted communication transitions) and $\rho \subseteq T \times T$ is a pairwise priority relation over $T$. We call $N$ the net part of $P$.

**Definition 1.** Let $P = (N, \rho)$ be a priority M-net, $M$ a marking of $N = (S, T, i)$ and $t$ a transition of $N$ such that $M[t]$; then $t$ is $\rho$-enabled in $P$ at $M$, denoted $M[t]_{\rho}$, if $\not\exists t' \in T$ such that $M[t']$ and $(t, t') \in \rho$.  

Notice that $\rho$ allows to disable a transition which would have been enabled with usual M-nets transition rule, but not the reverse. In other words, we have $M[t]_{\rho} \Rightarrow M[t]$.

The notion of step and step sequence defined for M-nets could be directly reused for priority M-nets. But, this way, they would lead to inconsistencies in the semantics. Consider for example the priority M-net $P = (N, \rho)$ shown in figure 1 (taken from [6]), if we do not take $\rho$ into account, we have the step semantics:

$$\text{steps} \ (N) = \{ \emptyset, \{ t_1 \}, \{ t_3 \}, \{ t_1, t_3 \}, \{ t_3 \}, \{ t_1 \}, \{ t_1 \} \{ t_2 \} \},$$

where $\emptyset$ is the empty step sequence.  

We can see that it contains the sequence $\{ t_1 \} \{ t_3 \}$ which violates $\rho$. Removing this sequence is necessary but not enough since it introduces inconsistency. Actually, the semantics cannot contain $\{ t_1 \} \{ t_3 \}$ because $\{ t_1 \} \{ t_3 \}$ is one of its linearisations. The consistent step semantics of $P$, denoted $\text{steps} \ (P)$, is thus the biggest sub-set of $\text{steps} \ (N)$ such that each step sequence $D \in \text{steps} \ (P)$ and each of its linearisations respect $\rho$. So, we have:

$$\text{steps} \ (P) = \{ \emptyset, \{ t_1 \}, \{ t_3 \}, \{ t_3 \} \{ t_1 \}, \{ t_1 \} \{ t_2 \} \}.$$
According to [6], this consistent step semantics is one of the most concurrent semantics one can expect for priority systems.

The unfolding of priority M-nets is a natural extension of the unfolding of M-nets.

**Definition 2.** Let $P = (N, \rho)$ be a priority M-net. The unfolding of $P$, $U(P)$, is a pair $(U(N), U(\rho))$ where $U(N)$ is the usual M-net unfolding and $U(\rho)$ is defined as the smallest set such that: for each pair $(t, t') \in \rho$ such that $t$ is unfolded into a set of low-level transitions $\{(t, \sigma_1), \ldots, (t, \sigma_n)\}$ and $t'$ is unfolded into $\{(t', \sigma_1'), \ldots, (t', \sigma_k')\}$, we have $\{(t_i, \sigma_i), (t'_j, \sigma'_j)\}$ $1 \leq i \leq n$ and $1 \leq j \leq k \subseteq U(\rho)$. If $M$ is a marking of $N$, then $U(M)$ is defined as it is for M-nets.

As for M-nets, an extension of the unfolding of priority M-nets to consistent steps and consistent step sequences is straightforward and we still have:

$$U(\text{steps}(P)) = \text{steps}(U(P)).$$

### 4.3 Algebra of Priority M-Nets

The extension of usual M-net operations to priority M-nets is immediate for most of them. However, in the case of synchronization or refinement, several possible definitions of priority relation can be considered. Our choice is not the most general possible, we already have in mind the definition of operation $\pi$ and of P/M-nets. Priority M-nets are just an intermediate step which avoids circular definitions.

**Definition 3.** Let $P_i = (N_i, \rho_i)$, for $i \in \{1, 2, 3\}$, be priority M-nets, where $N_i = (S_i, T_i, \iota_i)$, and let $X$ be a hierarchical symbol, $A$ a synchronous communication symbol, $b$ an asynchronous link symbol, and $f$ a renaming function on communication symbols. The usual M-net operations are extended as follows for priority M-nets:

- $P_1[X \leftarrow P_2] = (N_1[X \leftarrow N_2], \rho)$ where $\rho = \{(t, t') \in \rho_1 | \lambda_1(t) \neq X \neq \lambda_1(t')\}$
  $\cup \{(t, t') \in \rho_2 \land t, t' \in T_1 \land \lambda_1(t) = X\}$
  $\cup \{(t, t') | t \not\in \text{cod}(\rho_2) \land (t, t') \in \rho_1 \land t, t' \in T_1 \land \lambda_1(t) = X\}$;

- $P_1 \text{tie } b = (N_1 \text{tie } b, \rho_1)$;

- $P_1[f] = (N_1[f], \rho_1)$;

- $P_1 \text{sy } A = (N_1 \text{sy } A, \rho)$ where $N_1 \text{sy } A = (S, T, \iota)$ and $\rho$ is is the smallest set including $\rho_1$ such that if $t' \in T$ results from a basic synchronization of $t_1$ with $t_2$, and
  - if $\exists t''$ such that $(t_1, t'') \in \rho$ or $(t_2, t'') \in \rho$, then $(t', t'') \in \rho$,
  - if $\exists t''$ such that $(t_1, t) \in \rho$ or $(t_2, t') \in \rho$, then $(t', t') \in \rho$.

- $P_1 \text{rs } A = (N_1 \text{rs } A, \rho)$, where $N_1 \text{rs } A = (S, T, \iota)$ and $\rho = \rho_1 \cap (T \times T)$.

Control flow operators (sequential composition $;$, iteration $([* *])$, parallel composition $\langle\rangle$ and choice $\square$) are based on refinement and so defined canonically. Scoping is defined as a synchronisation followed by a restriction: $[A : P] = (P \text{sy } A) \text{rs } A$. 


Fig. 2. Example of synchronization of priority M-nets. (Only synchronous labels are represented.) Restricting on $A$ would remove from the net $t_1$ and $t_2$ (with their surrounding arcs) and $(t_3, t_2)$ from its priority relation.

Fig. 3. $N_\pi$, net part of $P_\pi$ where type($c$) = $\mathbb{N}$ and $i(i_2) = i_2.$ Large black border of some transitions just indicates that they belong to $\text{cod}(\rho_\pi).$ Inscriptions on the arcs are simplified: singletons are replaced by their unique element.

4.4 A New Operation for Preemption

In order to define operation $\pi$, we use the priority M-net $P_\pi = (N_\pi, \rho_\pi)$ where $N_\pi$ is represented in figure 3 and the priority relation is

$$\rho_\pi = \{(t_7, t_4), (t_7, t_5), (t_8, t_4), (t_8, t_5), (t_\emptyset, t_5), (t_2, t_3)\}.$$

The usage we make of $N_\pi$ is rather simple, even if the net may look quite complex:
the top part \((e_1, t_N, i_1)\) embeds the net \(N\) from \((N, \rho)\) which should be made abortable. When \(N\) terminates normally (with no preemption), transition \(t_0\) fires, consuming the token in place \(c_2\) and producing the exit marking in \(x\);

- all the rest is used for preemption: internal abortion starts with a firing of \(t_1\) which produces a token in place \(i_1\); external abortion starts when \(t_2\) or \(t_3\) fires, producing a token \(\circ\). These two tokens allow to start the emptying of \(N\); they are different in order to know, after the emptying is done, if \(N_\pi\) may terminate producing the exit marking (for \(\bullet\)). with transition \(t_7\) or should be completely emptied (for \(\circ\), with \(t_8\));

- transition \(t_1\) is for internal preemption: it will be synchronized with all transitions in \(N\) having a synchronous action \(\text{quit}\) in their label. This way, \(N\) may abort itself by firing such a transition. After \(t_1\) has fired and \(N\) has been completely emptied, \(t_7\) can fire and terminate \(N_\pi\);

- transition \(t_2\) is for external abortion: it may be synchronized later on with a transition such as \(t_5\), coming from another \(N_\pi\) in which the present one is nested. This synchronization is not yet possible on the net of figure 3, but a further renaming \(\text{kill}' \mapsto \text{kill}\) will allow it. After the emptying was completed, transition \(t_8\) can fire and fully empty \(N_\pi\); \(t_8\) is intended to be synchronized with a transition such as \(t_6\) and, here again coming from an external \(N_\pi\); it will be made possible thanks to a further renaming \(\text{empty}' \mapsto \text{empty}\);

- transition \(t_3\) is similar to \(t_2\) but is used when an external abortion occurs while an internal one is already in progress. It just replaces the token \(\bullet\) in \(i_2\) with a token \(\circ\). This corresponds to a switch from internal to external abortion mode. The priority \((t_2, t_3)\) in \(\rho_\pi\) ensures that \(t_3\) is always preferred to \(t_2\) when both are enabled;

- emptying is performed by transitions \(t_4, t_5, t_6\). We already had an intuition about the rôle of \(t_5\) and \(t_6\): the former triggers an abortion into a \(N_\pi\) nested in \(N\), doing this, it increments a counter handled through the links on \(c\) (this counter was initialized to zero by either \(t_1\) or \(t_2\)); the latter allows such an aborted net to empty completely (by firing its transition \(t_8\)), decrementing the counter;

- the counter in \(c\) of killed sub-nets is used to ensure that each aborted sub-net is also emptied: as stated in definition 4, \(t_4\) and \(t_5\) have the priority over \(t_7\) and \(t_8\), but \(t_6\) has not. So, any possible firing of \(t_4\) and \(t_5\) will be done before \(t_7\) or \(t_8\) have a chance to fire, then they must wait for \(t_6\) to perform all the needed (and possible) emptying and thus to decrease the counter until zero. Finally, and not before, \(t_7\) or \(t_8\) is allowed to perform communication \(c \xleftarrow{} (0)\);

- the way \(t_4\) works is not directly visible in \(N_\pi\). Roughly speaking, for every place \(s\) in \(N\), we will add an emptying transition \(t_4\) with a synchronous action \(\text{abort}\) in its label and an arc \(s \rightarrow t_4\) labelled with a singleton \(\{a\}\), where \(a\) is an arbitrary variable. These transitions will be synchronized with \(t_4\) and so, the loop on \(t_4\) will be used to empty \(N\), taking tokens one by one;

- for convenience, the type of \(c\) has been set to \(\mathbb{N}\) which leads, after unfolding, to an infinite number of places in the obtained low-level net. Fortunately, this type can easily be bounded in practice.
In order to have things working properly, we assume that actions symbols \texttt{quit}, \texttt{kill}, \texttt{kill'}, \texttt{empty}, \texttt{empty'} and \texttt{abort}, their conjugated symbols and link symbol \texttt{c} are reserved for operation \( \pi \), and so are never used somewhere else.

Operation \( \pi \) relies on \( P_\pi \): it first refines the net which should be made preemptible into \( P_\pi \) and then adds the emptying transitions (such as \( t_s \)) described above (and formalized below). Scoping on \texttt{quit} allows internal preemption, scoping on \texttt{abort} allows to control the emptying transitions and scoping on \{\texttt{kill}, \texttt{empty}\} allows the transmission of abortion to the potentially nested priority M-nets. A \texttt{tie} on \texttt{c} is also made so the counter can work properly. Finally, actions \( ^\hat{\texttt{kill}}' \) and \( ^\hat{\texttt{empty}}' \) are renamed in order to allow the result to be nested in another \( \pi \).

\textbf{Definition 4.} Let \( P \) be a priority M-net. Then,

\[
\pi(P) = \left[ \{ \texttt{abort, quit, kill, empty} \} : Ab(P_\pi[X \leftarrow P]) \right] \text{tie } c \right]
\]

where \( P_\pi \) is the priority M-net defined above and \( Ab \) is an auxiliary operation which includes the additional emptying transitions; if \( P_\pi[X \leftarrow P] = P' = (S', T', \iota'), \rho' \), then \( Ab(P') = ((S'', T'', \iota''), \rho'') \) with:

- \( S'' = S' \), and \( \forall s \in S'' : \iota''(s) = \iota'(s) \);
- \( T'' = T' \uplus T_s \) where \( T_s = \{ t_s | s \in S' \setminus \{ x \} \wedge s^* \cap \text{cod}(\rho') = \emptyset \} \)
  and \( \forall t \in T'' : \iota''(t) = \begin{cases} \iota'(t) & \text{if } t \in T', \\ \{ \texttt{abort} \}, \emptyset & \text{if } t \in T_s \end{cases} \);
- \( \forall (t, s) \in T'' \times S'' : \iota''(t, s) = \begin{cases} \iota'(t, s) & \text{if } t \in T', \\ \emptyset & \text{if } t \in T_s \end{cases} \);
- \( \forall (s, t) \in S'' \times T'' : \iota''(s, t) = \begin{cases} \{ \texttt{a} \} \subset \text{Var} & \text{if } t = t_s \in T_s, \\ \emptyset & \text{if } t \in T_s \setminus \{ t_s \} \end{cases} \);
- \( \rho'' = \rho' \uplus \{ (t, t_s) | t_s \in T_s \wedge t \in (\bullet t_s)^* \} \).

Let us make two remarks about this definition:

- places which are input places for transitions belonging to \text{cod}(\rho) (if \( P = (N, \rho) \)) are left untouched; the reason is that such places are already “under control”, i.e., belong to a nested emptying mechanism. We do not need additional emptying transitions for them;
- the added emptying transitions are \textit{always} synchronized with transition \( t_4 \) in \( N_s \), whose enabling is well controlled. So, net \( \pi(P) \) never empties in an uncontrolled way. Another consequence is that the only non T-restricted transitions are those such as \( t_2, t_3 \) and \( t_8 \).
4.5 P/M-Nets

We are now in position to define preemptible M-nets (P/M-nets). They are defined as a sub-class of priority M-nets with some structural constraints. This sub-class happens to be reasonably wide (see section 6) and sound with respect to the semantics of preemption (see section 4.6).

Definition 5. Let \( P = (N, \rho) \) be a priority M-net. \( P \) is a P/M-net iff either:

- \( N \) is an \( \text{ex-good} \) M-net and \( \rho = \emptyset \), or;
- \( P \) is defined as \( \pi(P_1), P_1[X \leftarrow P_2], P_1 \parallel P_2, P_1; P_2, P_1 \biguplus P_2, [P_1 \ast P_2 \ast P_3], P_1 \text{ sy } A, P_1 \text{ rs } A, [A : P_1], P_1 \text{ tie } b \text{ or } P_1[f] \), where \( P_i \), for \( i \in \{1, 2, 3\} \), are P/M-nets, \( X \) is a hierarchical symbol, \( A \) is a synchronous communication symbol, \( b \) is an asynchronous link symbol, and \( f \) is a renaming function on communication symbols.

Definition 6. A P/M-net \( (N, \rho) \) is said: valid if \( N \) is an \( \text{ex-good} \) M-net; finite if \( \mathcal{U}(N) \) is finite (in number of places and transitions); 1-safe if \( \mathcal{U}(N) \) is 1-safe.

4.6 Soundness

In order to define the soundness of a P/M-net \( P = (N, \rho) \) with respect to the semantics of preemption, we split the set \( T \) of transitions of \( N \) in four disjoint parts: \( T = T_r \uplus T_i \uplus T_i \uplus T_p \).

If \( P = \pi(P') \), then, \( T_r \) (\( r \) is for root) is the set of transitions involved in an abortion of \( P \) at the top level. In other words, \( T_r \) contains all the transitions coming from a synchronisation with \( t_4, t_5 \) or \( t_6 \) in \( N_{\pi} \) used in \( \pi(P') \). \( T_i \) (internal) contains the same kind of transitions, but issued from a possible \( \pi \) nested in \( P' \). \( T_t \) (termination) contains only transition \( t_7 \) coming from \( P_\pi \) in \( \pi(P') \). All the remaining transitions are in \( T_p \) (program). If \( P \neq \pi(P') \), we have \( T_r = T_i = T_t = \emptyset \).

Definition 7 below formalizes the intuition of the expected behavior of a preemptible M-net \( P \). Actually, \( P \) can evolve in one of the following ways:

- \( P \) runs without any abortion. In this case, it fires only transitions from \( T_p \);
- \( P \) runs as before by firing transitions in \( T_p \) until some abortions occur in some nested \( \pi \)'s. In this case, all the next fired transitions are in \( T_i \uplus T_p \);
- \( P = \pi(P') \) and it completely aborts at a point of its execution. In this case, it starts like in the previous case, until it begins aborting and thus only fires transitions from \( T_r \) or from \( T_i \) because of some possible abortions transmitted to nested \( \pi \)'s. After the end of the abortion, it may fire \( t_7 \) from \( T_t \) to produce its exit marking.

Definition 7. Let \( P \) be a P/M-net. \( \text{steps}(P) \) is said sound with respect to the semantics of preemption iff each \( D \in \text{steps}(P) \) is of the form \( \delta_1 \ldots \delta_n \ldots \delta_m \) or \( \delta_1 \ldots \delta_n \delta'_1 \ldots \delta_m \) if \( P = \pi(P') \), otherwise \( D = \delta_1 \ldots \delta_n \), where for all \( j, \delta_j \in \mathcal{M}_f(T_i \uplus T_p) \), for all \( k, \delta'_k \in \mathcal{M}_f(T_i \uplus T_r) \), and \( \delta'' = \{t_7 \in T_t\} \).
A Concurrent and Compositional Petri Net Semantics

Fig. 4. Net part of P/M-net $P_1 = (N_1, \rho_1)$ with $\rho_1 = \emptyset$. All the places of $N_1$ have type $\{\bullet\}$, and arcs should be annotated $\{\bullet\}$.\n
Fig. 5. The scheme for $P_2$, its priority relation is $\rho_2 = \emptyset$.\n
By induction on the algebraic structure, we get that what we structurally defined through P/M-nets is what we expected as a model preserving the semantics of preemption.

**Theorem 2.** Let $P$ be a P/M-net. Its step semantics, $\text{steps}(P)$, is sound with respect to the semantics of preemption.

### 5 A Detailed Example

This section gives an example of P/M-net. We show the construction of the P/M-net

$$P = \pi \left( \pi(P_1) \parallel P_2 \right)$$

where $P_1$ is a P/M-net with two parallel transitions $t_a$, labelled $\{\mathcal{A}\}.\emptyset.\emptyset$, and $t_b$ labelled $\{\text{quit}\}.\emptyset.\emptyset$ (see figure 4). $P_2$ is a net from which we will just consider one transition $t_c$ having one internal input place $i_c$; we assume that this net is valid, well defined and does not use $\pi$ in its construction (this would demonstrate transmission of abortion but, for the sake of simplicity, we prefer to show this feature with $P_1$ only). $P_2$ is schematized in figure 5. This example allows us to illustrate internal and external abortion as well as the propagation of abortion to nested $\pi$ operations. In the following, most annotations are omitted from figures in order to keep them as readable as possible.

First, let us look carefully at the P/M-net produced by $\pi(P_1)$ whose construction is detailed in figures 6 to 8. The first step for building $\pi(P_1)$ is to refine $P_1$ into $P_2$ and to add the emptying transitions; the net part of the result is shown in figure 6 and the corresponding priority relation is

$$\rho'_1 = \{(t_7, t_4), (t_8, t_4), (t_7, t_5), (t_8, t_5), (t_a, t_a), (t_b, t_b), (t_0, t'_a), (t_a, t_a), (t_b, t_b), (t_2, t_3)\}.$$ 

The next step consists in performing the scoping over $\text{kill}$ and $\text{empty}$, making asynchronous links over $c$, and renaming $\text{kill}'$ into $\text{kill}$ and $\text{empty}'$ into $\text{empty}$, as stated in definition 4. The result is sketched in figure 7 (most annotations are
omitted as well as place $s_c$ for asynchronous links on $c$. Transitions $t_b$ and $t_1$ yield a new transition $t'_4$ whose firing starts internal abortion stage. The synchronization of $t_4$ with the added emptying transitions yield $t'_5$ and $t'_7$. Transitions $t_5$ and $t_6$ are removed because they are nothing to synchronize with (it will not be the case later in our example, when we will consider the nesting of $\pi(P_1)$ into another $\pi$). The priority relation of $\pi(P_1)$ is

$$\rho_1^\pi = \{(t_0,t_4'), (t_0,t_7), (t_7,t_4'), (t_0,t_4), (t_7,t_4), (t_8,t_4'), (t_8,t_4), (t_2,t_3)\}.$$

One can observe that everything works right when this net is started from its entry marking: if $t'_4$ (internal abortion) or $t_2$ (external abortion) fires, then the net is correctly emptied, whenever $t_0$ did fire or not. Similarly, if $t_3$ fires after $t'_4$ did. The difference between external and internal abortion appears at the very end: for internal abortion, $t_7$ fires, producing the exit marking; for external abortion, only $t_8$ can fire. We will see later in the example how $t_2$, $t_3$ and $t_8$ are synchronized in order to achieve a correct transmission of abortion when $\pi(P_1)$ is nested into another $\pi$.

The communication transitions, related to the preemption and visible from the outside of $\pi(P_1)$, are $t_2$, $t_3$ and $t_8$. So, in the following, $\pi(P_1)$ will be schematized as shown in figure 8. (Notice that we show only one entry place while there are actually two, this has no other consequence than simplifying the figures.)

In order to finish our example, we have to put $\pi(P_1)$ in parallel with $P_2$ and to apply $\pi$ on the result. We get the P/M-net depicted (in a simplified version) in figure 9. The net resulting from $\pi(P_1)\parallel P_2$ is splitted in two parts, the part coming from $P_2$ corresponds to the gray box in the top and the part

---

**Fig. 6.** Net part of $P_n[X \leftarrow P_1]$ with the added emptying transitions. See figure 3 for the detailed annotations.
which comes from $\pi(P_1)$ corresponds to the box in the bottom. The middle part, which is not boxed, comes from the last application of $\pi$. For this final $P/M$-net, the priority relation is $\rho = \{(t_c, t_2^4), (t_7, t_3^4), (t_8, t_3^4), (t_3, t_2^4), (t_7, t_4^4), (t_8, t_4^4), (t_7, t_5^4), (t_7, t_5^5), \ldots\}$.

We can see that an abortion has three effects: one is the emptying of $P_2$ with transition $t_2^4$ (there would be much more transitions like $t_2^4$ if $P_2$ would be larger), one other is the emptying of entry and exit places for the parallel composition $\pi(P_1) \parallel P_2$ (with transitions $t_1^4$ and $t_3^4$), and the last is the propagation of the emptying to $\pi(P_1)$ (with transition $t_6^4$ or $t_6^5$). When this sub-abortion is done and when all $P_2$ is empty (which can be done in any order, even in parallel), $t_6^4$ can fire, ending the emptying of $\pi(P_1)$.

We can also observe that suspension is immediately effective. Thanks to the definition of $\rho$, $t_c$ is suspended as soon as $t_2^4$ is enabled (and it is disabled after $t_2^4$ fired). Similarly, transitions in $\pi(P_1)$ are first suspended as long as $t_6^4$ and $t_6^5$ are enabled and, when of them one fires, the suspension is done internally because $t_6^4$, $t_6^4$ and $t_6^5$ in $\pi(P_1)$ are then enabled (see figure 7).

In this final net, transition $t_7$ is dead because the net never uses internal abortion; even if sub-net $\pi(P_1)$ does, it is not the case for $\pi(\pi(P_1) \parallel P_2)$ as a whole.
Asynchronous links on \( e \) (even if not shown in the figures) ensure that the propagation of abortion to \( \pi(P_1) \) is always followed by its complete emptying, through a firing of transition \( t_6' \) (see figure 9).

Let us conclude this example noticing that the net we obtain is ready to be nested in another \( \pi \), like \( \pi(P_1) \) was, thanks to its own transitions \( t_2, t_3 \) and \( t_8 \). If no such nesting is to be made, the net should be restricted over \( \text{kill} \) and \( \text{empty} \), resulting in a valid P/M-net. This restriction would be necessary, not only for validness, but also in order to avoid a spontaneous firing of transition \( t_2 \) or \( t_3 \) which would result in an unexpected abortion of the net.

6 Properties of P/M-Nets and Links with Existing Works

Let us observe first that a P/M-net \((N, \rho)\) which has been constructed without operation \( \pi \) is always valid and has \( \rho = \emptyset \). This property states that the introduced extension is conservative and does not disturb the existing model if one makes no usage of operation \( \pi \). This is the reason why we consider P/M-nets as a “reasonably wide” model: it contains M-nets which already proved being useful.

However, in general, a P/M-net \( P = (N, \rho) \) can have some crippled transitions. It turns out that they can easily be identified by their synchronous label \( \{\text{empty}\} \) or \( \{\text{kill}\} \). These transitions belong to the communication interface of \( P \) and are crucial in order to nest \( P \) in another operation \( \pi \), e.g., in \( P' = \pi(\ldots P \ldots) \). Operation \( \pi \), thanks to the scoping on \( \text{kill} \) and \( \text{empty} \), al-
low $P'$ to abort its preemptible parts such as $P$. However, if $P$ is not to be nested in an operation $\pi$, then these crippled transitions should be removed. Actually, in that case, the P/M-net of interest is $P_{rs}\{\text{kill, empty}\}$. Moreover, $P_{rs}\{\text{kill, empty}\}$ is valid; this property is important because it shows that even if our modeling needs to relax the T-restrictness of some transitions, the final result can always be T-restricted.

Valid P/M-net semantics may still appear as somehow unsatisfactory, because of the use of priorities. It turns out that some results in the field of semantics of priority systems may be applied for P/M-nets. In [6], the authors define a transformation of a finite 1-safe Petri net $\Sigma$, equipped with a pairwise priority relation $\rho$, into a bounded Petri net which retains as much as possible of the concurrency of $(\Sigma, \rho)$. In this context, as much as possible means that only semantics composed of consistent steps are considered (see [6, section 3]). This result can be directly applied to the unfolding of a valid, finite and 1-safe P/M-net. (One can see that if $P$ is 1-safe, so is $\pi(P)$.) Then, applying the result from [7], the obtained bounded Petri net can be transformed into a 1-safe Petri net which has the same pomset semantics (partially ordered multi-sets semantics), and we can state:

**Proposition 1.** Let $P$ be a valid, finite and 1-safe P/M-net. Then, $P$ can be transformed into a low-level 1-safe Petri net having the same consistent step semantics.

So, P/M-nets can be transformed, in most of reasonable cases (i.e., finite ones), into 1-safe Petri nets having an equivalent concurrent semantics. However, the construction given in [7] leads to really huge nets and so is not intended to be used in practice. Nevertheless, the above proposition is important since it means that 1-safe Petri nets are expressive enough to model preemption with a concurrent semantics.

In practice, it should be possible to modify the existing model checker of PEP [13,11] in order to have it dealing with priorities. The model checker relies on finite prefixes computation: it develops the branching process semantics of the analyzed low-level net until it finds a maximal cut. (This is always the case in finite 1-safe Petri nets since the number of states is finite.) Under such conditions, the influence of priorities should be to prune some branches in the computation. For two enabled transitions $t_1$ and $t_2$, the existing algorithm would build one branch starting with a firing of $t_1$ and another for $t_2$. With priorities, if $(t_1, t_2) \in \rho$, the branch starting with $t_1$ does not have to exist anymore. This does not mean that the computation would be more efficient; actually, the contrary should hold: adding priorities would force the examination of much more cases in order to take into account some possibly disabled transitions, because of priorities. However, the model checking remains decidable.

### 7 Conclusion

We presented what is, to the best of our knowledge, a first attempt to provide a fully compositional model of Petri nets with a preemption operator, with a
concurrent semantics. Our construction is based on M-nets which are extended with priorities, structured into an algebra and structurally restricted leading to preemptibles M-nets (P/M-nets). The P/M-net algebra is similar to the M-net algebra, but having an additional operation $\pi$, which transforms any P/M-net into its preemptible equivalent.

We show that P/M-nets can be considered as a high-level version of so called priority systems (as defined in [6]) by defining an unfolding operation which transforms a P/M-net into a low-level Petri net having priorities between its transitions. Thus, applying results from [6] and [7], any reasonable P/M-net can be transformed into a 1-safe Petri net (without priorities) which retains as much as possible of the concurrency present in the P/M-net. This transformation leads to enormous nets and is not tractable in practice, but it shows that 1-safe Petri nets are powerful enough to model preemption with a concurrent semantics.

The presented work has been already applied for giving the semantics of some preemption-related extensions of the parallel programming language B(PN)$^2$, introducing abortable blocks, treatment of exceptions, a generalized timeout construct and a small Unix-like process manager.

Acknowledgments. We are very grateful to all the participants of the three last PORTA/BAT workshops for their remarks on this work and especially for their thoughts about model checking nets with priorities. Particular thanks go to Raymond Devillers.

References

An Approach to Symbolic Test Generation

Vlad Rusu, Lydie du Bousquet, and Thierry Jéron

IRISA/INRIA, Rennes, France
{rusu|jeron|ldubousq}@irisa.fr

Abstract. Test generation is a program-synthesis problem: starting from the formal specification of a system under test, and from a test purpose describing a set of behaviours to be tested, compute a reactive program that observes an implementation of the system to detect non-conformant behaviour, while trying to control it towards satisfying the test purpose. In this paper we describe an approach for generating symbolic test cases, in the form of input-output automata with variables and parameters.

1 Introduction

It is widely recognized that testing is an essential component of the full lifecycle of software systems. Among the many different testing techniques, conformance testing [11] is one of the most rigorous. The usual theoretical approach [5,16] is to consider a formal specification of the intended behaviour of the Implementation Under Test (IUT). It allows to define the notion of conformance relation, which defines the correct implementations with respect to the specification. It also allows to formally define test cases, their execution on an IUT, and the notion of verdict associated to the execution. However, the process of writing test cases for large specifications is complicated, error-prone and expensive. For example, the paper [12] identifies various errors in about 15% of the test cases from a commercially available test suite for the SSCOP protocol [21]. Another difficulty comes from the black-box nature of the implementation, whose behaviour is only observable and controllable at the interfaces. In this context, a formal framework is a prerequisite for giving precise and consistent meanings to test cases.

During the last decade, testing theories and algorithms for the automatic generation of tests have been developed from specifications modelled by variants of the Labeled Transition System model (LTS). Some efficient algorithms are based on adaptations of on-the-fly model-checking algorithms [13]. Academic tools such as TorX [1], TGV [6] and industrial tools such as TestComposer (Verilog) already exist, which implement these algorithms and produce correct test cases in a formal framework. However, these theories and tools do not explicitly take into account the program data, as the underlying model of LTS implies that values of variables are expanded during state exploration. This may result in the

The specifications and proofs for the example treated in the paper can be found at http://www.irisa.fr/pampa/perso/rusu/IFM00/.
classical state-explosion problem, but also has the effect of obtaining test cases
where all variables are instantiated. This is in contradiction with industrial prac-
tice, where test cases (written, for example, in the TTCN language [11]) are real
programs with parameters and variables. Generating such test cases requires new
models and techniques. The models should explicitly include parameters and va-
riables, and the techniques should treat them symbolically by combining model
checking with constraint propagation, static analysis, and theorem proving. In
this paper, we present some steps towards such an approach.

The rest of the paper is organized as follows. In Section 2 we define an exten-
sion of the LTS model (called IOSTS) to include parameters and variables. In
Section 3 we define subclasses of the IOSTS model that are used for specificati-
ons, test purposes and test cases. We adapt a conformance relation from [16] to
the model of IOSTS. In Section 4 we define a notion of correctness of test cases
with respect to specifications and test purposes. In Section 5 we describe how to
generate test cases by successively computing a synchronous product between
a specification and a test purpose, eliminating internal actions and nondeter-
minism, and selecting the behaviours that are accepted by the test purpose.
(Currently, the method works only for specifications for which internal actions
and non-determinism can be eliminated using a given set of heuristics.) We also
prove the correctness of the generated test cases. In Section 6 we show how to
simplify the test case using static analysis and theorem proving to remove irrele-
vant control and data. The approach is demonstrated on a simple example: the
BRP protocol [8].

2 Model: Input-Output Symbolic Transition Systems

We define a model of extended transition systems called Input-Output Symbolic
Transition Systems (IOSTS), inspired from I/O automata [15] and CSP [10]. The
IOSTS model was designed to be easily translatable into the input languages of
tools such as the PVS theorem prover [17] and the HyTech model checker [9].

Syntax. Let $D$ be a set of typed data. We denote by $\text{type}(d)$ the type of element
$d \in D$, by $\mathcal{E}(D)$ the set of type-correct expressions on the data $D$, and by $\mathcal{B}(D)$
the subset of boolean expressions.

Definition 1 (IOSTS). An IOSTS is a tuple $(D, \Theta, Q, q^0, \Sigma, T)$ where

- $D$ is a nonempty, finite set of typed data, which is the disjoint union of a
  set $V$ of variables, a set $P$ of parameters and a set $M$ of messages,
- $\Theta \in \mathcal{B}(V \cup P)$ is the initial condition,
- $Q$ is a nonempty, finite set of locations,
- $q^0 \in Q$ is the initial location,
- $\Sigma$ is a nonempty, finite alphabet, which is the disjoint union of a set $\Sigma^i$ of
  input actions, a set $\Sigma^o$ of output actions, and a set $\Sigma^{\text{int}}$ of internal actions.
  For each action $a \in \Sigma^i \cup \Sigma^o \cup \Sigma^{\text{int}}$, there is a (possibly empty) tuple of types
  $\text{sig}(a) = (\vartheta_1, \ldots, \vartheta_k)$, called the signature of the action (the signature of an
  internal action $a \in \Sigma^{\text{int}}$ is the empty tuple),
\( T \) is a set of transitions. Each transition consists of:

- a location \( q \in Q \), called the origin of the transition,
- an action \( a \in \Sigma \) called the action of the transition,
- a tuple of messages \( \mu = \langle m_1, \ldots, m_k \rangle \) such that if \( \text{sig}(a) = \langle \vartheta_1, \ldots, \vartheta_k \rangle \) is the signature of action \( a \), then, for all \( i \in [1, k] \), \( \text{type}(m_i) = \vartheta_i \),
- a boolean expression \( G \in B(V \cup P \cup \mu) \), called the guard of the transition,
- a set of expressions \( A \), called the assignments of the transition, such that for each variable \( x \in V \) there is exactly one assignment in \( A \), of the form \( x := A^x \), where \( A^x \) is an expression on \( V \cup P \cup \mu \),
- a location \( q' \in Q \) called the destination of the transition.

Figure 1 illustrates an example of IOSTS, which has variables \( f, rn, head \), parameters \( max \) and \( last \), and messages \( F \) and \( m \). We assume that \( rn, head, max, \) and \( last \) are natural numbers (with \( max > 1, last > 0 \)), variable \( f \) and message \( F \) are functions from natural numbers to some uninterpreted type (which is also the type of message \( m \)), and for all \( i \in \mathbb{N}, F(i) \neq F(i+1) \).

Consider for example the transition with origin \( \text{Send}_\text{File} \) and destination \( \text{Wait}_\text{Ack} \). It has the action \( \text{MSG} \) with message \( m \), guard \( m = f(\text{head} - 1) \), and assignment \( rn := rn + 1, f := f, head := head \). In the graphical representation, the symbol ? (resp. !) denotes an input (resp. an output) action, and the assignments without effect (such as \( f := f \)) are omitted.

**Semantics.** For \( D' \subseteq D \) a subset of the data, we denote by \( \text{Val}(D') \) the set of (type-consistent) valuations of \( D' \), which associate to each element of \( d \in D' \) a value of type \( \text{type}(d) \). A state is a pair \((l, v)\) where \( l \in Q \) is a location and \( v \in \text{Val}(V \cup P) \) is a valuation for the variables and the parameters. We denote the set of states by \( S \). An initial state is a state \((l_0, v_0)\) such that \( l_0 = q^0 \) (the
initial location) and $v_0 \models \Theta$. We denote by $S^0$ the set of initial states. A \textit{valued input} (respectively a \textit{valued output}) is a tuple consisting of an input action $a \in \Sigma^i$ (resp. an output action $a \in \Sigma^o$) and a tuple of values of the types given by the action’s signature. That is, if $a \in \Sigma^i$ and $\text{sig}(a) = \langle \theta_1, \ldots, \theta_l \rangle$, then valued inputs are of the form $(a, v_1, \ldots, v_l)$ where for all $i \in [1, l]$, $v_i$ is a value of type $\theta_i$ (and similarly for valued outputs). We denote by $T$ (resp. $\Omega$) the set of valued inputs (resp. of valued outputs.)

Let $D', D'' \subseteq D$ denote two disjoint subsets of $D$. For $v \in \text{Val}(D')$ and $v' \in \text{Val}(D'')$, we denote by $v \cdot w$ the valuation of $D' \cup D''$ such that $v \cdot w(d) = v(d)$ if $d \in D'$ and $v \cdot w(d) = w(d)$ if $d \in D''$. Given an expression $e \in \mathcal{E}(D)$ and a valuation $v \in \text{Val}(D)$, we denote by $v(e)$ the value obtained by replacing in $e$ each element $d \in D$ by its value $v(d)$. The semantics of an IOSTS is given by a labeled transition system with set of states $S$, initial states $S^0$, labels $T \cup \Omega \cup \Sigma^{\text{int}}$, and the transition relation $\xrightarrow{} \subseteq S \times (T \cup \Omega \cup \Sigma^{\text{int}}) \times S$, which is the smallest relation defined by the following rule:

$$(q, a, \mu, G, A, q') \in T$$

$v, v' \in \text{Val}(V \cup P)$, $w \in \text{Val}(\mu)$, $v \cdot w(G) = true$

$\forall x \in V : v'(x) = v \cdot w(A) \quad \forall x \in P : v'(x) = v(x)$

$<q, v> \xrightarrow{a,w} <q', v'>$

The relation $\xrightarrow{}$ induces the relation $\Rightarrow \subseteq S \times (\Omega \cup T)^* \times S$ by dropping all internal actions:

- $s \xrightarrow{a} s' \overset{\text{def}}{=} \exists \tau_1, \ldots, \tau_n \in \Sigma^{\text{int}}, \exists s_1, \ldots, s_{n-1} \in S, s \xrightarrow{\tau_1} s_1 \cdots \xrightarrow{\tau_n} s'$,
- for $\alpha \in \Omega \cup T$, $s \xrightarrow{\alpha} s' \overset{\text{def}}{=} \exists s_2 \in S, s \xrightarrow{\alpha} s_1 \xrightarrow{\tau_2} s'$,
- for $\sigma = \alpha_1 \cdots \alpha_n \in (\Omega \cup T)^n$ with $n > 1$, $s \xrightarrow{\sigma} s' \overset{\text{def}}{=} \exists s_1, \ldots, s_{n-1} \in S, \sigma \overset{\text{def}}{=} s \xrightarrow{\alpha_1} s_1 \cdots \xrightarrow{\alpha_{n-1}} s_{n-1} \xrightarrow{\alpha_n} s'$.

For an IOSTS $S$ we denote by traces($S$) the set $\{ \sigma \in (\Omega \cup T)^* | \exists s_0 \in S^0, \exists s \in S, s_0 \xrightarrow{\sigma} s \}$, For $\sigma \in (\Omega \cup T)^*$, we denote by $S$ after $\sigma$ the set of states $\{ s \in S | \exists s' \in S^0, \exists s_0 \xrightarrow{\sigma} s \}$. For $S^1 \subseteq S$ a set of states, we denote by out($S^1$) the set of valued outputs that can be “observed” in states $s' \in S^1$ possibly after internal actions, that is, out($S^1$) = $\{ \sigma \in (\Omega \cup T)^* | \exists s' \in S^1, \exists s \in S, s \xrightarrow{\sigma} s' \}$. For a set of traces $L$, we denote by pref($L$) the set of strict prefixes of sequences in $L$.

**Subclasses of IOSTS.** Let $S$ be an IOSTS, $P$ its set of parameters, and $\pi \in \text{Val}(P)$ a valuation of the parameters. By replacing each parameter $p \in P$ by its value $\pi(p)$, we obtain an IOSTS denoted $S(\pi)$, called an instance of $S$. In this case, we also say that $S(\pi)$ is instantiated. An instantiated IOSTS is initialized if the initial condition $\Theta$ assigns to each variable $v \in X$ exactly one value. An arbitrary IOSTS is initialized if all its instances are initialized. An IOSTS $S$ is deterministic if for all $s \in S$, $\text{card}(\cup_{\alpha \in \Sigma^{\text{int}}} \{ s' \in S | s \xrightarrow{\alpha} s' \}) \leq 1$ and for all $\alpha \in \Omega \cup T$, $\text{card}(\{ s' \in S | s \xrightarrow{\alpha} s' \}) \leq 1$. That is, for each state $s \in S$,
if an internal action is executed, then the next state depends only on s, and if a valued input (or valued output) α is executed, then the next state depends only on s and α. We then have the following proposition:

**Proposition 1.** If TC is an initialized, deterministic IOSTS without internal actions, then for all σ ∈ traces(TC), the set of states TC after σ is a singleton.

A location q of an IOSTS is input-complete if for each state s =< q, v > ∈ S and each valued input α ∈ T, the set \{s′ | s ↠^α s′ \} is nonempty. An IOSTS is complete if for each s ∈ S, α ∈ Ω ∪ T ∪ Σ^{int}, the set \{s′ | s ↠^α s′ \} is nonempty.

**Operations on IOSTS.** For an IOSTS S, there are two systems that may share some parameters, and that a variable of one system may affect actions, which are \{Σ \} and alphabet Σ. If \( q \) is a singleton.

We now define a generic composition operation for IOSTS, which we then specialize to define two operations (parallel and product) that are needed in the sequel. For two IOSTS \( S_1 = (D_1, Θ_1, Q_1, q_1, S_1, T_1) \) with data \( D_1 = V_1 ∪ P_1 ∪ M_1 \) and alphabet \( S_1 = Σ_1^v ∪ Σ_1^o ∪ Σ_1^{int} \) (resp. \( S_2 = (D_2, Θ_2, Q_2, q_2, S_2, T_2) \) with data \( D_2 = V_2 ∪ P_2 ∪ M_2 \) and alphabet \( S_2 = Σ_2^v ∪ Σ_2^o ∪ Σ_2^{int} \), we say \( S_1, S_2 \) are compatible for composition if \((V_1 ∪ M_1) ∩ (V_2 ∪ M_2) = \emptyset\), \((P_1 ∪ M_1) ∩ M_2 = \emptyset\), \((P_2 ∪ M_2) ∩ M_1 = \emptyset\), \( S_1 = S_2 \), \( Σ_1^o = Σ_2^o \), \( Σ_1^{int} = Σ_2^{int} \), and each action \( a ∈ Σ_1^v ∪ Σ_2^v \) (which also means \( a ∈ Σ_1^{v,v} \)) has the same signature in both systems. Note that the two systems may share some parameters, and that a variable of one system may be a parameter of the other (meaning that it can be read, but not modified).

The composition operation lets each system perform independently its internal actions (that are not in the alphabet of the other), and imposes synchronization on the shared actions. Formally, the composition \( S = S_1 ∣ S_2 \) of compatible IOSTS \( S_1, S_2 \) is the IOSTS \( (D, P, Θ, Q, q, Σ, T) \) that consists of the following elements: \( V = V_1 ∪ V_2, P = (P_1 ∪ P_2) \) with \( (V_1 ∪ V_2), M = M_1 ∪ M_2, Θ = Θ_1 ∧ Θ_2, Q = Q_1 × Q_2, q = (q_1, q_2), Σ^o = \emptyset, Σ^{int} = Σ_1^{int} ∪ Σ_2^{int}, Σ^{v,v} = Σ_1^{v,v} ∪ Σ_2^{v,v}, \) and each action \( a ∈ Σ_1^{v,v} \). The set \( T \) of transitions of the composed system is defined as follows:

- For each transition \((q_1, a, μ, G, A, q_1') \in T_1 \) with \( a ∈ Σ_1^{v,v} \) and for each location of the form \((q_1, q_2) \) with \( q_2 ∈ Q_2 \), there is a transition in \( S_1 ∣ S_2 \) a transition \((q_1, q_2, a, μ, G, A, q_1') \in T_1 \) with \( (y := y), (q_1', q_2) \) (and similarly for all actions \( a ∈ Σ_2^{v,v} \)).

- For each transitions \((q_1, a, μ_1, G_1, A_1, q_1') \in T_1 \) with \( a ∈ Σ_1^v \), let \( G_1 \) denote the expression obtained by replacing in \( G_2 \) each message from \( μ_2 \) by the corresponding, same-position message from \( μ_1 \). Then, in \( S_1 ∣ S_2 \) there is a transition \((q_1, q_2, a, μ_1 ∧ G_1, A_1 ∪ A_2, q_1') \) (and similarly when \( a ∈ Σ_2^v \)).
Parallel operation. The parallel operation $||$ is the same as the composition $\cdot$, except that it is defined only for IOSTS $S_1, S_2$ that satisfy the stronger compatibility requirements $D_1 \cap D_2 = \emptyset$, $\Sigma_1^{int} \cap \Sigma_2^{int} = \emptyset$. We use this operation to model the execution of a test case on a black-box implementation.

Product operation. The product operation $\times$ of IOSTS $S_1, S_2$ is defined as $S_1 \times S_2 = \text{mirror}(S_1|\text{mirror}(S_2), \Sigma_1', \Sigma_2')$. The compatibility requirement for the product operation (in addition to the fact that $S_1$ and $\text{mirror}(S_2)$ must be compatible for composition) is $\Sigma_1^{int} = \Sigma_2^{int}$. We use this operation during test generation to select a part of a specification, by computing the product with another IOSTS called a test purpose. For precise selection, we allow the test purpose to have access to the specification’s internal actions and data.

We are interested in the following relationships between the traces of the parallel and product-composed systems and the traces of their components.

Proposition 2 (Traces of the Parallel and Product Compositions). For IOSTS $S_1, S_2$ that are compatible for the parallel (resp. product) composition:

1. $\text{traces}(S_1||S_2) = \text{traces}(S_1) \cap \text{traces}(S_2)$
2. if $S_2$ is complete, then $\text{traces}(S_1 \times S_2) = \text{traces}(S_1)$.

Sketch of Proof. To prove the first item, we show that for all states $s_1, s'_1$ of $S_1$ (resp. $s_2, s'_2$ of $S_2$), and for all valued input or output $\alpha \in D_1 \cup D_2$ (which also means $\alpha \in T_2 \cup \Omega_2$), the relation $(s_1, s_2) \xRightarrow{\alpha} (s'_1, s'_2)$ holds in $S_1||S_2$ iff $s_1 \xRightarrow{\alpha} s'_1$ holds in $S_1$ and $s_2 \xRightarrow{\alpha} s'_2$ holds in $S_2$. For this, we first prove that internal actions of each system leave the state of the other unchanged (which is true essentially because of the hypothesis that the data of the two systems are disjoint), and that, for a valued input (or output) $\alpha$, a transition relation involving $\alpha$ exists in the composition if and only if it exists in both components. For the second item, we first note that the traces of an IOSTS $S_2$, and those of $S_2'$ modified by a mirror operation, are the same. Thus, it is enough to show that, for IOSTS $S_1$ and $S_2'$ that are compatible for the $|-$composition, which have the same internal actions, and such that $S_2'$ is complete, the equality $\text{traces}(S_1|S_2') = \text{traces}(S_1) \cap \text{traces}(S_2')$ holds. For the $\subseteq$ inclusion, we proceed in the same manner as for the corresponding inclusion in item 1 (here, the data independence hypothesis is not used) and we use the fact that (since $S_2'$ is complete), $\text{traces}(S_2') = (T_2 \cup \Omega_2)^* = (D_1 \cup T_1)^*$. For the $\supseteq$ inclusion of item 2, we use again the fact that $S_2'$ is complete: by composing it with $S_1$, any sequence of valued actions which is present in the latter is also allowed by the former, therefore it is also present in the composition $S_1|S_2'$.  

3 Conformance Testing with IOSTS

Conformance testing [11] is a methodology for validating reactive systems. It consists in testing a conformance relation between a specification, which is a formal model of a system, and an implementation of the system, which is a
physical process (typically, a program running on a machine) that can only be
controlled and observed at its interfaces. This is done by running test cases on the
implementation and obtaining verdicts about the conformance. In general, three
types of verdicts are distinguished. Informally, Fail means that non-conformance
was detected, Pass means that the implementation behaved in conformance with
the specification and the goal of the test experiment (described by the a so-called
test purpose) has been reached, and Inconclusive means that the implementation
behaved correctly but, due to the lack of control on the implementation, it did
not allow to reach the expected goal. We model specifications, implementations,
test purposes and test cases as IOSTS, with some restrictions for each category.

Specifications. A specification is an initialized IOSTS. In this paper we con-
sider (for test generation) specifications without cycles of internal actions. We
then indicate how to extend the approach to allow cycles of internal actions that
perform internal, deterministic, computations, during which the system does not
communicate with its environment.

As a running example, we use the specification of the BRP protocol [8], which
was designed to safely transmit files over an unsafe medium. The protocol uses a
combination of alternating-bit and resend-on-timeout mechanisms. Here, we are
only interested in the sender of the protocol, which is represented in Figure 1.
The parameters last (resp. max) stand for the number of packets in the file being
sent (resp. the maximum number of retransmissions of a packet). The variable
head is used to count the packets (which are $f(0)$ to $f(last - 1)$) and variable rn
is for counting retransmissions. On reception of a REQ input with message $F$
from its client, the sender saves it in variable $f$, then iteratively sends the packets
using the output action MSG with message $m$. A packet can be acknowledged
(input ACK), or there may be a timeout (an internal action), after which the
sender decides to resend the same packet. The status of the whole transmission
is described by either CONF_OK, meaning that all the packets were sent and
acknowledged, CONF_DONT_KNOW, meaning that all the packets were sent
and all but the last were acknowledged, or CONF_NOT_OK, meaning that some
intermediary packet was not acknowledged.

Implementations. An implementation is a physical object. However, in order
to be able to formally reason with it, we assume (this is a usual “testing hypo-
thesis”) that the implementation can be modelled by an IOSTS, which is unknown
except for its input and output alphabet and the signatures of its actions, which
are supposed to be the same as those of the specification.

Test Purposes. A test purpose is used to select a part of the specification,
for which a test case will be generated. A “good” test purpose should be simple
(typically, much simpler than the specification) and should select exactly the
scenarios that the user has in mind. In practice, as our experience with the tool
TGV [13], the process is iterative: a first test purpose only grossly selects a part
of the specification, then the user has to examine the result and modify the test purpose, and repeatedly so until a satisfactory result is obtained. Here, we use IOSTS as test purposes and the product operation as the selection mechanism. This gives enough expressiveness for describing, e.g., parameterized scenarios that would be hard to describe with, e.g., finite automata or temporal logic.

Formally, let $S = \langle D_S, \Theta_S, Q_S, q^0_S, \Sigma_S, T_S \rangle$ be a specification IOSTS. A test purpose of $S$ is an IOSTS $TP = \langle D_{TP}, \Theta_{TP}, Q_{TP}, q^0_{TP}, \Sigma_{TP}, T_{TP} \rangle$ together with a set of locations $Accept_{TP} \subseteq Q_{TP}$ such that $TP$ is initialized, complete, $\times$-compatible with $S$, and all the transitions with origin in the set $Accept_{TP}$ are self-loops.

The IOSTS represented in Figure 2 is a test purpose for the specification of the BRP sender. It is used to select the scenarios in which the latter sends all messages exactly once (without retransmissions) and exits successfully with a CONF_OK confirmation. It has the variable $j$, parameters $f$ and $last$ (which are also a variable, respectively a parameter of the specification), and one Accept location. The location Reject is used to discard executions in which the BRP sender does not behave as intended (i.e., it executes the internal action timeout, known to be followed by a retransmission, or it exits with a wrong confirmation).

This test purpose is not complete but, by convention, it is implicitly completed as follows: in each location, any action that does not (syntactically) label an outgoing transition, will produce a self-loop; and any action that labels some outgoing transition with a guard $G$, will generate a transition to Reject with the guard $\neg G$. This simplifies the writing of test purposes by allowing to focus on the intended behaviour for testing, leaving practically irrelevant details (such as completeness) to the test generation tool.

Given a specification $S$ and a test purpose $TP$ of $S$, we define the product $P = S \times TP$, the set of locations $Accept_P = Q_S \times Accept_{TP}$, and the set of traces of the product $Atraces(P) = \{ \sigma \in (\Omega_P \cup T_P)^* | \exists s_0 \in S^0_P, \exists s = (l, v) \in S_P : l \in Accept_P, s_0 \xrightarrow{\sigma} s \}$. By Proposition 2 item 2, we know that $Atraces(P) \subseteq traces(S)$. Intuitively, $Atraces(P)$ is the set of traces of the specification that are selected (through the product operation) by the test purpose.

The product between the BRP sender and its test purpose is partially represented in Figure 3.
Test Cases. Test cases are used to assign verdicts to implementations. This is done by running the two systems in parallel, and by observing the state of the test case at the end of the interaction. Formally, a test case is an initialized, deterministic IOSTS, together with three disjoint sets of locations Pass, Inconclusive and Fail. Moreover, a test case should react promptly to any input from the implementation, thus, it should not have internal actions, and all its locations (except those in the set Fail \cup Pass \cup Inconclusive) should be input-complete.

An example of test case is represented in Figure 4. It starts by giving to the sender’s implementation a file \( F \) to transmit. Then it expects to receive the successive packets in the file, and tries to acknowledge each packet. If it succeeds and receives the CONF_OK confirmation, the verdict is Pass, as the implementation behaved in conformance with the specification and the test purpose is satisfied: all packets were sent without retransmission. If, however, the test case gets another copy of the packet that it has just received, this means that the sender performed a retransmission, and the verdict is Inconclusive: the implementation behaved in conformance with the specification, but it did not allow to satisfy the test purpose. Finally, if some other input is received, the verdict is Fail, as the implementation emitted an output that is not allowed by the specification. For simplicity, the locations Fail are not represented in Figure 4.

Verdicts. We formalize the notion of verdict. Let \( TC \) be a test case. We denote by Pass (respectively Inconclusive, Fail) the set of states of \( TC \) whose locations are in the set Pass (resp. Inconclusive, Fail). Let \( I \) be the implementation
An Approach to Symbolic Test Generation 347

\[ m = f(head - 1) \]

\[ m = f(head - 1) \]

\[ m = f(head - 1) \]

\[ f := F \]

\[ f := F \]

\[ \text{head} = 1 \land \text{last} > 0 \]

\[ \text{head} = \text{last} \]

\[ \text{head} < \text{last} \]

\[ \text{head} := \text{head} + 1 \]

\[ \text{head} = \text{last} \]

\[ \text{ACK!} \]

\[ \text{WR} \_	ext{P1} \]

\[ \text{SP} \_	ext{P1} \]

\[ \text{WA} \_	ext{P1} \]

\[ \text{SC} \_	ext{P1} \]

\[ \text{CONF} \_\text{OK?} \]

\[ \text{Pass} \]

**Fig. 4.** Example of Test Case for the BRP Sender.

under test. For a trace \( \sigma \in \text{traces}(I||TC) \), we write \( \text{verdict}(TC, I, \sigma) = \text{Pass} \) if \( TC \text{after} \sigma \subseteq \text{Pass} \). We introduce the notations \( \text{verdict}(TC, I, \sigma) = \text{Fail} \) and \( \text{verdict}(TC, I, \sigma) = \text{Inconclusive} \) in a similar way.

**Conformance.** The conformance relation links an implementation with instances of the specification and the test purpose. Indeed, in practice, implementations are compared against specifications with known values of the parameters.

**Definition 2 (Conformance Relations).** Let \( S(\pi) \) be an instance of IOSTS \( S \), \( TP \) be a test purpose for \( S \), \( P = S \times TP \) their product, and \( P(\pi) = (S \times TP)(\pi) \) the corresponding instance of the product.

1. An implementation \( I \) is conformant to the instance \( S(\pi) \) of the specification \( S \), denoted \( \text{Conf}(S(\pi)) \), if for all traces \( \sigma \in \text{traces}(S(\pi)) : \text{out}(I \text{ after } \sigma) \subseteq \text{out}(S(\pi) \text{ after } \sigma) \).

2. An implementation \( I \) is conformant to the instance \( S(\pi) \) of the specification \( S \) and relative to the test purpose \( TP \), denoted \( \text{Conf}_{TP}(S(\pi)) \), if for all \( \sigma \in \text{pref}(\text{Atraces}((S \times TP)(\pi))) : \text{out}(I \text{ after } \sigma) \subseteq \text{out}(S(\pi) \text{ after } \sigma) \).

That is, after each trace of the specification, the possible outputs of the implementation are included in those of the specification. In the second case, the inclusion should hold after each prefix of a trace of the specification that is selected (through the product mechanism) by the test purpose.
4 Correctness of Test Cases

We formalize what it means for a test case to be correct, relative to a given specification and test purpose and for a given class of implementations. The intuition is that any instance of the test case should always give the right verdict when executed on an implementation in the class.

**Definition 3 (Correctness of Test Cases).** Let $S$ be a specification, $TP$ a test purpose of $S$, and $I$ a set of implementations. Let $TC$ be a test case such that the parameter sets of $TC$ and $S$ are the same, and $\pi$ a valuation of the parameters.

- $TC$ is **sound** for $S$, $I$ if for every instance $TC(\pi)$ and implementation $I \in I$, if there exists $\sigma \in \text{traces}(I||TC(\pi))$ such that $\text{verdict}(TC(\pi), I, \sigma) = \text{Fail}$, then $I$ is not conformant to $S(\pi)$.
- $TC$ is **relatively complete** for $S$, $TP$, $I$ if for every instance $TC(\pi)$ and implementation $I \in I$, if $I$ is not conformant to $S(\pi)$ relative to $TP$, then there exists $\sigma \in \text{traces}(I||TC(\pi))$ such that $\text{verdict}(TC(\pi), I, \sigma) = \text{Fail}$.
- $TC$ is **accurate** for $S$, $TP$, $I$ if for every instance $TC(\pi)$, implementation $I \in I$, trace $\sigma \in \text{traces}(I||TC(\pi))$: $\text{verdict}(TC(\pi), I, \sigma) = \text{Pass}$ if $\sigma \in \text{Atraces}((S \times TP)(\pi))$.
- $TC$ is **conclusive** for $S$, $TP$, $I$ if for every instance $TC(\pi)$, implementation $I \in I$, trace $\sigma \in \text{traces}(I||TC(\pi))$: $\text{verdict}(TC(\pi), I, \sigma) = \text{Inconclusive}$ implies $\sigma \in \text{traces}(S(\pi))$ and $\sigma \notin \text{pref}(\text{Atraces}((S \times TP)(\pi)))$.

A test case $TC$ is correct for $S$, $TP$, $I$ if it is sound for $S$, $I$ and relatively complete, accurate and conclusive for $S$, $TP$, $I$.

Intuitively, **soundness** means that the test case rejects only non-conformant implementations (in the given class). **Relative completeness** means that the test case can detect all implementations in the given class, which are non-conformant with the specification relative to the test purpose. (It does not mean all these implementations will actually be detected, because of nondeterminism of the implementation and because there may be an infinity of traces to consider.) **Accuracy** means that the verdict Pass is given when the observed trace of the implementation is a trace of the specification that is selected by the test purpose. Finally, **conclusiveness** means that the verdict Inconclusive is given when the observed trace of the implementation is a trace of the specification, but it cannot be extended into a trace that eventually produces the verdict Pass.

5 Test Case Generation

We present a procedure for generating symbolic test cases that are correct in the sense of Definition 3. The first step is computing the product between the test purpose and the specification, which was defined in Section 3. The following steps consist in transforming and simplifying the product (e.g., Figure 3) to obtain a test case (e.g., Figure 4) with all the required properties.
Closure: Eliminating Internal Actions. A test case should react promptly to inputs from the implementation. A natural way to obtain this is by requiring that every location of a test case (except the verdict locations) is input-complete. However, the possible inputs in a location may be hidden by internal actions. For example, consider the location \( WA_P1 \) of the IOSTS represented in Figure 3. Here, the system can only send an output \( ACK! \) or execute the internal action \( \text{timeout} \), but the inputs (\( MSG? \), \( CONF.OK? \), etc) can only be executed after a \( \text{timeout} \). To make the location input-complete, we first have to eliminate the latter actions. Thus, to obtain test cases where all locations are input-complete, all internal actions have to be eliminated from the product \( \mathcal{P} = S \times \mathcal{T} \mathcal{P} \). For this, the idea is to compute the effect of any sequence of internal actions that leads to an input-labeled transition, and to encode it in the last transition. This gives a simple syntactical procedure, which works if the specification does not contain cycles of internal actions. (A way to handle cycles is proposed at the end of this section.)

More formally, let \( \tau_1, \tau_2, \ldots, \tau_n \) be a sequence of internal actions that leads to an input action \( a \) (cf. Figure 5). The guard and assignments corresponding to \( \tau_i \) are \( G_i, A_i \), and the guard (resp. assignments) corresponding to \( a \) are \( G, A \). Then, it is not hard to see that a trace-equivalent system is obtained by replacing the whole sequence by a transition with origin \( l_1 \), destination \( l_{n+2} \), action \( a \), guard \( G_1 \land (G_2 \circ A_1) \land \cdots (G_n \circ A_{n-1} \circ \cdots \circ A_1) \land (G \circ A_n \circ A_{n-1} \circ \cdots \circ A_1) \), and assignments \( A \circ A_n \circ A_{n-1} \circ \cdots \circ A_1 \), where \( \circ \) denotes function composition.

We obtain an IOSTS \( \mathcal{P} \) with the same traces as \( \mathcal{P} \), but without internal actions. The set of accepting locations \( \text{Accept}_{\mathcal{P}} \) is defined by \( \text{Accept}_{\mathcal{P}} = Q_{\mathcal{P}} \cap \text{Accept}_{\mathcal{P}} \). For the product IOSTS represented in Figure 3, the corresponding IOSTS without internal actions is represented in Figure 6. The differences come from the transitions labeled \( \text{timeout} \), which have disappeared, and whose guards \( (rn < \text{max}, \text{resp. } rn = \text{max}) \) have been propagated to the nearest observable transitions.

Determinization. Nondeterminism is prohibited in test cases, because the verdicts should not depend on internal choices of the tester. Thus, the following step is to eliminate nondeterminism from the IOSTS \( \mathcal{P} \). This means computing another IOSTS denoted \( [\mathcal{P}] \), which has the same traces as \( \mathcal{P} \) (and thus, the same traces as the product \( \mathcal{P} = S \times \mathcal{T} \mathcal{P} \), but without nondeterministic choices.

The typical case of nondeterministic choice is represented in Figure 7: an (input or output) action \( a \) leading to two different effects on the variables and/or the control.

![Fig. 5. Sequence of Internal Actions.](image-url)
Eliminating nondeterminism in a symbolic transition system is a difficult problem in general. Here, we propose a simple heuristic that takes care of common situations, such as the one represented in Figure 7. The idea is to postpone the effect of a nondeterministic choice on the observable actions that follow it. In the situation represented below, this amounts to splitting the two transitions (with guards $G_1$, $G_2$ and assignments $A_1$, $A_2$) into three: one for the case $G_1 \land \neg G_2$ holds, another for the case when $G_2 \land \neg G_1$ holds, and the last for the case $G_1 \land G_2$ holds. In the latter case, the choice whether to assign the variables according to $A_1$ or to $A_2$ is postponed until the observable action that follows. A new location $q_{2,3}$ is introduced, which we call a copy of locations $q_2$, $q_3$. Thus, if $b$ is the next action, then the assignment $A_1$ should have been executed, and it is composed with the guard and the assignment corresponding to $b$ (which pro-
An Approach to Symbolic Test Generation 351

ducles the guard \( G_3 \circ A_1 \) and assignment \( A_3 \circ A_1 \). Similarly, if \( c \) is the next action, then \( A_3 \) should have been executed, which produces the guard \( G_3 \circ A_2 \) and assignment \( A_4 \circ A_2 \). Of course, if \( b \) and \( c \) are actually the same action, then the same scheme has to be applied again, and the whole procedure might not terminate. It can be shown that it does terminate if the maximal sequence of actions involved in nondeterministic choices (such as the sequence composed of action \( a \) in Figure 7) does not produce cycles in the graph of the IOSTS. Also, knowing that some guards are mutually exclusive can help the determinization procedure to terminate. For example, determinization leaves the IOSTS represented in Figure 6 unchanged, because wherever there are transitions with the same origin and labeled with the same action \( (MSG?, ACK!) \) and with several possible outcomes, the corresponding guards are mutually exclusive.

We associate to the IOSTS \( \mathcal{P} \) \( \{ \text{obtained after product, closure and determinization} \} \) a set of accepting locations \( \text{Accept}_{\mathcal{P}} \), which is the union of \( \text{Accept}_{\mathcal{P}} \) and of the set of new locations produced by determinization, which are copies of some location in \( \text{Accept}_{\mathcal{P}} \).

**Selection.** The next step consists in selecting a part of the control graph of the IOSTS \( \mathcal{P} \) that leads to the set of locations \( \text{Accept}_{\mathcal{P}} \), and in adding transitions to a new location \( \text{fail} \), such as to make the IOSTS input-complete. We define three subsets of the set of locations \( Q_{\mathcal{P}} \) as follows:

- let \( \text{Pass} \) denote the set \( \text{Accept}_{\mathcal{P}} \);  
- let \( \text{Leads2Pass} \) denote the set of locations \( q \in Q_{\mathcal{P}} \setminus \text{Pass} \) such that, in the graph \( \langle Q_{\mathcal{P}}, T_{\mathcal{P}} \rangle \), there exists a path (i.e., a sequence of contiguous locations and transitions) from the initial location \( q^0 \) to \( q \) and from \( q \) to \( \text{Pass} \);  
- let \( \text{Inconclusive} \) denote the set of locations \( q' \in Q_{\mathcal{P}} \setminus \text{Pass} \setminus \text{Leads2Pass} \) such that there is a transition in \( T_{\mathcal{P}} \) with origin in \( \text{Leads2Pass} \), destination \( q' \), labeled by an input \( a \in \Sigma_{\mathcal{P}} \).

Let also \( \text{fail} \) denote a new location \( (\text{fail} \notin Q_{\mathcal{P}}) \). We define the test case as an IOSTS \( \mathcal{T}_c = \langle D_{\mathcal{T}_c}, \Theta_{\mathcal{T}_c}, Q_{\mathcal{T}_c}, \Sigma_{\mathcal{T}_c}, \Sigma_{\mathcal{T}_c}, T_{\mathcal{T}_c} \rangle \) as follows: \( D_{\mathcal{T}_c} = D_{\mathcal{P}}, \Theta_{\mathcal{T}_c} = \Theta_{\mathcal{P}}, Q_{\mathcal{T}_c} = \text{Pass} \cup \text{Leads2Pass} \cup \text{Inconclusive} \cup \{ \text{fail} \}, q^0_{\mathcal{T}_c} = q^0_{\mathcal{P}} \) (if \( q^0_{\mathcal{P}} \notin \text{Pass} \cup \text{Leads2Pass} \) then the test case is not defined). The alphabet of the test case is \( \Sigma_{\mathcal{T}_c} = \Sigma_{\mathcal{P}}, \Sigma_{\mathcal{T}_c} = \Sigma_{\mathcal{P}}, \Sigma_{\mathcal{T}_c} = \emptyset \). The set of transitions \( T_{\mathcal{T}_c} \) consists of the transitions of \( T_{\mathcal{P}} \) with origin and destination in \( \text{Pass} \cup \text{Leads2Pass} \cup \text{Inconclusive} \), and of a set of new transitions, with origin in every \( q \in \text{Leads2Pass} \) and for each input \( a \in \Sigma_{\mathcal{T}_c} \). Every such new transition leads to \( \text{fail} \), and, for a given origin \( q \) and action \( a \), its guard is the negation of the disjunction of all the guards of transitions in \( T_{\mathcal{P}} \), with same origin \( q \) and action \( a \). The assignments of the new transitions are empty. They are used to make the test case input-complete, by letting go to \( \text{fail} \), from any state, any valued input that is not allowed by the specification.

For the IOSTS partially represented in Figure 6, the selection operation eliminates precisely the parts that are not represented. The set of locations \( \text{Pass} \) consists of only one element \( (\text{WT2Accept}) \). The set \( \text{Inconclusive} \) consists of
locations WA Reject, WT2 Reject. For simplicity, the location fail, and the transitions leading to it, are not represented. By comparing the IOSTS in Figures 6 and 4, it is clear that the former still needs considerable simplification. This is the role of the simplification operations, which we describe in more detail in the next section.

However, the IOSTS TC obtained so far has all the properties required from a test case: it is input-complete (this is obtained by adding the new transitions to fail), it is initialized (this is inherited from the specification and the test purpose, and is preserved by subsequent transformations), it is deterministic (this is obtained by closure and determinization), and furthermore:

**Proposition 3 (Correctness of Test Generation).** For any specification $S$ and test purpose $TP$ of $S$, the test case $TC$ obtained from $S$ and $TP$ by test generation is correct for $S$, $TP$, and the class of implementations that have the same inputs, outputs, and signatures as $S$.

**Proof outline.** We have to prove that the four properties of definition 3 (soundness, relative completeness, accuracy and conclusiveness) hold. First, it is easy to see that, by construction, $TC$ has the same parameter set as $S$ and that, for an arbitrary implementation $I$ which has the same inputs, outputs, and signatures as $S$, $I$ is compatible with $TC$ for parallel composition. (Thus, the valued inputs of $I$ are the valued outputs of $TC$ and reciprocally.) We consider an arbitrary instance $TC(\pi)$ of $TC$, and the corresponding instances $S(\pi)$ of the specification $S$ and $(S \times TP)(\pi)$ of the synchronous product $S \times TP$. As in Section 3, we denote by $Pass$ (respectively Inconclusive and Fail) the set of states of $TC(\pi)$ whose locations are in the set $Pass$ (resp. Inconclusive and {fail}). In the following proofs, we implicitly make use of Proposition 1 (i.e., when we say a trace $\sigma$ brings the $TC(\pi)$ in a given location, we implicitly assume that $TC(\pi)$ after $\sigma$ is a singleton).

**Soundness:** Let $\sigma \in traces(I)[TC(\pi)]$ such that $verdict(TC(\pi), I, \sigma) = Fail$. By the first item of Proposition 2, $\sigma$ is a trace of $I$ and a trace of $TC(\pi)$. By definition of $verdict$, we obtain $TC(\pi)$ after $\sigma \subseteq Fail$, meaning that $\sigma$ leads the test case to the location fail. Consider the prefix $\sigma'$ of $\sigma$ obtained by removing its last valued action $\alpha$. By construction of the test case, we obtain that $\alpha$ is a valued input of $TC(\pi)$ (thus, a valued output of $I$). $\sigma' \in traces((S \times TP)(\pi))$, and $\sigma = \sigma' \cdot \alpha \in traces((S \times TP)(\pi))$. Using the second item of Proposition 2, we obtain that $\sigma' \in traces(S(\pi))$ and $\sigma \notin traces(S(\pi))$, thus, $\alpha \notin out(S(\pi)\ after\ \sigma')$. But $\sigma = \sigma' \cdot \alpha$ is a trace of $I$ and $\alpha$ is a valued output of $I$, thus $\alpha \in out(I\ after\ \sigma')$. Hence, $I$ does not conform to $S(\pi)$.

**Relative completeness:** Suppose that $I$ does not conform to $S(\pi)$ relative to $TP$. By definition of $conf$, there exists a trace $\sigma' \in prefix(Atraces((S \times TP)(\pi)))$ and a valued output $\alpha$ of the implementation such that $\sigma \not\in out(I \ after \ \sigma')$ but $\alpha \not\in out(S(\pi)\ after\ \sigma')$. We choose a trace $\sigma'$ which is minimal with this property, and let $\sigma = \sigma' \cdot \alpha$. Thus, $\sigma \in traces(I)$ and $\sigma \not\in traces(S(\pi))$. From $\sigma' \in prefix(Atraces((S \times TP)(\pi)))$ and the minimality of $\sigma$, we obtain (by construction of the test case) also $\sigma' \in traces(TC(\pi))$ and $(TC(\pi)\ after\ \sigma') \cap
(\text{Fail} \cup \text{Pass} \cup \text{Incon}) = \emptyset$; thus, $\sigma'$ brings the test case to a location $q$ which is not in $\text{Pass} \cup \text{Inconclusive} \cup \{\text{fail}\}$, thus $q$ is input-complete. But $\alpha$ is also an input of the test case, thus, the trace $\sigma = \sigma' \cdot \alpha$ is also in $\text{traces}(TC(\pi))$. Now, $\sigma \in \text{traces}(TC(\pi))$ and $\sigma \notin \text{traces}(S(\pi))$ implies, by construction of the test case, that $TC(\pi)$ after $\sigma \subseteq \text{Fail}$. As $\sigma \in \text{traces}(I)$ and $\sigma \in \text{traces}(TC(\pi))$, we obtain, by the first item of Proposition 2, that $\sigma \in \text{traces}(I||TC(\pi))$. The latter together with $TC(\pi)$ after $\sigma \subseteq \text{Fail}$ implies $\text{verdict}(TC(\pi), I, \sigma) = \text{Fail}$.

Accuracy: By definition, for a trace $\sigma \in \text{traces}(I||TC(\pi))$, $\text{verdict}(TC(\pi), I, \sigma) = \text{Pass}$ if $TC(\pi)$ after $\sigma \subseteq \text{Pass}$. By construction of the test case, the traces $TC(\pi)$ leading to a state in the set $\text{Pass}$ are exactly those in $\text{Atraces}((S \times TP)(\pi))$, which proves the property.

Conclusiveness: By definition, for $\sigma \in \text{traces}(I||TC(\pi))$, $\text{verdict}(TC(\pi), I, \sigma) = \text{Inconclusive}$ if $TC(\pi)$ after $\sigma \subseteq \text{Incon}$. If $\sigma$ leads to a state in the set $\text{Incon}$, by construction of the test case, $\sigma$ is a trace of $(S \times TP)(\pi)$, thus (by the second item of Proposition 2), $\sigma \in \text{traces}(S(\pi))$. We prove the second part of the property by contradiction. If we had $\sigma \in \text{pref(Atraces}((S \times TP)(\pi)))$, then, by construction of the test case, the location of $TC(\pi)$ after $\sigma$ would be in the set $\text{Leads2Accept}$, not in $\text{Inconclusive}$. This concludes the proof.

Cycles of Internal Actions. We consider a class of specifications that contain cycles of internal actions, and for which it is still possible to generate test cases with all the correctness properties except relative completeness (cf. Definition 3). This class includes specifications described, e.g., in an imperative programming language, where cycles of internal actions correspond to deterministic control structures like iterations or recursions. For a specification $S$ with locations $Q$ and transitions $T$, let $T^{\text{int}} \subseteq T$ denote the subset of transitions labeled by internal actions (which we call for short internal transitions). Let $Q^{\text{int}} \subseteq Q$ denote the set of locations such that all transitions with origin in $Q^{\text{int}}$ are internal. We call $Q^{\text{int}}$ the set of internal locations.

The test generation process is redefined in the following way. The product operation is unchanged. The elimination of internal actions is applied only to internal transitions whose origins are in $Q \setminus Q^{\text{int}}$. (Thus, if these transitions do not form cycles, the operation terminates, as for example in the case of the IOSTS represented in Figure 3). The determinization operation is the same, and the selection operation is unchanged, except that it does not add transitions leading to $\text{fail}$ from the internal locations.

To show that the obtained test case $TC$ is correct (except for relative completeness), we prove the soundness, accuracy and conclusiveness like in the proof of Proposition 3. (However, we cannot use Proposition 1 any more, since $TC$ now has internal actions). The proof of relative completeness does not work, because of the internal locations, which are not input-complete. Intuitively, the test case can miss some inputs while it is executing its internal actions. In practice, this problem can be fixed by buffering the inputs for later observation.

6 Simplifying the Test Cases

The test generation procedure defined in the previous section consists essentially of syntactical transformations (which extend the algorithms implemented in the tool TGV [13] to handle symbolic data). Although it does produce correct test cases (e.g., the one represented in Figure 6), the latter are almost never what the user would expect (e.g., something like Figure 4). Indeed, there are unreachable parts, redundant variables, guards and assignments can be simplified, etc. Thus, we need semantic-based simplification techniques that preserve the correctness of test cases. We present an experiment with the HyTech model checker [9] and the PVS theorem prover [17] for simplifying the test case of the BRP, and indicate other potentially useful techniques.

We define translations of IOSTS into the HyTech and PVS input languages. All the corresponding files (PVS specifications and proofs, and HyTech scripts) can be downloaded from http://www.irisa.fr/pampa/perso/rusu/IFM00/.

Translation to HyTech. HyTech is a model checker originally designed for hybrid systems verification. It handles discrete variables (integers) through a polyhedral library [7]. We had to modify the tool to re-include a widening operator that forces convergence of fixpoints. As HyTech does not handle uninterpreted function symbols, the translation of the test case represented in Figure 6 involves replacing every conjunct containing the uninterpreted function symbol $f$ with \textit{true}. Also, the actions become uninterpreted labels of the HyTech transitions.
Translation to PVS. PVS is a general-purpose theorem prover with a rich input language (typed higher-order logic). Consequently, the translation of IOSTS into PVS preserves all the information and is straightforward.

Simplifying the Test Case. The simplification of the test case represented in Figure 6 is done in two steps. First, HyTech (with widening) is used to automatically generate a set of linear invariants\(^1\). This gives the following output:

Location: Pass
\[ \text{rn} = 1 \land \text{head} = \text{last} + 1 \land \text{head} = j \land \text{head} \geq 2 \land \text{max} \geq 2 \]

Location: Inconclusive
\[ j = \text{head} + 1 \land \text{head} \geq 1 \land \text{rn} \geq 1 \land \text{max} \geq 2 \land \text{head} \leq \text{last} \]

Location: SC\(_{P1}\)
\[ \text{rn} = 1 \land \text{head} = \text{last} + 1 \land \text{head} = j \land \text{head} \geq 2 \land \text{max} \geq 2 \]

Location: WA\(_{P1}\)
\[ \text{rn} = 1 \land j = \text{head} + 1 \land \text{head} = j \land \text{head} \geq 2 \land \text{max} \geq 2 \land \text{head} \leq \text{last} \]

Location: SF\(_{P1}\)
\[ \text{rn} = 0 \land \text{head} = j \land \text{head} = 1 \land \text{max} \geq 2 \land \text{head} \leq \text{last} \]

Location: WR\(_{P1}\)
\[ \text{rn} = 0 \land \text{head} = 1 \land j = 1 \land \text{max} \geq 2 \land \text{last} \geq 1 \]

Thus, e.g., whenever control is at location SF\(_{P1}\), the relation \(\text{rn} = 0 \land \text{head} = j \land \text{head} \leq \text{last}\) holds between the variables of the IOSTS. This is useful information: in particular, it shows that location WT2\_Reject is not reachable, thus, it can be eliminated, together with all six transitions leading to it. This also indicates another transition that could be eliminated: the transition from SF\(_{P1}\) to WA\(_{Reject}\). This is because its guard \(f(head - 1) \neq f(j - 1) \lor j > last\), in conjunction with the invariant produced by HyTech, evaluates to \text{false}. However, HyTech cannot evaluate conditions with uninterpreted function symbols, thus, the simplified IOSTS, together with the generated invariants, are translated to PVS, which automatically proves the following invariant:

\[
\%\text{proved (transition from SF\(_{P1}\) to Inconclusive is never fireable)}
\]
\[
\text{trans1_never_fireable: LEMMA invariant(\text{NOT LAMBDA(s):State):EXISTS(m:Msg): control(s)=SF\(_{P1}\) AND m=f(s)(head(s)-1) AND (j(s)>last OR m/=f(s)(j(s)-1)))}
\]

This allows to eliminate the transition from SF\(_{P1}\) to WA\(_{Reject}\), which produces the IOSTS represented in Fig. 8. Other simplifications are then made: given the invariant relations between head and j in locations SF\(_{P1}\), WA\(_{P1}\), and SC\(_{P1}\), it is clear that j can be replaced by head and eliminated altogether from the IOSTS, since it has no more influence on the variables. The same holds for \text{rn}, as the only test \text{rn} < \text{max} on this variable always evaluates to \text{true}. Finally, we obtain the test case represented in Figure 4.

Clearly, other program analysis and verification techniques can be useful for symbolic test generation. We are investigating slicing [22] (see [4] for a use of slicing techniques in the context of conformance test generation), and automatic theorem-proving in rich, yet decidable theories such as WS1S [14] or the

\(^1\) Inconclusive in the HyTech output corresponds to WA\(_{Reject}\) in Figure 6.
quantifier-free fragment of Presburger arithmetic with uninterpreted function symbols [20]. Integrating these algorithmic and deductive techniques into an environment for symbolic test generation is a longer-term project.

7 Conclusion and Future Work

In this paper we describe some basic steps towards the generation of symbolic test cases in the form of extended transition systems with parameters and variables. The approach generalizes existing work on test generation using enumerative methods [1,6]. The generated test cases satisfy some correctness properties, which means essentially that they always emit the right verdict. To be useful in practice, the test cases have to be simplified using automated static analysis and proof strategies. This is a first direction for continuing work.

Another future work direction (with the same goal of obtaining simpler test cases) is to generate test cases from an abstraction of the specification, rather than from the (concrete) specification itself. The usual notion of abstraction (over-approximation, cf. [2]) cannot be applied here, because of the asymmetry between inputs and outputs in the conformance relation. Over-approximating the outputs of the specification is acceptable (the test cases remain sound, although they lose completeness and accuracy), but over-approximating the inputs may lead to unsoundness (the test case explores behaviours that are not in the specification, which may produce false Fails). We need abstractions that over-approximate the outputs and under-approximate the inputs.

Third, symbolic test cases need to be instantiated before they can be executed. An analysis of their behaviours, depending on their parameters and messages, may be useful to find domains of equivalent values. Then, under uniformity hypotheses [3] it is enough to instantiate the parameters and messages to one value in each domain. Also, test generation needs two inputs: specifications and test purposes. Designing test purposes by hand from the specification can be difficult, in particular if a good coverage of the specification is targeted. We believe that classical structural coverage criteria [19], combined with symbolic analysis, may yield interesting test purposes.

Finally, the techniques described in this paper can be used in (or fertilized by) techniques of other domains. In particular, there are obvious similarities between test generation and controller synthesis [18], where a controller has to be derived from the specification of a program with respect to a control plan.

References

Behavioral Conformance Verification in an Integrated Approach Using UML and B

Eric Meyer\(^1\) and Thomas Santen\(^2\)

\(^1\) LORIA - Université Nancy 2 - UMR 7503, Campus scientifique, BP 239, 54506 Vandœuvre-les-Nancy Cedex - France, \(^2\) Technische Universität Berlin

Abstract. We propose an integration of diagrammatic object oriented modeling techniques with a formal specification and verification technique. We translate UML class diagrams to B abstract machines in a way that does not only provide a formal interpretation of the class diagrams but that also allows us to verify properties of object oriented models within the framework of the B method. Specifically, we address translating generalization / specialization hierarchies to B. An appropriate construction of B components allows us to express and formally verify behavioral conformance, which ensures that polymorphism can be used safely. Expressing the proof obligations associated with behavioral conformance by constructing B components makes it possible to use the tool AtelierB for mechanically verifying them.

1 Introduction

The Unified Modeling Language (UML) [14] has become a de-facto standard notation for describing analysis and design models of object-oriented software systems. The mostly graphical description of models is easily accessible. Developers and their customers intuitively grasp the general structure of a model and thus have a good basis for discussing the requirements of a system and their possible implementation.

When it comes to more detailed questions, however, the fact that the UML lacks a precise semantics is a serious drawback. Thus, even if it is in principle possible to document requirements in some detail, for example, describe a behavioral constraint on a method, its semantics is unclear. This can result in misconceptions by developers or customers. The imprecise semantics also hinders the development of tools for analyzing models that achieve more than just a syntactic check of mutual consistency between models.

An approach to ameliorate that deficiency is to map part of the UML to a formal language with a precise mathematical semantics and – hopefully – tool support. We use the B abstract machine notation [1] for that purpose. In previous work [11], we have shown how to represent class diagrams and statecharts by B abstract machines.

In the present paper, we focus on the generalization / specialization relationship in UML class diagrams. We present an automatable mapping from UML class diagrams to B
abstract machines. Roughly, we map a UML class to a B abstract machine. Thus, one machine represents the objects of a particular class. We then augment the resulting B specification by logical properties that are not described in the UML class diagram.

We use the refinement relation defined for B abstract machines to analyze the classes in a generalization / specialization relation for behavioral conformance. This is the semantic relation between classes that allows the clients of a class hierarchy to use polymorphism safely. The tool \texttt{AtelierB} [19] generates verification conditions for B abstract machines that allow us to prove behavioral conformance between classes that are described in UML. \texttt{AtelierB} also supports proving those verification conditions in an interactive fashion.

In Sect. 2, we motivate the concept of behavioral conformance by way of an example. We define behavioral conformance independently of the B formalism. In Sect. 3, we represent UML classes by B machines. In Sect. 4 and Sect. 5, we show how the proof obligations generated by \texttt{AtelierB} relate to the definition of behavioral conformance given in Sect. 2. Section 4 considers the operation refinement conditions on the methods that are present in the more general class of a generalization / specialization relation. Section 5 deals with the condition on history constraints that is relevant for so-called extra methods of the more specialized class of a generalization / specialization relation that do not have a counterpart in the more general class. We use a number of auxiliary machines to obtain proof obligations in B that correspond to the obligations for the history constraint. Section 6 summarizes our experience with the tool \texttt{AtelierB}. We conclude in Sect. 7 by relating our work to the work of others and discussing what we have achieved.

## 2 From Class Diagrams to Behavioral Conformance

We motivate the analysis we carry out for generalization / specialization relationships by way of an example taken from [9]. The purpose of this section is to introduce the concepts informally. Formal definitions have been given for algebraic specifications in [9] and for an object oriented extension of Z [18] in [15].

Figure 1 shows a UML class diagram for two specializations of bags. The class \texttt{Bag} models a container data structure that is a multi-set. \texttt{Bag} is a generic class: it is parameterized with the type \texttt{ELEMENT}. For each item \texttt{i} of type \texttt{ELEMENT}, the attribute \texttt{elements} counts the number of occurrences of \texttt{i} in the bag. The attribute \texttt{bound} is the maximal number of items that a given bag can contain. An instance of the class has three operations: \texttt{newBag} creates a bag with a given bound, \texttt{get} returns an arbitrary item in the bag and deletes it from the bag, and \texttt{put} adds a new item to the bag.

The classes \texttt{DynamicBag} and \texttt{StaticBag} are two specializations of \texttt{Bag}. An instance of \texttt{DynamicBag} may change the value of its \texttt{bound}: the operation \texttt{chg bound} may increase the value of \texttt{bound} to the value of its parameter, but no operation on a dynamic bag may decrease the bound of the bag. The constraint attached to the class box of \texttt{StaticBag} requires that the bound of a \texttt{StaticBag} cannot be changed. Additionally, the operation \texttt{get} is specialized by \texttt{StaticBag}: for an instance of \texttt{StaticBag}, \texttt{get} returns the item that the bag contains most often.
Behavior specification. The class diagram in Fig. 1 does not provide information about
the intended behavior of the classes as described by the preceding explanatory text. When we translate that diagram to a B model in Sect. 3, we will augment the information derived from the class diagram with specifications of the behavior of the operations of the classes.

Specialization versus behavioral conformance. If we model a generalization / specialization relationship as the one of Fig. 1, we wish to know whether or not that relation is a behaviorally conforming specialization. If a specialization is behaviorally conforming, then instances of the subclass can safely be used where instances of the superclass are expected, i.e. polymorphism is safe. By “safe”, we mean that the behavior that an object exhibits if it is accessed through the interface of the superclass conforms the expectations one has about the behavior of instances of the superclass – there are no “surprises”. For example, if we access an object through the interface of Bag, we should have no way of determining whether that object is an instance of DynamicBag, StaticBag, or a proper instance of Bag. If the specialization was not behaviorally conforming, then it would be unsafe to polymorphically assign an instance sbag of StaticBag, say, to a variable of type Bag, because then using sbag as if it were an instance of Bag might lead to runtime errors or otherwise undesired behavior of the implemented system.

Extra methods. As America [2] pointed out, behavioral conformance\(^1\) is strongly related to classical data refinement. Liskov and Wing [9] extend America’s definition by taking

\(^1\) America and others call behavioral conformance “behavioral subtyping”. We avoid the term “subtyping”, because behavioral conformance is a logical relation that is independent of a particular type system.
extra methods into account. Extra methods are the operations of a subclass that are not present in the interface of its superclass. For example, the operation `chg_bound` of `DynamicBag` is not defined for arbitrary `Bag`

In the presence of aliasing, extra methods may provoke “unexpected” behavior. Consider an instance `dbag` of `DynamicBag`. If we access `dbag` only through the interface of `Bag`, there is no way of changing `dbag.bound`. If, however, `dbag` is accessed through the interface of `DynamicBag`, then the bound may be changed by calling `dbag.chg_bound`. That change then is observable through the interface of `Bag`, for example, by reading the value of the attribute `bound`.

Is changing the value of `bound` “unexpected” for an instance of `Bag`? The answer depends on the interpretation of what behavior `Bag` specifies. A very conservative view might argue that changing the bound is indeed unexpected, because `Bag` provides no way of doing so. With that view, the expected behavior of a class would be inductively defined by the state transitions that its operations allow. That view is too restrictive for realistic cases. In [15], we have studied the container hierarchy of the Eiffel Base Libraries [10]. The root class in that hierarchy does not have an operation which changes the state of an object. With the restrictive notion of behavioral conformance, any specialization of that class which modifies the container would not behaviorally conform to the superclass.

Posing no restrictions on the behavior of extra methods is also not adequate. Then, a `DynamicBag` could be a specialization of `StaticBag`, contradicting the constraint that the bound of a static bag must not change.

**History constraints.** Those considerations have led Liskov and Wing [9] to propose the concept of a history constraint as an additional means to specify the acceptable behavior of the instances of a class. We [15] have translated their definition from the context of algebraic specifications to model-based specifications in Object-Z [17]. In that context, a history constraint `HC(s, s')` is a binary relation between the states of the objects of a class, such that

- it respects the invariant `I` of the class, i.e. `I(s) ∧ HC(s, s') ⇒ I(s')`, and that
- the specifications of all operations `Op` – viewed as relations between before and after states – respect the history constraint, i.e. `I(s) ∧ I(s') ∧ Op(s, s') ⇒ HC(s, s')`.

Thus, a history constraint may be regarded as an anonymous operation that provides all possible behaviors of objects of a class – including the objects of any specialization of the class. If `H(s, s')` satisfies those two conditions, we call the class specification history consistent2.

**Behavioral conformance.** Given the concept of a history constraint, we can define behavioral conformance more precisely: A class `C` behaviorally conforms to a class `A` if

- `C` is a data refinement of `A` if we view both classes as abstract data types, and

2 History consistency is a necessary condition for the internal consistency of a class specification [15]. The other conditions that make up internal consistency are similar to the ones on B abstract machines enforced by the B method (c.f. Sect. 4.2), e.g., that the initialization of a class establishes the invariant and that operations preserve the invariant.
– the history constraint of $C$ implies the history constraint of $A$, $HC_c \Rightarrow HC_a$.

The first condition ensures that any redefinition of an operation in a specialization is a refinement of the overwritten operation. The second condition ensures that any extra method – which must conform to the history constraint of $C$ – conforms to the history constraint of $A$, i.e. it does not exhibit unexpected behavior.

3 From UML Class Diagrams to B Abstract Machines

3.1 The B Method

B [1] is a formal software development method that covers the software process from the specification to the implementation. The B notation is based on set theory, the language of generalized substitutions, and first order logic. Specifications are composed of abstract machines similar to modules or classes; they consist of a set of variables, invariance properties relating those variables, and operations. The state of the system, i.e. the set of variable values, is only modifiable by operations. Machines can be composed in various ways. Thus, large systems can be specified in a modular way, possibly reusing parts of other specifications. Refinement of a B model allows developers to derive a correct implementation in a systematic way. Refinement can be seen as an implementation technique but also as a specification technique to progressively augment a specification with more detail. At every stage of the specification, proof obligations ensure that operations preserve the system invariant. A set of proof obligations that is sufficient for correctness must be discharged when a refinement is postulated between two machines.

3.2 Integrating UML and B

This section presents the translation of some of the UML class diagram concepts into a B specification. We focus on the formalization of classes and inheritance. More information on our integration of UML and B can be found in [11]. Most of the B machines we present in this and the following sections can be generated automatically from the UML model. Only the parts of the machines shaded gray must be added manually.

Classes, attributes, and methods. A formal representation of the class Bag shown in Fig. 1 is given in Fig. 2. We describe a class by an abstract machine which models a collection of instances. Names of formal parameters are expressed in brackets after the name of the machine. The abstract machine defines the constant $BAG$ which is the set of the identities of possible instances of the class. Therefore, it is a subset of the set $INSTANCES$ of all possible object identities. The set of identities of existing instances is modeled by the variable $bag$.

Because a B machine represents all instances of a class, not just a particular object, we model attributes by relations between object identities and attribute values. Depending on the attribute characteristics specified in the UML class diagram, a relation between object identities and attribute values can be restricted further. For instance, if the attribute is mandatory and mono-valued, then the relation modeling that attribute is required to be a total function. Thus, we have two functions in Fig. 2, elements and bound, which
Behavioral Conformance Verification in an Integrated Approach

MACHINE Bag (ELEMENT)

CONSTANTS Bag

PROPERTIES Bag ⊆ INSTANCES

VARIABLES bag, elements, bound

INVARIANT

bag ⊆ Bag ∧ elements ∈ bag ∧ bound ∈ Bag → NAT ∧

INITIALISATION

bag, elements, bound := ∅, ∅, ∅

OPERATIONS

newBag(bnd) =
  pre
  bnd ∈ NAT ∧ not(Bag = bag)
  then
  any new where
  new ∈ Bag ⇒ bag ∧
  elements(new) := ELEMENT x [0] ∧
  bound(new) := bnd
  end

rmBag(oo) =
  pre
  oo ⊆ bag
  then
  bag := bag \ oo || elements := oo \ elements ||
  bound := oo \ bound
  end

elt ← get(oo) =
  pre
  oo ∈ bag
  then
  any ii where
  ii ∈ ELEMENT ∧
  elements(oo)(ii) > 0
  then
  elements(oo)(ii) :=
  elt := ii
  end

put(oo, ii) =
  pre
  oo ∈ bag ∧
  ii ∈ ELEMENT ∧
  elements(oo)(ii) < MAXINT
  then
  elements(oo)(ii) := elements(oo)(ii) + 1
  end

END

Fig. 2. A formal representation of the class Bag

model the attributes of class Bag. The INVARIANT clause declares them to be functions mapping object identities to attribute values. All variables are initialized to the empty set, because, upon system startup, no instances of Bag exist.

Constructors and destructors of objects are defined as B operations. The return value of the constructor newBag is an instance of the class. This instance is nondeterministically chosen from the set Bag - bag. A constructor may have parameters for setting attribute initial values, such as bnd in the example.

A class diagram usually specifies just the signature of a method but not its functionality. Therefore, we augment the B specification of operations modeling methods by preconditions and generalized substitutions. For the methods get and put, those are the parts shaded gray in Fig. 2: for get, we nondeterministically choose an element ii of the bag oo, return that element and reduce the number of ii in bag oo by one; for put, we extend the number of elements ii in oo by one. The precondition elements(oo)(ii) < maxint is necessary, because the set NAT is the finite interval from 0 up to maxint.

Inheritance. We simulate inheritance in B according to the idea that an object of a subclass is an object of its ancestor classes and that the set of objects of a subclass is a subset of the set of objects of its superclass. We generate an abstract machine for each subclass. That abstract machine uses the same set of possible object instances as the
MACHINE Dynamic_Bag
USES Bag
VARIABLES dynamic_Bag
INVARIANT dynamic_Bag ⊆ bag
INITIALISATION dynamic_Bag := ∅
OPERATIONS
  addDynamic_Bag(new) = pre new ∈ BAG − bag then dynamic_Bag := dynamic_Bag ∪ {new} end;
  supprDynamic_Bag(oo) = pre oo ⊆ dynamic_Bag then dynamic_Bag := dynamic_Bag − oo end;
END

MACHINE Static_Bag
USES Bag
VARIABLES static_Bag, sizeSB
INVARIANT static_Bag ⊆ bag ∧ sizeSB ∈ static_Bag → NAT
INITIALISATION static_Bag, sizeSB := ∅, ∅
OPERATIONS
  addStatic_Bag(new) = pre new ∈ BAG − bag then static_Bag := static_Bag ∪ {new} || sizeSB(new) := 0 end;
  supprStatic_Bag(oo) = pre oo ⊆ static_Bag then static_Bag := static_Bag − oo || sizeSB := oo < sizeSB end;
END

Fig. 3. A formal representation of subclasses Dynamic_Bag and Static_Bag

machine representing its superclass. The set of existing instances of the subclass is a subset of the one of the superclass. New attributes of the subclass are modeled as before by binary relations between the set of existing instances and the types associated to the attributes.

Figure 3 shows the formalization of the subclasses Dynamic_Bag and Static_Bag of Bag (c.f. Fig. 1). The variables dynamic_Bag and static_Bag are the instance variables of the two classes. They are subsets of bag, which is visible in the machines representing the two classes, because they refer to the machine Bag by a USES clause. Thus, they have read-only access to the variables of Bag. The variables of the used machine may be referred to in invariants and in operation specifications.

However, the B notation does not allow a machine to change the state of a machine it uses. This is possible for a machine that INCLUDES another one. In contrast to the USES construct, a machine may only be included by at most one other machine. Therefore, neither construct alone is suited to model subclasses by referring to a machine that models their (common) superclass.

Figure 4 shows the hierarchy of B machines that models the inheritance hierarchy of Fig. 1. Each class gives rise to an abstract machine, the superclass machine (Bag) is used by all its subclass machines (Static_Bag, Dynamic_Bag). These machines are all
Behavioral Conformance Verification in an Integrated Approach

Fig. 4. Machine hierarchy modeling the subclasses of Bag

included in the top-level machine called Bag Interface that declares the interface of all objects belonging to one of the classes of the inheritance hierarchy.

Typically, if a machine is referred to in a number of specification machines, then they should all be included (by INCLUDES) in another one. The operations of included machines are not regarded as operations of the including machine. They can, however, be promoted (by the clause PROMOTES) to become operations proper of the including machine. Those visibility rules of B machines ensure that the consistency of the constructions in a B specification can be checked by relatively simple verification conditions. Those rules force us, however, to distribute the formalization of methods over several machines, and to model late binding by dispatching in a top-level “interface machine”, as we will explain in the following.

The methods of all classes of our example inheritance hierarchy are accessible via the interface machine shown in Fig. 5. To model inheritance, we must revise the definition of the constructor newBag (c.f. Fig. 2): We split that definition into two. One called addBag replaces newBag in the machine Bag. It adds an object to the set of existing instances of bags and initializes its attribute values. The second is the definition of the constructor newBag, which now resides in the interface machine. It nondeterministically chooses the object instance new to be created and calls addBag. With that mechanism, we can create a new instance of the class Bag as well as one of a subclass as a specialization of Bag. The operations newDynamicBag and newStaticBag accomplish this.

We must also modify the definitions of operations representing methods such as put and get which are inherited and possibly redefined by the subclasses. We add the name of the class as a suffix to the names of the operations in the machines defining the functionality of the single classes (put_bag, put_static_bag, etc.). In the interface machine, we define new operations put and get, which call the other operations according to membership of the parameter object oo in one of the instance sets. Thus, we simulate late binding by explicitly dispatching operation calls in the interface machine.

In general, operation definitions must to be put either in the interface or in the machine defining the class, depending on the state variables that need to be modified. Operations are defined in the machine defining the class if the variables they modify are declared in this machine; operations modifying variables of other machines must be defined in the interface machine. For example, chg_bound and get_static_bag are defined in the interface because they modify two variables of the subclass and of the superclass. In contrast,
MACHINE Bag_Interface(ELEMENT)

INCLUDES
	Bag(ELEMENT),
	Dynamic_Bag, Static_Bag

DEFINITIONS

get_static_bag(oo) = ...
put_static_bag(oo, ii) = ...

OPERATIONS

oo ← newBag(bnd) = pre
	bnd ∈ NAT ∧
	not [BAG = bag]
then
	any new where
	new ∈ BAG = bag
then
	addBag(new, bnd) ∥
	oo := new
end
end;

oo ← newDynamic_Bag(bnd) = pre
	bnd ∈ NAT ∧
	not [BAG = bag]
then
	any new where
	new ∈ BAG = bag
then
	addBag(new, bnd) ∥
	addDynamic_Bag(new) ∥
	oo := new
end
end;

oo ← newStatic_Bag(bnd) = ...
rmvBag(oo) = pre
	oo ⊆ bag − (dynamic_Bag ∪ static_Bag)
then
	supprBag(oo)
end;

rmvDynamic_Bag(oo) = pre
	oo ⊆ dynamic_Bag
then
	supprDynamic_Bag(oo) ∥
	supprBag(oo)
end;

rmvStatic_Bag(oo) = ...
put(oo, ii) = pre
	oo ∈ bag ∧
	ii ∈ ELEMENT ∧

elements(oo)(ii) < maxint
then
	If oo ∈ static_Bag then
	put_bag(oo, ii) ∥
	put_static_bag(oo, ii)
else
	put_bag(oo, ii)
end
end;

elem ← get(oo) = pre
	oo ∈ bag
then
	If oo ∈ static_Bag then

get_static_bag(oo)
else

elem ← get_bag(oo)
end
end;

chg_bound(oo, ii) = pre
	oo ∈ dynamic_Bag ∧
	ii ∈ NAT ∧
	ii ≥ bound(oo)
then
	chg_state_bag(oo, elements(oo), ii)
end;

END

Fig. 5. Interface machine of the Bag inheritance hierarchy
1. $\forall bg. (bg \in bag \Rightarrow \sum_{xx. (xx \in ELEMENT \mid elements(bg)(xx)) \leq bound(bg)})$

2. $\forall oo. (oo \in static_Bag \Rightarrow \sum_{xx. (xx \in ELEMENT \mid elements(oo)(xx)) = sizeSB(oo)})$

3. $\text{dynamic}_Bag \cap \text{static}_Bag = \emptyset$

Fig. 6. Additional invariants for bags

put_bag is defined in the machine Bag, because it only changes the variables of this machine. The operation chg_bound refers to an operation called chg_state_bag that sets the attribute values of a bag to the values of its actual parameters. That operation, and others to change the attributes defined on other B machines are internal operations. We need them to provide a means of changing attribute values by operations that are not defined in the B machine that declares the state variables representing those attributes. Because the specifications of those “change state” operations are just parallel assignments, we do not show them in the figures.

Constraints. Additional properties that restrict the semantics of one or more elements of the UML model can be added in the B specification. They are expressed as predicates in the invariant clause of either the interface machine or the class machine. Again, which machine is the appropriate place to put them depends on the variables they need to refer to.

Figure 6 shows some of the constraints we used to augment the model derived from the UML diagram. The first one is defined in the machine Bag and restricts the numbers of elements of a bag to be smaller than its bound. The second constraint is specified in the Interface machine and forces the number of elements of a static_bag to be strictly equal to its size. The third one is formalized in the Interface machine and requires the specializations static_bag and dynamic_bag to be disjoint.

As a consequence of augmenting the model with those properties, we need to strengthen the preconditions of several operations such that they are guaranteed to establish those constraints as invariants. The figures in the following sections show some of the properties we had to add to the Bag model.

4 Behavioral Conformance I: Data Refinement

Translating UML class hierarchies to B models and augmenting them with behavioral specifications, we provide a formal basis to semi-automatically prove properties of the classes in the hierarchy. In the present and the following section, we show how to use the B method and the tool AtelierB to prove behavioral conformance in a gen / spec hierarchy (c.f. Sect. 2). In doing so, we face two problems: first, we must provide a definition of behavioral conformance in terms of the B language; second, we must find a way to make AtelierB generate proof obligations that are sufficient for showing behavioral conformance. In the present section, we focus on the first condition of behavioral conformance, namely data refinement. Section 5 deals with the proof obligations concerning history constraints.
4.1 Data Refinement in B

The definition of behavioral conformance in [15] is based on the classical notion of data refinement for Z [18], i.e. it is a forward simulation based on Z-style relational pre- and postcondition specifications of operations. More formally, a class $C$ data refines a class $A$ if there exists a relation $CI(c, a)$, the coupling invariant between the states $c$ of instances of $C$ and the states $a$ of instances of $A$ such that the following conditions hold for $op$:

- the initialization of $C$ is sufficient for the initialization of $A$
  \[ \forall c. \, CInit(c) \Rightarrow (\exists a. \, CI(c, a) \land AInit(a)) \]

and for each pair of operations $op_a$ and $op_c$ which are the respective specifications of the same method in $A$ and $C$, respectively:

- the precondition of $op_a$ is sufficient for the precondition of $op_c$
  \[ \forall a, c. \, (pre\, op_a)(a) \land CI(c, a) \Rightarrow (pre\, op_c)(c) \]

- each concrete state transition has an abstract counterpart
  \[ \forall a, c, c', (pre\, op_a)(a) \land CI(c, a) \land opc(c, c') \Rightarrow (\exists a'. \, CI(c', a') \land op_a(a, a')) \]

Here, the precondition $(pre\, op)(s)$ of an operation specification is defined by

\[ (pre\, op)(s) \Leftrightarrow (\exists s'. \, op(s, s')) \]

The refinement of B machines [1] is defined in the semantic framework of set transformers, which are isomorphic to predicate transformers. Relational forward simulation can be considered as a special case of data refinement in a predicate transformer setting [6]. Therefore, it is plausible\(^3\) that proving the refinement of properly defined B machines is an adequate basis to prove the data refinement condition of behavioral conformance.

4.2 B Machines for Data Refinement

According to the B method [1], each way of constructing a B abstract machine (from scratch, by including another, by refining another, etc.) comes with a number of proof obligations that are sufficient for the consistency of the entire model. For each operation, for example, we must prove that it preserves the state invariant of the machine in which it is defined. If we construct a machine from a given one by a **REFINEMENT**, then this implicitly defines a B machine that is supposed to be a data refinement of the base machine. Processing such a refinement definition, AtelierB will generate proof obligations that are sufficient for a data refinement (see [1] for the theory justifying those conditions).

For representing a UML gen / spec hierarchy in B, we do not directly use refinement of B machines (c.f. Fig. 4), because if safe use of polymorphism is not an issue, then the

\(^3\) We do not attempt to present a formal proof here, because that would exceed the limits of this paper.
Behavioral Conformance Verification in an Integrated Approach

MACHINE Operation_Refinement_Bag_Put(ELEMENT)

INCLUDES Bag_INTERFACE(ELEMENT)

OPERATIONS

Operation_Refinement_Bag_Put(oo, ii) =
pre
  oo ∈ bag ∧
  ii ∈ ELEMENT ∧
  ∑ xx (xx ∈ ELEMENT | elements(oo)(xx) < bound(oo) ∧
  elements(oo)(ii) < maxint)
then
  put_bag(oo, ii)
end

END

REFINEMENT Operation_Refinement_Static_Bag_Put(ELEMENT)

REFINES Operation_Refinement_Bag_Put

INCLUDES concrete_Bag_INTERFACE(ELEMENT)

INVARIANT
  concrete_bag = bag ∧
  concrete_elements = elements ∧
  concrete_bound = bound ∧

OPERATIONS

Operation_Bag_Put(oo, ii) =
pre
  oo ∈ concrete_static_bag ∧
  ii ∈ ELEMENT ∧
  ∑ xx (xx ∈ ELEMENT | concrete_elements(oo)(xx) < concrete_bound(oo) ∧
  concrete_elements(oo)(ii) < maxint)
then
  concrete_put(oo, ii)
end

END

Fig. 7. Operation verification for Put

specializations in such a hierarchy need not behaviorally conform to the general class. Therefore, to analyze behavioral conformance in a class hierarchy, we define auxiliary B machines whose associated proof obligations are the ones we wish to prove for behavioral conformance.

To prove the data refinement condition between two classes A and C, we need not consider the initialization condition explicitly: Because we construct a specialization C from the machine representing the general class A (c.f. Fig. 4) by USES clauses, the initialization of C by definition includes the initialization of the general machine representing A.

Addressing the refinement of operations, we construct two B machines for each operation op of the more general class A. The first machine defines an operation for the abstract version of op. The second machine is a refinement of the first, using the version of C for op. For example, to prove the data refinement condition for Bag and StaticBag and the operation put, we define the two machines shown in Fig. 7.
The first machine defines Operation\textunderscore Bag\_Put, which basically calls the definition of \textit{put} in class \textit{Bag}, which is \textit{put\_bag}. It is based on the interface machine \textit{Bag\_Interface}. Directly including the machine \textit{Bag} is impossible, because a machine may be included by at most one other machine and \textit{Bag\_Interface} already includes \textit{Bag}. If we included a copy of \textit{Bag}, however, then we would get a type checking error, because the declaration of the base set \textit{BAG} of object identifiers (c.f. Fig. 2) would be duplicated. Refining the first machine, the second includes a copy of \textit{Bag\_Interface} called \textit{concrete}. The coupling invariant for the refinement identifies all state components of \textit{concrete} with the ones of the original \textit{Bag\_Interface}. Unless the mathematical model of the data in the specialization differs from the one of the more general class, the coupling invariant in an object-oriented setting usually is just a projection from the state of the specialization (which may have additional attributes) to the state of the more general class [15]. We redefine the operation \textit{Operation\_Bag\_Put} in the second machine to call the version of \textit{put} of \textit{Static\_Bag}. We achieve this by stating the precondition \textit{oo \in concrete\_static\_Bag} and calling \textit{concrete\_put} in the body of the operation. As before, the predicates shaded gray describe information not present in the UML model: To successfully augment a bag with an element, the number of elements already present in the bag must be strictly less than the bound of the bag. The refinement conditions \textit{AtelierB} generates for that pair of machines are exactly the ones needed to prove the conditions of data refinement concerned with the operation \textit{put}. For the other operations of \textit{Bag}, we construct similar pairs of machines to relate them to the ones of \textit{Static\_Bag}. Conceptually, we could define only one pair of machines including all the necessary operation definitions. We prefer to define many small machines to keep the single machines more accessible and facilitate change. Those machines are not used further because they are constructed only to generate appropriate proof obligations for data refinement. Therefore, they do not clutter any further use of the classes in the gen / spec hierarchy.

5 Behavioral Conformance II: History Constraints

Mapping the concept of history constraints into the framework of B, we must take two kinds of proof obligations into account (c.f. Sect. 2). First, we must find a place to specify history constraints of a class, and verify that each class specification – made up of a number of B machines – is history consistent. Second, we must verify that history constraints are preserved along gen / spec relations.

5.1 History Consistency

To specify the history constraints of a class, we define a machine that includes the interface machine of the class hierarchy we consider. Figure 8 shows that machine for the class \textit{Bag}. The predicate \textit{HC} is the history constraint of \textit{Bag}. The history of \textit{BAG} is unconstrained, because \textit{HC} is equivalent to \textit{true}. The syntactic form of \textit{HC} serves to illustrate how the history constraint depends on the values of the attributes before and after a state transition. Given that history constraint of \textit{Bag}, the extra methods of
MACHINE History_Bag(ELEMENT)

INCLUDES
Bag_Interface(ELEMENT)

DEFINITIONS
HC(previousE, previousB, nextE, nextB) ≜
   (previousE = nextE ∨ not(previousE = nextE)) ∧
   (previousB = nextB ∨ not(previousB = nextB))

OPERATIONS
   history(oo) =
   pre
      oo ∈ bag
   then
      any newb, newe where
         newe ∈ ELEMENT −→ nat ∧
         newb ∈ nat ∧
         newb ≥ ∑ xx.x ∈ ELEMENT | newe(xx)) ∧
         HC(elements(oo), bound(oo), newe, newb)
      then
         chg_state(oo, newe, newb)
      end
   end

END

Fig. 8. History specification for Bag

a specialization of Bag, such as Dynamic_Bag, may perform arbitrary state changes (within the limits of the class invariant) without destroying behavioral conformance.

The machine History_Bag defines one operation history that changes the state of the parameter object oo nondeterministically within the restrictions specified by the history constraint: It selects new values newb and newe of the attributes (bound and elements) of oo such that the relation HC holds between those new values and the given values of the attributes of oo. Then it changes the attribute values of oo with the internal operation chg_state, which is defined in Bag_Interface. Thus, the precondition of the operation history is \( I(s) ∧ (∃s'). HC(s, s') \). The state transition relation of history is HC.

Verification. To verify history consistency, we need to verify the proof obligations mentioned in Sect. 2 for each operation of the class. The first condition, \( I(s) ∧ HC(s, s') \Rightarrow I(s') \), is one of the proof obligations associated with the definition of the machine History_Bag: The operation history must respect the invariant of Bag_Interface. Because the transition relation of history is HC, we get exactly the proof obligation we wish to prove.

For the second condition, \( I(s) ∧ I(s') ∧ Op(s, s') \Rightarrow HC(s, s') \), we define a refinement of the machine History_Bag for each operation of Bag. Figure 9 shows the one for put. In that refinement, we use the proof obligations for operation refinement to formalize the condition of history consistency: In Sect. 2, we mentioned that we may consider the history constraint as an anonymous operation that is refined by all (named) operations of the class. Therefore, the refined version of the operation history (in Fig. 9) calls the
operation \textit{put} (of \textit{Bag}) with an arbitrary input parameter \(ii\) satisfying the precondition of that operation.

The first condition of operation refinement (c.f. Sect. 4.1) considers the preconditions. It is satisfied because the precondition of the concrete operation is not stronger than the one of the abstract operation (the latter is just the invariant).

The second condition of operation refinement considers the transition relations. For the abstract operation, the transition relation is given by the history constraint \(H\). By construction, the coupling invariant of the refinement is equivalent to the equation \(c = a\).

Using the one point rule, we calculate

\[
(\forall a, c, c'. (\text{pre}_{\text{op}}) (a) \land \text{CI} (c, a) \land \text{op}_{a} (c, c') \Rightarrow (\exists a'. \text{CI} (c', a') \land \text{op}_{a} (a, a'))) \\
\Leftrightarrow (\forall a, c, c'. c = a \land \text{op}_{a} (c, c') \Rightarrow (\exists a'. c' = a' \land \text{HC} (a, a'))) \\
\Leftrightarrow (\forall c, c'. \text{op}_{c} (c, c') \Rightarrow \text{HC} (c, c'))
\]

In our example, we get:

\[
(\forall \text{elements}, \text{bound}, \text{New.elements}, \text{New.bound}, \ldots \text{New.elements}', \text{New.bound}') \land \text{history}_{a} (\text{elements}, \text{bound}) \land \text{New.bound} = \text{bound}' \Rightarrow (\exists \text{elements}', \text{bound}') \land \text{New.elements}' = \text{elements}' \land \text{New.bound}' = \text{bound}' \land \ldots\)
\]

Fig. 9. History consistency of \textit{put}
\[ \Leftrightarrow \forall \text{elements}, \text{bound}, \text{elements}', \text{bound}' \ldots \]
\[ \text{put}(oo, ii)(\text{elements}, \text{bound}, \text{elements}', \text{bound'}) \Rightarrow HC(\text{elements}, \text{bound}, \text{elements}', \text{bound'}) \]

Again, for lack of space, we cannot formally derive that equivalence from the semantics of the involved B machines in this paper. The derivation involves reducing the operation definitions to set transformers, and deriving the preconditions and transition relations for the resulting transformers. The general theory for doing so is elaborated in [1]. Note that the preconditions of both versions of history require that oo \in bag and that the invariant holds, but they are independent of HC.

To completely verify history consistency of the class hierarchy shown in Fig. 1, we must construct “history machines” as the one of Fig. 8 for all classes of the hierarchy, and we must construct refinements similar to the one shown in Fig. 9 for all operations of those classes. The history constraint Static_Bag is non-trivial, because it requires that bound does not change:

\[
HC(\text{previousE}, \text{previousB}, \text{nextE}, \text{nextB}) \equiv
(\text{previousE} = \text{nextE} \lor \neg(\text{previousE} = \text{nextE}) \land
(\text{previousB} = \text{nextB}))
\]

Proving that Static_Bag is history consistent, we show that none of the operations of that class changes the bound of its parameter object.

5.2 History Preservation

Having shown that all classes of the gen/spec hierarchy of Fig. 1 are history consistent, and knowing that Dynamic_Bag and Static_Bag data refine Bag, it remains to show that the two specializations of Bag preserve the history constraint of Bag. Remember from Sect. 2 that the second condition of behavioral conformance requires us to show that the history constraint of Dynamic_Bag and Static_Bag each imply the history constraint of Bag.

In terms of predicate logic, we just wish to prove two implications of the form \( H_i \Rightarrow H_j \). Technically, we are faced with the problem of encoding the history constraints in related B machines for which the B method requires prove obligations that are equivalent to that simple implication. As for verifying history consistency in the previous section, we use refinements of B machines to reach that aim.

For Bag, we again use the machine History_Bag (c.f. Fig. 8). To verify history preservation of Dynamic_Bag with respect to Bag, we define the machine shown in Fig. 10.

The definition of HC in that machine shows the history constraint of Dynamic_Bag, which requires that the bound must not decrease. As before, the operation history is constructed in such a way that its transition relation is given by the history constraint. The coupling invariant also is just a conjunction of equations identifying the states of the two copies of Bag_Interface. Therefore, with a calculation similar to the one of the previous section, we verify that proving the refinement conditions for the operation
REFINEMENT  History Refinement Dynamic Bag (ELEMENT)

REFINES  History Bag

INCLUDES  New, Bag Interface (ELEMENT)

DEFINITIONS

\[
HC(\text{previous}E, \text{previous}B, \text{next}E, \text{next}B) \triangleq \\
(\text{previous}E = \text{next}E \lor \neg (\text{previous}E = \text{next}E) \\
\land (\text{previous}B \leq \text{next}B))
\]

INVARIANT

\[
\text{New}.\text{bag} = \text{bag} \land \\
\text{New}.\text{elements} = \text{elements} \land \\
\text{New}.\text{bound} = \text{bound} \land
\]

OPERATIONS

\[
\text{history}(oo) = \\
\quad \text{pre} \\
\quad \ oo \in \text{New}.\text{dynamic}\_\text{Bag} \\
\quad \quad \text{then} \\
\quad \quad \ any \ new, newe \ where \\
\quad \quad \quad \ newe \in \text{ELEMENT} \rightarrow \text{nat} \land \\
\quad \quad \quad \ newb \in \text{nat} \land \\
\quad \quad \quad \ \sum_{xx} \ (xx \in \text{ELEMENT} | \ newe(xx)) \land \\
\quad \quad \quad \ \text{HC}(\text{New}.\text{elements}(oo), \text{New}.\text{bound}(oo), newe, newb) \\
\quad \quad \text{then} \\
\quad \quad \ \text{New}.\text{chg}\_\text{state}(oo, newe, newb) \\
\quad \quad \text{end} \\
\quad \text{end}
\]

END

Fig. 10. History preservation of Dynamic Bag wrt. Bag

\[
history
\]

amounts to proving the implication between history constraints that we wish to prove.

There is, however, a grain of salt in this argument: The precondition of the abstract version of history in Fig. 8 is not sufficient for the precondition of the “refinement” of history in Fig. 10. That condition essentially requires that all bags are dynamic bags:

\[
oo \in \text{bag} \Rightarrow oo \in \text{dynamic}\_\text{Bag}
\]

That implication obviously is not true. Conceptionally, our reasoning is nevertheless sound, because the relation between the classes Bag and Dynamic Bag is important only for instances of the specialization Dynamic Bag. For other instances of Bag (e.g. static bags) that are not instances of Dynamic Bag, the question whether Dynamic Bag behaviorally conforms to Bag is not of interest.

Technically, there are two ways of resolving that problem. First, we could change the precondition of history in the abstract machine History Bag to \( oo \in \text{dynamic}\_\text{Bag} \). But then, the resulting machine could serve only for proving behavioral refinement of Bag by Dynamic Bag. For Static Bag, we would have to use a copy of History Bag with precondition \( oo \in \text{static}\_\text{Bag} \). In general, we would have to construct a new “history machine” of the more general class for each proof of behavioral conformance in which
that class is involved. All those machines would differ only in the precondition of the
operation \textit{history}.
Second, we could take a pragmatic view and just accept that the refinement condition
for preconditions is not provable when we verify history preservation. This is what we
currently do. It spares us the effort to define (and manage) different versions of "history
machines" for a single class. It also reduces the proof load for \texttt{AtelierB}. We believe
that this procedure is still safe, because an attempt to prove a false implication between
history constraints manifests itself in the condition on the transition relations and not in
the condition on the preconditions.

6 Experience

In the present section, we give an impression of the size of the B model that we have
derived from the UML model in Fig. 1. We also discuss how verifying B models with
the tool \texttt{AtelierB} compares to proving in a model of object oriented specifications [15]
that we have developed for Isabelle/HOL [13].
The complete B model of the gen / spec hierarchy of bags comprises 21 components
in B (abstract machines or refinements). Table 1 shows a statistics about the proofs we
conducted on those components. The components written in italics stem from translating
the UML model to B. The ones written in bold face serve to verify behavioral confor-
mance. The column \texttt{Obv} shows the number of obvious proof obligations the B method
requires for each component, the column \texttt{nPO} stipulates the number of proof obligati-
on that are not "evident"; i.e. immediately discharged by \texttt{AtelierB} without invoking a
prover. The column \texttt{nUn} presents the number of unproved proof obligations, and \texttt{% Pr}
shows the percentage of discharged obligations. The two unproven obligations are the
false implications between preconditions that we discussed at the end of the preceeding
section.
Table 1 illustrates that the part of the B model representing the UML model is relatively
small: it comprises 5 machines for the three classes; the machine \texttt{Types} is a quite small one
containing a few global declarations of sets to which the other machines refer. The other
16 machines are constructions supporting the verification of behavioral conformance.
While that may seem quite a large number, the important fact to note is that those
machines can be derived from the others automatically, and that they are used solely
in the verification of behavioral conformance. A UML model that uses the hierarchy of
bags does not need to consider those machines.

As a tool specialized for the B language and method, \texttt{AtelierB} provides a quite elaborate
interface for working with B models. All proof obligations are set up fully automatically.
The ones that are not "evident" are presented to the user for interactive proof. With a few
proof commands, users who do not have extensive experience with general purpose pro-
vers can still conduct substantial proofs. In our experience, the performance of \texttt{AtelierB}
is quite satisfactory.
These observations are valid as long as the B models refer to the parts of the underlying
mathematical theory that are often used and, therefore, well-supported by the tool. When
we used to summation operator $\sum$ to express properties of the bound on the elements
of bags, it became evident that other parts of the theory are not sufficiently supported by
<table>
<thead>
<tr>
<th>COMPONENT</th>
<th>Obv nPO nUn % Pr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bag</td>
<td>15 26 0 100</td>
</tr>
<tr>
<td>Bag Interface</td>
<td>87 46 0 100</td>
</tr>
<tr>
<td>Dynamic_Bag</td>
<td>3 0 0 100</td>
</tr>
<tr>
<td>History_Bag</td>
<td>40 0 0 100</td>
</tr>
<tr>
<td>History_Bag_Get</td>
<td>14 6 0 100</td>
</tr>
<tr>
<td>History_Bag_Put</td>
<td>17 6 0 100</td>
</tr>
<tr>
<td>History_Dynamic_Bag</td>
<td>20 2 0 100</td>
</tr>
<tr>
<td>History_Dynamic_Bag_Chg_bound</td>
<td>15 3 0 100</td>
</tr>
<tr>
<td>History_Dynamic_Bag_Get</td>
<td>13 7 0 100</td>
</tr>
<tr>
<td>History_Dynamic_Bag_Put</td>
<td>16 7 0 100</td>
</tr>
<tr>
<td>History_Refinement_Dynamic_Bag</td>
<td>50 79 1 99</td>
</tr>
<tr>
<td>History_Refinement_Static_Bag</td>
<td>36 41 1 98</td>
</tr>
<tr>
<td>History_Static_Bag</td>
<td>20 2 0 100</td>
</tr>
<tr>
<td>History_Static_Bag_Get</td>
<td>11 7 0 100</td>
</tr>
<tr>
<td>History_Static_Bag_Put</td>
<td>14 7 0 100</td>
</tr>
<tr>
<td>Operation_Refinement_Bag_Get</td>
<td>6 0 0 100</td>
</tr>
<tr>
<td>Operation_Refinement_Bag_Put</td>
<td>6 0 0 100</td>
</tr>
<tr>
<td>Operation_Refinement_Static_Bag_Get</td>
<td>9 11 0 100</td>
</tr>
<tr>
<td>Operation_Refinement_Static_Bag_Put</td>
<td>13 8 0 100</td>
</tr>
<tr>
<td>Static_Bag</td>
<td>3 8 0 100</td>
</tr>
<tr>
<td>Types</td>
<td>1 0 0 100</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>409 266 2 99</strong></td>
</tr>
</tbody>
</table>

AtelierB. Properties of \( \sum \) such as

\[
\forall (A, x). \sum x. (x \in A \mid 0) = 0
\]

for some finite set \( A \) are not part of the set of rules used by AtelierB. This property is not derivable using the implemented set of rules. Therefore, we assumed this and three other properties of \( \sum \) for the proofs in AtelierB. The only way to make use of those properties in the proofs is to interactively specialize the universal quantifiers. It is impossible to introduce new rules that can be applied automatically during proofs, because those rules are all hand-coded into the implementation of the prover.

To make sure that we did not introduce an inconsistency in our theory, we derived those properties of \( \sum \) in Isabelle/HOL. There, we have a theory supporting the Z mathematical toolkit, whose definitions are quite similar to the ones used in B.

Compared to working with AtelierB, proving theorems in Isabelle tends to be more interactive. This is mostly because the automated support for the Z toolkit is not developed as far as the one implemented in AtelierB. It is, however, very easy to extend the theory by new theorems, and make the proof tactics, such as the simplifier, use those theorems automatically. This clearly is an advantage of a proof engine like Isabelle that is designed to be modified and extended.

There also is another difference between the two tools: The theory we have developed to support object oriented specification [15] allows us to derive theorems about relations of
classes \textit{in general}, which we may specialize for concrete classes later (without having to reprove the theorem). Thus, we could derive mechanically that behavioral conformance is transitive. AtelierB does not allow us to conduct similar proofs about B machines in general. Thus, we cannot establish a theory in AtelierB that would allow us to conclude that, e.g., a behaviorally conforming specialization of static bags also is a behaviorally conforming specialization of bags. Instead, we would have to prove behavioral conformance with respect to bags directly. For the small hierarchy of classes that we considered in this paper, the effort involved in those proofs is not an issue, but for a large hierarchy, the effort of conducting redundant proofs would quickly become unbearable. For such a task, a more open tool environment for B is desirable that would allow for extending proof procedures and defining new verification conditions for particular contexts.

7 Conclusions

We are not the first to propose a combination of UML (or OMT) notations with a formal technique. Others have studied integrations of OMT and LOTOS [20], OMT and VDM [4], and UML and Z [3]. The following research more specifically addresses translating models of the UML to B specifications:

- Nagui-Raiss [12] proposes a translation from extended entity association diagrams to B.
- Shore [16] presents an alternative method for producing formal specifications from objects. The approach taken here represents the object structure at the implementation level by abstractly representing objects as structures (in B) at the machine level that are refined during a development using machines representing those objects.
- Facon et al. [5] define rules from both OMT static and dynamic models to B specifications. They are interested in modularizing the formal specification resulting from the object model transformation.
- Lano [7,8] illustrates the use of object-oriented methods to support the development process in B. He proposes several translations to map analysis models of OMT to B machines.

Compared to that research, our approach is novel in that we use the framework of B to verify properties of class hierarchies such as behavioral conformance, which may or may not be expressed in the UML model. Constructing B components in an appropriate way, we utilize the verification conditions that the B method associates with those components to verify properties that stem from an object oriented world and are not genuinely properties one would verify of a B model made up from scratch. An advantage of that approach is not only that we can rely on the theoretically well understood formal framework of B, but also that we can use a tool such as AtelierB to generate and verify the properties we wish to show about the UML model. Building on AtelierB, we set up a productive working environment for generating and proving the proof obligations that stem from the object oriented world in a very short time. Implementing a tool from scratch to support the same tasks would have taken much longer, and it would have been much more error prone. Although AtelierB has its deficiencies in terms of robustness,
extensibility and openness, it relieved us from many routine tasks that take much effort to implement, such as tracking dependencies between assertions, and recording and maintaining proofs. The construction of auxiliary B machines to generate appropriate proof obligations may seem complicated, but it surely is much simpler and much more safe with respect to soundness of the logical derivations than implementing a tool that would produce and prove them directly.

The translation from an UML class diagram to a system of B components can be automated to a large extent. We are currently working on an implementation that will automatically carry out the construction of B machines illustrated in this paper. To date, we augment the B model resulting from such a translation with behavioral descriptions of operations and with properties expressing invariants, preconditions, and history constraints. Most of those could be expressed by constraints in a formal language such as OCL within the UML model. If a particular formal notation for expressing constraints of a UML model will be widely accepted in the future, it will be easy to provide a translation from that language to B. For now, we do not see it as a heavy burden for specifiers to express formal constraints directly in B.

References


Abstract. We define a class of diagrams that represent abstractions of—possibly infinite-state—reactive systems described by specifications written in temporal logic. Our diagrams are intended as the basis for the verification of both safety and liveness properties of such systems. Non-temporal proof obligations establish the correspondence between the original specification and the diagram, whereas model checking can be used to verify properties over finite-state abstractions. We describe the use of abstract interpretation techniques to generate proof diagrams from a given specification and user-defined predicates that represent sets of states.

1 Introduction

Verification techniques for reactive systems are traditionally classified as either deductive or algorithmic. Whereas deductive verification can in principle establish properties of arbitrarily complex systems, algorithmic verification such as model checking is usually restricted to finite-state systems, although some classes of infinite-state systems have been identified for which certain properties are decidable. On the other hand, algorithmic verification is (essentially) automatic and, hence, easy to use by system engineers, whereas computer support for deductive verification in the form of interactive proof assistants requires careful guidance by experienced users. Combinations of both kinds of technique such as deductive model checking have also been suggested, with the aim of providing a unified framework that combines the advantages of deductive and algorithmic verification. More generally, abstraction and composition have been recognized as the key concepts that bridge the two approaches [13,9]. Our paper proposes the concept of predicate diagrams to integrate algorithmic and deductive verification techniques; it allows to relate specifications and properties by constructing an abstract finite model. We consider safety as well as liveness properties by annotating diagrams with information relating to fairness and ordering conditions. We also show how predicate diagrams can be constructed semi-automatically, based on techniques of abstract interpretation. Predicate diagrams intend to help users to understand how a reactive system (or a distributed system) is working. We have focused on defining a framework that is as powerful as traditional deductive verification and can integrate safety, fairness, and general liveness properties, yet can abstractly represent the given system at a level that is intuitive to understand for the system designer. We have applied predicate diagrams to some case

* This work has been partially supported by a PROCOPE grant from EGIDE and DAAD.
studies such as a reader-writer algorithm developed at Siemens Corporate Research [11] and a self-stabilizing algorithm due to Dijkstra [26]. On the other hand, an abstract animator [4] based on techniques of abstract interpretation has been shown to allow models for non-trivial systems to be built automatically once the user has defined the relevant abstract predicates. Our method can therefore be seen as a natural generalization of traditional model checking, except that it is applied to an abstract state space. In particular, there is no need for the user to provide inductive invariants as in deductive verification. We assume the system to be modeled as a specification written in a variant of TLA [17] and we use the temporal language of linear-time temporal logic for stating required assertions or properties. Nevertheless, the basic ideas that underly our approach apply more widely. We define verification conditions that ensure that a predicate diagram adequately represents a system specification; these verification conditions must in general be discharged using an automatic or interactive theorem prover. The second task is to show that every run of the diagram satisfies the asserted properties. Because predicate diagrams are finite, this can be established using standard model checkers such as Spin [12], obviating the need for tedious temporal reasoning that plagues standard deductive verification. We have shown that the constructions of the diagram can often be automated using abstract interpretation. The user need then only choose the predicates that define the abstraction function. Besides their use as a formal basis for verification, predicate diagrams can also serve as support for explaining how systems are working and for documenting them.

Using diagrams to simplify, to visualize or to structure a proof is not new. There are many works like proof diagrams proposed by Manna et al [7,22,23], predicate-action diagrams from Lamport [18], proof lattices proposed by Lamport and Owicki [27] and proof charts proposed by R. Cousot [5].

The contributions of our work are the following:

- We emphasize a clear separation of concerns and identify those subtasks that are within the realm of, respectively, theorem proving and model checking.
- Following ideas present in [25,14], edges of a predicate diagram can be labelled by well-founded orderings to exclude runs where such an edge is taken infinitely often; we can thus prove liveness properties (these annotations simulate the lattice rule of temporal logic).
- Fairness assumptions about transitions can be stated on the level of diagrams; these assumptions are taken into account when proving liveness properties using model checking.
- We propose an automatic construction of a predicate diagram by using abstract interpretation. This construction may require approximations introduced via maybe edges, and we propose solutions to eliminate such maybe edges via a reconsideration of the corresponding transitions at the concrete level.

The paper is structured as follows. Section 2 presents predicate diagrams. Section 3 describes the relationship between specifications and predicate diagrams. Sections 4 and 5 explain how we use abstract interpretation to construct a predicate diagram for a given specification. Sections 6 and 7 compare with related works and conclude the paper.
2 Predicate Diagrams

We express system specifications and properties in a variant of linear-time temporal logic whose formulas are built from state predicates and actions, which may contain primed state variables. For example, \( x > 3 \) is a state predicate, and \( x \leq y' + 1 \) is an action. For an action \( A \), we denote by \( \text{enabled}_A \) the state predicate obtained from \( A \) by existential quantification over the primed state variables. For a state predicate \( P \), we denote by \( P' \) the action obtained from \( P \) by replacing all flexible variables \( v \) by \( v' \). Temporal formulas are formed using boolean connectives, the always operator \( \Box \), and quantification over rigid (state-independent) variables.

The semantics of state formulas is defined with respect to a state, i.e., an assignment of values to state variables, and a valuation of the rigid variables. Actions are interpreted relative to a pair \((s, t)\) of states, where \( s \) and \( t \) interpret respectively the unprimed and primed state variables. Temporal formulas are evaluated over behaviors, which are \( \omega \)-sequences \( \sigma = s_0 s_1 \ldots \) of states [15, 24]. We write \( s \models P, (s, t) \models A, \) and \( \sigma \models F \) to denote that a state predicate \( P \), an action \( A \) or a temporal formula \( F \) hold of a state, a pair of states or of a behavior. Validity of a temporal formula \( F \) (over the class of first-order interpretations of interest) is denoted by \( \models F \), and similarly for state predicates and actions.

Derived connectives include the eventually operator defined by \( \Diamond F \equiv \neg \Box \neg F \), the leads-to operator \( F \Rightarrow G \equiv \Box (F \Rightarrow G) \) and, for an action \( A \), the formulas

\[
\text{WF}(A) \equiv (\Box \Box \text{enabled}_A) \Rightarrow \Box \Diamond A \\
\text{SF}(A) \equiv (\Diamond \Box \text{enabled}_A) \Rightarrow \Diamond \Box A
\]

that assert weak and strong fairness conditions for \( A \). This logic is similar to Lamport’s Temporal Logic of Actions [17], but it is not invariant under stuttering.\(^1\) As in TLA, specifications of reactive systems can be expressed as temporal formulas; these are usually written in the form \( \text{Init} \land \Box \text{Next} \land L \) where \( \text{Init} \) is a state predicate characterizing the system’s initial state, \( \text{Next} \) is an action that represents the next-state relation, and \( L \) is a conjunction of formulas \( \text{WF}(A) \) or \( \text{SF}(A) \).

We assume the underlying assertion language to contain a finite set \( \mathcal{O} \) of binary relation symbols \( \prec \) that are interpreted by well-founded orderings. For \( \prec \in \mathcal{O} \), we denote by \( \preceq \) its reflexive closure. We write \( \mathcal{O}^\omega \) to denote the set of relation symbols \( \prec \) and \( \preceq \), for \( \prec \in \mathcal{O} \).

The definition of predicate diagrams is relative to finite sets \( \mathcal{P} \) and \( \mathcal{A} \) that contain the state predicates and actions of interest. We denote by \( \mathcal{P} \) the set containing the predicates in \( \mathcal{P} \) and their negations.

**Definition 1.** A predicate diagram \( G = (N, I, \delta, o, \zeta) \) over \( \mathcal{P} \) and \( \mathcal{A} \) consists of

- a finite set \( N \subseteq 2^\mathcal{P} \) of nodes,
- a finite set \( I \subseteq N \) of initial nodes,

\(^1\) Stuttering invariance is unimportant for the verification of some fixed specification. It becomes essential when predicate diagrams are used for the top-down development of reactive systems by stepwise refinement, a topic to be described in a separate paper.
We say that the action $A \in \mathcal{A}$ can be taken at node $n \in N$ if $(n, m) \in \delta_A$ holds for some $m \in N$.

A predicate diagram $G$ is a finite labelled transition system whose nodes are sets of state predicates from $\mathcal{P}$ or their negations. Intuitively, a node represents the set of system states that satisfy the formulas contained in the node. (In the following, we indifferently write $n$ for the set and the conjunction of its elements.) Transitions are labelled by actions and by ordering annotations that indicate which terms decrease with respect to relations from $O^N$. An action $A$ may have an associated fairness condition; it applies to all transitions in $\delta_A$ rather than to individual edges.

We now define traces through a diagram as behaviors that correspond to fair runs satisfying the node and edge labels. To evaluate the fairness conditions, we identify the enabling condition of an action $A \in \mathcal{A}$; the possible values represent no fairness, weak fairness, and strong fairness.

**Definition 2.** Let $G = (N, I, \delta, o, \zeta)$ be a predicate diagram over $\mathcal{P}$ and $\mathcal{A}$. The set $\text{tr}(G)$ of traces through $G$ consists of all $\omega$-sequences $\sigma = s_0 s_1 \ldots$ of states such that there exist sequences $n_0 n_1 \ldots$ and $A_0 A_1 \ldots$ of nodes $n_i \in N$ and actions $A_i \in \mathcal{A}$ such that all of the following conditions hold:

- $n_0 \in I$ is an initial node,
- $(n_i, n_{i+1}) \in \delta_A$ holds for all $i \in \mathbb{N}$,
- $s_i \models n_i$ holds for all $i \in \mathbb{N}$,
- $(s_i, s_{i+1}) \models A_i$ holds for all $i \in \mathbb{N}$,
- $(s_i, s_{i+1}) \models t' \prec t$ holds for all $i \in \mathbb{N}$ and $(t, \prec) \in o_A(n_i, n_{i+1})$,
- for every action $A \in \mathcal{A}$ such that $\zeta(A) = \text{WF}$ there are infinitely many $i \in \mathbb{N}$ such that either $A_i = A$ or $A_i$ cannot be taken at $n_i$,
- for every action $A \in \mathcal{A}$ such that $\zeta(A) = \text{SF}$, either $A_i = A$ holds for infinitely many $i \in \mathbb{N}$ or there are only finitely many $i \in \mathbb{N}$ such that $A$ can be taken at $n_i$.

### 3 Reasoning about Systems via Predicate Diagrams

#### 3.1 Relating Specifications and Predicate Diagrams

We say that a predicate diagram $G$ conforms to a specification $Spec$, written $Spec \preceq G$, if every behavior that satisfies $Spec$ is a trace through $G$. In general, proving $Spec \preceq G$ requires reasoning about entire behaviors. The following theorem allows us to (essentially) reduce this temporal reasoning to “local” proof obligations that concern single states or pairs of states.
Theorem 3. Let $G = (N, I, \delta, \alpha, \zeta)$ be a predicate diagram over $P$ and $A$, and $Spec \equiv \text{Init} \land \text{Next} \land L$ be a system specification. If all of the following conditions hold, then $\sigma \in \text{tr}(G)$ holds for all models $\sigma$ of $Spec$.

1. $\models \text{Init} \Rightarrow \bigvee_{n \in I} n$
2. $\models n \land \text{Next} \Rightarrow \bigvee_{\{(A,m):(n,m) \in s_k\}} A \land m'$ holds for every node $n \in N$.
3. $\models n \land A \land m \Rightarrow t' \prec t$ holds for all $A \in \mathcal{A}$, $(n, m) \in \delta_A$, and $(t, \prec) \in o_A(n, m)$.
4. For every action $A \in \mathcal{A}$ such that $\zeta(A) \neq \text{NF}$:
   a) If $\zeta(A) = \text{WF}$ then $\models \text{Spec} \Rightarrow \text{WF}(A)$.
   b) If $\zeta(A) = \text{SF}$ then $\models \text{Spec} \Rightarrow \text{SF}(A)$.
   c) $\models n \Rightarrow \text{ENABLED } A$ holds whenever $A$ can be taken at node $n$.
   d) $\models n \land A \Rightarrow \neg m'$ holds for all $n, m \in N$ such that $(n, m) \notin \delta_A$.

Proof. Assume that $Spec$ and $G$ are such that all the conditions hold, and assume that $\sigma = s_0s_1 \ldots$ is a behavior that satisfies $Spec$. In order to show that $\sigma \in \text{tr}(G)$, we inductively define a sequence $s_0n_1 \ldots$ of nodes $s_i \in N$ and a sequence $A_0A_1 \ldots$ of sets of actions $\emptyset \neq A_i \subseteq \mathcal{A}$ such that for all $i \in \mathbb{N}$ the following conditions hold:

(i) $s_i \models n_i$,
(ii) $n_0 \in I$.
(iii) $A_i = \{A \in \mathcal{A} : (n_i, n_{i+1}) \in \delta_A \land (s_i, s_{i+1}) \models A\}$, and
(iv) $(s_i, s_{i+1}) \models t' \prec t$ for all $A \in A_i$ and all $(t, \prec) \in o_A(n_i, n_{i+1})$.

Obviously, conditions (i)–(iv) ensure the conditions stated in definition 2 for any choice of $A_i \in A_i$, except for the conditions related to fairness requirements.

For the induction base, we choose some node $n_0 \in I$ such that $s_0 \models n_0$ holds; the existence of some such node is ensured by condition (1), since $s_0 \models \text{Init}$ holds by assumption.

For the induction step, we assume that $n_0 \ldots n_i$ and $A_0 \ldots A_{i-1}$ have already been defined such that conditions (i) and (ii) hold for all $j \leq i$ and conditions (iii) and (iv) hold for all $j < i$. In particular, we have $s_i \models n_i$. Moreover, since $\sigma \models \text{Spec}$, we know that $(s_i, s_{i+1}) \models \text{Next}$, and condition (2) ensures that $(s_i, s_{i+1}) \models A \land m'$ holds for some action $A$ and some transition $(n_i, m) \in \delta_A$. Define $n_{i+1} \equiv m$ for some such transition, and choose $A_i$ as given in the right-hand side of condition (iii). These choices imply that $s_{i+1} \models n_{i+1}$ and that $A_i \neq \emptyset$. To prove condition (iv), assume that $A \in A_i$ and that $(t, \prec) \in o_A(n_i, n_{i+1})$. By induction hypothesis and the choices of $n_{i+1}$ and $A_i$, we have $(s_i, s_{i+1}) \models n_i \land A \land n_{i+1}'$ and $(n_i, n_{i+1}) \in \delta_A$, hence condition (3) implies $(s_i, s_{i+1}) \models t' \prec t$ as required.

It remains to pick a sequence $A_0A_1 \ldots$ of actions $A_i \in A_i$ such that the last two conditions from definition 2, which concern the fairness annotations in $G$, are satisfied. Choose a sequence such that every action $A \in \mathcal{A}$ that appears in infinitely many $A_i$ is chosen infinitely often. (Such a choice is possible because $\mathcal{A}$ is finite.)

Now assume that $\zeta(A) = \text{WF}$ for some action $A \in \mathcal{A}$ and that for all but finitely many $i \in \mathbb{N}$, $(n_i, m) \in \delta_A$ holds for some $m \in N$ (otherwise the condition is obviously satisfied). Because we already know that $s_i \models n_i$ holds for all $i \in \mathbb{N}$, condition (4c)
implies that \( s_i \models \text{enabled} \ A \) holds for all but finitely many \( i \in \mathbb{N} \). Moreover, since \( \sigma \models Spec \) and \( \sigma \models Spec \Rightarrow WF(A) \) (by condition (4a)), it follows that \( (s_i, s_{i+1}) \models A \) holds for infinitely many \( i \in \mathbb{N} \). For every such \( i \in \mathbb{N} \), we have \((n_i, n_{i+1}) \in \delta_A\); otherwise, we would obtain \( s_{i+1} \models \neg n_{i+1} \) from condition (4d) and have a contradiction. But then, condition (iii) above implies that \( A \in A_i \) holds for infinitely many \( i \in \mathbb{N} \), hence \( A = A \) holds for infinitely many \( i \in \mathbb{N} \), which completes the proof.

For actions \( A \in A \) such that \( \zeta(A) = SF \), the proof is analogous, replacing “all but finitely many” by “infinitely many” and using (4b) instead of (4a).

\[\square\]

### 3.2 Model Checking Predicate Diagrams

Since predicate diagrams are finite transition systems, it is straightforward to encode their runs in the input language of standard LTL model checkers such as Spin [12] or STeP [21]. We briefly sketch our encoding of predicate diagrams in Promela, the modelling language of Spin. Two variables are used to indicate the current node and the last action taken. The predicates in \( P \) are represented by boolean variables, which are updated according to the label of the current node (nondeterministically if that label contains neither \( P \) nor \( \neg P \)). We also add variables \( b(t, \prec) \), for every term \( t \) and relation \( \prec \in O \) such that \((t, \prec)\) appears in some ordering annotation \( a_A \). These variables are set to 2 if the last transition taken is labelled by \((t, \prec)\), 0 if it is labelled by \((t, \preceq)\), and to 0 otherwise.

The fairness conditions can be stated as temporal logic assumptions when properties are verified of the Promela model. In order to do so, we assume that action \( A \) is enabled whenever the currently active node has an outgoing edge in \( \delta_A \), as asserted by condition (4c) of theorem 3. To capture the effect of the ordering annotations, we add the assumptions

\[\square \Box (b(t, \prec) = 2) \Rightarrow \square \Box (b(t, \prec) = 0)\]

for every variable \( b(t, \prec) \). These assumptions ensure that transitions known to strictly decrease \( t \) with respect to \( \prec \) cannot be taken infinitely often unless infinitely often some transitions are taken that may increase the value of \( t \).

It is easy to see that whenever \( Spec \preceq G \) holds, then any property verified for the Promela encoding of \( G \) also holds of \( Spec \). We can therefore use model checking to exhaustively consider the traces of \( G \) in order to establish temporal logic properties of \( Spec \). In combination with the non-temporal proof obligations stated in theorem 3 to establish the correctness of the abstraction, we thus achieve the desired separation of concerns for the entire verification of \( Spec \). Obviously, the size of diagrams for which model checking is feasible is mostly limited by the number of fairness conditions and ordering annotations that appear in the diagram.

### 3.3 Two Examples

We illustrate the use of predicate diagrams at the hand of two examples: a two-process version of Lamport’s Bakery algorithm and the “dining mathematicians” mutual-exclusion protocol. We present the specifications as they are input to the generator of
The Bakery Algorithm. We consider a two-process version of Lamport’s well-known protocol [16], shown in figure 1. Mutual exclusion between two processes is ensured via “ticket” variables \(y_1\) and \(y_2\) that processes draw when they wish to enter the critical section. The variables \(p_{c1}\) and \(p_{c2}\) represent the control states of the processes (UNCHANGED \(x\) abbreviates \(x' = x\)).

\[
\text{VARIABLES } y_1, y_2, p_{c1}, p_{c2}
\]

\[
\begin{align*}
\text{Init} & \triangleq y_1 = 0 \land y_2 = 0 \land p_{c1} = "11" \land p_{c2} = "m1" \\
L1 & \triangleq p_{c1} = "11" \land p_{c1}' = "12" \land \text{UNCHANGED } (y_1, y_2, p_{c2}) \\
L2 & \triangleq p_{c1} = "12" \land p_{c1}' = "13" \land y_1' = y_2 + 1 \land \text{UNCHANGED } (y_2, p_{c2}) \\
L3 & \triangleq y \land p_{c1} = "13" \land (y_2 = 0 \lor (y_1 \leq y_2)) \\
& \land p_{c1}' = "14" \land \text{UNCHANGED } (y_1, y_2, p_{c2}) \\
L4 & \triangleq p_{c1} = "14" \land p_{c1}' = "15" \land \text{UNCHANGED } (y_1, y_2, p_{c2}) \\
L5 & \triangleq p_{c1} = "15" \land p_{c1}' = "11" \land y_1' = 0 \land \text{UNCHANGED } (y_2, p_{c2}) \\
M1 & \triangleq p_{c2} = "m1" \land p_{c2}' = "m2" \land \text{UNCHANGED } (y_1, y_2, p_{c1}) \\
M2 & \triangleq p_{c2} = "m2" \land p_{c2}' = "m3" \land y_2' = y_1 + 1 \land \text{UNCHANGED } (y_1, p_{c1}) \\
M3 & \triangleq y \land p_{c2} = "m3" \land (y_1 = 0 \lor \neg(y_1 \leq y_2)) \\
& \land p_{c2}' = "m4" \land \text{UNCHANGED } (y_1, y_2, p_{c1}) \\
M4 & \triangleq p_{c2} = "m4" \land p_{c2}' = "m5" \land \text{UNCHANGED } (y_1, y_2, p_{c1}) \\
M5 & \triangleq p_{c2} = "m5" \land p_{c2}' = "m1" \land y_2' = 0 \land \text{UNCHANGED } (y_1, p_{c1}) \\
\text{Next} & \triangleq L1 \lor L2 \lor L3 \lor L4 \lor L5 \\
& \lor M1 \lor M2 \lor M3 \lor M4 \lor M5 \\
\text{Spec} & \triangleq \text{Init} \land \Box \text{Next}
\end{align*}
\]

\textbf{Fig. 1.} System specification of the Bakery algorithm.

Consider the predicate diagram \(G\) shown in figure 2. We cannot show \(\text{Spec} \subseteq G\) for the Bakery specification of figure 1 using theorem 3 because the attempt to prove obligation (2) would require proving

\[
\begin{align*}
\land \neg(p_{c1} = "14" \land p_{c2} = "m4") \\
\land p_{c2} = "m3" \land (y_1 = 0 \lor y_2 < y_1) \\
\land p_{c2}' = "m4" \land \text{UNCHANGED } (y_1, y_2, p_{c1}) \\
\Rightarrow \neg(p_{c1} = "14" \land p_{c2} = "m4")
\end{align*}
\]
In section 4 we explain how we can construct semi-automatically a predicate diagram for the Bakery specification from which one can prove mutual exclusion as well as the desired liveness properties of the Bakery algorithm.

The “Dining Mathematicians” Example. This is another two-process mutual exclusion protocol introduced in [6]. Processes alternate between “thinking” and “eating” states; they may eat only if $n$ is even or odd, respectively. The temporal logic specification of the protocol appears in figure 3.

Let $\text{DiningGraph}$ be the predicate diagram shown in figure 4. It is an easy exercise to show that $\text{DM} \cdot \text{Spec} \preceq \text{DiningGraph}$ by an application of theorem 3. Using the encoding of predicate diagrams for Spin described in section 3.2, we can establish the following properties:

\begin{align*}
\text{(Pos)} & \quad \falsesc (n \in \text{Nat} \land n > 0) \\
\text{(Live}_0) & \quad \falsesc (c_0 = \text{“e”}) \\
\text{(Live}_1) & \quad \falsesc (c_1 = \text{“e”}) 
\end{align*}

The invariants assert that $n$ remains a positive natural number throughout any run of specification $\text{DM}$, and that mutual exclusion is ensured. The other properties are liveness properties and show starvation-freedom for both processes. In particular, property \((\text{Live}_1)\) can be established despite the cycle between the two leftmost states in the diagram, because the ordering annotations forbid that cycle to be followed indefinitely.
4 Using Abstract Interpretation to Construct Predicate Diagrams

Theorem 3 can be used to justify that a given predicate diagram conforms to a given specification, but it may generate a number of proof obligations that is quadratic in the number of nodes. It is therefore desirable to construct predicate diagrams semi-automatically from a specification whenever this is possible.

In [4] we have defined an “abstract animator” of temporal logic specifications that constructs behaviors of system specifications where concrete values may be abstracted via a user-defined abstraction function. Successor states are computed by abstractly evaluating the system’s next-state relation w.r.t. the abstract values at the previous state. If the abstract evaluation succeeds for all reachable states, we obtain a finite approximation of the system’s behaviors that can be used to infer system invariants. The implementation of our prototype was based on the rewrite engine Logic-Solver that underlies Atelier B [29], an environment supporting the B method [1]. We have since extended the functionality of our tool in order to produce predicate diagrams that conform to the given system specification.

Traditionally, the relationship between concrete and abstract states that underlies abstract interpretation is described by a Galois connection. We recall that a Galois connection between two partially ordered sets \((L_1, \sqsubseteq_1)\) and \((L_2, \sqsubseteq_2)\) is given by two mappings 

\[
(L_1, \sqsubseteq_1) \xrightarrow{\alpha} (L_2, \sqsubseteq_2)
\]

such that \(x_1 \sqsubseteq_1 \gamma(x_2)\) holds iff \(\alpha(x_1) \sqsubseteq_2 x_2\), for all \(x_1 \in L_1\) and \(x_2 \in L_2\). In our setting, elements of \(L_1\) are sets of states and \(L_2\) is the Boolean algebra \(B(\mathcal{P})\) whose atoms are the predicates in \(\mathcal{P}\). The abstraction function \(\alpha\) returns the set of predicates true or false of a set of states, and the concretization function \(\gamma\) produces the set of states that are models of a set of (negated) predicates.

Given an abstract state \(n \in B(\mathcal{P})\), the main problem is to compute an abstract representation of the set of successor states of the states in \(\gamma(n)\) with respect to some action \(A\) (including the next-state relation \(\text{Next}\)) of the given specification.
Definition 4. For $m, n \in B(P)$ and an action $A \in \mathcal{A}$, we say that $n$ is an abstract successor of $m$ iff for all states $s, t$, if $s \in \gamma(m)$ and $(s, t) \models A$, then $t \in \gamma(n)$.

Because we may alternately interpret abstract states as elements of $B(P)$ and as predicates over concrete states, the above definition can be restated as requiring

$$\models m \land A \Rightarrow n'.$$

Since nodes of predicate diagrams are sets of $\mathcal{P}$, interpreted conjunctively, we may assume $m$ to be a conjunction of literals, and try to compute some successor $n$ in disjunctive normal form.

As the first step of the abstract evaluation of $A$ from $m$, we try to evaluate those subformulas of $A$ that contain only unprimed variables to either true or false in order to simplify the action formula. This amounts to abstractly evaluating the guards of the action. In the second step, we try to evaluate as many formulas $P'$, for $P \in \mathcal{P}$, as possible in order to assemble information about the predicates that are true or false after the action has been executed. Throughout, we maintain $A$ and the results of simplification in disjunctive normal form.

Let us first describe the abstract evaluation procedure for monadic predicates. If $\mathcal{P}$ contains $k$ monadic predicates $P_1(x), \ldots, P_k(x)$ that contain the same (concrete) state variable $x$, we let the abstract states contain $k$ state variables $x_1, \ldots, x_k$ such that $x_i$ takes values $P_i$ or not-$P_i$. For example, the predicate $y_1 = 0$ might be represented by the variable $y_1 \in \{\text{zero, not-zero}\}$. Now, every unprimed occurrence of variable $x$ in $A$ replaced by the value assigned to $x_i$ by the abstract source state $m$, for every $x_i$ that appears in $m$. User-supplied rewrite rules of the form are used to simplify the resulting expressions. The evaluation is successful if the resulting formula is of the form

$$\text{res} \equiv \bigvee_p \bigwedge_q x' = e_{pq}$$

for some expressions $e_{pq}$ that contain the “abstract values” $P_i$ and not-$P_i$. The set of abstract successor states is extracted from $\text{res}$ by picking those subformulas for which $e_{pq}$ is either $P_i$ or not-$P_i$.

Example 5. In the context of the Bakery specification considered in section 3.3, consider the evaluation of the next-state action $\text{Next}$ from the state

$$m = \{\text{pc}_1 = \text{"ll"}, \text{pc}_2 = \text{"m2"}, \overline{y_1} = \text{zero, } \overline{y_2} = \text{zero}\}$$

(Note that our tool allows concrete state variables to be left unabstracted; this is useful when variables take values from a finite domain.)

The underlying set of rewrite rules include

- $\text{zero} + 1 \rightarrow \text{not-zero}$
- $\text{not-zero} + 1 \rightarrow \text{not-zero}$
- $\text{zero} = \text{zero} \rightarrow \text{true}$
- $\text{zero} \equiv \text{not-zero} \rightarrow \text{false}$
- $\text{zero} \leq \text{zero} \rightarrow \text{true}$
- $\text{zero} \leq \text{not-zero} \rightarrow \text{true}$
- $\text{not-zero} \leq \text{zero} \rightarrow \text{false}$
- $\text{not-zero} \leq \text{not-zero} \rightarrow \text{true}$
Note that expressions such as \( \text{not-zero} = \text{not-zero} \) or \( \text{not-zero} \leq \text{not-zero} \) cannot be simplified.

Application of the abstract evaluation procedure outlined above produces the two successor states

\[
\begin{align*}
\{ \text{pc}_1 = \text{"l2"}, \text{pc}_2 = \text{"m2"}, y_1 = \text{zero}, y_2 = \text{zero} \} \\
\{ \text{pc}_1 = \text{"l1"}, \text{pc}_2 = \text{"m3"}, y_1 = \text{zero}, y_2 = \text{not-zero} \}
\end{align*}
\]

that correspond to either process taking a step in the protocol.

Given a specification \( \text{Spec} \equiv \text{Init} \land \Box \text{Next} \land L \), we construct a predicate diagram \( G \) as follows: first \( \text{Init} \) is rewritten to disjunctive normal form, producing a set \( I \) of abstract initial states. Starting from \( I \), we repeatedly apply abstract evaluation for \( \text{Next} \) and for any action such that \( \text{WF}(A) \) or \( \text{SF}(A) \) appears as a conjunct in \( L \) in order to arrive at the set \( N \) of abstract states and the relations \( \delta_A \) for the chosen actions. Assuming that all rewrite rules are correct, the resulting predicate diagram is guaranteed to satisfy the conditions of theorem 3.

The procedure described above can be extended to sets \( \mathcal{P} \) that contain other than just monadic predicates, but it becomes somewhat more tedious to describe and implement because it can no longer simply be based on syntactic substitution. We prefer to rely on the strengthening technique described in section 5.2.

## 5 Improving Abstract Evaluation

If the abstraction is not fine enough, it is not always possible to successfully evaluate an action \( A \) with respect to a source state \( m \), because some of the guards cannot be reduced to true or false. In such a situation, we can try to compute an approximation, introducing \( \text{maybe} \) edges. Any property proved from the resulting over-approximation is still valid of the original specification \( \text{Spec} \). On the other hand, we may be able to eliminate \( \text{maybe} \) edges introducing an “on-the-fly” refinement of the abstraction with the help of an automatic prover. Alternatively, we may introduce non-determinism as allowed in condition (2) of theorem 3 by relying on a lattice of abstractions.

### 5.1 Maybe Edges

Suppose we choose \( \{ \text{zero}, \text{not-zero} \} \) as the abstract domain for the variables \( y_1 \) and \( y_2 \) of the Bakery algorithm. Then the abstract animation based on the evaluation rules of section 4 will fail because the expressions \( \text{not-zero} \leq \text{not-zero} \) that appear in the guards of actions \( L_3 \) and \( M_3 \) cannot be evaluated. Our first approximation is to assign the value \( \text{maybe} \) to such uninterpreted expressions:

\[
\text{not-zero} \leq \text{not-zero} \rightarrow \text{maybe}
\]

“Maybe” values are propagated using rewrite rules such as

\[
\begin{align*}
\text{maybe} \land \text{maybe} & \rightarrow \text{maybe} \\
\text{false} \land \text{maybe} & \rightarrow \text{false} \\
\lnot \text{maybe} & \rightarrow \text{maybe}
\end{align*}
\]
Successor states can be extracted as described above even in the presence of *maybe* conjuncts, but we remember such situations and indicate edges obtained in this way using dashed edges. For the Bakery specification, we obtain the graph shown in figure 5 (the values assigned to the abstract variables are shown in a separate table). The diagram gives a clear indication of the progress of the two processes with the first process moving in the vertical and the second in the horizontal direction.

The predicate diagram corresponds to an over-approximation of the Bakery specification and abstractly represents every behavior of the Bakery algorithm. It cannot be used to prove the invariant

\[\text{Bakery Spec} \Rightarrow \Box \neg (pc_1 = "l4" \land pc_2 = "m4")\]

that asserts mutual exclusion for the two processes. However, every path leading to the abstract state l4m4 contains a dashed edge, and these edges indicate opportunities...
for refining the approximation by reconsidering the transitions in view of the concrete specification. We now describe a technique that can perform such a refinement during the computation of the diagram.

5.2 “On-the-fly” Refinement of Diagrams

Maybe edges are introduced when predicates that appear in the concrete-level next-state relation (such as $y_1 \leq y_2$ in the Bakery example) cannot be interpreted in the abstract states. We suggest to reconsider the offending transitions based on the predecessors of the abstract source state. More precisely, suppose that we cannot interpret a sub-formula $Q$ in an abstract state $m$ during the evaluation of action $A$. We may then try to evaluate $Q'$ with respect to the formula $n \land A \land m'$, for every predecessor $n$ of $m$. In other words, we try to prove either of

$$n \land A \land m' \Rightarrow Q' \quad \text{or} \quad n \land A \land m' \Rightarrow \neg Q'$$

By construction, it is never the case that both formulas are provable, and so we consider the three following cases:

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n \land A \land m' \Rightarrow Q'$</td>
<td>Proved</td>
<td>Unproved</td>
<td>Unproved</td>
</tr>
<tr>
<td>$n \land A \land m' \Rightarrow \neg Q'$</td>
<td>Unproved</td>
<td>Proved</td>
<td>Unproved</td>
</tr>
</tbody>
</table>

In the first and second case, we may add $Q$ resp. $\neg Q$ to the label of $m$ (if $m$ has more than one predecessor, the node is split into two copies). If all subformulas that originally evaluated to $maybe$ can be decided one way or another, the $maybe$ edge can be replaced by a standard transition.

In the case of the Bakery example we can prove

$$13m2 \land Next \land 13m3' \Rightarrow \neg(y'_1 \leq y'_2)$$
$$12m3 \land Next \land 13m3' \Rightarrow y'_1 \leq y'_2$$

Consequently, state $13m3$ is split into two states $13m3 \land y_1 \leq y_2$ and $13m3 \land \neg(y_1 \leq y_2)$ as indicated in figure 6. Continuing the abstract evaluation, both $maybe$ edges leading to node 14m4 disappear, and mutual exclusion can be proven.

In our implementation, we remember the subformulas of the concrete-level action that evaluated to $maybe$. If such expressions remain after rewriting completes, the tool calls the automatic prover Simplify [8] in order to try and solve the resulting proof obligations,
and splits the node as necessary. Applying this modified generation procedure to the Bakery specification, we arrive at the predicate diagram schematically shown in figure 7. In this diagram, the node at which both $pc_1 = "l3"$ and $pc_2 = "m3"$ are true has been split into two nodes that differ in the value assigned to the predicate $y_1 \leq y_2$, and the node $l4m4$ has been eliminated. In particular, mutual exclusion can now be established, although the diagram still contains two maybe edges.

Because of these edges, we can still not prove liveness properties about the Bakery algorithm. However, the edges can be eliminated by forward propagation of the inferred properties in a similar manner. In this way we obtain the predicate diagram shown in figure 8 for the Bakery example which precisely represents the behaviors of the specification modulo the chosen abstraction. Similar graphs have been produced manually, for example, in [14], but to our knowledge our tool is the first that is able to obtain this diagram semi-automatically, with the user only indicating that zero and non-zero values for the “ticket” variables should be distinguished.

Running Spin on the Promela encoding of the diagram of figure 8, we can prove liveness properties such as $Bakery.Spec \Rightarrow (pc_1 = "l2" \iff pc_1 = "l4")$ as well as precedence properties such as one-bounded overtaking for either process.

5.3 Using a Lattice of Abstractions

Sometimes when we cannot evaluate an expression in the abstract world we can change the abstract world using a lattice abstraction. For example, with the classical calculation rules [10] on $\{\text{odd}, \text{even}\}$ we cannot evaluate $\text{even div 2}$ and we cannot prove that $n$ is always greater than 0 for the “dining mathematicians” specification. Instead, we can use an abstraction using the three abstract values $\text{zero}, \text{even} > 0$ and $\text{odd}$; this amounts to abstracting on both predicates $\text{even}(n)$ and $n = 0$. However, the expression $\text{even} > 0 \text{ div 2}$ is yet uninterpreted. We can replace this abstract expression by another abstract value.

Fig. 7. Predicate diagram for Bakery with some maybe edges removed.
not-zero which represents every natural greater than 0. After this evaluation we transform 
\(x = \text{not-zero} \) in \(x = \text{even}^>0 \lor x = \text{odd}\). Formally, this transformation can be explained 
as a change of the degree of abstraction, cf. figure 9. When successful evaluation of 
the predicates at the more abstract level has established that \(\text{even}^>0 \) \ div 2 is a positive 
natural number, we can re-descend to the first abstraction based on \(\text{zero}\), \(\text{even}^>0\) and 
\(\text{odd}\) using the following calculation rules (where the variable \(x\) that appears in the last 
rule can be instantiated arbitrarily).

\[
\begin{align*}
\text{even}^>0 + \text{even}^>0 & \rightarrow \text{even}^>0 \\
\text{odd} + \text{even}^>0 & \rightarrow \text{odd} \\
\text{even}^>0 \cdot \text{even}^>0 & \rightarrow \text{even}^>0 \\
\text{odd} \cdot \text{odd} & \rightarrow \text{odd} \\
\text{even}^>0 \div 2 & \rightarrow \text{not-zero} \\
\text{even}(\text{odd}) & \rightarrow \text{FALSE} \\
x = \text{not-zero} & \rightarrow x = \text{even}^>0 \lor x = \text{odd}
\end{align*}
\]

In this way, we can automatically construct the predicate diagram \(\text{DiningGraph}\) of 
figure 4, except for the ordering annotation that must still be justified interactively. In 
principle, similar methods of abstract evaluation could be used to compute ordering 
notations, but we believe that they should be used only judiciously as they greatly 
affect the effectiveness of model checking.

6 Related Work

The first work using graphs to visualize and structure temporal proofs for concurrent 
programs is due to Lamport and Owicki [27]. There, proof lattices are used to better 
explain logic rules and to “see” and so verify what a program is supposed to do. Building 
on these ideas, proof charts were introduced in Radha Cousot’s thesis [5]. A proof chart 
for a transition system is a finite graph with a unique start and final state. Proof obligations
are associated with every sub-chart, and “return” edges can be labelled by well-founded orderings to guarantee termination of the system. In contrast to our diagrams, these approaches concentrate on illustrating the structure of the proof rather than of the system under analysis.

Predicate-action diagrams proposed by Lamport [18] represent the safety part of specifications. An interesting point is that different, complementary views about a specification can be illustrated by different diagrams (even for the same set of predicates), and that the proof of refinement relations via diagrams is considered. Manna et al [7, 22, 23] have also advocated the diagrammatic verification of temporal logic properties. The main difference is in the representation of fairness conditions, which we assert on the level of entire diagrams rather than for individual edges. This simplifies the verification conditions related to fairness assumptions, and allows to take them into account during abstract evaluation. We also allow an arbitrary number of ordering annotations, which reduces the number of proof obligations, and should be more intuitive for a system designer. We have prototypically implemented refinement rules discussed in [22] that allow to split nodes or remove edges in our tool. Probably closest in spirit, although different in detail, to our work is the work by Bensalem, Saïdi et al on the generation of invariants that is incorporated in the InVest and SAL tools [3, 2, 28]. While our approach focuses more on abstract evaluation, they use the theorem prover PVS as their primary workhorse. Another conceptual difference is our emphasis on liveness properties; indeed, we believe that it is here that abstraction-based techniques can be used to full advantage because the deductive verification of liveness properties is inherently tedious and difficult.

7 Conclusion and Future Work

In this paper we have proposed predicate diagrams as a general framework for presenting Boolean abstractions of reactive systems and as a adequate theoretical basis for the integration of deductive and algorithmic verification techniques. We have shown that both safety and liveness properties can be verified from predicate diagrams thanks to the presence of fairness assumptions and of annotations related to well-founded orderings that can be associated with state transitions. While the focus of this paper has been a
bottom-up approach to verification of given specifications, which could be characterized as *abstract model checking*, we have found predicate diagrams to be equally useful for proving refinement relations between systems in a top-down development method. For the acceptance of formal methods in practice it is essential to achieve as much automation as possible, and to be able to supply relevant feedback to the developer. While obviously the process of computing abstractions for infinite-state systems can never be fully automatic, we have shown that a substantial degree of automation is possible. We also believe that predicate diagrams provide an adequate level of abstraction for system engineers, and that the abstract counter-examples that are produced when model checking a predicate diagram indicate how to refine the chosen abstraction. The generation of *maybe* edges should also help clarify missing predicates or evaluation rules for the computation of predicate diagrams. Some small-scale case studies have convinced us that the method is powerful and really helps in verifying reactive systems. Our current prototype implementation is based on the rewriting engine *Logic Solver* from Atelier B [29], the automatic prover Simplify [8], and the model checker Spin [12]. Besides carrying out further case studies, we intend to concentrate on the representation of hierarchy and refinement in the formal development of reactive systems via predicate diagrams.

**References**

Modular Verification for a Class of PLTL Properties

Pierre-Alain Masson, Hassan Mountassir, and Jacques Julliand

LIFC - Laboratoire d’Informatique de l’université de Franche-Comté
16, route de Gray, 25030 Besançon Cedex
Ph:(33) 3 81 66 64 52, Fax:(33) 3 81 66 64 50
{masson,mountass,julliand}@lifc.univ-fcomte.fr
Web : http://lifc.univ-fcomte.fr

Abstract. The verification of dynamic properties of a reactive systems by model-checking leads to a potential combinatorial explosion of the state space that has to be checked. In order to deal with this problem, we define a strategy based on local verifications rather than on a global verification. The idea is to split the system into subsystems called modules, and to verify the properties on each module in separation. We prove for a class of PLTL properties that if a property is satisfied on each module, then it is globally satisfied. We call such properties modular properties. We propose a modular decomposition based on the B refinement process.

We present in this paper an usual class of dynamic properties in the shape of $\Box (p \Rightarrow Q)$, where $p$ is a proposition and $Q$ is a simple temporal formula, such as $\diamond q$, $\Box q$, or $q U r$ (with $q$ and $r$ being propositions). We prove that these dynamic properties are modular. For these specific patterns, we have exhibited some syntactic conditions of modularity on their corresponding Büchi automata. These conditions define a larger class which contains other patterns such as $\Box (p \Rightarrow \Box (q U r))$.

Finally, we show through the example of an industrial Robot that this method is valid in a practical way.

Keywords. Refinement, modularity, Verification, model-checking, Büchi automata, Propositional linear temporal logic PLTL, B specification.

1 Introduction

This paper is about the problem of the verification of finite reactive systems [18,3,4]. The work that we present takes the B events system specification as a context [1], in which we introduce and verify dynamic properties of liveness and safety [16,17].

A solution for the verification of dynamic properties is to use a model-checker [10,11]. In comparison with proof techniques, such an approach offers the advantage of a possible and entire automatization, without the use of variants and loop invariant. The well known inconveniences of it are its limitation to finite
state systems and the potential combinatorial explosion of the number of states to be checked.

The problem of combinatorial explosion has been treated through various methods with time or memory space gain as a result. Let us cite by the way the memory compression techniques [12], the partial verification techniques based on heuristics [9], the memory efficient algorithms [6], the partial order techniques [21,8] suppressing useless interleaving, the techniques of symbolic representation of the states with BDD and the techniques of state vector abstraction [7].

The approach that we expand below, based on the verification by model-checking, attacks the problem of the combinatorial explosion in a different way. We use a decomposition of the reachability graph into several modules. We then verify the properties module after module. We prove that an interesting class of PLTL properties can be verified modularly. In others words, if the property holds on every module, we prove that it holds on the whole system.

All the decompositions are not equivalent. For some decompositions and for dynamic properties that can be verified in a modular way, the modular verification is valid whereas for some other decompositions, the modular verification might fail even when a property is true. Thus, the problem consists in finding a “good” decomposition. We use the notion of refinement induced by the B method to produce the modular split. The refinement of an abstract model by a concrete one induces a split of the concrete graph into subgraphs, according to the structure of the abstract graph. On such subgraphs, a modular verification is valid.

As a matter of fact, the B refinement process introduces new dynamic properties at each level of the refinement, together with the new events to which they are linked. So, the decomposition by refinement “keeps” the sequences of new events within subgraphs. This is the reason why such a split, which relies on a partition of the state space, is a “good” modular split. In other words, the refinement process produces a semantic split.

Some other modular approaches have been proposed that use compositional verification [13]. They are based on the parallel operationality which allows to split the model into components that contribute to the entire state space in a multiplicative way. Our approach splits the model of a component in an additive way. Since on one hand the B refinement allows the expression of parallel composition and, on the other hand, it preserves all the PLTL properties but the ones that contain the next operator, the two techniques are compatible. It is possible to verify separately a process component with a modular approach in the sense of our proposition.

Conversely, in [15], Karen Laster and Orna Grumberg present a modular approach in our sense to temporal logic model-checking of software, in which a program text is partitioned into sequentially composed subprograms. The model-checking is then performed on each subprogram, separately but not independently. As a matter of fact, some information (by means of an assumption function) needs to be transmitted from one module to another. This is due to the fact that the partition proposed is a syntactic one.
Based on the refinement, the modularity that we propose is a semantic one. Furthermore, our modular approach is compatible with the use of the techniques that reduce the combinatorial explosion mentioned above.

Section 2 describes the concept of modular verification. In section 3, we define a class of dynamic properties by means of Büchi automata, and we prove that the properties in this class can be verified in a modular way. Section 4 defines the refinement relation between the abstract and refined specifications in \( B \). To illustrate this work, we give in section 5 a simple example of an industrial Robot in order to exhibit the abstract and detailed specifications with their PLTL properties. Finally, we conclude this work and give some future directions of research.

2 Modular Verification

2.1 Principle of the Modular Verification

The basic idea of the modular verification is simple. Since a transition system might be too large to be verified by model-checking in an exhaustive way, we shall split it into several smaller pieces called modules in order to perform the verification on each module in separation.

Notice that every state and transition of the original transition system must belong to one module at least, so that the modular split must consist of a partition for the transitions and of an overlapping for the states, that are possibly both the target of a last transition of a module and the source of a first transition of another module.

Thanks to the modular split, a property can then be verified by model-checking on each module in separation, so that it is never necessary to keep the whole transition system in memory. With such a verification, we want to be able to tell whether the property is globally true or not.

2.2 Modular Property

When a property is true on every module, it is not always possible to conclude that it is globally true. Further in this section, we look at the PLTL property pattern \( \Box(p \Rightarrow \bigcirc q) \) which is possibly true on every module although globally false. Thus, we have to distinguish the properties that can be verified in a modular way and the ones that can not. We will say that a PLTL property \( P \) is modular if and only if

\[
P \text{ true on every module } \Rightarrow P \text{ globally true.}
\]

We will give a formal definition of a modular property in definition 4.

Thus, given a specification with dynamic properties in PLTL, we first have to make sure that these properties are modular before we verify them in a modular way. Section 3.3 shows that the three PLTL property patterns \( \Box(p \Rightarrow \bigcirc q) \),
\(\Box(p \Rightarrow \Diamond q)\) and \(\Box(p \Rightarrow qtr)\), where \(p, q\) and \(r\) are propositional formulae over the states of a transition system, are modular.

Notice that

\[ P \text{ true on every module } \Rightarrow P \text{ globally true} \]

is equivalent to

\[ P \text{ globally false } \Rightarrow P \text{ false on one module at least.} \]

Thus, in order to detect the modularity of a property, we will suppose that this property is globally false. If we can prove that in this case it also false on a module, then we know that the property is modular.

As examples, let us look at the two property patterns \(\Box(p \Rightarrow \Diamond q)\) and \(\Box(p \Rightarrow \Box q)\). The first pattern is modular, as will be proved in section 3.3. The second pattern is not modular. In both cases, we suppose that the properties are globally false. That is, we consider sequences that satisfy the negations of the properties. The modular split will possibly “cut” the sequences, and so we look whether the cut sequences still negate the properties or not.

**A Modular Property:** \(\Box(p \Rightarrow \Diamond q)\). Let us suppose that the property \(P = \Box(p \Rightarrow \Diamond q)\) is globally false. Then, the negation of the property is true. Notice that \(\neg P\) is equivalent to \(\Diamond(p \land \Box \neg q)\). This means that there is a sub-sequence satisfying \(p \land \Box \neg q\). Fig. 1 shows such a sequence, and how it is cut by a two modules split.

\[ \neg \Box(p \Rightarrow \Diamond q)c \equiv \Diamond(p \land \Box \neg q) \]

![Fig. 1. Modular property \(\Box(p \Rightarrow \Diamond q)\)](image)

In module \(M_2\), there is no state satisfying \(p\) and so \(P\) is trivially true in this module. In module \(M_1\), a state satisfies \(p\) which is followed by states all satisfying \(\neg q\). As a consequence, \(P\) is indeed false in this module.

This is due to the fact that, with property pattern \(\Box(p \Rightarrow \Diamond q)\), once a state satisfying \(p\) has occurred, a response (in the shape of a state satisfying \(q\)) is waited for. If this response never comes (\(\Box \neg p\)), one may cut the sequence, it still will not come.
A Non-modular Property: \( \Box(p \Rightarrow \Box q) \). Let us suppose that the property
\( P = \Box(p \Rightarrow \Box q) \) is globally false. Then, the negation of the property is true.
Notice that \( \neg P \) is equivalent to \( \Diamond(p \land \Box \neg q) \). This means that there is a
subsequence satisfying \( p \land \Box \neg q \). Fig. 2 shows such a sequence, and how it is cut by
a two modules split.

\[
\neg \Box(p \Rightarrow \Box q) \Leftrightarrow \Diamond(p \land \Box \neg q)
\]

![Diagram](image)

**Fig. 2. Non-modular property \( \Box(p \Rightarrow \Box q) \)**

In module \( M_2 \), there is no state satisfying \( p \) and so \( P \) is trivially true in
this module. In module \( M_1 \), the state satisfying \( p \) is now followed by states all
satisfying \( q \), so that this cut sequence no longer negates \( P \). As a consequence, \( P \)
is possibly true in that module as well, although it is globally false.

This is due to the fact that, once a state satisfying \( p \) has occurred, the
sequence remains possibly true as long as a state satisfying \( \neg q \) has not occurred.
If the sequence is cut before that state, then it possibly satisfies \( P \).

### 3 Verification of the Modularity

Before we perform a modular verification, we have to prove that the dynamic
properties that we want to verify are modular.

In this section, we are looking at a class of dynamic properties defined from
the Büchi automata [20] that recognize their negation. We call this class \( BA_2 \).
We prove that every property in the class \( BA_2 \) is a modular property.

As examples of properties that belong to the class \( BA_2 \), we prove that, amongst
others, the dynamic properties suggested by the three patterns of dynamic
constraints introduced by J.-R. Abrial in [2] are modular. They consist in the
three following patterns of PLTTL properties: \( \Box(p \Rightarrow \Box q) \), \( \Box(p \Rightarrow \Diamond q) \) and
\( \Box(p \Rightarrow p \lor q) \) where \( p \) and \( q \) are propositional formulae over the states of a transition
system.

Let \( P \) be a PLTTL property. We have ever noticed that we prove \( P \) modular
equivalently by proving that if \( P \) is false on the whole transition system, then it
is false on one module at least. For this we consider the Büchi automaton that
Modular Verification for a Class of PLTL Properties

recognizes the negation of $P$. We know that every path $\sigma$ in the whole graph that negates the property is recognized by this automaton. We then prove that there is a part of such a path $\sigma$ that belongs to one module and that is recognized by the Büchi automaton. This proves that the property is false on that module.

3.1 Definitions for the Verification of the Modularity of Dynamic Properties

In this section, we introduce the notations and the definitions that we use to establish that the three above patterns are modular.

Definition 1 (Transition System) A transition system is a 6-tuple $TS = <A, S, S_0, \rightarrow, AP, L>$ where $A$ is an alphabet labelling the transitions, $S$ is the set of states, $S_0 \subseteq S$ is the set of initial states, $\rightarrow$ is the transition relation ($\rightarrow \subseteq S \times A \times S$), $AP$ is a set of propositions and $L : S \rightarrow 2^{AP}$ is the assignment function.

Let $TS = <A, S, S_0, \rightarrow, AP, L>$ be a transition system. Let us consider a partition of the set $S$ of states. Each part of the partition can be regarded as an equivalence class $EQ_i$ such that $S = \bigcup_i EQ_i$. Some states in $EQ_i$ are the source states of some transitions in $\rightarrow$ whose target states are not in $EQ_i$. We call these transitions the exiting transitions of $EQ_i$. A module $M$ of a transition system $TS$ is a transition system whose set of states is composed of the states in $EQ_i$ augmented with the target states of the exiting transitions.

Definition 2 (Module) Let $TS = <A, S, S_0, \rightarrow, AP, L>$ be a transition system. Let $P_S$ be a partition of $S$. Each equivalence class $EQ_i$ allows the definition of a module which is a transition system $TS' = <A, S', S'_0, \rightarrow', AP, L'>$ defined in the following way:

- $S' = EQ \cup \{s \in S / \exists s_1 \overset{a}{\rightarrow} s \in \rightarrow \wedge s_1 \in S_1 \wedge s \notin S_1\}$
  $S'$ is the set of states of the class augmented with the exiting states,
- $\rightarrow' = \{s_1 \overset{a}{\rightarrow} s \in \rightarrow / s_1 \in S' \wedge s \in S'\}$
  $\rightarrow'$ is the set of transitions that link the states of $S'$,
- $S'_0 = \{s' \in S' / \exists s \overset{a}{\rightarrow} s' \in \rightarrow \wedge s \overset{a}{\rightarrow} s' \notin \rightarrow'\}$
  $S'_0$ is the set of states of $S'$ reachable from a transition in $\rightarrow$ that is not in $\rightarrow'$,
- $L'$ is the restriction of $L$ on $S'$.

We call internal states the states of the modules that are in $EQ$, whereas we call exiting states the target states of the exiting transitions. With such a definition, we have indeed a partition of the transitions of $TS$ and an overlapping of the states of $TS$.

Notations 3 Let $P$ be a dynamic property in PLTL and $TS$ a transition system. We denote $TS \models P$ the fact that the property $P$ is satisfied on $TS$. Its meaning is defined by the semantics of PLTL. We denote $s \models p$ the fact that a state $s$ satisfies a boolean proposition $p$. Thus, we have $s \models p$ iff $p \in L(s)$. 

Given a PLTL property $P$, a transition system $TS$ and its split into a set $\mathcal{M}$ of modules $M$, we want to prove that $P$ is modular.

**Definition 4 (Modular Property)** Let $P$ be a PLTL property. Let $\mathcal{M}$ be the split of a transition system $TS$ into modules. The property $P$ is modular iff

$$(\forall M \in \mathcal{M}, M \vdash P) \Rightarrow TS \vdash P$$

which is equivalent to

$TS \not\vdash P \Rightarrow \exists M \in \mathcal{M} \land \exists \sigma \text{ path of } M \text{ s.t. } \sigma \vdash \neg P.$

**Definition 5 (Path of a Transition System)** A path $\sigma = s_0s_1...s_i...$ of a transition system $TS = <A,S,S_0,\rightarrow,AP,L>$ is defined as a sequence (finite or infinite) of states linked by a transition.

$\forall s_i \in \sigma, s_{i+1} \in \sigma, \exists t \in \rightarrow \text{ s.t. } t = s_i \rightarrow s_{i+1}.$

**Definition 6 (Büchi Automaton)** A Büchi automaton is a 5-tuple $B = <b_0,B,AP,\rightarrow_B,Accept>$ where:

- $b_0$ is the initial state,
- $B$ is the finite set of states ($b_0 \in B$),
- $AP$ is a set of boolean propositions (the labels of the transitions),
- $\rightarrow_B$ is the finite set of transitions labeled by propositions of $P_B : \rightarrow_B \subseteq B \times AP \times B$,
- $Accept \subseteq B$ is the set of accepting states of the automaton.

Remark. A Büchi automaton is defined as a transition system in which we distinguish accepting states, and whose transitions are labeled with boolean propositions. As the states of a Büchi automaton are not valued, we need not introduce the assignment function $L$ in its definition.

**Definition 7 (Recognition of a Finite Path)** A finite path $\sigma = s_0s_1...s_n$ of a transition system $TS = <A,S,S_0,\rightarrow,AP,L>$ is recognized by a path $\tau = b_0b_1...b_{n+1}$ of a Büchi automaton $B = <b_0,B,AP,\rightarrow_B,Accept>$ if and only if

i) $\forall i, 0 \leq i \leq n, s_i \in \sigma \Rightarrow \exists p_i \in AP \text{ s.t. } b_i \xrightarrow{p_i} b_{i+1} \in T_B \text{ and } s_i \models p_i$

ii) $b_{n+1} \in Accept$

**Definition 8 (Recognition of an Infinite Path)** An infinite path $\sigma = s_0s_1...$ of a transition system is recognized by a path $\tau = b_0b_1...$ of a Büchi automaton if and only if

i) $\forall i \geq 0, s_i \in \sigma \Rightarrow \exists p_i \in AP \text{ s.t. } b_i \xrightarrow{p_i} b_{i+1} \in T_B \text{ and } s_i \models p_i$

ii) accepting states appear infinitely often in $\tau$. 
3.2 Intuitive Presentation of the Demonstration of the Modularity Verification Correctness

Given a PLTL property $P$ and a transition system $TS$ split into a set $M$ of modules $M$, we want to prove that:

$$TS \not
\models P \Rightarrow \exists M \in M \land \sigma \text{ path of } M \text{ s.t. } \sigma \models \neg P.$$ 

For a path $\sigma$ of the whole transition system that negates $P$, we prove that a part of $\sigma$ exists that is entirely within a module and that negates $P$ as well. In order to do this, we isolate in $\sigma$ the suffix that is sufficient to negate $P$ (we call it the minimal suffix of the negation of $P$ and we prove that every prefix of this suffix still negates $P$. As a consequence, $\sigma$ can be “cut” anywhere by a module, it still negates $P$.

Let us now justify the usefulness of this notion of minimal suffix of the negation of $P$. Let $\sigma$ be a path of the whole graph that negates $P$: $\sigma \models \neg P$. The path $\sigma$ possibly contains useless information as for the negation of $P$. Consider as an example the following property $P$: $\Box(p \Rightarrow \Diamond q)$. A path including a state $s_k$ satisfying $p \land \neg q$ followed by states all satisfying $\neg q$ is a path that satisfies $\neg P$, whatever the states preceding $s_k$. So, we are only interested in the part of the path that starts with state $s_k$. Thus, given $P$ and $\sigma$, we consider a suffix $\sigma_m$, which is the suffix of $\sigma$ still satisfying $\neg P$ obtained by removing from $\sigma$ the longest possible prefix. This suffix $\sigma_m$ is called the minimal suffix of negation of $P$.

**Definition 9 (Minimal Suffix of Negation of a Property)** Let $P$ be a PLTL property and let $\sigma = s_0s_1...s_i...$ be a path of a transition system such that $\sigma \models \neg P$. We denote $\sigma_i$ the suffix $s_is_{i+1}...$ of $\sigma$.

If an integer $m$ exists such that: $\forall i \in [0...m], \sigma_i \models \neg P$ and $\forall i > m, \neg(\sigma_i \models \neg P)$, then the suffix $\sigma_m$ is called the minimal suffix of negation of $P$.

Two situations have to be considered.

1. the states involved in $\sigma_m$ all belong to a unique module $M$. Thus, $M \models \neg P$
2. the states involved in $\sigma_m$ belong to distinct modules. We consider in this case a prefix $\sigma'_m = s_ms_{m+1}...s_k$ such that all the states involved in $\sigma'_m$ belong to the same module. Thus, we have to prove that $\sigma'_m \models \neg P$.

3.3 Demonstration of the Modularity of the PLTL Properties Encoded by a Büchi Automaton in the Class $BA_2$

Let us consider a class of Büchi automata that we call $BA_2$, for which all states of the automata are accepting states, with the exception of at most the initial state and its direct successors, provided that there is no transition between any of such (non-accepting) direct successors. In other words, every path $\tau = b_0b_1b_2...$ of an automaton in the class $BA_2$ that recognizes the minimal suffix of negation of a property is such that $\forall i \geq 2, b_i \in \text{ Accept}$. 

Definition 10 (The Class $\text{BA}_2$) Let $\mathcal{B} = \langle b_0, B, AP, \rightarrow_B, \text{Accept} \rangle$ be a Büchi automaton. $B \in \text{BA}_2$ iff

$$\forall \tau = b_0 b_1 \ldots \text{ path of } B, \exists k > 0 \text{ s.t. } \forall i, 0 \leq i < k, b_i = b_0,$$

and $\forall j > k, b_j \in \text{Accept}.$

The automata encoding the properties $2(p \Rightarrow \Box q), 2(p \Rightarrow \diamond q)$ and $2(p \Rightarrow q U r)$ (represented in figures 3, 4 and 5) belong to the class $\text{BA}_2.$ We prove that all the PLTL properties that can be encoded by an automaton of the class $\text{BA}_2$ are modular properties.

Let $P$ be a PLTL formula, and let $\sigma$ be a path of a transition system $TS$ such that $\sigma \vdash \neg P.$ Let $\sigma_m$ be the minimal suffix of $\sigma$ that negates $P,$ and let $BA$ be the Büchi automaton that encodes $\neg P.$ We suppose $BA \in \text{BA}_2.$ Lemma 1 proves that a prefix of $\sigma_m$ whose all the states belong to the same module always exists. Moreover, it proves that this prefix is at least two states long. Lemma 2 proves that when recognizing $\sigma_m,$ $BA$ can not get back to its initial state once it has left it. In other words, we eliminate the loop on the initial state of $BA$ (see figures 3, 4, 5). Theorem 3 concludes that if the above conditions are true, $P$ is still negated on a module. It is obvious as $BA$ is only composed of accepting states from the third one, and so the recognition will be “cut” on an accepting state. In other words, $BA \in \text{BA}_2$ is a sufficient condition for $P$ being a modular property.

Lemma 1 Let $P$ be a PLTL formula, $\sigma$ a path of a transition system $TS$ such that $\sigma \vdash \neg P,$ and $\sigma_m$ the minimal suffix of $\sigma$ that negates $P.$

A prefix $\sigma'_m$ of $\sigma_m$ such that all the states in $\sigma'_m$ belong to the same module always exists. The prefix $\sigma'_m$ is at least 2 states long.

Proof. Consider the prefix only composed of the first two states of $\sigma_m$: $\sigma'_m = s_ms_{m+1} \ldots$ the minimal suffix of $\sigma$ that negates $P.$

$\sigma_m = s_0s_1 \ldots s_{m+1} \ldots$ is recognized by a path $\tau = b_0b_1 \ldots$ of $BA$ where initial state $b_0$ only occurs once.

Proof. Let $i > 0$ be an integer such that $b_i = b_0.$ Then, $\sigma_i = s_is_{i+1} \ldots$ is a suffix of $\sigma$ recognized by $\tau_i = b_ib_{i+1} \ldots.$ So, $\sigma_i \vdash \neg P,$ which is a contradiction since $\sigma_m$ is the minimal suffix of $\sigma$ that negates $P.$
Theorem 3 All the PLTL properties encoded by a Büchi automaton in the class $BA_2$ are modular properties.

Proof. Let $P$ be a PLTL formula, and let $TS$ be a transition system which is split into a set $\mathcal{M}$ of modules. Let $\sigma$ be a path of $TS$ such that $\sigma \vdash \neg P$, and $\sigma_m = s_0s_1\ldots$ be the minimal suffix of $\sigma$ that negates $P$. Let finally $BA = < b_0, B, AP, \rightarrow_{BA}, Accept >$ be a Büchi automaton such that $BA \in BA_2$.

A prefix $\sigma'_m = s_0s_1\ldots s_k$ of $\sigma_m$ exists such that all the states involved in $\sigma'_m$ belong to a module $M \in \mathcal{M}$. Moreover, $\sigma'_m$ is at least 2 states long (lemma 1). Let us prove that $\sigma'_m \vdash \neg P$.

We have $\sigma_m \vdash \neg P$. So, a path $\tau_m = b_0b_1\ldots$ of $BA$ exists that recognizes $\sigma_m$. Since $BA \in BA_2$, we have: $\forall i \leq 2, b_i \in Accept$ (lemma 2). Let us consider the two following cases:

- let $\sigma'_m = s_0s_1$. Then $\tau'_m = b_0b_1b_2$ recognizes $\sigma'_m$ as $b_2 \in Accept$
- let $\sigma'_m = s_0\ldots s_k$ with $k > 1$. Then $\tau'_m = b_0\ldots b_{k+1}$ recognizes $\sigma'_m$ as $b_{k+1} \in Accept$.

As a consequence, $\sigma'_m \vdash \neg P$. Finally,

$$TS \not\models P \Rightarrow \exists M \in \mathcal{M} \land \sigma'_m \text{ path of } M \text{ s.t. } \sigma'_m \vdash \neg P.$$

3.4 Modularity of the Three Modalities Introduced in $B$

In order to prove that $\Box(p \Rightarrow \Diamond q)$, $\Box(p \Rightarrow q)$ and $\Box(p \Rightarrow qUr)$ are modular, we simply prove that the automatons of the negations of these patterns belong to $BA_2$.

**Pattern 1:** $\Box(p \Rightarrow \Diamond q)$. Suppose this property is false on the global graph. Then a path $\sigma$ exists such that $\sigma \vdash \neg \Box(p \Rightarrow \Diamond q)$ (which is equivalent to $\sigma \vdash \Diamond(p \land \Box \neg q)$). The Büchi automaton that encodes such a property is given in Figure 3.

![Fig. 3. Büchi automaton encoding $\neg \Box(p \Rightarrow \Diamond q)$](image)

Figure 3. It belongs to the $BA_2$ class and thus $\Box(p \Rightarrow \Diamond q)$ is modular.

**Pattern 2:** $\Box(p \Rightarrow \Diamond q)$. Suppose this property is false on the global graph. Then a path $\sigma$ exists such that $\sigma \vdash \neg \Box(p \Rightarrow \Diamond q)$ (which is equivalent to $\sigma \vdash \Diamond(p \land \Box \neg q)$). The Büchi automaton that encodes such a property is given in Figure 4. It belongs to $BA_2$ class and thus $\Box(p \Rightarrow \Diamond q)$ is modular.
Pattern 3: $\Box(p \Rightarrow q U r)$. Suppose this property is false on the global graph. Then a path $\sigma$ exists such that $\sigma \vdash \neg \Box(p \Rightarrow q U r)$ (which is equivalent to $\sigma \vdash \Box(p \land \neg (q U r))$). The Büchi automaton which encodes such a property is given in Figure 5. It belongs to the class $BA_2$ and therefore $\Box(p \Rightarrow q U r)$ is modular.

4 Refinement and Modules

4.1 Modularity and Modules

Until now, we have only considered abstract partitions of the state space, without giving any indication on how to find a “good” partition. As a matter of fact, in order to prove in a modular way that a property $P$ is true, $P$ has to be true on every module. But with a random split of the transition system, it is possible that some modules find that $P$ is false, whereas another split would have found $P$ true on every module.

The question is, for a given modular property, how can we find a partition that makes the verification module by module successful?

Our purpose here is to suggest a partition of the state space of a transition system into several components, this partition being based on a refinement relation. The refinement that we use is temporal in the sense that the system is
observed more often. Thus, details such as new variables and new events are introduced step by step. They are observed between the old events in the abstract specification. The new properties are concerned with sequences of news events and it is suitable to verify them only on derived subsystems. We call old what is concerned with the abstract specification and new what is concerned with the refined specification.

![Fig. 6. Refinement of an abstract transition](image)

Let us recall some notions given in a paper published at IFM’99 [14]. Intuitively, the figure 6 schematizes the refinement of an abstract transition $t$ labeled with an old event into a family of refined transitions in which some conditions hold. We briefly define these notions informally.

- each labeled transition $t$ is refined by a set of refined paths made of transitions labeled with the new events and terminated with a transition labeled by the label of $t$,
- the initial state of the abstract transition is refined by the initial states and all the intermediate states of the paths,
- the final state of the abstract transition is refined by the final states of all paths.

The modular verification of a $PLTL$ property $P$ consists in

- computing the set of modules,
- verifying on each module by model-checking if $P$ is satisfied or not,
- concluding that $P$ is satisfied if all the modules satisfy $P$.

In the rest of this section we explain why the modular verification concept concerns the modules of the refined specification, rather than the full specification. The introduced dynamic properties concern the new events in the modules obtained at this step from the abstract specification. Each module is described as a transition system obtained from a state by application of the new events. In the modules, the new states are pointed in and pointed out by the old events of the abstract specification. The new transitions are labeled by the new events.
The verification of the dynamic properties is only performed on this state space augmented with its immediate neighbour states, on the border. Thus, semantically, each old transition is refined by some chains of new events terminated by a transition labeled by an old event. We establish the relation between a verification on modules and a verification on the whole system. In other words, we are essentially looking for conditions allowing us to perform local verifications rather than global ones.

4.2 B Events and Transitions Systems

We give some formal definitions for a B specification extended with PLTL formulae, and its semantics with a labeled transition system TS. The transition system semantics of a B specification is directly obtained from the Before/After predicate semantics of the B event systems given in [1], for the class of finite systems.

Modules are defined as particular labeled transition systems that refine a state and the set of transitions whose this state is source in the abstract specification.

Definition 11 (Event System) An event system is a 6-tuple \( \text{ES} = <V, I, F, \text{Init}, A, G_A> \) which consists of

- a set \( V \) of variables,
- an invariant \( I \),
- a set \( F \) of formulae,
- an initial action \( \text{Init} \) to initialize variables,
- an alphabet \( A \) of label of events,
- a set \( G_A \) of events definitions in the shape select \( g \) then a end where \( g \) denotes a proposition and \( a \) is a generalized substitution as defined in the B method [1].

Consider now two labeled transition systems \( \text{TS}_1 = <A_1, S_1, S_{i_0}, \rightarrow_1, \text{AP}, L_1> \) and \( \text{TS}_2 = <A_2, S_2, S_{i_2}, \rightarrow_2, \text{AP}, L_2> \) semantics of \( \text{ES}_1 = <V_1, I_1, \text{Init}_1, A_1, G_{A_1}> \) and \( \text{ES}_2 = <V_2, I_2, \text{Init}_2, A_2, G_{A_2}> \).

Definition 12 (Refinement of an Abstract Transition) A path \( \sigma_2 \) of \( \text{TS}_2 \) refines an abstract transition \( t = s \to s' \in \rightarrow_1 \) of \( \text{TS}_1 \) denoted \( \sigma_2 \sqsubseteq t \) iff

- \( \sigma_2 = s_1 \ldots s_n \) is a finite path s.t. the transitions \( t_1 = s_0 \xrightarrow{a_1} s_1, \ldots, t_{n-1} = s_{n-2} \xrightarrow{a_{n-1}} s_{n-1} \) are labeled with new events

\[
\forall t_i = s_{i-1} \xrightarrow{a_i} s_i \in \rightarrow_2 \ s.t. \ 1 \leq i < n, a_i \notin A_1 \land a_i \in A_2 ,
\]

- the label of the transition \( t_n = s_{n-1} \xrightarrow{a_n} s_n \) is the same than the one of \( t \):

\[
a_n = a,
\]

- the valuations of all the states, except the last one, of the path \( \sigma_2 \), do not contradict the valuation of the source state of \( t \).
the valuation of the final state of $\sigma_2$ does not contradict the valuation of the target state of $t$.

**Definition 13 (Refinement of a Transition System)** Let $TS_1$ and $TS_2$ be two labeled transition systems. We define the set $\Sigma$ of paths $\sigma_2$ of LTS$_2$ which refine each abstract transition $t \in \rightarrow_1$ enabled from $s \in S_1$ as follows:

$$\forall t = s \xrightarrow{a} s' \in \rightarrow_1, \forall \sigma_2 = s_0 \ldots s_n \text{ path of } TS_2, \sigma_2 \sqsubseteq t \Rightarrow \sigma_2 \in \Sigma.$$ 

**Definition 14 (Module Derived from the Refinement)** Let $I_2$ be the gluing invariant of $ES_2$. A module of $TS_2$, associated with a state $s \in S_1$ and an abstract transition $t \in \rightarrow_1$ is a labeled transition system $TS =< A, S, S_0, \rightarrow, AP, L >$ where

- $\rightarrow$ is the set of transitions such that
  $$\forall \sigma_2 = t_1 \ldots t_n \in \Sigma, \forall i \text{ s.t. } 1 \leq i \leq n, t_i \in \rightarrow,$$
- $S$ is the set of states such that
  $$S = \{s'/s' \in S_2 \text{ and } s' \wedge I_2 \Rightarrow s\} \cup$$
  $$\{s''/\forall t = s_{i-1} \xrightarrow{a} s_i \in \rightarrow, s'' = s_i \vee s'' = s_i\},$$
- $S_0$ is the set of initial states such that
  $$S_0 = \{s'/s' \in S \wedge \forall t = s_{i-1} \xrightarrow{a} s_i \in T, s' \neq s_i\}.$$

Informally, a module is a labeled transition system defining all paths $\sigma_2$ of $TS_2$ which refines each abstract transition $t$ enabled from one state $s \in S_1$. Notice that the number of modules in $TS_2$ is the number of states in $TS_1$.

The specification refinement guarantees essentially three points for a labeled transition system

- the refinement of each transition of the abstract transition system by a set of refined paths,
- the connectivity between the paths of the modules with the abstract transitions,
- the modules are free from deadlocks and livelocks.

All these definitions, and the consequences of the refinement of specification are established and proved in [5].

**5 Specifications of the Robot Example**

In order to make these notions clear, we illustrate them with the example of an industrial robot composed of an *Arrival Device*, an *Exit Device* and a *Carrier Device*. This robot carries some parts by moving from the Arrival Device to the Exit Device. We first give an abstract specification and then we give two refinement levels of the system. The indices 0, 1 and 2 of the variables indicate the level of refinement.
5.1 Informal Specification

The figure 7 represents the physical system. The Carrier Device (called \( CD \)) takes a part from the Arrival Device (called \( AD \)) and places it on the Exit Device (called \( ED \)). We first describe it as an abstract system, considering only two operations to load and unload a part. It simply consists in the Carrier Device taking a part and putting it onto the Exit Device.

![Diagram of the robot]

**Fig. 7.** The robot

**Informal Presentation of an Extended \( B \) Specification.** We describe a specification as an abstract system as in \( B[1] \). It is composed of two parts.

- a descriptive specification that indicates what the system does,
- an operational specification that indicates how the system proceeds.

The descriptive specification is essentially composed of a list of variables and of an invariant expressing safety properties restricting the set of valid states. In order to complete this part, we have proposed an extension of those systems with dynamic properties such as liveness, in the shape of PLTL formulae. In the example produced below, we use the four future temporal operators called usually *Always*, *Next*, *Eventually* and *Until*, denoted respectively by the symbols \( \Box, \bigcirc, \lozenge \) and \( \mathcal{U} \).

The operational specification is also composed of two parts.

- the description of the initial states in the *Initialization* section,
- the description of all the events in the shape of guarded actions. The semantic of such an event is that it is enabled when the guard is true, and in that case the action is performed, so that it transforms the state of the system.

The verification phase makes sure that both invariant and dynamic properties hold with the operational specification. As in the \( B \) method, the invariant is proved by means of a theorem-prover, but we will use a model-checker in order to verify the dynamic properties on the derived transition systems.
Abstract Specification. In this first approach of the system, we focalize our observation on one component and we only look at the behaviour of the Carrier Device. We ignore what happens to the other devices. There are only two states: one when the device is free and one when it is busy. The specification proposed in figure 8 uses only one variable that gives the status of the Carrier Device denoted $CD_0 \in \{free, busy\}$. The invariant property gives the sorting of the variables. Two events are introduced in order to load and unload the part.

$$\text{System} \quad Robot_0$$

$$\text{Variables} \quad V_0 = \{CD_0\}$$

$$\text{Invariant} \quad I_0 = CD_0 \in \{free, busy\}$$

$$\text{Dynamic properties} \quad P_{0,1} = \Box(CD_0 = busy \Rightarrow \Diamond CD_0 = free)$$

$$\text{Initialization} \quad Init_{0} = CD_0 := free$$

$$\text{Events} \quad Load_0 \triangleleft Select \; CD_0 = free \; \text{then} \; CD_0 := busy \; \text{end}$$

$$\quad Unload_0 \triangleleft Select \; CD_0 = busy \; \text{then} \; CD_0 := free \; \text{end}$$

$$\quad end \quad Robot_0$$

Fig. 8. Abstract specification of the robot

The dynamic property $P_{0,1}$ is a liveness property that states that when the Carrier Device is busy, then eventually it becomes free.

The corresponding transition system is represented in figure 9. It has two states and two events. It represents all the execution paths by means of a finite state system [3]. In this case, there is only one variable with two possible values. The transitions are labeled with the events Load and Unload representing the evolution of the system from one state to another.

Fig. 9. The abstract transition system

5.2 Refined Specification

We now present a possible refined specification of the specification given above.
Informal Presentation of a Refinement of Specification. As in the \textit{B} method \cite{1}, we use the concept of refinement. The idea is to enhance the detail level of observation of the system, so that the refined specification gives further details on what the system does and how it proceeds. Thus, refinement after refinement, the description will go from a high level specification to a specification that can be directly implemented. The refined specification contains the same sections than the abstract one: \textit{Variables, Invariant, Dynamic Properties, Initialization} and \textit{Events}.

The figure 10 gives a refined specification of the Robot. Both abstract and refined specifications must respect the following constraints:

- the two sets of variables are disjoint,
- the new invariant, also called the gluing invariant, expresses a relation between the two sets of variables of the two levels,
- the dynamic properties are new properties concerning the behaviour induced by the new events,
- the old events are described again in such a way that they are refined,
- the new events are introduced.

In the \textit{B} method, the refinement relation is verified and warrants the abstract invariant properties. The old dynamic properties are preserved by refinement notion as defined in \cite{2}. This question is not treated in this paper in which we consider only the verification of new dynamic properties.

First Refinement of the Robot. The refined specification in figure 10 settles the following point: the Carrier Device moves into two directions denoted \textit{Up} and \textit{Down}. The \textit{Up} movement specifies that the Carrier Device is busy and goes up to the Exit Device in order to put a part. The \textit{Down} movement specifies that it goes to the Arrival Device in order to take another part. We only need one new variable $\text{PosCD}_1 = \{\text{Down}, \text{Up}\}$ which statuses the position of the Carrier Device.

The variables of the levels 0 and 1 are glued by the gluing invariant. The old events are refined with the new constraints and keep the same label. The guards of the events are reinforced. We have two new properties for this specification and they concern only the movement of the Carrier Device. They are safety properties. The properties $P_{1_1}$ and $P_{1_2}$ ensure that when the Carrier Device goes up it is busy, and when it goes down it is free. The transition system associated to this first refinement of the specification is given in figure 11.

Second Refinement of the Robot. From this specification and with the same process, we give another refinement through a third detailed specification by introducing the component \textit{Exit Device}. When a part is in it, it can be removed. Two new variables $\text{ED}_2$ and $\text{AD}_2$ are introduced with the possible values \textit{free} or \textit{busy}. Once again, the old events are refined and the guards are reinforced. Two new events are introduced, denoted \textit{ArrivalPart}_2 and \textit{Evac}_2. The first one expresses that when a part is on the Arrival Device, the Carrier Device
System  

Robot_1 refines Robot_0

Variables

V_1 \equiv \{CD_1, PosCD_1\} 

Invariant

I_1 \equiv \{CD_1 = CD_0 \land PosCD_1 \in \{Down, Up\}\}

New Dynamic Properties

P_{11} \equiv \Box(PosCD_1 = Down \land \Diamond(PosCD_1 = Up) \Rightarrow CD_1 = \text{busy})

P_{12} \equiv \Box(PosCD_1 = Up \land \Diamond(PosCD_1 = Down) \Rightarrow CD_1 = \text{free})

Initialization

Init_1 \equiv CD_1 := free \land PosCD_1 := Down

Old Events

Load_1 \equiv Select CD_1 = free \land PosCD_1 = Down

then CD_1 := busy end

Unload_1 \equiv Select CD_1 = busy \land PosCD_1 = Up

then CD_1 := free end

New Events

StopUp_1 \equiv Select CD_1 = busy \land PosCD_1 = Down

then PosCD_1 := Up end

StopDown_1 \equiv Select CD_1 = free \land PosCD_1 = Up

then PosCD_1 := Down end

End

Robot_1

Fig. 10. First refinement of the robot specification

Fig. 11. The first refinement transition system
transports it to the Exit Device. The second event removes the part from the Exit Device. The figure 12 gives the obtained specification.

\begin{verbatim}
System
Robot2 refines Robot1

Variables
V2 = \{CD2, PosCD2, AD2, ED2\}

Invariant
I2 = \{CD2 = CD1 \land PosCD2 = PosCD1 \\
\land AD2 \in \{\text{free, busy}\} \land ED2 \in \{\text{free, busy}\}\}

New Dynamic properties
P2 \equiv \Box(\text{CD2 = busy} \Rightarrow \Diamond \text{CD2 = free})
U (CD2 = busy \land ED2 = free))

Initialization
Init2 = CD2 := free \land PosCD2 := Down

Old Events
Load2 = Select CD2 = free \land PosCD2 = Down \\
then CD2 := busy end

Unload2 = Select CD2 = busy \land PosCD2 = Up \\
then CD2 := free end

StopUp2 = select CD2 = busy \land PosCD2 = Down \\
then PosCD2 := Up end

StopDown2 = Select CD2 = free \land PosCD2 = Up \\
then PosCD2 := Down end

New Events
PartArrival2 = Select AD2 = free then AD2 := busy end

Evac2 = Select ED2 = busy then ED2 := free end

End
Robot2
\end{verbatim}

Fig. 12. The second refinement specification

The gluing invariant states that the variables are identical to their abstraction. The dynamic property \(P_2\) states that the Carrier Device remains busy until the Exit Device is free. The labeled transition system given in figure 13 refines the labeled transition system of figure 11. It is divided into 4 modules. The modules \(M_1'\) and \(M_3'\) both contain 6 states and 6 transitions, and the modules \(M_2'\) and \(M_4'\) both contain 8 states and 8 transitions.

5.3 Verification of the Properties

We summarize the results of the modular verification of the properties \(P_0\), \(P_1\), \(P_2\), and \(P_3\) presented above. These 4 dynamic properties have the Büchi automata of their negations in \(BA_2\) and so they are modular properties. It is sufficient that we verify them on each module in separation. They can be verified by a simple model-checking.

- \(P_0 \equiv \Box(\text{CD0 = busy} \Rightarrow \Diamond \text{CD0 = free})\) is satisfied in the abstract specification at the level 0.
- \(P_1 \equiv \Box(\text{PosCD1 = Down} \land \Box(\text{PosCD1 = Up}) \Rightarrow \text{CD1 = busy})\) from level 1 is satisfied in \(M_2\). In the module \(M_1\), there is only one state such that \(\text{PosCD1 = Down}\) and there is no successor. We then conclude that \(\text{PosCD1 = Down} \land \Box(\text{PosCD1 = Up})\) is false and globally \(P_1\) is satisfied.
Fig. 13. The second refined transition system

- $P_{1,2} \equiv \square (\text{Pos}CD_1 = \text{Up}) \land \Box (\text{Pos}CD_1 = \text{Down}) \Rightarrow CD_1 = \text{free}$ from level 1 is satisfied in $M_1$. We use the same reasoning than $P_{1,1}$.
- $P_{2,2} \equiv \Box (CD_2 = \text{busy} \Rightarrow CD_2 = \text{busy} \cup (CD_2 = \text{busy} \land ED_2 = \text{free}))$ defined on level 2 is satisfied modularly on $M'_2$ and on $M'_3$. In the other modules $M'_1$ and $M'_2$ we have $\bigcup (CD_2 = \text{free})$. We then conclude that $P_{2,2}$ is globally satisfied.

6 Conclusion

We propose a technique that allows the modular verification of a class of usual dynamic properties of reactive systems. These properties are safety and liveness properties, and they are checked by means of a model-checker. The main problem in applying this technique based on reachability analysis is the potential combinatorial explosion of the state space. In order to attack this problem, we define a strategy based on local verifications which uses the refinement process. From the abstract specification of a $B$ event system, we derive a detailed specification by introducing new events. We then look at the new properties that have to be checked on the chains of new events. These properties are verified on modules
rather than on the whole transition system, and consequently the verification of a \textit{PLTL} property reduces to the verification of a local \textit{PLTL} property.

For some specific dynamic property patterns, we have exhibited sufficient conditions to assert that if a \textit{PLTL} property is satisfied on each module, then it is globally satisfied. The negations of these modular properties are recognized by particular Büchi automata that we describe. However, in some cases, if the verification of the property fails on a module, we cannot conclude about the satisfaction of this property. This situation is possibly due to the fact that the property should have been established at an abstract level of abstraction because it uses a chain of old events. This point has already been discussed and published at the Conference IFM’99 in York [14].

We have shown in [19] another possibility to reduce the complexity of verification by model-checking for large systems which consists of a combination of model-checking and theorem-proving techniques. Patterns in the shape of $\Box(p \Rightarrow Q)$ can be studied in two verifications. Invariance properties $\Box-p$ can be verified with the prover and temporal properties $\Diamond p \land \Box(p \Rightarrow Q)$ with a model-checker. Then, there are two kinds of modules: those in which we have $\Box-p$ and those in which some states verify $p$. In the first case, the $B$ prover is the proper tool for proving $\Box-p$. Thus, the modules are treated separately without communication. In practice, we are working at the implementation of a verification tool using both the $B$ and $SPIN$ environments.

The work that we present in this paper only uses the syntactic way to characterize the modular dynamic properties, without taking into account their semantics. The modular class is formed of patterns in the shape of $\Box(p \Rightarrow Q)$, where $p$ is a proposition and $Q$ is a temporal formula such as $\Box q$, $\Diamond q$, and $qU r$.

We project in future research to extend this class of properties by using the semantics level of the specification. We know that a module is exited with transitions that correspond to the occurrence of old events. The idea is to use this information in order to prove that some properties (not in $BA_2$) are modular in the very context of such a semantic modular split. This class of \textit{PLTL} properties could combine many different temporal operators.

References

Towards Model Checking
Stochastic Process Algebra

Holger Hermanns¹, Joost-Pieter Katoen¹, Joachim Meyer-Kayser², and Markus Siegle²

¹ Formal Methods and Tools Group, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
{hermanns|katoen}@cs.utwente.nl
² Lehrstuhl für Informatik 7, University of Erlangen-Nürnberg, Martensstraße 3, 91058 Erlangen, Germany
{jmmeyerk|siegle}@informatik.uni-erlangen.de

Abstract. Stochastic process algebras have been proven useful because they allow behaviour-oriented performance and reliability modelling. As opposed to traditional performance modelling techniques, the behaviour-oriented style supports composition and abstraction in a natural way. However, analysis of stochastic process algebra models is state-oriented, because standard numerical analysis is typically based on the calculation of (transient and steady) state probabilities. This shift of paradigms hampers the acceptance of the process algebraic approach by performance modellers. In this paper, we develop an entirely behaviour-oriented analysis technique for stochastic process algebras. The key contribution is an action-based temporal logic to describe behaviours-of-interest, together with a model checking algorithm to derive the probability with which a stochastic process algebra model exhibits a given behaviour-of-interest.

1 Introduction

The analysis of systems with respect to their performance is a crucial aspect in the design cycle of concurrent information systems. Although huge efforts are often made to analyse and tune system performance, these efforts are usually isolated from contemporary hardware and software design methodology [15,18, 28]. This insularity of performance analysis has numerous drawbacks. Most severe, it is unclear how to incorporate performance analysis into the early stages of a design, where substantial changes are still not too costly. In these design stages, system models are nowadays developed by means of semi-formal methods such as UML or SDL.

In order to overcome the insularity problem, there is a growing tendency towards the integration of performance modelling and analysis into (semi-)formal methods, such as Petri nets [1], process algebra [21], or SDL [12]. This integration has potential benefits for the application of both formal methods and performance analysis: Using a formal method, performance models of interest are

© Springer-Verlag Berlin Heidelberg 2000
readily available for analysis. Conversely, the availability of quantitative insight into a design clearly adds extra value to a formal design.

Process algebra is an influential approach to the modelling of concurrent systems using formal methods. Developed in the 80ies, process algebra is radically behaviour-oriented. Systems are modelled by describing the possible behaviours they can exhibit to the external environment. This approach led to powerful composition operators as means to compose behaviours hierarchically. The behaviour-oriented approach also enables one to employ abstraction mechanisms to compress behaviours to only those fragments relevant in a specific environment.

In a behaviour-oriented setting, the notion of a state is an auxiliary one. To identify a state is of no importance, since a state is completely characterized by the behaviour it exhibits. As a consequence, states exhibiting the same behaviour are considered to be indistinguishable, and hence are (or can be) collapsed to just a single state, using an appropriate notion of equivalence (such as bisimulation).

During the last decade, stochastic process algebra (SPA) has emerged as a promising way to carry out compositional performance and reliability modelling, mostly on the basis of continuous-time Markov chains (CTMCs) \[21\]. Following the same philosophy as ordinary process algebra, the stochastic behaviour of a system is described as the composition of the stochastic behaviours of its components.

However, all standard analysis algorithms for stochastic models are purely state-based. They compute interesting information about the model on the basis of state probabilities derived by either transient or steady-state analysis \[35\]. As a consequence, there is a disturbing shift of paradigms when it comes to the analysis of stochastic process algebra models: While the model is specified in a behaviour-oriented style, the performance properties-of-interest are defined in terms of states, on a very different level of abstraction. This shift of paradigms clearly hampers the acceptance of the SPA approach to performance modellers.

In the context of model checking of ordinary (i.e. non-stochastic) process algebra models, a similar mismatch has been attacked successfully. Model checking is a successful technique to establish the correctness of a given model, relative to a set of temporal logic properties which the model should satisfy \[9,10\]. The most efficient model checkers use the logics LTL or CTL. Though different in nature, both logics are state-oriented, their basic building blocks are state propositions. So, at first sight they do not fit well to a behaviour-based formalism.

To import the success of model checking to behaviour-oriented formalisms, De Nicola and Vaandrager have pioneered the development of an action-based variant of CTL, called $\text{aCTL}$ \[33,34\]. $\text{aCTL}$ is behaviour-oriented, yet it naturally corresponds to CTL. In particular, \[33,34\] provide a translation from $\text{aCTL}$ to CTL that allows one to perform (behaviour-oriented) $\text{aCTL}$ model checking by means of a (state-oriented) CTL model checker (on a transformed

\[1\] The logic $\text{aCTL}$ should not be confused with the logic $\text{ACTL}$, the restriction of CTL to universal path quantifiers.
model) with only linear overhead. (It should however be noted that direct aCTL
model checkers are more popular by now [14,32].)

In this paper, we develop a behaviour-oriented analysis technique for CT-
MCs, and hence for the SPA approach modelling and analysis become entirely
behaviour-oriented. This is the central contribution of the paper. Our analysis
complements behaviour-oriented CTMC modelling with SPA in the same sense
as De Nicola and Vaandrager’s work complements ordinary process algebraic
modelling.

We develop an action-based, branching-time stochastic logic, called aCSL
(action-based Continuous Stochastic Logic), that is strongly inspired by CSL,
the continuous stochastic logic first proposed in [2] and further refined in [5,3].
Similar to CSL, aCSL provides means to reason about CTMCs, but opposed to
CSL, it is not state-oriented. Its basic constructors are sets of actions, instead
of atomic state propositions. The logic provides means to specify temporal and
timed properties, and means to quantify their probability. aCSL allows one to
specify properties such as “there is at least a 30% chance that action SEND will
be observed within at most 4 time units”. After defining syntax and semantics,
we develop a dedicated model-checking algorithm for aCSL. As an application
example, we study behaviour-oriented performance and reliability properties of a
multiprocessor mainframe example taken from [23]. Furthermore, we show that
Markovian bisimulation, an equivalence notion that can be used to compress
SPA specifications compositionally, preserves aCSL-formulas. This property is
exploited in our case study.

For efficiency reasons, our model checking algorithm is not based on a trans-
lation from aCSL to CSL. Instead, it checks aCSL properties directly. A trans-
lational approach would allow one to use a state-based CSL model checker (such
as E TMC² [24]), but with an increase of the state space. We briefly sketch the
translation from aCSL to CSL, which is inspired by Emerson and Lei [13], and
discuss why the linear translation in the style of De Nicola and Vaandrager [33,
34] fails in the stochastic setting.

The paper is organised as follows. Section 2 introduces action-labelled Mar-
kov chains, the basic model considered in this paper. In Section 3, we define
syntax and semantics of aCSL, derive a number of convenient operators, and
discuss Markovian bisimulation. Section 4 focuses on model checking of aCSL.
Section 5 studies aCSL-properties of the multiprocessor example, while Section
6 briefly discusses the translational approach to model checking aCSL. Section
7 concludes the paper.

2 Action-Labelled Markov Chains

The operational semantics of purely Markovian process algebra such as
TIPP [16], PEPA [29] and (the core of) EMPA [7] is defined in terms of la-
belled transition systems where transitions are labelled with pairs of actions and
\footnote{We call a stochastic process algebra purely Markovian if the delay of any action is
governed by an exponential distribution.}
rates. In this section we briefly recall this notion and define some notations that are convenient for our purpose.

**Action-labelled Markov chains.** Let Act denote a set of actions, ranged over by \( a, b \). We will use \( A, B \) as subsets of Act and adopt the convention that for singleton sets curly brackets are omitted; i.e., we write \( a \) for \( \{a\} \).

**Definition 1.** An action-labelled Markov chain (AMC) \( \mathcal{M} \) is a triple \((S, A, \rightarrow)\) where \( S \) is a set of states, \( A \subseteq \text{Act} \) is a set of actions, and \( \rightarrow \subseteq S \times (A \times \mathbb{R}_{\geq 0}) \times S \) is the transition relation.

Throughout this paper we assume that any AMC is finite, i.e., has a finite number of states and is finitely branching. Transition \( s \xrightarrow{a,\lambda} s' \) denotes that the system can move from state \( s \) to \( s' \) while offering action \( a \) after a delay determined by an exponential distribution with rate \( \lambda \). We use the following notations:

\[
\begin{align*}
\mathbf{R}_A(s,s') &= \sum_{a \in A} \{ \lambda : s \xrightarrow{a,\lambda} s' \} \\
\mathbf{E}(s) &= \sum_{s' \in S} \mathbf{R}_{\text{Act}}(s,s') \\
\mathbf{P}_A(s,s') &= \mathbf{R}_A(s,s')/\mathbf{E}(s).
\end{align*}
\]

Stated in words, \( \mathbf{R}_A(s,s') \) denotes the cumulative rate of moving from state \( s \) to \( s' \) while offering some action from \( A \), \( \mathbf{E}(s) \) denotes the total rate with which some transition emanating from \( s \) is taken, and finally, \( \mathbf{P}_A(s,s') \) is the probability of moving from state \( s \) to \( s' \) by offering an action in \( A \). For absorbing states, \( \mathbf{E}(s) = 0 \) and \( \mathbf{P}_A(s,s') = 0 \) for any state \( s' \) and any set \( A \). Further note that \( \mathbf{R}_A(s,s') = \mathbf{P}_A(s,s') = 0 \) for any states \( s, s' \).

**Paths.** An infinite path \( \sigma \) is a sequence \( s_0 \xrightarrow{a_0,t_0} s_1 \xrightarrow{a_1,t_1} s_2 \xrightarrow{a_2,t_2} \ldots \) with for \( i \in \mathbb{N} \), \( s_i \in S \), \( a_i \in \text{Act} \) and \( t_i \in \mathbb{R}_{\geq 0} \) such that \( \mathbf{R}_{\text{Act}}(s_i, s_{i+1}) > 0 \). For \( i \in \mathbb{N} \) let \( \sigma[i] = s_i \), the \((i+1)\)-th state of \( \sigma \), and \( \delta(\sigma, i) = t_i \), the time spent in \( s_i \). For \( t \in \mathbb{R}_{\geq 0} \) and \( i \) the smallest index with \( t \leq \sum_{j=0}^{i-1} t_j \) let \( \sigma[0t] = \sigma[i] \), the state in \( \sigma \) at time \( t \).

A finite path \( \sigma \) is a sequence \( s_0 \xrightarrow{a_0,t_0} s_1 \xrightarrow{a_1,t_1} s_2 \ldots \xrightarrow{a_{l-1},t_{l-1}} s_l \) where \( s_i \) is absorbing, and \( \mathbf{R}(s_i, s_{i+1}) > 0 \) for all \( i < l \).

For finite \( \sigma, \sigma[i] \) and \( \delta(\sigma, i) \) are only defined for \( i \leq l \); they are defined as above for \( i < l \), and \( \delta(\sigma, l) = \infty \). For \( t > \sum_{j=0}^{l-1} t_j \) let \( \sigma[0t] = s_l \); otherwise, \( \sigma[0t] \) is as above.

We denote \( \sigma[i] \xrightarrow{\Delta} \sigma[i+1] \) whenever \( \sigma[i] \) can move to \( \sigma[i+1] \) by performing some action in \( A \), i.e., if \( a_i \in A \). Note that \( \sigma[i] \notin \mathcal{F} \). Let \( \text{Path}(s) \) denote the set of paths starting in \( s \). A Borel space over \( \text{Path}(s) \) can be defined in a similar way as in [5] and is omitted here.
3 An Action-Based Continuous Stochastic Logic

This section describes the action-based stochastic logic aCSL which is inspired by the action-based logic aCTL by De Nicola and Vaandrager [33] and the stochastic logic CSL by Baier et al. [5], which in turn is based on the work of Aziz et al. [2].

3.1 Syntax and Semantics of aCSL

Syntax. For \( p \in [0, 1] \) and \( \triangleleft \in \{ \leq, <, \geq, > \} \), the state-formulas of aCSL are defined by the grammar

\[
\Phi ::= \text{true} \mid \Phi \land \Phi \mid \neg \Phi \mid S_{\triangleleft \phi} (\Phi) \mid P_{\triangleleft \phi} (\varphi)
\]

where path-formulas are defined for \( t \in \mathbb{R}_{>0} \cup \{ \infty \} \) by

\[
\varphi ::= \Phi \_A U_t \Phi \mid \Phi \_A U_t B \Phi.
\]

Note that atomic propositions are absent. The boolean connectives such as \( \lor \) and \( \Rightarrow \) are derived in the obvious way. The probabilistic operator \( P_{\triangleleft \phi} (\cdot) \) replaces the CTL path quantifiers \( \exists \) and \( \forall \) that can be re-invented — up to fairness [6] — as the extremal probabilities \( P_{>0} (\cdot) \) and \( P_{\geq 1} (\cdot) \). The state formulas are directly adopted from CSL: \( S_{\triangleleft \phi} (\Phi) \) asserts that the steady-state probability for a \( \Phi \)-state meets the bound \( \triangleleft \phi \) and \( P_{\triangleleft \phi} (\varphi) \) asserts that the probability measure of the paths satisfying \( \varphi \) meets the bound \( \infty p \).

The path-formula \( \Phi_1 \_A U_t \Phi_2 \) is fulfilled by a path if a \( \Phi_2 \)-state is eventually reached via visiting only \( \Phi_1 \)-states before, while taking only \( A \)-transitions; besides, going from the beginning of the path until reaching the \( \Phi_2 \)-state should last at most \( t \) time units. The formula \( \Phi_1 \_A U_t B \Phi_2 \) requires in addition that (i) a move to a \( \Phi_2 \)-state is actually made and that (ii) this transition is labelled by some action in \( B \). We remark the following. Due to the fact that the \( \Phi_2 \)-state must be reached via a \( B \)-transition, the formula \( \Phi_1 \_A U_t B \Phi_2 \) is invalid in a \( (\neg \Phi_1 \land \Phi_2) \)-state \( s \); although the state satisfies \( \Phi_2 \), it is not able to move from a \( \Phi_1 \)-state to a \( \Phi_2 \)-state via a \( B \)-transition as it does not fulfill \( \Phi_1 \). The formula \( \Phi_1 \_A U_t \Phi_2 \) is, however, valid in state \( s \), since for the validity of this formula it is not required that a transition into a \( \Phi_2 \)-state is made. Thus, whereas for \( \Phi_1 \_A U_t \Phi_2 \) it suffices to currently be in a \( \Phi_2 \)-state, this is not the case for \( \Phi_1 \_A U_t B \Phi_2 \).3

---

3 If we enlarged the set of path-formulas such that conjunction and negation of path-formulas is allowed (in a similar way as for CTL*), the relationship between \( \_A U_t B \) and \( \_A U_t \), could be made precise as follows:

\[
\Phi_1 \_A U_t \Phi_2 = \Phi_2 \lor (\Phi_1 \_A U_t A \Phi_2).
\]
The major differences with a ‘standard’ until-formula $\Phi_1 U \Phi_2$ of linear and branching temporal logics are that restrictions are put on (i) the action labels of transitions to be taken and on (ii) the amount of time that is needed to reach a $\Phi_2$-state. This can be made precise in the following way:

$$\Phi_1 U \Phi_2 = \Phi_1 A_\tau U <\infty \Phi_2.$$  

In the sequel, we use $\Phi_1 A_\tau U B_\tau \Phi_2$ as an abbreviation of $\Phi_1 A_\tau U <\infty B_\tau \Phi_2$ and $\Phi_1 A_\tau U \Phi_2$ as an abbreviation of $\Phi_1 A_\tau U <\infty \Phi_2$. These are the untimed versions of the until-operators $A_\tau U <t B_\tau$ and $A_\tau U <t$.

An interesting aspect of $\mathbf{aCSL}$ is that the following set of next-operators are all derived operators:

- $X_A^{<t} \Phi = true_{\emptyset} U <t A_\tau \Phi$
- $X_A \Phi = X_A^{<\infty} \Phi$
- $X \Phi = X_{A_\tau} \Phi$

The formula $X_A^{<t} \Phi$ asserts that from the current state an $A$-transition can be made to a $\Phi$-state before time $t$. Remark that the $\Phi$-state must be reached by the first transition, as — due to the empty set of actions — further transitions are disallowed. $X_A$ is the action-labelled next-operator from $\mathbf{aCTL}$, whereas $X$ is the traditional state-based next-operator.

**Note 1.** In our logic, the next operator is derived from the until-operator. In $\mathbf{aCTL}$ the reverse holds [33]. This stems from the special treatment of internal, i.e., $\tau$-labelled, transitions in $\mathbf{aCTL}$. For instance in $\mathbf{aCTL}$, $X_\emptyset \Phi$ allows to reach a $\Phi$-state by an internal transition (but not any other). In our setting, internal transitions are treated as any other transition, and accordingly, $X_\emptyset \Phi$ is invalid for any state. We have made this difference deliberately: whereas $\mathbf{aCTL}$ is aimed to characterize branching bisimulation – a slight variant of weak bisimulation equivalence — we focus on characterizing a strong equivalence like lumping equivalence (since exact weak equivalences on AMCs cannot be obtained [21]).

The temporal operator $\Diamond$ and its variants are derived in the following way:

- $A_\Diamond^{<t} \Phi = true_{\tau} U <t A_\tau \Phi$
- $A_\Diamond \Phi = A_\Diamond^{<\infty} \Phi$
- $\Diamond^{<t} \Phi = A_\tau A_\Diamond^{<t} \Phi$

A path fulfills $A_\Diamond^{<t} \Phi$ if it reaches a $\Phi$-state within $t$ time units by only performing $A$-actions. Formulas $A_\Diamond \Phi$ and $\Diamond^{<t} \Phi$ denote the generalisations to infinite time and arbitrary actions. Their combination, $\Diamond \Phi$, corresponds to the well-known “eventually” operator. An even more discerning $\Diamond$-operator can be defined by

- $A_\Diamond^{<t} B_\Diamond \Phi = true_{A_\tau U <t B_\tau} \Phi$ and $A_\Diamond B_\Diamond \Phi = A_\Diamond^{<\infty} B_\Diamond \Phi$

Here, the path leading to the $\Phi$-state consists of an arbitrary number of $A$-actions, followed by a single $B$-action. Dual to these $\Diamond$-operators is the set of
\(\Box\)-operators, of which we only mention the following:

\[
P_{\Box p} (A^\llcorner t \Phi) = \neg P_{\Box p} (A^\llcorner \llcorner t \neg \Phi) \quad \text{and} \quad P_{\Box p} (A^\llcorner \llcorner t \Phi) = \neg P_{\Box p} (A^\llcorner \llcorner t \neg \Phi)
\]

with the obvious generalisations to infinite time and/or arbitrary sets of actions.

Finally, existential and universal quantification are introduced as

\[
\exists \varphi = P_{>0} (\varphi) \quad \text{and} \quad \forall \varphi = P_{\geq 1} (\varphi)
\]

Note that by this definition formula \(\forall \varphi\) holds even if there exists a path that does not satisfy \(\varphi\), if that path has zero probability mass.

We consider the modal operators from Hennessy-Milner logic [19] and the \(\mu\)-calculus [30] as derived operators. They are obtained as follows:

\[
\langle A \rangle \Phi = P_{>0} (X_A \Phi) \quad \text{and} \quad [A] \Phi = \neg \langle A \rangle \neg \Phi.
\]

The modal operator \(\langle A \rangle \Phi\) states that there is some \(A\)-transition from the current state to a \(\Phi\)-state, whereas \([A] \Phi\) states that for all \(A\)-transitions from the current state a \(\Phi\)-state is reached.

Note 2. The modal operator \(\langle a \rangle_p \Phi\) from the probabilistic modal logic PML [31] cannot be obtained as a derived operator in our setting. The state-formula \(\langle a \rangle_p \Phi\) asserts that, given that an \(a\)-transition happens, the probability of moving to a \(\Phi\)-state is at least \(p\). This interpretation fits well to the reactive probabilistic setting used in [31] in which over each set of equally labelled transitions a discrete probability space is defined. Since we consider a generative setting — having a discrete probability space over all, possibly different labelled, transitions — the probability in a formula like \(\langle a \rangle_p \Phi\) is relative to all transitions, and not just the ones labelled with \(a\). In the continuous variant of PML [8] a similar approach as in [31] is taken, and a reactive interpretation is used.

Semantics. The aCSL state-formulas are interpreted over the states of an AMC \((S,A,\rightarrow)\). Let \(Sat(\Phi) = \{ s \in S \mid s \models \Phi \}\).

\[
s \models \text{true} \quad \text{for all} \quad s \in S \quad s \models \Phi_1 \land \Phi_2 \quad \text{iff} \quad s \models \Phi_i, \quad \text{for} \quad i = 1, 2
\]

\[
s \models \neg \Phi \quad \text{iff} \quad s \not\models \Phi
\]

\[
s \models S_{\Box p} (\Phi) \quad \text{iff} \quad \pi(s,Sat(\Phi)) \ll p
\]

\[
s \models P_{\Box p} (\varphi) \quad \text{iff} \quad \text{Prob}(s,\varphi) \ll p
\]

Here, \(\pi(s,S')\) denotes the steady-state probability to be in a state of set \(S'\) when starting in \(s\). It is defined by means of a probability measure\(^4\) \(\Pr\) on the set of paths \(Path(s)\) emanating from \(s\).

\[
\pi(s,S') = \lim_{t \to \infty} \Pr\{ \sigma \in Path(s) \mid \sigma @ t \in S' \}
\]

\(\text{Prob}(s,\varphi)\) denotes the probability measure of all paths satisfying \(\varphi\) given that the system starts in state \(s\), i.e.,

\[
\text{Prob}(s,\varphi) = \Pr\{ \sigma \in Path(s) \mid \sigma \models \varphi \}.
\]

\(^4\) The probability measure \(\Pr\) is defined by means of a Borel space construction on paths. We refer to [5] for a formal definition.
The fact that these sets are measurable follows by easy verification from the Borel space construction given in [5].

The meaning of the path-operators is defined by a satisfaction relation, also denoted by $|=\,$, between a path and a path-formula. We define: $\sigma |= \Phi_1 A U^<t \Phi_2$ if and only if:

$$\exists k \geq 0. \left( \sigma[k] = \Phi_2 \right)$$

$$\land \left( \forall i < k. \sigma[i] = \Phi_1 \land \sigma[i] \xrightarrow{A} \sigma[i+1] \right) \land t > \sum_{i=0}^{k-1} \delta(\sigma, i) \right)$$

where we recall that $\delta(\sigma, i)$ denotes the sojourn time in state $\sigma[i]$. Thus, $\Phi_1 A U^<t \Phi_2$ is valid for a path if at some time instant before $t$ a $\Phi_2$-state is reached — assume this is the $(k+1)$-st state (for $k \geq 0$) in the path so far — by visiting only $\Phi_1$-states, while taking only $A$-transitions along the entire path.

For the other until-formula we have: $\sigma |= \Phi_1 A U^<t B \Phi_2$ if and only if:

$$\exists k > 0. \left( \sigma[k] = \Phi_2 \land \left( \forall i < k-1. \sigma[i] = \Phi_1 \land \sigma[i] \xrightarrow{A} \sigma[i+1] \right) \land \sigma[k-1] = \Phi_1 \land \sigma[k-1] \xrightarrow{B} \sigma[k] \land t > \sum_{i=0}^{k-1} \delta(\sigma, i) \right)$$

Note the subtle difference with (1): For $\Phi_1 A U^<t B \Phi_2$ to be valid, there should be a single transition leading to a $\Phi_2$-state labelled by some action in $B$.

It is left to the interested reader to check that $s |= X^<t_A \Phi$ iff

$$\sigma[1] = \Phi \land \sigma[0] \xrightarrow{A} \sigma[1] \land t > \delta(\sigma, 0).$$

This agrees with the intuitively expected semantics for $X^<t_A \Phi$.

3.2 Markovian Bisimulation

In this section, we show that aCSL is invariant under the application of Markovian bisimulation. Markovian bisimulation, a variant of Larsen-Skou bisimulation [31], is a congruence for the stochastic process algebras TIPP [16] and PEPA [29]. In the context of process algebraic composition operators, a congruence relation can be used to compress the state space of components before composition, in order to alleviate the state space explosion problem, under the condition that the relation equates only components obeying the same properties. Hence the question arises whether a Markovian bisimulation $R$ can be applied to compress models (or model components) prior to model checking aCSL-formulas. In general, this requires that the validity of aCSL-formulas is preserved when moving from an AMC $\mathcal{M}$ to its quotient AMC $\mathcal{M}/R$. We establish this property in Theorem 1.

Definition 2. A Markovian bisimulation on $\mathcal{M} = (S, A, \rightarrow)$ is an equivalence $R$ on $S$ such that whenever $(s, s') \in R$ then $R_A(s, C) = R_A(s', C)$ for all $C \in S/R$ and all $a \in \text{Act}$. States $s$ and $s'$ are Markovian bisimilar iff there exists a Markovian bisimulation $R$ that contains $(s, s')$. 
Here, $S/R$ denotes the quotient space and $R_a(s,C)$ abbreviates $\sum_{s' \in C} R_a(s,s')$. Let $M/R$ be the AMC that results from building the quotient space of $M$ under $R$, i.e., $M/R = (S/R,A,\rightarrow)$. In the following we write $|=M$ for the satisfaction relation $|=\text{aCSL}$ on $M$.

**Theorem 1.** Let $R$ be a Markovian bisimulation on $M$ and $s$ a state in $M$. Then:

(a) For all state-formulas $\Phi$: $s |\leq_M \Phi$ iff $[s]_R |\leq_{M/R} \Phi$

(b) For all path-formulas $\varphi$: $\text{Prob}^{M}(s,\varphi) = \text{Prob}^{M/R}([s]_R,\varphi)$.

In particular, Markovian bisimilar states satisfy the same aCSL formulas.

In the appendix, we sketch the proof of Theorem 1. The detailed proof can be found in [25]. This result allows to verify aCSL-formulas on the potentially much smaller AMC $M/R$ rather than on $M$. The quotient with respect to Markovian bisimilarity can be computed by a modified version of the partition refinement algorithm for ordinary bisimulation without an increase in complexity [26]. In addition, due to the congruence property of Markovian bisimilarity on TIPP and PEPA, a specification can be reduced in a compositional way, thus avoiding the need to model check the (possibly very large) state space $S$. This feature is exploited in the case study discussed in Section 5.

## 4 Model Checking aCSL

The general strategy for model checking aCSL proceeds in the standard way: For a given state formula $\Phi$, the algorithm recursively computes the sets of states satisfying the sub-formulas of $\Phi$, and constructs from them the set of states satisfying $\Phi$. For boolean connectives, the strategy is obvious. Model checking steady-state properties $S_{\text{cap}}(\varphi)$ involves solving linear systems of equations, after determining (bottom) strongly connected components, exactly as in the CSL context [5].

Model checking the probabilistic quantifier $\mathcal{P}_{\text{cap}}(\varphi)$ is the crucial difficulty. It relies on the following characterizations of $\text{Prob}(s,\varphi)$. We discuss the characterizations by structural induction over $\varphi$. For the sake of simplicity, we first treat the simple untimed until-formulas.

**Untimed until.** For $\varphi = \Phi_1 A_\mathcal{U} \Phi_2$ we have that $\text{Prob}(s,\varphi)$ is given by the following equations: $\text{Prob}(s,\varphi) = 1$ if $s \models \Phi_2$, 

$$\sum_{s' \in S} P_A(s,s') \cdot \text{Prob}(s',\varphi)$$

if $s \models \Phi_1 \land \neg \Phi_2$, and 0 otherwise. For $A = \text{Act}$ we obtain the equation for standard until as for DTMCs [17].

Let $\varphi = \Phi_1 A_\mathcal{U} \Phi_2$. For $s \not\models \Phi_1$, the formula is invalid. As for $s \models \Phi_1$, the situation is more involved let us, for the sake of simplicity, assume that $A$ and
Towards Model Checking Stochastic Process Algebra 429

B are disjoint, i.e. \( A \cap B = \emptyset \). Then the only interesting possibilities starting from \( s \) are (i) to directly move to a \( \Phi_2 \)-state via a \( B \)-transition, in which case the formula \( \varphi \) is satisfied with probability 1, or (ii) to take an \( A \)-transition leading to \( \Phi_1 \)-state \( s' \) which satisfies \( \varphi \) with probability \( \text{Prob}(s', \varphi) \). Accordingly, for \( A \cap B = \emptyset \), \( \text{Prob}(s, \varphi) \) can be characterized by:

\[
\sum_{s' \models \Phi_2} P_B(s, s') + \sum_{s' \models \Phi_1} P_A(s, s') \cdot \text{Prob}(s', \varphi).
\]

(2)

In the general case we have to take into account that \( A \) and \( B \) may not be disjoint. Equation (2) does not apply now, since an \( (A \cap B) \)-transition into a state that satisfies both \( \Phi_1 \) and \( \Phi_2 \) is "counted" twice. We therefore obtain that \( \text{Prob}(s, \varphi) \) is the least solution of the following set of equations:

\[
\sum_{s' \models \Phi_2} P_B(s, s') + \sum_{s' \models \Phi_1} P_A(s, s') \cdot \text{Prob}(s', \varphi) - \sum_{s' \models \Phi_1 \land \Phi_2} P_A \cap B(s, s') \cdot \text{Prob}(s', \varphi)
\]

if \( s \models \Phi_1 \), and 0 otherwise. Note that

\[
\text{Prob}(s, X_B \Phi) = \text{Prob}(s, \text{true} \not\in \mathcal{U}_B \Phi) = \sum_{s' \models \Phi} P_B(s, s')
\]

which coincides, for \( B = \text{Act} \), with the characterization of next for DTMCs [17]. Thus, the probability that \( s \) satisfies \( X_B \Phi \) equals the sum of the probabilities to move to a \( \Phi \)-state via a single \( B \)-transition. The reader is also invited to check that for \( B = \emptyset \) there is no state that satisfies \( \Phi_1 \land \mathcal{U}_B \Phi_2 \) with positive probability.

Timed until. For \( \varphi = \Phi_1 \mathcal{U}^< t \Phi_2 \) we have that \( \text{Prob}(s, \varphi) \) is the least solution of the following set of equations: \( \text{Prob}(s, \varphi) = 1 \) if \( s \models \Phi_2 \), and

\[
\int_0^t e^{-E(s) \cdot x} \sum_{s' \in S} R_A(s, s') \cdot \text{Prob}(s', \Phi_1 \mathcal{U}^< t-x \Phi_2) \ dx
\]

if \( s \models \Phi_1 \land \neg \Phi_2 \), and 0 otherwise. For state \( s \) satisfying \( \Phi_1 \land \neg \Phi_2 \), the probability of reaching a \( \Phi_2 \)-state within \( t \) time units from \( s \) equals the probability of reaching some direct successor \( s' \) of \( s \) within \( x \) time units, multiplied by the probability of reaching a \( \Phi_2 \)-state from \( s' \) within the remaining time \( t-x \). Since there may be different paths from \( s \) to \( \Phi_2 \)-states, the sum is taken over all these possibilities. (Note that by taking \( t = \infty \) we obtain, after some straight-forward calculations, the characterisation for untimed until \( \mathcal{U} \) given before).

For \( \varphi = \Phi_1 \mathcal{U}^< t - B \Phi_2 \) we have that \( \text{Prob}(s, \varphi) \) is the least solution of the following set of equations:

\[
\int_0^t e^{-E(s) \cdot x} \left( \sum_{s' \models \Phi_2} R_B(s, s') + \sum_{s' \models \Phi_1} R_A(s, s') \cdot \text{Prob}(s', \Phi_1 \mathcal{U}^< t-x - B \Phi_2) \right. \\
\left. - \sum_{s' \models \Phi_1 \land \Phi_2} R_A \cap B(s, s') \cdot \text{Prob}(s', \Phi_1 \mathcal{U}^< t-x \Phi_2) \right) \ dx
\]
if $s \models \Phi_1$, and 0 otherwise. This characterization can be justified in the same way as for its untimed counterpart, i.e., $\Phi_1 \land U_B \Phi_2$, given the above explanation for the simpler timed until variant. Let us consider what this yields for $X^< t \Phi$:

$$\text{Prob}(s, X^< t \Phi) = \text{Prob}(s, \text{true} \land U^< t \Phi) = \int_0^t e^{-E(s) \cdot x} \cdot \sum_{s' \models \Phi} R_B(s, s') \, dx$$

which, after some straight-forward calculations, leads to

$$\sum_{s' \models \Phi} P_B(s, s') \cdot \left(1 - e^{-E(s) \cdot t}\right).$$

The first term of the product denotes the discrete probability to move via a single $B$-transition to a $\Phi$-state, whereas the second term denotes the probability to leave state $s$ within $t$ time units.

This equational characterization allows one to model check $\mathcal{aCSL}$ formulas by means of approximate numerical techniques. The concrete implementation closely follows the one for $\mathcal{CSL}$ outlined in [5] and implemented in [24]. We are currently investigating whether the solution of the above integral equations can be reduced to standard transient analysis via uniformisation, as in [3].

5 Case Study

We consider a multiprocessor mainframe which was first introduced in [27] and has since then served as a standard SPA example, see e.g. [23,11]. Here we only briefly repeat the main features of the model.

5.1 Specification of Multiprocessor Mainframe

The multiprocessor mainframe serves two purposes: It has to process database transactions submitted by users, and it provides computing capacity to programmers maintaining the database. The system is subject to software failures which are modelled as special jobs. On the top level, the system is composed of two processes (cf. Fig. 1).

$$\text{System} := \text{Load} | [\text{putUserJob}, \text{putProgJob}, \text{fail}] | \text{Machine}$$

Process $\text{Load}$ represents the system load caused by the database users, the programmers and the failures. The mainframe itself is modelled by the $\text{Machine}$ process. The three different system load components are modelled as so-called Markov modulated Poisson processes, see [27]. The intensity of the load alters between different levels. To realize a synchronous change of load level, a synchronizing action $c$ is used.

$$\text{Load} := \text{ProgLoad} | [c] | \text{UserLoad} | [c] | \text{FailLoad}$$
The Machine component consists of two finite queues and four identical processors. The queues buffer incoming jobs. They are controlled by a priority mechanism to ensure that programmer jobs have the lowest priority, while failures have the highest priority. The priority mechanism is realised by appropriate synchronisation of the queue processes. For instance, process $Q$ can only deliver a job to a processor if queue $R$ is empty and no failure is present. Furthermore, the insertion of new jobs into the system is prohibited once a failure has occurred, until the system is repaired.

Each of the four processors executes user or programmer jobs waiting in the respective queues, unless a failure occurs. As failures have preemptive priority over the other two job classes, all processors stop working once action $fail$ has occurred and then wait until the system will recover (via action $repair$).

5.2 Properties of Interest

This section contains some example properties which are of interest for the multiprocessor mainframe model. For each property, a description in plain English, its aCSL formulation and some explanation are given. We first introduce some purely functional requirements to ensure that the priority mechanism is properly realised by the model. Then we turn to the performance and reliability requirements which the system should satisfy. For $A \subseteq \text{Act}$ we let $\overline{A}$ denote $\text{Act} \setminus A$. We use the following sets of actions: Get := \{getUserJob, getProgJob\}, Put := \{putUserJob, putProgJob\}, Fin := \{finishUserJob, finishProgJob\} and FailRep := \{fail, repair\}. We omit brackets for singleton sets.

$\Phi_1$: If there are user jobs waiting, the processors will not start programmer jobs.

$\Phi_{UserJobWaiting} \Rightarrow \neg (getProgJob \text{ true})$

where $\Phi_{UserJobWaiting}$ is defined by $\exists (putUserJob \odot getUserJob \text{ true})$, characterizing at least one user job waiting in the queue.
\( \Phi_2 \): Whenever a failure occurs, no jobs can be inserted into the queues until the system is repaired.

\[
\text{[fail]} \lor (\text{repair} \lor \text{true})
\]

\( \Phi_3 \): Whenever a failure occurs, the processors will be blocked until the system is repaired.

\[
\text{[fail]} \lor (\text{Get} \cup \text{Fin} \lor \text{repair} \lor \text{true})
\]

\( \Phi_4 \): After a repair, both queues are empty.

\[
\text{[repair]} \ (\neg \Phi_{\text{UserJobWaiting}} \land \neg \Phi_{\text{ProgJobWaiting}})
\]

where \( \Phi_{\text{ProgJobWaiting}} \) characterizes at least one waiting programming job, defined in a similar way as \( \Phi_{\text{UserJobWaiting}} \). This is an example of a property which is not true, since a failure does not cause the queues to be flushed.

\( \Phi_5 \): In steady state, the probability of low priority programming jobs having to wait because of user jobs being served is smaller than 0.01.

\[
S_{<0.01} ([\text{finishUserJob}] \land \Phi_{\text{ProgJobWaiting}})
\]

\( \Phi_6 \): At least two processors are occupied by user jobs.

\[
([\text{finishUserJob}] \land [\text{finishUserJob}] \land \text{true})
\]

\( \Phi_7 \): In steady state, the probability that at least two processors are occupied by user jobs is greater than 0.002.

\[
S_{>0.002} (\Phi_6)
\]

\( \Phi_8 \): There is at least a 30% chance that some job will be finished within at most 4 time units.

\[
P_{>0.3} ([\text{atRep}] \lor [\text{true}])
\]

\( \Phi_9 \): In steady state, the probability of the system being unavailable (i.e. waiting for repair) is at most 0.05.

\[
S_{\leq0.05} ([\text{FailRep}] \lor [\text{repair} \lor \text{true}])
\]

\( \Phi_{10} \): After a system failure, there is a chance of more than 90% that it will come up again within the next 5 time units.

\[
\text{[fail]} \ P_{>0.9} ([\text{repair} \land [\text{repair} \lor \text{true}])
\]

The fact that the above property holds for all states can be expressed by \( \forall \ \Box \ \Phi_{10} \). Slightly weaker, one might require the above property to hold on the long run, formulated as \( S_{\geq1} (\Phi_{10}) \).
Table 1. Verification runtimes

<table>
<thead>
<tr>
<th>property</th>
<th>verification runtimes (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Φ₁</td>
<td>0.012 0.037 0.268</td>
</tr>
<tr>
<td>Φ₂</td>
<td>0.008 0.049 0.864</td>
</tr>
<tr>
<td>Φ₃</td>
<td>0.008 0.039 0.319</td>
</tr>
<tr>
<td>Φ₄</td>
<td>0.003 0.005 0.036</td>
</tr>
<tr>
<td>Φ₅</td>
<td>0.642 2.371 18.750</td>
</tr>
<tr>
<td>Φ₆</td>
<td>0.001 0.002 0.014</td>
</tr>
<tr>
<td>Φ₇</td>
<td>0.558 2.122 18.814</td>
</tr>
<tr>
<td>Φ₈</td>
<td>0.554 2.009 18.819</td>
</tr>
<tr>
<td>Φ₁₀</td>
<td>2.557 11.404 92.324</td>
</tr>
</tbody>
</table>

5.3 Verification Results

In this section we report on our experience with the verification of the above properties. The results have been obtained by means of a trial implementation, basically an extension of the model checker EIMC² [24]. The implementation does not yet support the full logic aCSL, therefore property Φ₈ has not been checked.

For the properties listed in the previous section we present the verification runtimes in Table 1. We checked three models: A small model with 4 (2) programmer (user) buffer places, an intermediate model with 10 (4) programmer (user) buffer places and a large model with 40 (10) programmer (user) buffer places. The small model has 3690 states and 24009 transitions, the intermediate model has 13530 states and 91069 transitions and the large model has 110946 states and 761989 transitions. However, we did not perform model checking on the original models but on models with compressed state spaces which we gained through the application of Markovian bisimilarity (in the example multiprocessor system, the main potential for reduction stems from the symmetry of the four identical processors). By Theorem 1, the compressed models satisfy the same properties as the original ones. After bisimilarity compression, the small model has 720 states and 3219 transitions, the intermediate model has 2640 states and 12295 transitions and the large model has 21648 states and 103471 transitions. All steady state properties given in the table were double checked with TIPPTOOL [22].
6 On Translating aCSL to CSL

The design of aCSL closely follows the work of De Nicola and Vaandrager on aCTL [33]. For what concerns model checking, they propose a translation $K$ from aCTL into CTL, and a transformation (also denoted $K$) from action-labelled to state-labelled transition systems in such a way that for an arbitrary aCTL formula $\Phi$ and arbitrary action-labelled transition system $M$ (with the obvious notation):

$$ s \models_{M,aCTL} \Phi \text{ iff } K(s) \models_{K(M),CTL} K(\Phi) \quad (3) $$

In this way, aCTL model checking can be reduced to CTL model checking, by checking a translated formula $K(\Phi)$ on a transformed model $K(M)$. The bypass via $K$ blows up the model and the formula, but only by a factor of 2, whence it follows that model checking aCTL has the same worst case (space and time) complexity as CTL. The key idea of this transformation is to break each transition of $M$ in two, connected by a new auxiliary state. The new state is labelled with the action label of the original transition, playing the role of an atomic state proposition. (The original source and target states are labelled with a distinguished symbol $\bot$). Formula $\Phi$ is manipulated by $K$ in such a way that starting from some state $K(s)$ essentially all the labellings of original states ($\bot$) do not matter, while the ones of auxiliary states do. Unfortunately, this approach does not carry over to the Markov chain setting, because splitting a Markovian transition in two implies splitting an exponential distribution in two. However, no sequence of two exponential distribution agrees with an exponential distribution. Since aCSL is powerful enough to detect differences in transient probabilities, this approach is infeasible.

Even though a translation in the style of De Nicola and Vaandrager does not allow one to reduce aCSL to CSL, this does not imply that such a reduction is generally infeasible. For the sake of completeness, we remark that it is indeed possible to reduce model checking aCSL to model checking (slight variants of) CSL. We briefly sketch two possibilities:

- Apply the transformation of [33] and map AMCs to interactive Markov chains (IMC) [20]. This transformation is exemplified in Figure 2 (from left to middle), where state labellings appear as sets, and dashed transitions are supposed to be immediate. In general, IMC allow for nondeterminism, but this phenomenon is not introduced by the translation. Therefore, the model checking algorithm of [5] can be lifted to this subset of IMC.

- Transform AMCs to state-labelled CTMCs (SMC), using a transformation inspired by Emerson and Lei [13]. The main idea is to split each state into a number of duplicates, given by the number of different incoming actions it possesses, and label each duplicate with a different action, and distribute the incoming transitions accordingly. (In order to track the first transition delay correctly, one additional $\bot$-labelled duplicate per state is needed.) To give an intuitive idea, this transformation is depicted in Figure 2 (from left to right). A mapping $K$ from aCSL to a minor variant of CSL exists that
Towards Model Checking Stochastic Process Algebra

Fig. 2. Transformation example from AMC (left) to IMC (middle) and SMC (right)

eff\satisfaction relation $s \models_{\mathcal{M}} \varphi$ holds if and only if $(s, \bot) \models_{\mathcal{K}(\mathcal{M})} \mathcal{K}(\varphi)$ holds. (The satisfaction relation $\models_{\mathcal{K}(\mathcal{M})}$ on CSL requires a subtle – but straightforward to implement – modification.) Details can be found in [25]. In the worst case, the state space is blown up by a factor given by the maximal number of distinct actions entering a state.

Notice that both translations sketched above require a small modification of the model checking algorithm for CSL [3,5]. Furthermore, both approaches induce a blow up of the model by a linear factor. To avoid these drawbacks, we have decided to develop a direct model checking algorithm, as sketched in Section 4. Remark that despite the aforementioned translations from aCTL to CTL, dedicated model checkers for aCTL are more popular by now [14,32].

7 Concluding Remarks

This paper has introduced a behaviour-oriented analysis approach for Markovian stochastic process algebra. From a conceptual as well as from a pragmatic point of view, this approach closes a disturbing gap in the process algebraic approach to performance and dependability modelling. In particular, performance engineers are no longer confronted with the need to switch from a behaviour-oriented to a state-oriented view when it comes to model analysis.

The behaviour-oriented modelling and analysis approach outlined in this paper has four ingredients: (1) A standard stochastic process algebra (such as TIPP, PEPA, EMPA) is used to model the system under consideration as an action-labelled CTMC. (2) The action-based logic aCSL serves as a powerful means to specify properties of interest. (3) A model checking algorithm decides which properties are satisfied by the Markov chain model. (4) Since Markovian bisimilarity preserves aCSL properties, it can be used to compress the model (or the model components, due to the congruence property for TIPP and PEPA)
before model checking. We have illustrated all four ingredients by means of the multiprocessor mainframe case study.

**PML** \(\mu\) [8], the continuous-time variant of **PML** [31], is another logic on action-labelled CTMCs. **PML** \(\mu\) and **aCSL** are incomparable, because **PML** \(\mu\) takes a reactive point of view, while our view is generative (see Note 2). **PML** \(\mu\) is not considered in the context of model checking, instead it serves as the foundation of a formalism to assign rewards to states, i.e., to construct Markov reward models. The thus obtained models are then analyzed with standard (steady-state) numerical analysis. **PML** \(\mu\) does neither provide means to quantify probability nor to reason about time intervals.

For the future, we intend to study to what extent **aCSL** can be extended towards the analysis of Markov reward models. In the state-based setting, we have recently developed a continuous reward logic (**CRL**) that allows bounds on rewards to be checked, and naturally combines with **CSL** [4].

Acknowledgements. The authors are grateful to Lennard Kerber, Frits Vaandrager and Moshe Vardi for highly valuable discussions. Joachim Meyer-Kayser is supported by the German Research Council DFG under HE 1408/6-1.

References


A Appendix: Sketch of the Proof of Theorem 1

In order to verify Theorem 1(a), we prove that \((u, v) \in R\) implies \(\forall \phi. (u \models_M \phi \iff v \models_M \phi)\). We do so by structural induction on \(\phi\). The only non-trivial cases are that \(\phi\) is of the form \(S_{\text{cop}}(\Psi)\), or \(P_{\text{cop}}(\varphi)\). In the former case, \(S_{\text{cop}}(\Psi)\), we use the induction hypothesis, the fact that Markovian bisimulation implies lumpability, and that lumpability ensures that steady-state probabilities can be obtained from the lumped quotient Markov chain [21]. In the latter case, \(P_{\text{cop}}(\varphi)\), we can apply Theorem 1(b), together with the induction hypothesis. So, only Theorem 1(b) remains to be verified. For this purpose, it is sufficient to show that \((u, v) \in R\) implies

\[
\Pr\{\sigma \in \text{Path}(u) \mid \sigma \models \varphi\} = \Pr\{\sigma \in \text{Path}(v) \mid \sigma \models \varphi\}
\]

We have to distinguish two cases, \(\varphi = \Phi_1 A U^t B \Phi_2\) and \(\varphi = \Phi_1 A U^t B\Phi_2\). Only the first of them is elaborated below. The other case proceeds in a similar, but simpler, way. For \(n \geq 1\) and \(t > 0\) we define the set of paths \(A_n^u(t)\) as

\[
A_n^u(t) = \{ \sigma \in \text{Path}(u) \mid \sigma[n] \models \Phi_2 \\
\land \forall 0 \leq i < n, \sigma[i] \models \Phi_1 \\
\land \forall 0 \leq i < n - 1, \sigma[i] \to \sigma[i + 1] \\
\land \sigma[n - 1] \not\to \sigma[n] \\
\land \sum_{i=0}^{\leq n} \delta(\sigma, i) < t \}
\]

(observe the similarity to the semantics of \(\Phi_1 A U^t B \Phi_2\) and the set of paths \(B_i^u(t)\) for \(i \geq 1\) by \(B_i^u(t) = A_i^u(t)\), and \(B_{n+1}^u(t) = A_{n+1}^u(t) \setminus \bigcup_{i=1}^{n} B_i^u(t)\). Intuitively, \(A_n^u(t)\) is the set of paths starting in \(u\) and reaching a \(\Phi_2\)-state within \(t\) time units in \(n\) steps, where the first \(n - 1\) steps are \(A\)-transitions and the last step is a \(B\)-transition. \(B_{n+1}^u(t)\) denotes the subset of \(A_{n+1}^u(t)\) consisting of paths that reach a \(\Phi_2\)-state in \(n\) steps without performing an \(A \land B\)-transition to a \(\Phi_2\)-state in the previous steps.

Note that \(B_i^u(t), B_i^u(t)\) are pairwise disjoint (for \(i \neq j\)). By exploiting the fact that \(\{ \sigma \in \text{Path}(u) \mid \sigma \models \Phi_1 A U^t B \Phi_2\} = \bigcup_{i=1}^{n} B_i^u(t)\), we obtain:

\[
\Pr\{\sigma \in \text{Path}(u) \mid \sigma \models \Phi_1 A U^t B \Phi_2\} = \sum_{i=1}^{\infty} \Pr\{\sigma \in B_i^u(t)\}.
\]

Hence, it is sufficient to show that for arbitrary \(t > 0\),

\[
\sum_{i=1}^{\infty} \Pr\{\sigma \in B_i^u(t)\} = \sum_{i=1}^{\infty} \Pr\{\sigma \in B_i^u(t)\}.
\]
Towards Model Checking Stochastic Process Algebra 439

We fix some \( t > 0 \), and prove the above by showing the stronger property that for all positive \( n \), \( \Pr\{ \sigma \in B_n^u(t) \} = \Pr\{ \sigma \in B_n^v(t) \} \). This proof proceeds by induction on \( n \), the length of the paths in \( B_n^u(t) \) and \( B_n^v(t) \). So, we perform a nested induction, the (inner) induction on \( n \) is nested in the (outer) induction on the structure of the formula \( \Phi \).

In the base case \( n = 1 \) of the inner induction, let us first assume \( u \not\models \Phi_1 \). But then \( v \not\models \Phi_1 \) (by the outer induction hypothesis) and hence \( \Pr\{ \sigma \in B_1^u(t) \} = 0 = \Pr\{ \sigma \in B_1^v(t) \} \). If, conversely, \( u \models \Phi_1 \), we obtain \( v \models \Phi_1 \) by the outer induction hypothesis, and therefore

\[
\Pr\{ \sigma \in B_1^u(t) \} = \sum_{w \in \Phi_2} P_B(u, w) \cdot (1 - e^{-E(u) \cdot t})
\]

\[
\sum_{C \in M|_R, C \models \Phi_2} \sum_{w \in C} P_B(u, w) \cdot (1 - e^{-E(u) \cdot t}) \quad (\ast)
\]

\[
\sum_{C \in M|_R, C \models \Phi_2} \sum_{w \in C} P_B(v, w) \cdot (1 - e^{-E(v) \cdot t}) = \Pr\{ \sigma \in B_1^v(t) \}
\]

Here, \( C \models \Phi_2 \) denotes that all states in the equivalence class \( C \) satisfy \( \Phi_2 \), which we can assume by the outer induction hypothesis. The transformation labelled \((\ast)\) uses that \( (u, v) \in R \) implies \( \sum_{w \in C} P_A(u, w) = \sum_{w \in C} P_A(v, w) \), since \( C \) is the class of a Markovian bisimulation.

To complete the inner induction we now assume that for arbitrary \( n > 1 \) we have that \( \Pr\{ \sigma \in B_n^u(t) \} = \Pr\{ \sigma \in B_n^v(t) \} \), and aim to show that this also holds for \( n + 1 \). The case \( u \not\models \Phi_1 \) proceeds as above. The remaining case, \( u \models \Phi_1 \), leads to the following transformation, using the same arguments as above.

\[
\Pr\{ \sigma \in B_{n+1}^u(t) \} = \int_0^t e^{-E(u) \cdot x} \sum_{w \models \Phi_1} R_A(u, w) \cdot \Pr\{ \sigma \in B_n^w(t - x) \} \, dx
\]

\[
\int_0^t e^{-E(v) \cdot x} \sum_{C \in M|_R, C \models \Phi_1} \sum_{w \in C} R_A(u, w) \cdot \Pr\{ \sigma \models B_n^w(t - x) \} \, dx \quad (\ast)
\]

\[
\int_0^t e^{-E(v) \cdot x} \sum_{C \in M|_R, C \models \Phi_1} \sum_{w \in C} R_A(v, w) \cdot \Pr\{ \sigma \models B_n^w(t - x) \} \, dx
\]

\[
\Pr\{ \sigma \in B_{n+1}^v(t) \}
\]

This completes the proof sketch, details can be found in [25].
Author Index

Ait-Ameur, Yamine 136
Baumeister, Hubert 3
Bert, Didier 235
Blair, Lynne 176
Bolton, Christie 77
du Bousquet, Lydie 338
Bowen, Jonathan P. 277
Bowman, Howard 176
Bryans, Jeremy 176
Butler, Michael 61
Cansell, Dominique 380
Cave, Francis 235
Chen, Yifeng 299
Conrad, Stefan 45
Davies, Jim 77
Derrick, John 176, 194
Dingel, Jürgen 214
Ferreira, Carla 61
Fischer, Jörg 45
Haxthausen, Anne Elisabeth 25
Hayes, Ian 97
Hermanns, Holger 420
Hoare, Tony 1
Jéron, Thierry 338
Jonkers, Hans B.M. 116
Jullian, Jacques 398
Katoen, Joost-Pieter 420
Klaudel, Hanna 318
Masson, Pierre-Alain 398
Méry, Dominique 380
Menz, Stephan 380
Meyer, Eric 358
Meyer-Kayser, Joachim 420
Mountassir, Hassan 398
Petre, Luigia 156
Pommereau, Franck 318
Reeve, Greg 255
Reeves, Steve 255
Rusu, Vlad 338
Santen, Thomas 358
Schulte, Wolfram 297
Sere, Kaisa 156
Siegle, Markus 420
Smith, Graeme 97, 194
Xia, Yong 25
Zamulin, Alexandre 3