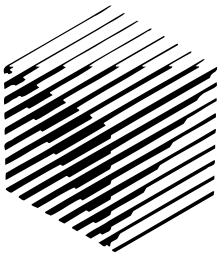

GMD Report
91

GMD –
Forschungszentrum
Informationstechnik
GmbH

European Research Consortium
for Informatics and Mathematics
ERCIM



Stefania Gnesi, Ina Schieferdecker,
Axel Rennoch (Eds.)

# 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems

Proceedings of FMICS'2000

April 3-4, 2000 in Berlin

April 2000

GMD –
Forschungszentrum Informationstechnik GmbH
Schloß Birlinghoven
D-53754 Sankt Augustin
Germany
Telefon +49 -2241 -14 -0
Telefax +49 -2241 -14 -2618
http://www.gmd.de



Anschriften der Herausgeber/Addresses of the editors:
Dr. Stefania Gnesi
Istituto di Elaborazione della Informazione
CNR - Consiglio Nazionale delle Ricerche
Area della Ricerca di Pisa
Via Alfieri, 1
I-56010 Ghezzano - Pisa
E-mail: gnesi@iei.pi.cnr.it

Dr. Ina Schieferdecker
Axel Rennoch
Institut für Offene Kommunikationssysteme
GMD – Forschungszentrum Informationstechnik GmbH
Kaiserin-Augusta-Allee 31
D-10589 Berlin
E-mail: {schieferdecker, rennoch}@fokus.gmd.de

ISSN 1435-2702

# Preface

The European Research Consortium for Informatics and Mathematics (ERCIM) has recently celebrated its 10th anniversary. The ERCIM Working Group on Formal Methods for Industrial Critical Systems (FMICS) is organizing its 5th International Workshop. FMICS workshops are dedicated to interested researchers at ERCIM sites, universities and industry active in the industrial application of formal methods. Among a variety of formal methods conferences and workshops FMICS is increasing its popularity. The idea of FMICS workshops is to attract people with industrial relevant topics, with internationally well-known invited speakers and with high-quality technical papers in combination with a discussion podium for the exchange of ideas. The workshop character of FMICS is realized on a minimal cost base.This time, FMICS is organized right after ETAPS'2000 - the European Joint Conferences on Theory and Practice of Software in Berlin.

After starting the FMICS workshop series 1996 in Oxford (UK) further workshops followed 1997 in Cesena (I), 1998 in Amsterdam (NL) and 1999 in Trento (I). In 2000, the workshop is hosted and organized at the GMD Research Institute for Open Communication Systems (FOKUS) in Berlin, Germany.

This year' workshop includes sessions on modelling, verification, testing and software development, MSC/SDL, and various applications and case studies. We are pleased to present two interesting invited talks: Günter Karjoth, IBM Zurich (CH), addresses the value of formal methods for security properties such as confidentiality and authenticity. Holger Hermanns, University of Twente (NL), investigates in the performance and reliability model checking and construction.

We wish to thank the members of the programme committee, especially the FMICS working group chairman Hubert Garavel, for the excellent assistance during the planing of the workshop, the invited speakers, the authors and the reviewers for their scientific contributions, the people from the GMD Fokus Competence Center TIP for preparing the workshop event, and ERCIM and GMD for their financial and organizational support of FMICS.

Berlin, April 2000

Stefania Gnesi, Ina Schieferdecker, Axel Rennoch

**Keywords:** Formal Methods, Formal Description Techniques (FDT), Modelling, Specification, Verification, Prototyping, Testing, Software development, Industrial applications.

**Further information:** FMICS homepage http://www.inrialpes.fr/vasy/fmics/

# Vorwort

Das Europäische Forschungskonsortium für Informatik und Mathematik (ERCIM) hat gerade sein 10jähriges Jubiläum gefeiert und die ERCIM Arbeitsgruppe zu Formalen Methoden für Industrie-kritische Systeme (FMICS) organisiert bereits ihren fünften internationalen Workshop. FMICS Workshops wenden sich an interessierte Forscher aus ERCIM Instituten, Universitäten und der Industrie, die sich aktiv an der Anwendung formaler Methoden für industrielle Anwendungen beteiligen. Trotz der Vielzahl von Konferenzen und Workshops über formale Methoden erfreut sich FMICS wachsender Beliebtheit. Es ist der Gedanke von FMICS Workshops die Fachleute mit industrie-relevanten Themen anzusprechen, mit international anerkannten eingeladenen Vortragenden und mit hochqualifizierten technischen Beiträgen in Kombination mit einem Forum für den Austausch von Ideen. Der Workshop Charakter von FMICS wird auf der Basis niedriger Kosten durchgeführt. Diesmal wird FMICS direkt im Anschluß an ETAPS'2000 - den Europäischen Konferenzen für Softwaretheorie und -praxis in Berlin organisiert.

Nach dem Start der FMICS Workshops 1996 in Oxford (UK) folgten Workshops 1997 in Cesena (I), 1998 in Amsterdam (NL) und 1999 in Trento (I). Im Jahr 2000 findet der Workshop beim GMD Forschungsinstitut für Offene Kommunikationssysteme (FOKUS) in Berlin statt.

Der diesjährige Workshop umfaßt die Themengebiete Modelling, Verification, Testing und Software Entwicklung, MSC/SDL, sowie vielfältige Anwendungen und Fallstudien. Wir freuen uns sehr zwei interessante eingeladene Vorträge zu präsentieren: Günter Karjoth, IBM Zürich (CH), erörtert den Wert formaler Methoden für Sicherheitsaspekte wie Vertraulichkeit und Authentizität. Holger Hermanns, Universität Twente (NL), geht ein auf die Konstruktion und Überprüfung von Leistungs- und Zuverlässigkeitsmodellen.

Abschießend möchten wir den Mitgliedern des Programmkommitees danken, insbesondere dem Vorsitzenden der FMICS Arbeitsgruppe Hubert Garavel, für die hervorragende Unterstützung bei der Vorbereitung des Workshops, außerdem den eingeladenen Vortragenden, den Autoren der Beiträge und den Gutachtern für ihre wissenschaftlichen Beiträge, den Mitarbeitern des GMD Fokus Competence Center TIP bei der Ausrichtung des Workshops, sowie ERCIM und der GMD für ihre finanzielle und organisatorische Unterstützung von FMICS.

Berlin, April 2000

Stefania Gnesi, Ina Schieferdecker, Axel Rennoch

**Schlagworte:** Formal Methods, Formal Description Techniques (FDT), Modelling, Specification, Verification, Prototyping, Testing, Software development, Industrial applications.

**Weitere Informationen:** FMICS homepage http://www.inrialpes.fr/vasy/fmics/

# Programme Committee

Juan Bicarregui (CLRC Abington, UK)
Lars-åke Fredlund (SICS Stockholm, S)
Hubert Garavel (INRIA Rhone-Alpes, F), FMICS chair
Stefania Gnesi (CNR/IEI Pisa, I), PC co-chair
Jan Frisco Groote (CWI Amsterdam, NL)
Diego Latella (CNR/CNUCE Pisa, I)
Axel Poigné (GMD/AiS Birlinghofen, D)
Ina Schieferdecker (GMD/Fokus Berlin, D), PC co-chair
Jan Tretmans (University of Twente, NL)
Ulrich Ultes-Nitsche (University of Southampton, UK)
Adam Wolisz (TU Berlin, D)

# List of Reviewers

Axel Belinfante, Pierfrancesco Bellini, Juan Bicarregui, Michael J. Butler, Gennady Chugunov, Alessandro Fantechi, Lars-åke Fredlund, Hubert Garavel, Pablo Giambiagi, Stefania Gnesi, Jan Friso Groote, Dilian Gurov, Izak van Langevelde, Diego Latella, Gabriele Lenzini, Mang Li, Giuseppe Manco, Andrew Martin, Mieke Massink, Radu Mateescu, Brian M. Matthews, Franco Mazzanti, Thomas Noll, Axel Poigné, Jaco van de Pol, Michel Reniers, Axel Rennoch, Brian Ritchie, Eric Rutten, Ina Schieferdecker, Jan Tretmans, Ulrich Ultes-Nitsche, Adam Wolisz.

# Organizing Committee

(GMD/Fokus Berlin, D)

Birgit Benner
Axel Rennoch
Ina Schieferdecker
Theofanis Vassiliou-Gioles

# Contents

**Session 5: *Modelling***



**Session 6: *Cases Studies***

# From Dining Philosophers to Dining Cryptographers

Günter Karjoth
IBM Research
Zurich Research Laboratory

**Abstract**

In theory, formal methods give us the ability to determine whether properties we ascribe to specifications or software systems hold for certain. However, the assurance that can be obtained from formal methods comes at a price. In the eighties, the computer networks community invested a lot in tools, theories, and case studies. They used formal methods to provide a rigorous and unambiguous way of designing and documenting protocols, to allow formal analysis (verification/performance analysis) before protocols are implemented, and to allow automatic code generation from the formal specification. The seminal work on Communicating Finite State Machines was followed by approaches based on process algebra and temporal logic, to give an example. In the last decade, however, attention shifted to computer security as an application area where the expense of faulty software would make the application of formal methods cost-effective. But security properties such as confidentiality and authenticity are often difficult to characterize formally (or even informally). In our presentation, we review ways in which above communities describe their domain-specific properties, how mechanisms are captured, and how protocols are analyzed. We conclude that despite the different objectives, even "traditional" methods can be sucessfully applied in the field of computer security. As an example, we describe our work on giving an operational semantics of the JavaCard Virtual Machine and using a well-known model checker for analysis.

# Performance and reliability model checking and model construction

H. Hermanns

University of Twente, Faculty of Computer Science, Formal Methods and Tools Group,
P.O. Box 217, 7500 AE Enschede, the Netherlands

hermanns@cs.utwente.nl

**Abstract**

Over the last decade formal methods have been extended towards performance and reliability evaluation. This paper tries to provide a rather intuitive explanation of the basic concepts and features in this area. The intention is to give an illustrative introduction to the basics of stochastic models, to stochastic modelling using process algebra, and to model checking as a technique to analyse stochastic models.

## 1 Introduction

Modern industrial systems, such as communication networks, transport systems, or manufacturing systems, are more and more operating in a stochastic context: communication lines can break, buffers can overflow, a lorry with material for a just-in-time production line might get stuck in a traffic jam. Each of these phenomena is stochastic by nature, its absence or presence can only be predicted up to some probability. Since these stochastic phenomena have impact on the system under consideration, it is nowadays commonly agreed that the systems themselves exhibit stochastic behaviour. As a consequence, performance and reliability studies of industrial systems have to take into account that rigid assessments ("It is impossible that the system fails") only hold under unrealistic assumptions.

The construction and analysis of models suited for performance and reliability studies of real-world phenomena is a difficult task. To a large extent this problem is attacked using human intelligence and experience. Due to increasing size and complexity of systems, this tendency seems even growing: performance as well as reliability modelling becomes a task dedicated to specialists, in particular for systems exhibiting a high degree of irregularity. Traditional performance models such as queueing networks lack hierarchical composition an abstraction means, significantly hampering the modelling of systems that are developed nowadays.

On the other hand, for describing the plain functional behaviour of systems various specification formalisms have been developed that are strongly focussed on the facility to model systems in a compositional, hierarchical manner. A prominent example of such specification formalisms is the class of *process algebra* [14]. Developed on a strong mathematical basis, process algebra has emerged as an important framework to achieve compositionality. Process algebra provides a formal apparatus for reasoning about structure and behaviour of systems in a compositional way.

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). 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.

To analyse properties of formally specified models *model checking* is a very successful technique to establish the correctness of the model, relative to a given set of temporal logic properties the model is supposed to satisfy [9, 10]. Using efficient encoding techniques, model checking has been applied to industrial size designs involving more than $10^{100}$ states.

It appears valuable to apply efficient model checking techniques also to performance and reliability properties of industrial systems. Since performance and reliability models are stochastic in nature, the properties of interest are stochastic as well, and have to be described in an appropriate extension of a temporal logic. The model checking algorithm then involves the calculation (or approximation) of probabilities of certain properties to hold.

This paper tries to provide a rather intuitive explanation of the basic concepts and features of stochastic models, of stochastic modelling using process algebra, and of model checking as a technique to analyse stochastic models. For the sake of being illustrative the paper tends to treat various fine points much more simplistic than the advanced reader probably desires.

The paper is organised as follows. Section 2 introduces the basic concepts of stochastic models. Section 3 exemplifies the use of process algebra for modelling stochastic phenomena by means of a real-world example, and Section 4 describes the model checking approach to analyse stochastic models. Section 5 concludes the paper.

## 2 Stochastic models

A stochastic model is basically a means to describe the evolution of a real-world phenomenon as time[1] passes, with a particular emphasis on phenomena with stochastic timing characteristics. In other words, repeated observations of the same phenomenon can have varying timing characteristics, but their variation exhibits a specific kind of randomness.


Figure 1: At the door of a gambler

As an example, consider a gambler that throws a die every minute. Observing the gambler, one might wish to study a phenomenon, such as the time that it takes to throw a six. Starting the observation at some arbitrary minute, one counts the minutes till the

[1] It is a bit narrow minded to consider the time domain as the only possible domain of variability. Spatial Markov processes, for instance, are used to describe the evolution of some phenomenon as its position in some appropriate space changes, as opposed to the time.

die shows a six. Obviously, repeated observations will usually lead to different results, at least if gambling with a fair die. Nevertheless, the variation among these observations exhibits a specific kind of randomness: The time needed to throw a six is known to follow the so-called geometric probability distribution.

**Probability distribution** A probability distribution is a function that assigns a probability (a real value between 0 and 1) to each element of some given set. For instance, the geometric probability distribution $P$ assigns probabilities to natural numbers. For the gambler, these numbers enumerate the minutes he is already gambling (remind that he throws the die once per minute). For some $t$, $P(t)$ is the probability to see the first six after $t$ minutes, and is given by:

$$P(\text{see the first six after } t \text{ minutes}) = 1 - \left(\frac{5}{6}\right)^t,$$

or complementary,

$$P(\text{still no six after } t \text{ minutes}) = \left(\frac{5}{6}\right)^t.$$

For instance, the probability of not having seen a six after $t = 2$ minutes (i.e. after throwing the die twice) is 25/36.

To make the example a bit more interesting, assume that the gambler is throwing the die somewhere outside his office. Before leaving his office he has put a note on the door, as depicted in Figure 1. In fact, his intention is to return to his office as soon as the die shows a six. Now let us assume that someone arrives at his door, finding the door closed. How long will he have to wait for the gambler? Probably just a minute, but probably (more likely) more than a minute, probably (unlikely) more than ten minutes. Since this experiment is governed by the above geometric distribution, the probability of having to wait more than a minute is 5/6, the probability of waiting more than ten minutes is $(5/6)^{10}$. Figure 2 depicts these probabilities for the first 15 minutes.

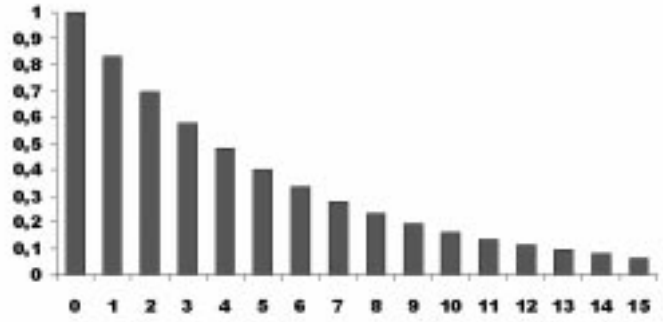



Figure 2: A geometric probability distribution: Will the gambler still be absent at time $t$?

**Markov chain** Having explained the gambler's behaviour, we are now in the position to specify a stochastic model of his behaviour. It is depicted in Figure 3. As many other (formal or semi-formal) models, the model is a graph, consisting of states and transitions. There are two states in this model. One state represents the absence of the gambler, one represents his presence in the office. The model contains three transitions representing possible events that might induce a change of state. One transition indicates that every

minute the absent gambler has a 1-out-of-6 chance to return to his office. Another transition indicates that with probability 5/6 the absent gambler will miss the six, and hence has to stay absent for at least another minute. In case he is back in his office, the third transition indicates that he stays there (ad infinitum). The small arrow on top of the left state indicates the initial state. i.e. the state occupied at time zero.

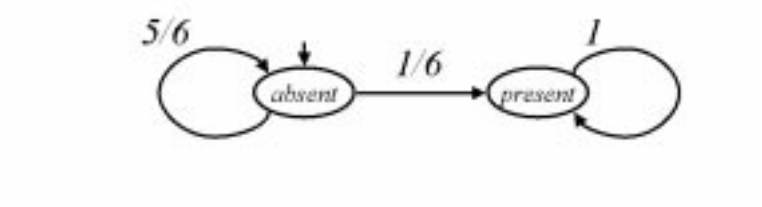



Figure 3: A discrete-time Markov chain describing the gambler's behaviour

The stochastic model of the gambler's behaviour is a very simple one. It is a Markov chain, named after A.A. Markov who studied models of this kind in the beginning of the last century. More specific, it is a discrete-time Markov chain (DTMC), since state changes are only possible at discrete points in time: The gambler can return to his office precisely every minute only. DTMCs restrict the possible time points for state changes to a discrete subset of dense real time. As in our example, these time points are often (but not necessarily) equidistant.

**Markov chain analysis** For a given stochastic model, such as a Markov chain, there is usually a variety of interesting properties that one might want to study. Two substantially different classes of properties can be distinguished. *Transient analysis* investigates the evolution of the model up to a given point in time. On the contrary, *steady-state analysis* focusses on the long-run average behaviour. It requires that on the long-run initial start-up effects (the transient phase) do not have a measurable impact.

A trivial steady-state property for the gambler is that with probability 0 he will be absent on the long-run. As an example for a transient property, we have already indicated that the probability of still being absent after 10 minutes is $(5/6)^{10}$. A variant of transient analysis gives us that on the average it takes the gambler six minutes to throw a six. So, the sign on the office door is essentially right, the gambler will be back in six minutes, on the average.

**Analysis techniques** In practice, three fundamentally different techniques are used to analyse stochastic models. They differ with respect to accuracy, applicability and computational requirements. Here, we only give a concise subjective summary on differences and similarities, and refer to Jain's textbook [25] for a more elaborate discussion.

*Simulation* The stochastic model is mimicked by a simulator throwing dice and producing statistics of simulation time spent in states. The fraction of simulation time spent in a particular state is used as an estimate for the state probability. This technique is generally applicable, in particular it is suitable also for non-Markov stochastic models. However, it should be noticed that good accuracy tends to require long simulation runs, and hence limits applicability in practice.

*Numerical solution* The transient or steady-state behaviour of a stochastic model is obtained by an exact or approximate algorithm where model parameters are instantiated with numerical values. This approach gives accurate results in general, up

to numerical precision. On the other hand, its applicability is restricted to finite Markov chains (with a few exceptions, see e.g. [17, 24]). Furthermore the number of states of the model is a limiting factor, because of computational requirements. A very readable textbook on numerical solution methods is [26].

*Analytical solution* The transient or steady state property of interest is expressed as a closed formula over the parameters of the model. This is the most simple, accurate and elegant technique. However, analytical solutions are available only for highly restricted classes of stochastic models.

**Absence of memory** Markov chains are widely used as stochastic models of real-world phenomena. This is mainly because they possess a distinguishing feature that simplifies both modelling and analysis. They obey the so called *memoryless property*: The future evolution of a Markov chain model is independent of the past, it only depends on the state currently occupied. This property is best explained in terms of the absent gambler. The probability that the gambler returns to his office after one minute from now is 1/6, independent of the fact that someone might be waiting for him in front of his door for ten minutes (or years) already. This is a direct consequence of the fact that a fair die has no memory; the die does not change if it has not shown a six for ages. This should not be mixed with the fact that the probability of actually having to wait for ten minutes is low, $(5/6)^{10}$. Under the assumption that this unlikely case becomes reality, it still needs another six minutes waiting time on the average, as the sign on the door indicates.

**Discrete vs. continuous time** Discrete-time Markov chains are convenient to describe the stochastic evolution of sequential systems. In each state, the outgoing transitions define how the probability mass will be spread at the next time instant. Since DTMCs evolve in a discrete time domain, the flow of probability is not continuous, instead it possesses jumps, and remains unchanged in the time interval between two relevant time points, such as between $t = 2$ and $t = 3$. This is relatively convenient for sequential systems. But it is not convenient in a concurrent probabilistic setting, for both theoretical as well as pragmatic reasons.

As an example, imagine that the gambler's office door is checked by some customer. In case he finds the door closed he probabilistically decides to check again after either 24 or 48 seconds. Note that the basic time unit of this DTMC is 24 seconds. For instance, one might want to study the probability that the customer finds an open door after 72 seconds.

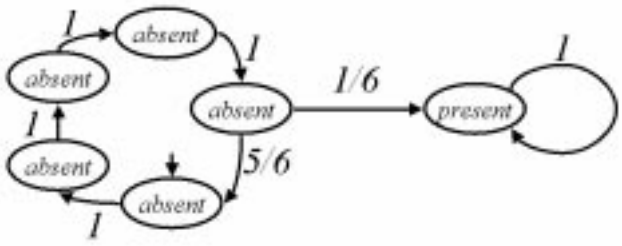



Figure 4: A discrete-time Markov chain describing the gambler's behaviour if observed every 12 seconds

Without specifying the model in all detail, we are already in the position to understand the problem: In order to develop a concurrent probabilistic model of both gambler and

customer, we have to relate events that may happen at every 24 seconds to events that happen may every 60 seconds. One solution is to change the basic time unit of both models to 12 seconds, the greatest common divisor of their basic time units. In other words, the gambler's model is blown up to record in 4 additional states that while being absent, four times 12 seconds pass till he may throw the die in the last twelve seconds of the minute (cf. Figure 4).[2] After a similar change in the customer's sub-model, one can combine both models (by essentially taking the crossproduct of states and the products of transition probabilities). To determine the concurrent stochastic behaviour at the next point in time (i.e. after 12 seconds) one synchronously updates the respective states in the two sub-models, because state changes now occur exactly at the same time. The probability for such a joined transition is given by the product of the transition probabilities in the sub-models.

This strategy has two practical limitations, at least. First, it tends to induce a tremendous blow-up of the size of the model, caused by the number of auxiliary states needed in general. Second, it fails if there is no greatest common divisor, for instance if the customer shows up every $\pi$ seconds, or if time points are not equidistant. As a consequence, virtually all stochastic models of concurrent systems are developed in a continuous time domain, including models of modern computer systems (even though each component of such a system can be considered as working in discrete time, changing state according to fixed frequency clock ticks).

**Continuous-time Markov chains** Continuous-time Markov chains (CTMCs) are Markov chains interpreted over continuous time, in contrast to DTMCs. They are widely used to model the stochastic behaviour of concurrent real-world phenomena, due to their mathematical simplicity, paired with modelling convenience.

How does the continuous-time variant of the gambler look like? In a continuous time setting, the absent gambler is able to return to his office at arbitrary time points. Still we may assume that he has a 1-out-of-6 chance to return within the first minute, and so on. Under these assumptions, we get the following probability distribution:

$$P(\text{still no six after } t \text{ minutes}) = (5/6)^t.$$

What is this? It perfectly resembles the geometric distribution appearing in the discrete time case, but it is different. The difference is that the domain of this function is the real line, instead of the natural numbers. In other words, the above function assigns a probability to all time points one may think of, instead of only to each minute. Hence, there is now a non-zero probability of returning within the first second already, namely $1 - (5/6)^{1/60}$. Instead of being a geometric distribution, this function belongs to the class of so-called (negative) exponential probability distributions, because $(5/6)^t$ can be rewritten to $e^{-\lambda t}$, with $\lambda = \ln 6 - \ln 5 \approx 0.18232$. The value $\lambda$ is a parameter of the distribution, usually called 'rate'. For $t < 15$, the probabilities determined by this exponential probability distribution are depicted (by the dark plot) in Figure 5. The expected value of an exponential distribution (i.e. the average duration) is $1/\lambda$, the reciprocal value of the rate. So, the (continuously gambling) gambler returns after 5.48 minutes on the average, not after six minutes.[3]

[2]Note that this change encodes some kind of memory in an otherwise memoryless model: A sequence of states is used to keep track of the time already spent in the original state.

[3]Remark that since the probability mass is flowing continuously, a sixth of the mass leaks prior to the first minute tick. Hence, to some extent the probability mass flows earlier than in the discrete-time case, where a sixth of the probability mass jumps a bit later, at each minute tick. As a consequence, the average time needed for the continuously gambling gambler is slightly smaller than 6 minutes. To obtain an average duration of 6 minutes, one has to adjust $\lambda$ to 1/6.

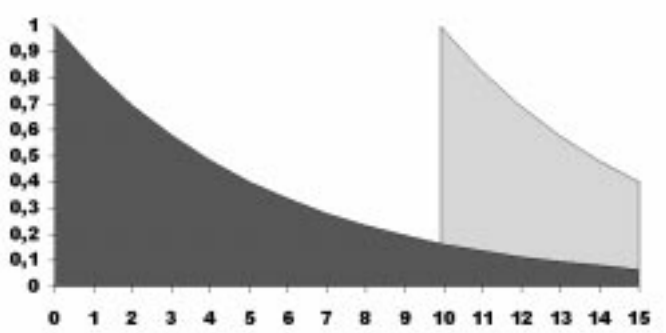



Figure 5: A negative exponential probability distribution with $\lambda = \ln 6 - \ln 5$: Will the gambler still be absent at time $t$?

A continuous-time Markov chain model of this absent gambler is depicted in Figure 6. It consists of two states, and one transition. The transition represents that the gambler can return to his office with rate $\lambda$. The gambler stays absent as long as needed to throw a six. According to the value of $\lambda$ the probability mass flows from state to state as time passes, that is, a fraction of $1 - e^{\lambda} = 1/6$ of the probability mass flows from the left state to the right state per minute.[4]

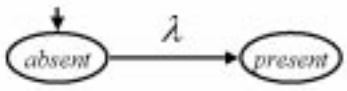



Figure 6: A continuous-time Markov chain describing the gambler's behaviour

Though the above example shows one of the simplest CTMCs one can think of, it exhibits all relevant ingredients: states and transitions, the latter labelled with rates of exponential distributions. It is worth to note that – in correspondence to geometric distributions – exponential distributions are memoryless: The future evolution of a CTMC model is independent of the past, it only depends on the state currently occupied. In terms of the gambler, the probability that the absent gambler returns to his office within the next minute is 1/6, independent of the fact he might have been absent for ages already.

Figure 5 allows us to illustrate the memoryless property in a graphical way [1]. Consider the case that the gambler is still gambling after minute 10. We obtain the probability that he will still be gambling at time $10 + t$ by stretching the tail of the distribution (from time 10 to $\infty$) upwards in such a way that it reaches probability 1 for minute 10, i.e. $t = 0$. As a matter of fact, this stretching returns precisely the original distribution, as indicated by the light-grey plot in Figure 5, except that it is shifted by 10 minutes. (The same graphical illustration holds for the geometric distribution, but for no other discrete or continuous distribution.)

From a pragmatic point of view, the memoryless property is rather convenient. It simplifies analysis, but it also simplifies modelling. In particular, it fits well to concurrent stochastic phenomena: If two sub-models, both described in terms of CTMCs, are to

[4]Since the gambler continuously tries to return to his office, there is no need to record by an explicit (looping) transition that he might fail for some (continuous) time. For CTMCs, this fact is implicit, while in the DTMC scenario it is not.

be considered concurrently, one can simply interleave their evolution: If one sub-model changes from one state to another, the other sub-model is not affected. The fact that the latter has been staying in some state for some time (the time it took the former sub-model to change state) does not need to be recorded somehow, because it does not alter the future behaviour of the latter sub-model, due to the memoryless property.

Anyway, it should be clearly stated that absence of memory is an assumption that is by far not always justified when modelling real-world phenomena.[5]

# 3 Formal specification of continuous-time Markov chains

In this section we illustrate the use of formal methods to model a specific aspect of a real-world example as a CTMC. Several formal notations exist that map on CTMCs, among them stochastic Petri nets and stochastic process algebra. Here we restrict ourselves to illustrate the use of process algebra; an introduction to the Petri net based approach can be found for instance in [1]. As opposed to Petri nets, process algebra allows one to compose models out of smaller sub-models, by means of general composition operators such as parallel composition and choice [14], and also more specific constructs, such as exception handling [16]. We will make use of these operators to model a simplified view on the performance and reliability of the Hubble space telescope.

**The Hubble Space Telescope** The Hubble space telescope (HST) is an orbiting astronomical observatory operating from the near-infrared into the ultraviolet (cf. Figure 7). Launched in 1990 and scheduled to operate through 2010, the HST carries a variety of instruments producing imaging, spectrographic, astrometric, and photometric data.

The HST was first conceived in the 1940. It was designed and built in the 1970s and 1980s, aiming at a life span of 15 years with on-orbit servicing taking place on 3 year intervals. The HST is a cooperative program of the National Aeronautics and Space Administration(NASA) and the European Space Agency (ESA). Originally, the HST was designed to be returned to earth via the space shuttle every 5 years with on-orbit servicing every 2.5 years as well. This concept was later scrapped as it was felt there was a too great risk of contamination and structural load to make the concept sound. By the time it was launched the HST cost $1.5 billion U.S. dollars.

Since the telescope has been launched in April 1990, three servicing missions were carried out: in December 1993, in February 1997, and in December 1999. During the last mission the stabilising unit of the HST was repaired. This was necessary, since severe problems with the reliability of the gyroscopes contained therein had forced the HST to turn into a sleep mode.

The gyroscopes are part of HST pointing system. They provide a frame of reference to determine where it is pointing and how that pointing changes as the telescope moves across the sky. They report any small movements of the spacecraft to the HST pointing and control system. The computers then command the spinning reaction wheels to keep the spacecraft stable or moving at the desired rate in order to avoid that the telescope pointing device staggers. This is of particular importance to avoid that pictures taken by the telescope are blurred. The gyroscopes work by comparing the HST motion relative to the axes of the spinning masses inside the gyroscopes.

[5] It is possible to incorporate a notion of memory into the model, similar to what we have used to realise synchronisation of DTMCs (cf. footnote 2). In this way, general non-exponential probability distributions (so-called phase-type distributions) can be represented. The price to pay for this is usually a blow up of the model.

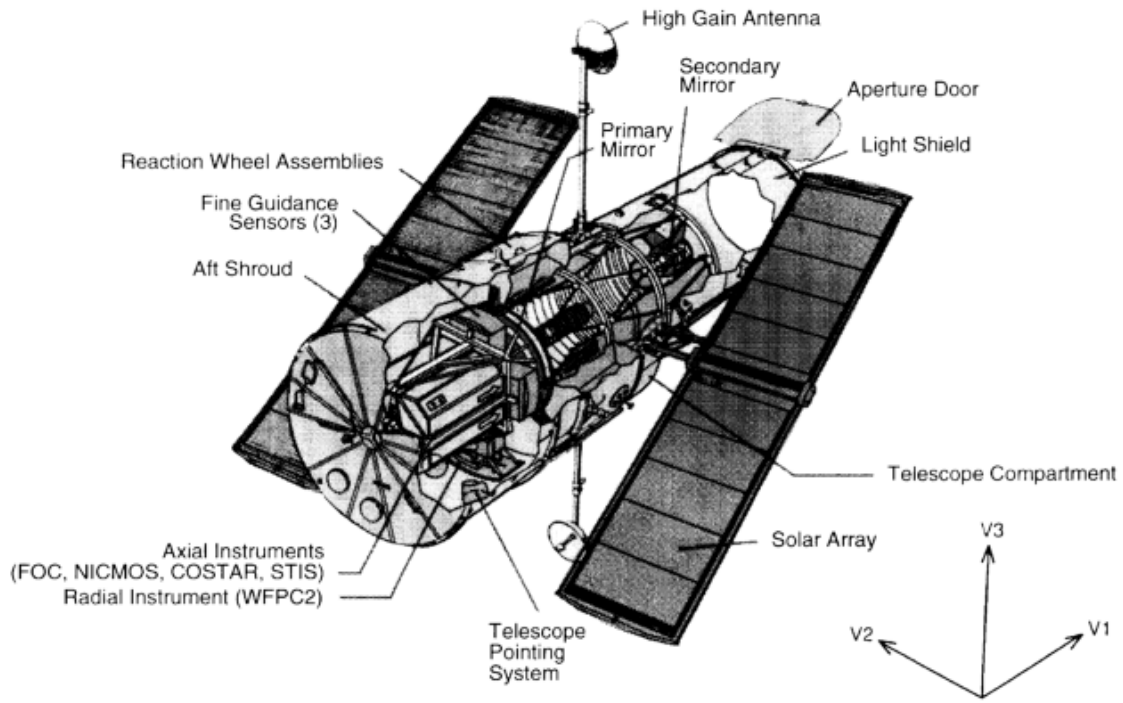



Figure 7: The Hubble space telescope [23].

The HST has a total of six gyroscopes, grouped into three fine guidance sensors. They are arranged in such a way that any three gyroscopes can keep the HST operating with full accuracy. Two fine guidance sensors had been replaced already during the first servicing mission in 1993. Till the end of the second servicing mission in 1997, all six gyroscopes were working normally, but then one after the other failed. Starting from January 1999 the HST had been operating with only 3 functional gyroscopes. As a consequence of a fourth gyro failure on November 13, 1999, HST turned itself into a sleep mode and the science program was suspended. Without operational gyro the telescope would have run the risk to crash. In December 1999, a space shuttle mission was sent to the HST to replace (among others) the complete stabilising unit. This mission was successful.

In order to judge whether the problems of the HST could have been expected beforehand, one might want to study the reliability of the stabilising unit by means of an abstract stochastic model. Here we construct a simple Markov chain model of the gyros, and of their controller. The model is a toy example, developed to give a flavour of Markov chain modelling with process algebra. The model is developed in the algebra of interactive Markov chains (IMC) [18, 20], an extension of basic Lotos [6].

**Basic processes** Each gyro might FAIL after an exponentially distributed amount of time (it is known that exponential distributions fit relatively well to failures of technical equipment). The failure rate $\lambda$ is the same for all gyros. A GYRO specification is as follows:

$$\text{GYRO} = (\lambda).\ \text{FAIL}.\ \text{STOP}$$

This specification corresponds to a graphical representation depicted in Figure 8. Apart from a transition labelled $\lambda$ representing the delay prior to failure, there is a second kind

of transition, indicated by a dotted arrow labelled FAIL. In abstract terms, this transition represents the potential of interaction, i.e. of synchronising with a partner transition (labelled with the same name) in a different sub-model. The potential of interaction between sub-models is one of the well known features offered by a process algebraic approach [6].

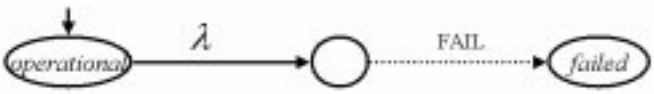



Figure 8: A simple interactive Markov chain describing the gyroscope's behaviour

**Parallel composition** Six of these gyros coexist independently in the stabilising unit, together with a controller that keeps track of the status of each gyro, by means of synchronisation on FAIL. This is realized using the operator |[FAIL]| for synchronisation, and ||| to denote independent parallelism (among the gyros):

$$\begin{aligned} \text{STABILISER} \;=\; & \text{CONTROLLER} \\ & |[\text{FAIL}]| \\ & (\text{GYRO}|||\text{GYRO}|||\text{GYRO}|||\text{GYRO}|||\text{GYRO}|||\text{GYRO}) \end{aligned}$$

The controller counts the number of failures, and mechanically turns the telescope into sleep mode in case four gyros have failed. To turn into sleep mode requires some time. For the moment we just assume an exponential distribution with rate $\mu$. We will explain shortly how to deal with other distributions. After turning on the sleep mode, the controller notifies the base station by means of a SLEEP signal. In the meantime, further gyro failures might occur. If the last gyro fails, a CRASH is assumed to be inevitable. The graphical representation of the controller is depicted in Figure 9.

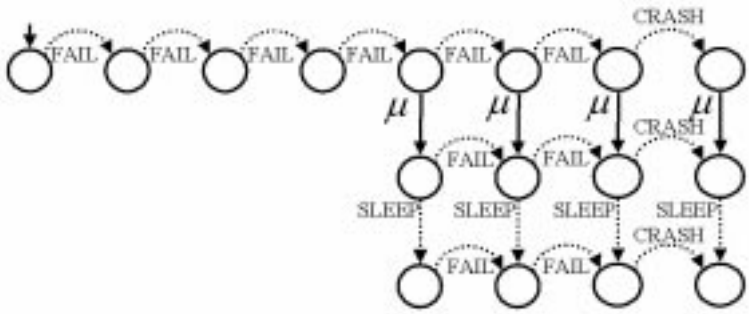



Figure 9: An interactive Markov chain describing the controller

$$\begin{aligned} \text{CONTROLLER} \;=\; & \text{FAIL. FAIL. FAIL. FAIL.} \\ & ((\mu).\ \text{SLEEP. STOP} \;|||\; \text{FAIL. FAIL. CRASH. STOP})) \end{aligned}$$

To complete the picture, we consider the stabilising unit of the HST in the context of the base station. The base station listens to the SLEEP notification and reacts accordingly:

launch a space shuttle mission to repair – and then restart – the telescope.

$$\text{BASE} \;=\; \textsc{sleep}.\ \textsc{prepare}.\ \textsc{launch}.\ \textsc{repair}.\ \textsc{restart}.\ \text{BASE}$$

**Exception handling** The complete specification consists of the STABILISER and the BASE station synchronising on SLEEP. Two events may alter the functioning of the system. If a CRASH occurs, the whole system is extinguished, but if the shuttle mission manages to repair the stabilising unit in time, the whole system will be restarted anew.[6]

$$\begin{array}{rl} \text{HST} \;=\; & \mathit{trap} \\ & \quad \textsc{crash} \;\text{->}\; \text{STOP} \\ & \quad \textsc{restart} \;\text{->}\; \text{HST} \\ & \mathit{in}\ \text{STABILISER} \;|[\textsc{sleep}]|\; \text{BASE} \end{array}$$

**Time constraints** Of course, preparing the shuttle mission takes time, and one might wish to incorporate the expected (random) delay in the model. To do so, we can use a constraint-oriented style, as advocated in [20]. This style allows one to add constraints on the timing of certain sequences of interactions, such as between PREPARE and LAUNCH by means of a dedicated operator. For instance,

$$\begin{array}{ll} \mathit{on} & \textsc{prepare} \\ \mathit{delay} & \textsc{launch} \\ \mathit{by} & (\nu).\ \text{STOP} \\ \mathit{in} & \text{HST} \end{array}$$

adds an exponentially distributed delay with rate $\nu$ between PREPARE and LAUNCH. Semantically speaking, this will have the same effect as specifying $\text{BASE} = \textsc{sleep}.\ \textsc{prepare}.\ (\nu).\ \textsc{launch}.\ \textsc{repair}.\ \textsc{restart}.\ \text{BASE}$, but it is much more modular and flexible, in particular because it can be used to impose very general time constraints, instead of only exponentially distributed ones, see [20]. In short, one can insert an arbitrary (phase-type distributed) delay between PREPARE and LAUNCH, by replacing $(\nu).\ \text{STOP}$ in the above expression by some appropriate term (in fact, an encoding of the distribution as a CTMC).

For the sake of the presentation we do not add further time constraints, even though a realistic model would at least impose some nontrivial delay between LAUNCH and REPAIR, (as well as a non-exponential delay to set up the SLEEP mode.)

**Extracting the Markov chain** The complete HST specification gives rise to a stochastic model, a CTMC depicted in Figure 10. It is obtained from the specification by applying the formal semantics of the process algebra, and compressing the model by means of an appropriate weak bisimulation afterwards.[7] The states are labelled from left to right with the number of gyros that are currently operational, except if the system is *sleep*ing, or *crash*ed.

Remark that in this CTMC the failure rate $\lambda$ appears weighted with different multiplying factors. The intuitive reason is that if six gyros are operational, the time to the first failure is six times smaller than if only one gyro is left. This increased failure rate for multiple identical components is correctly derived by the formal approach outlined above.

[6] The semantics of this exception handling is similar to [16].

[7] As explained in [20], constructing the Markov chain requires to hide all possible interactions beforehand. This is necessary but not always sufficient to extract a CTMC, since interactive Markov chains are strictly more expressive than CTMCs (because of the absence on nondeterminism in CTMCs).

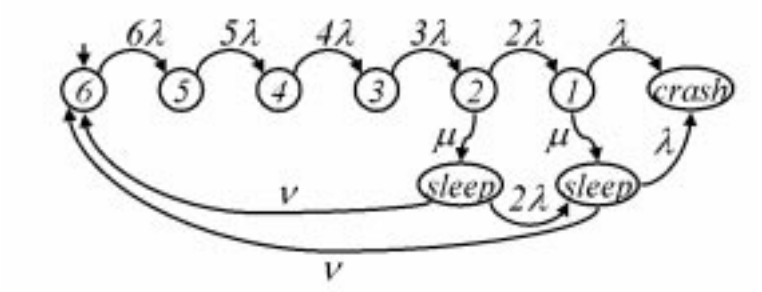



Figure 10: A continuous-time Markov chain corresponding to the stochastic behaviour of the telescope

# 4 Performance and reliability via model checking

In this section we illustrate the use of model checking to analyse performance and reliability properties of CTMC models. We discuss the main ingredients of this approach, and apply model checking to the simple Hubble space telescope example of Section 3.

**Temporal logic** The model checking approach relies on the use of temporal logic for specifying properties one is interested in. For this purpose temporal logic provides means to specify undesired (or – dually – desired) evolutions. Typical specifications of properties are `'something undesired never happens'` or `'eventually a desired state is reached'`. A temporal logic specification is usually considered in the context of a given model (provided by some process algebraic specification, for instance). The mechanic verification whether a model satisfies a temporal logic specification is called *model checking*. It is worth to mention that basic temporal logic does not allow one to reason about delays and time points (although the name might suggest the converse). It is 'temporal' in the sense that it allows one to refer to the ordering of events as the model evolves in time.

**Temporal logics for Markov chains** In the context of Markov chain models, the temporal logic approach turns into a probabilistic temporal one. It is not sufficient to decide whether `'eventually a desired state is reached'`. Instead the probability of eventually reaching a desired state is much more interesting. For the gambler example in Figure 3 the standard interpretation of `'eventually the gambler will be present'` would return false, because it is in principle possible to stay absent ad infinitum. However, this evolution is extremely unlikely, it has probability zero. So, a quantitative interpretation of temporal logic is needed, quantifying the likelihood of satisfying a given property. This allows one to specify properties such as `'a desired state is eventually reached with at least probability 0.95'`.

Moreover, since the evolution of a Markov chain model in time is measurable (in the true sense of the word), it is possible to reason about time instances within the temporal logic. Timed properties such as `'with at most probability 0.2 the gambler will still be absent after 10 minutes'` are possible.

**Continuous stochastic logic** The continuous stochastic logic (CSL), first proposed in [2] and further refined in [4, 3] provides means to reason about continuous-time Markov chain models. It is a branching time logic based on CTL [8] with dedicated means to specify time intervals, and to quantify probability. As explained in Section 2, there are

two substantially different classes of properties of a CTMC: transient and steady-state properties. Therefore, CSL provides two complementary means to quantify the probability mass: a steady-state operator $\mathcal{S}$, to quantify the long-run likelihood, and a transient probability operator $\mathcal{P}$.

For instance, a steady-state property $\mathcal{S}_{\leq p}(\Phi)$ is true if the long-run likelihood of property $\Phi$ is at most $p$.[8] $\Phi$ can be a basic property (usually called atomic proposition) valid (or invalid) in some state. It can also be an arbitrary nested property of the logic. The transient probability operator is used to quantify the likelihood of evolving in a specified way, from a given state and a given time point on. For example $\mathcal{P}_{\leq p}(X\,\Phi)$ is true in a particular state if the probability of moving (in one step) to a state where $\Phi$ holds is bounded by $p$. Apart from $X\,\Phi$, there can be various other arguments for the operator $\mathcal{P}$, such as

- $\Diamond\,\Phi$ quantifies the probability mass evolving in such a way that eventually a state is reached where $\Phi$ holds (called a $\Phi$-state in the sequel).
- $\Diamond^{[0,t]}\,\Phi$ characterises the amount of probability reaching a $\Phi$-state within $t$ time units.
- $\Phi_1\,\mathcal{U}\,\Phi_2$ characterises the amount of probability evolving only along $\Phi_1$-states until a $\Phi_2$-state is reached.
- $\Phi_1\,\mathcal{U}^{[t_1,t_2]}\,\Phi_2$ quantifies the probability mass evolving only along $\Phi_1$-states until a $\Phi_2$-state is reached, under the additional constraint that $\Phi_1$ holds at least up to time $t_1$, and $\Phi_2$ holds at time $t_2$ the latest.

**Model checking CSL** Model checking a CTMC with respect to a given CSL property involves different algorithms. Since the details are not of vital importance for a proper understanding of the approach – at least relative to the logical means to specify properties – we only give a concise overview of the ingredients.

As in other model checking strategies, a couple of graph algorithms are used. In addition, algorithms to quantify the probability mass of satisfying the above criteria are needed. In principle, these probabilities could be derived using simulation, numerical solution, or sometimes via analytical solutions. Since numerical solution of CTMCs is well studied and generally applicable, it seems wise to use numerical solution methods to model check CSL properties [4]. In this way, model checking involves matrix-vector multiplications (for $X$), solutions of linear systems of equations (for $\Diamond$, $\mathcal{U}$ and for $\mathcal{S}$), and solutions of systems of Volterra integral equations (for $\mathcal{U}^{[\cdots]}$). Linear systems of equations can be iteratively solved by standard numerical methods [26]. Systems of integral equations can be solved either by piecewise integration after discretisation, or they can be reduced to standard transient analysis [3]. A prototypical model checker for CSL, $\mathsf{E} \vdash \mathsf{MC}^2$, is available [21], We shall make use of $\mathsf{E} \vdash \mathsf{MC}^2$ to investigate properties of the Hubble space telescope.

**Properties of the telescope model** CSL provides a rich framework to study performance and reliability properties of the HST. Here we consider a few illustrative cases. In order to allow the calculation of numerical values, we first need to fix the model parameters $\lambda$, $\mu$, and $\nu$ of the CTMC in Figure 10. Assuming a basic time unit of one year, we set $\lambda = 0.1$, i.e. we assume that each gyro has an average lifetime of 10 years. (Remind that $1/\lambda$ gives the average duration of an exponential distribution with rate $\lambda$.) To turn

[8] Instead of '$\leq$' one may use arbitrary comparison operators, or specify intervals of probabilities instead.

on the sleep mode may require a hundredth of a year (a bit more than three days and a half) on the average, whence we set $\mu = 100$. Further assuming that preparing the repair mission will take about two months, we set $\nu = 6$. Unless otherwise stated we consider the validity of CSL properties in the initial state, i.e. the state labelled *6* in Figure 10. The state labels appearing in this figure serve as atomic state propositions for the logic.

First, let us look into long-run averages. An interesting property, often called *availability*, is the probability that the system will be available – i.e. neither *crash*ed nor *sleep*ing – on the long-run average. In CSL we assure an availability higher than $p$ by specifying

$$\mathcal{S}_{>p}(\neg\ (\ sleep\ \vee\ crash)\ ).$$

None of the states of the HST satisfies this property (whatever the value of $p$ may be). This should not be surprising, because the telescope is not constructed for the long run. In fact, the availability of the telescope is zero, because on the long run, the modelled telescope will crash, all the probability mass will eventually be cumulated in the *crash*-state (cf. Figure 10).[9]

While checking standard availability does not make much sense for the HST, the *instantaneous availability* is of interest. Instantaneous availability is a typical transient property, it is the probability that the system is operational at a given time point $t$. This time point could for instance be given by the need to observe a rare astronomic event. Assuming that an interesting comet passes the telescope in five years, we specify

$$\mathcal{P}_{\geq 0.95}(\Diamond^{[5,5]}\neg\ (sleep\ \vee\ crash))$$

in order to assure that with at least probability 0.95 the telescope is neither *sleep*ing nor *crash*ed then. (Note that the time interval $[t, t]$ denotes just a single time point.) This property is satisfied, we compute a probability of more than 0.98.

In the same direction, we may wonder about the probability to obtain blurred data at that time from the telescope, because less than three gyros are operational, but sleep mode is not yet turned on. This is a very unlikely situation, and one might accept at most a probability of $10^{-6}$. One way of characterising the relevant states is to isolate those (non-*sleep*) state that (with positive probability) can turn on the *sleep* mode in the next step. This gives us

$$\mathcal{P}_{\leq 10^{-6}}(\Diamond^{[5,5]}\ (\neg\ sleep\ \wedge\ \mathcal{P}_{>0}(X\ sleep)\ ),$$

a property that is not satisfied, because the probability of being in the specified states after 5 years is in the dimension of $10^{-5}$.

Another quantity of interest is the *time until first sleep*, i.e. the time span before the (fully operational) telescope has to be put into *sleep* mode for the first time. In reality, this happened within 2.7 years: All gyros were operational at the end of the second servicing mission in early 1997, and the *sleep* mode was turned on in November 1999. We specify a less than 10 % chance of such a first sleep within 2.7 years by

$$\mathcal{P}_{<0.1}(\ \neg sleep\,\mathcal{U}^{[0,2.7]}\ sleep)$$

It turns out that this property is valid, $\mathsf{E} \vdash \mathsf{MC}^2$ computes that the probability of a first sleep within 2.7 years amounts to about 0.03. A related question is whether it was likely not to witness any gyro failure within the four years between the first (1993) and the second

[9] Generally speaking, steady-state properties provide very useful insight in the model, in particular for the widespread class of models where the probability mass can flow forever without gradually leaking into some sink (so to speak), or where more than one sink exists. Each of these sink may in general consist of a set of mutually reachable states.

servicing mission (1997). We answer this question by checking whether the probability to leave the state *6* within 4 years is between, say, 0.3 and 0.7. (Notice that leaving state *6* corresponds to a gyro failure).

$$\mathcal{P}_{[0.3,0,7]}(\Diamond^{[0,4]} \neg\ 6)$$

In fact, this property is invalid, because the probability of a gyro failure within 4 years is approximately 0.9, thus exceeding the upper bound 0.7.

As a last example property, be reminded that the HST is planned to stay on orbit through 2010. Hence, it seems worth to study whether a *crash* before reaching the year 2010 can hardly be expected. To do so, we model check a property saying that there is at most a 1% chance that the system will *crash* within the next 10 years (given that the system was reset to state *6* in late 1999):

$$\mathcal{P}_{<0.01}(\Diamond^{[0,10]}\ crash).$$

This property is satisfied, the probability of crashing within 10 years is calculated by $\mathsf{E} \vdash \mathsf{MC}^2$ to be 0.00036. Be reminded that the model is a toy example, and that its timing parameters are not claimed to reflect reality.

# 5 Concluding remarks

In this paper, we have tried to give an illustrative introduction to the basics of stochastic models, to stochastic modelling using process algebra, and to model checking as a technique to analyse stochastic models.

A few questions have not been addressed to a satisfactory extent. In particular we have negligently skipped the discussion how to label states of a CTMC generated from a process algebra in such a way that these labels can be used in temporal logic property specifications. One solution to this problem is to move from a state based logic towards a transition-based formalism [22].

Another important issue for industrial strength formal analysis is the availability of tool support. At the current state, prototypical tool support is available for both the stochastic modelling and the analysis phase: A couple of prototypes exist that allow a process algebraic modelling of CTMCs [19, 7, 5]. So far, performance models with up to $10^7$ states have been modelled and analysed compositionally [20]. A prototypical model checker for Markov chains, $\mathsf{E} \vdash \mathsf{MC}^2$, is also available [21], it was used to check the above CSL properties of the Hublle space telescope. More effort is nevertheless needed to enhance modelling and analysis convenience. In addition, it seems favourable to link stochastic features to existing modelling and analysis tools with open architecture. We are currently making efforts to incorporate stochastic modelling and analysis features into the CADP toolset [13, 15].

Markov chain models have been the clear focus of this paper. Their memoryless property considerably simplifies both modelling and analysis, but the property also implies that many real-world phenomena can only roughly be approximated with Markov chains. Hence there is a need to extend the framework sketched in this paper beyond Markov models. The work of D'Argenio et al. [11, 12] develops a process algebra, called SPADES, to specify non-Markov performance and reliability models in an elegant way. So, the benefits of a process algebraic formalism extend to performance and reliability modelling in general. Anyhow, the analysis of such models needs further investigations. Since numerical solution methods are impractical in general, we are currently developing an open simulation environment to analyse SPADES specifications.

**Acknowledgements** Pedro R. D'Argenio and Joost-Pieter Katoen have provided valuable comments on an earlier version of this paper.

## References

[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* Wiley, 1995.

[2] A. Aziz, K. Sanwal, V. Singhal and R. Brayton. Verifying continuous time Markov chains. In *Computer Aided Verification (CAV 96)*, LNCS 1102, pp. 269–276, Springer, 1996.

[3] C. Baier, B.R. Haverkort, H. Hermanns and J.-P. Katoen. Model checking continuous-time Markov chains by transient analysis. In *Computer Aided Verification (CAV 2000)*, LNCS, Springer, 2000 (to appear).

[4] C. Baier, J.-P. Katoen and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In *Concurrency Theory (CONCUR 99)*, LNCS 1664, pp. 146–162, Springer, 1999.

[5] M. Bernardo, W.R. Cleaveland, S.T. Sims, and W.J. Stewart. TwoTowers: A tool integrating function and performance analysis of concurrent systems. In Proc. of IFIP Joint Int. Conf. on Formal Description Techniques and Protocol Specification, Testing and Verification. North Holland (IFIP), 1998.

[6] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems* 14:25-59, 1987.

[7] G. Clark, S. Gilmore, J. Hillston, and N. Thomas. Experiences with the PEPA performance modelling tools. *IEE Proceedings–Software* 146(1):11-19, February 1999.

[8] E.M. Clarke, E.A. Emerson and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Tr. on Progr. Lang. and Sys.* 8(2):244-263, 1986.

[9] E.M. Clarke and R.P. Kurshan. Computer-aided verification. IEEE Spectrum 33(6):61–67, 1996.

[10] E.M. Clarke, O. Grumberg and D. Peled. *Model Checking.* MIT Press, 1999.

[11] P.R D'Argenio. *Algebras and Automata for Timed and Stochastic Systems.* PhD-Thesis, University of Twente, November 1999.

[12] P.R. D'Argenio, J.-P. Katoen E. Brinksma. Specification and Analysis of Soft Real-Time Systems: Quantity and Quality. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pp. 104-114, Phoenix, Arizona, December 1999. IEEE Computer Society Press.

[13] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier and M. Sighireanu. CADP (Caesar/Aldébaran Development Package): A protocol validation and verification toolbox. In *Computer Aided Verification (CAV 96)*, LNCS 1102, pp. 437-440, Springer, 1996.

[14] W.J. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science, Springer, 2000.

[15] H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In B. Steffen, ed, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 98)*, LNCS 1384, pp. 68–84, Springer, 1998.

[16] H. Garavel and M. Sighireanu. On the Introduction of Exceptions in E-LOTOS. In R. Gotzhein and J. Bredereke, editors, *Formal Description Techniques IX*, pp. 469-484, Chapman and Hall, 1996.

[17] B. Haverkort. SPN2MGM: Tool support for matrix-geometric stochastic Petri nets. In *Proc. of IEEE International Computer Performance and Dependability Symposium*, pp. 219–228, Urbana-Champaign, Illinois, September 1996. IEEE Computer Society Press.

[18] H. Hermanns. *Interactive Markov Chains.* PhD thesis, Universität Erlangen-Nürnberg, 1998.

[19] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis and M. Siegle. Compositional performance modelling with the TIPPTOOL. *Performance Evaluation* 39(1-4):5–35, 2000.

[20] H. Hermanns and J.P. Katoen. Automated compositional Markov chain generation for a plain-old telephony system. *Science of Computer Programming* 36(1):97–127, 2000.

[21] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser and M. Siegle. A Markov chain model checker. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, LNCS 1786, Springer, 2000.

[22] H. Hermanns, J.-P. Katoen, and J. Meyer-Kayser. Towards model checking stochastic process algebra, 2000 (submitted).

[23] The Hubble space telescope. `http://astro.sau.edu/~astro/html/MARAC/HST.html`

[24] C. Lindemann and R. German. Modeling discrete event systems with state-dependent deterministic service times. *Discrete Event Dynamic Systems: Theory and Applications* 3:249–270, July 1993.

[25] Raj Jain. *The Art of Computer Systems Performance Analysis.* J. Wiley, New York, 1991.

[26] W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton Univ. Press, 1994.

# A B Model for Ensuring Soundness of a Large Subset of the Java Card Virtual Machine

Antoine Requet

Gemplus Research Laboratory, Av du Pic de Bertagne,
13881 Gémenos cedex BP 100.
antoine.requet@gemplus.com

**Abstract**: Java Cards are a new generation of smart cards that use the Java programming language. As smart cards are usually used to supply security to a system, security requirements are very strong and certification can become a competitive advantage. Such a certification to a high Common Criteria or ITSEC level requires the proof of all the security mechanisms. Those security mechanisms include the byte code interpreter and verifier of the virtual machine. Previous works have been done on methodology for proving the soundness of the byte code interpreter and verifier using the B method. It refines an abstract defensive interpreter into a byte code verifier and a byte code interpreter. However, this work had only been tested on a very small subset of the Java Card instruction set. This paper presents a work aiming at verifying the scalability of this previous work. The original instruction subset of about ten instructions has been extended to more than one hundred instructions, and the additional cost of the proof has been managed by modifying the specification in order to group opcodes by properties.

**Keywords**: B method, Java Card, formal specification

## 1. Introduction

A smart card is a small embedded system generally used to supply security to an information system. Traditionally, the application and the operating system were developed in a secure environment by the card issuer. For few years, platforms (*e.g.*, Java Card, MultOS and Smart Card for Windows) have provided new facilities for application developers. They allow dynamic storage and execution of downloaded executable content. Those platforms are based on a virtual machine both for portability across multiple smart card micro-controllers and for security reasons. Such architecture introduces new risks: the most important one is the possibility to attack the card from an applet by exploiting some implementation faults. In order to avoid such a risk, card manufacturers have a fairly extensive qualification process. Quality insurance requirements for smart cards are very strong. To convince the customer that the system is secure enough, card manufacturers propose to evaluate their system through a certification process.
This certification is a means for the card issuer to promote its products against its competitors. Sometimes the customer or the targeted market requires the certification.

For example, the German market requires each product that uses an electronic signature to be certified at the E4 level of the ITSEC scheme. According to the certification rule and the requested level, the card issuer must provide all the elements needed by the authority to guarantee the quality of the development process. At some high levels, it is required to use formal methods and to provide the proof that security mechanisms satisfy the security policy. One of the trickiest problems is to prove the coherence of the different security mechanisms of the system. Since there are strong size constraints on the chip, the amount of memory is small. This leads Java Card to modify the security scheme. It becomes more crucial to be able to prove the correctness of the whole system security.
After a brief presentation of the Java Card security mechanisms, we sum up the state-of-the art on the formal verification of the Java byte code semantics. We emphasise the proof of the static and dynamic semantics coherence using our approach. Then, we conclude with the extension of our work and its integration in the whole Java Card model.

## 2. Security of the Java Card

The Java Card 2.1 standard [Sun-99] defines the CAP file (Converted APplet) *i.e.*, the structure of the input files. For each byte code, the standard defines the conditions required for a correct execution, but not the way to ensure that those conditions are met. The Java Card virtual machine is specially designed for smart card; several features have been removed, compared to the Java virtual machine, while others features have been added (*e.g.,* the applet firewall). The Java Card API is a set of tools or services aimed to help programmers designing Java Card applets. Due to the limited resources of the smart card (CPU, memories…), most of the tests (the verifier and part of the loader) must be done statically, outside the card. A secure link mechanism allows the card to check the integrity of the cap file; *i.e.*, after having verified the signature, the card can safely assume that the downloaded program has the required properties, and that a valid verifier has checked it. Of course the certificate can only be provided by a trusted third-party authority.
In fact, the security provisions are scattered across different components: a verifier, a converter, an on-card loader, a firewall and an interpreter (see figure 1). Moreover a specific applet is used to manage the applet: the Java Card Runtime Environment (JCRE). It is used to select and deselect applets, and also contain the registers of the selected applets and of the currently active applet.
While the virtual machine insures Java language-level security, the firewall performs additional runtime checks. This mechanism is in charge of the applet isolation and of the control of object accesses. For example, it prevents unauthorized accesses to the fields and the methods of class instances. An applet may share objects with other applets, so the applet firewall must control the access to the shareable interface of these objects. This component is of prime importance for the system security.

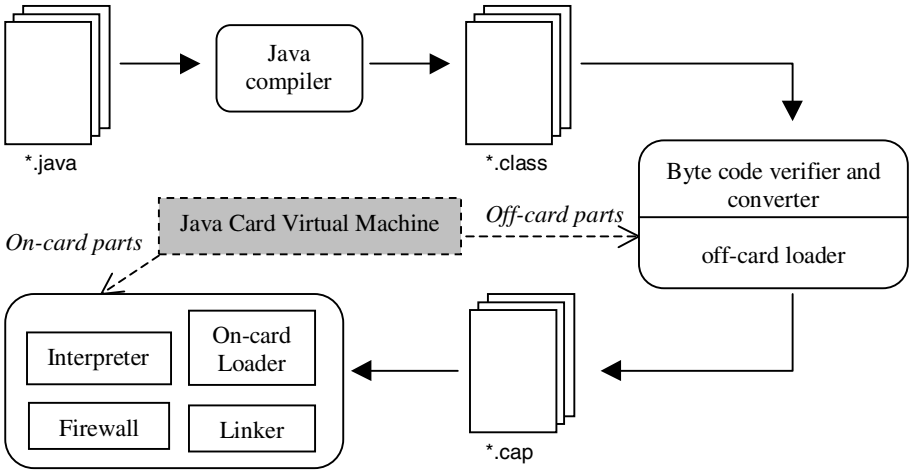



*Fig.1: Java Card environment*

The security policy has to express the correct confinement of the applets and the correct access to shared objects. The respect of the typing rules associated to the access rules of the firewall guarantee this security policy. Thus, we have to verify that the elements performing those checks are correctly implemented and that they are consistent. A formal specification of these mechanisms must be done even if the formal proof is costly. Several elements have already been modelled: the verifier [Cas-99] and partially the JCRE with an emphasis on the firewall [Mot-00]. We present here a method guaranteeing that the security policy is correctly implemented by the different mechanisms.

## 3. Related Work

There has been much work on a formal treatment of Java but no work has been done in order to formally verify whether a given security policy is correctly implemented by a virtual machine. All the works on Java and the Java byte code focus on a formal definition of the semantics. At the Java language level, [Nip-98] and [Sym-97] define a formal semantics for a subset of Java in order to prove the soundness of its type system. [Qia-98] considers a subset of the byte code and aims at proving the runtime correctness from its static typing. Then, he proposes the proof of a verifier that can be deducted from the virtual machine specification.
An interesting work has been done by [Coh-96]. He proposes a formal implementation of a defensive virtual machine. It is possible to prove that his model is equivalent to an offensive interpreter plus a sound byte code verifier. Posegga and Vogt [Pos-98] propose a verification mechanism based on a model checker. They shown the easiness of the proof process using the SMV tool. Goldberg [Gol-97] proposes a formal specification of the byte code verifier for the data flows analysis. His approach is close to the implementation but he simplifies the problem when

neglecting to check subroutines. In the Bali project [Pus-99], Push proves a part of the Java Virtual Machine using the prover Isabelle/HOL. Qian works [Qia-98] gives a specification of the byte code verifier and then proves its correctness.

## 4. The approach used

The main purpose of our approach is to ensure the soundness of the type system. Principles described in [Cas-99] are used to formally specify the Java byte code interpreter. The main idea is to start with a formal description of an abstract defensive byte code interpreter that defines the checks needed to ensure a safe byte code execution. This defensive byte code interpreter defines the expected security policy.

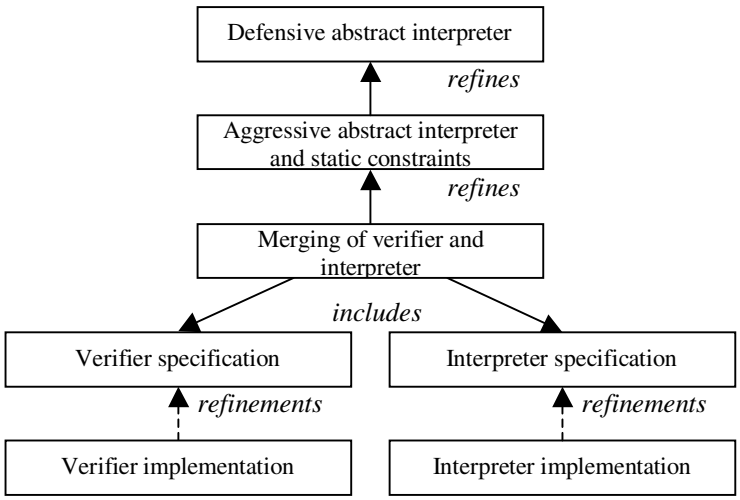



*Fig 2: Overview of the approach*

The runtime checks performed by the defensive interpreter are removed and converted to static constraints on the byte code during the refinement process. During this process, the proof obligations of the refinement ensure the validity of the static constraints specified.
At the last refinement step, the machine is separated in a byte code verifier, which enforces the static constraints, and an aggressive interpreter, corresponding to the implementation of the Java Card virtual machine. The refinement mechanism ensures that the security policy defined in the abstract interpreter is preserved by the aggressive one.
This approach ensures the soundness of the byte code verifier and the interpreter. That is, the byte code interpreter relies only on tests that are performed. Moreover, from the verifier point of view, this proves that the properties verified are enough to guarantee a safe byte code execution: a property that would not have been verified would generate unprovable proof obligations in the verifier part. Lastly, generating

the code for the interpreter and the verifier ensures the correctness of the implementation.
Initially, a small instruction set composed of about ten instructions and a simplified lattice has been used. This approach was adapted to this small instruction set, but extending it to the whole Java Card instruction set did not scale well. More exactly, each instruction needed several manual proofs and both the response time and memory requirement of the prover was too large to completely demonstrate the proofs. The next part focuses on describing how the approach has been extended for a large subset of the Java Card virtual machine.

## 5. Machine considered

### 5.1. Instruction set

The Java Card subset considered consists of all the stack manipulation instructions, most of the control flow instructions and instructions manipulating local variables.
As the aim of this work was to verify the scalability of the approach, instructions that would drastically increase the complexity of the model have been left out. Especially, those instructions include the instructions used for subroutines, for method calls and for objects handling. The difficulties implied by those instructions have already been widely studied, and there are known solutions for handling them. Moreover, those difficulties usually involve few instructions, and are not subject to scalability problems. The handling of exceptions and subroutines will be added later, when the scalability of the model will be resolved. We will use a model developed as an extension of [Lan-98] based on [Aba-98] and very close to [Fre-99].
So, the chosen instruction set is neither representative of the full Java Card instruction set nor representative of the tricky parts of the full instruction set. However, it appears as a valid choice to study the problems that can be encountered when extending a ten instructions subset to the full instruction set.
A subset of instructions manipulating the stack is created. Each of those instructions is considered as first removing elements from the stack, and adding new elements to the resulting stack. For example, the instruction *iadd*, which adds the two topmost elements of the stack together, and replaces them by the result, is considered as being an instruction that pops two integers from the stack and pushes an integer.
To model this, two constants have been added: *tpushed* and *tpopped*. Those constants are defined as partial maps from opcodes to sequence of types. *tpopped* defines the types that are expected to be removed from the top of the stack, and *tpushed* defines the types to be pushed onto the stack when the instruction is executed. In the previous example, *tpushed*(*iadd*) is equal to the one element sequence *[integer]*, and *tpopped*(*iadd*) is equal to the sequence *[integer, integer]*.
In order to simplify the specification and the proof process, the opcodes are grouped by properties. Sets are defined to contain opcodes with similar properties. For example, the following sets are used:

- *OP_NEXT*. This set contains opcodes that can go to the next instruction after execution. This include nearly all the instruction, excepted the unconditional jumps.
- *OP_BRANCH* and *OP_BRANCH_W:* the set of opcodes that may perform a relative branch, where the target is defined by the first parameter. There are two sets, since the branch can be defined by a signed byte parameter (*OP_BRANCH*) or a signed short parameter (*OP_BRANCH_W*)
- *OP_NEXT_FRAME_READ*: the set of opcodes reading a value from the local variables.

A given opcode can be part of several sets. For example, instructions that perform conditional branch are both elements of *OP_NEXT* and *OP_BRANCH*. Although every Java Card opcodes can not fit in a group, such a grouping scheme highly simplifies the specification.
One drawback is that grouping opcodes by properties generates more complicated proof obligations that require increased manual interaction. However, those proof obligations are more generic and can usually be used to discharge nearly all the proof obligations corresponding to the opcodes within the group.

### 5.2. State of the machine

We consider the execution of one method. This is enough to verify the consistency between the interpreter and the verifier. Thus the verification can be performed a method at a time, provided that some information about the global context is accessible.
A set *BYTE* is defined, the method being considered as a sequence of *BYTE*. Since its content does not change during the interpretation, it is defined as a constant. Some additional information on the method is added: *max_stack* corresponds to the maximum size of the local stack during the execution of the method, and *max_local* to the maximum number of local variables used. Lastly, the set *opcode_locations* corresponds to the set of valid adresses within the method. As this last information is not directly available within the classfile, it has to be computed before the method is executed.

$$
\begin{array}{l}
max\_locals \in \mathbf{NAT} \wedge \\
max\_stack \in \mathbf{NAT} \wedge \\
method \in \mathbf{seq1}(BYTE) \wedge \\
opcode\_locations \subseteq \mathbf{dom}(method)
\end{array}
$$

*Fig 3: Constants used to represent a method*

For the most abstract specification, we are only interested in the types contained in the stack and the frame. So, the state consists of:
- the program counter, which points to the instruction currently being executed,
- the typing of the runtime stack,
- the typing of the frame.

This state is defined by the variables shown on figure 4. For now, the variable *frame_type* contains the content of the frame, and is defined as a partial map from

integer to type (more exactly, from the interval 0 to the maximum variable number to type). The variable *stack_type* represents the content of the stack, and is defined as a sequence of types. *apc* is defined as beeing a value in *opcode_locations*, always ensuring the applet confinement. An additional invariant ensures that the stack never overflows.

$$
\begin{array}{l}
frame\_type \in 0..max\_locals\text{-}1 \rightarrow TYPE \wedge \\
stack\_type \in \mathbf{seq}(TYPE) \wedge \\
\mathbf{size}(stack\_type) \leq max\_stack \wedge \\
apc \in opcode\_locations
\end{array}
$$

*Fig 4: Variables representing the state of the machine*

Since we manipulate byte, and not more abstract data types, we need some functions converting bytes to opcodes or values. Figure 5 lists some of the B functions defined. The functions *BYTE_to_signed* and *BYTE2_to_signed* allow converting a byte or a short into a signed value useable within the specification. Those functions are defined as constants, and are used to get the opcodes and the parameters from the method.

$$
\begin{array}{l}
BYTE\_to\_OPCODE \in BYTE \rightarrow OPCODE \wedge \\
BYTE\_to\_signed \in BYTE \rightarrow \mathbf{INT} \wedge \\
BYTE2\_to\_signed : (BYTE \times BYTE) \rightarrow \mathbf{INT}
\end{array}
$$

*Fig 5: Functions handling byte conversions*

## 6. The defensive interpreter

The defensive interpreter performs an abstract execution of the method, and ensures that every instruction can be executed in a safe way by runtime tests. Each Java opcode has an associated B operation describing the expected semantics.
To simplify the specification, a few more definitions have to be introduced (Fig 6).

$$
\begin{array}{l}
opcode(pc) == BYTE\_to\_OPCODE(method(pc)); \\
parameter(pc, xx) == method(pc+xx); \\
succ\_pc(pc) == pc + parameters\_size(opcode(pc)) + 1; \\
can\_update\_stack(pc) == \mathbf{size}(stack\_type) \geq \mathbf{size}(tpopped(opcode(pc))) \wedge \\
\qquad \mathbf{size}(stack\_type)\text{-}\mathbf{size}(tpopped(opcode(pc)))+\mathbf{size}(tpushed(opcode(pc))) \\
\qquad\qquad \leq max\_stack \wedge \\
\qquad stack\_type \uparrow \mathbf{size}(tpopped(opcode\ (pc))) = tpopped(opcode\ (pc))
\end{array}
$$

*Fig 6: Definitions*

The first definition corresponds to a function returning the opcode for the specified location in the method. The second one is used to access parameters associated to opcodes. The next one computes the address of the next instruction based on the

number of additional parameters of the opcode. The last definition is a predicate ensuring that the stack can be updated according to the definition of the current opcode. That is, it ensures that the execution of the instruction will not introduce stack underflow or overflow, and that the types expected are present on top of the stack.
To specify the operations, we use event driven B, and associate a guard corresponding to the expected opcode of the operation. The operation will be triggered when the guard is true, that is, when the corresponding opcode is encountered.
Each operation performs tests ensuring that it can safely be executed and then updates the state of the machine. For example, the specification of the *iload* instruction, which loads an integer local variable onto the stack is given figure 7.

```
op_iload=
SELECT
        opcode(apc) = ILOAD
THEN
        IF
                BYTE_to_unsigned(parameter(apc, 1)) ∈ 0..max_locals-1 ∧
                frame_type(BYTE_to_unsigned(parameter(apc, 1))) =
                        frame_type_used(opcode(apc)) ∧
                succ_pc(apc) ∈  opcode_locations ∧
                can_update_stack(apc)
        THEN
                apc := succ_pc(apc) ||
                stack_type := tpushed(opcode(apc))
                                ^(stack_type↓size(tpopped(opcode(apc))))
        END
END;
```

*Fig 7: Specification of the operation corresponding to the iload opcode*

In this example, the content of the *SELECT* clause means that this operation will be triggered when an *iload* opcode is encountered within the method. Then, the tests within the *IF* clause correspond to the runtime tests performed when executing the instruction: the two first checks ensure that the local variable exists and is defined, and that the types it uses match with the expected types, ensuring correct typing. The next checks ensure the confinement of the applet execution, by testing if the program counter is still within the method body after the operation is performed. The last check tests for the stack underflow and overflow, and ensures that the types expected within the stack match with the types found.
As this defensive interpreter only operates on types, its specification cannot be deterministic: some instruction behaviour may depend on the values stored in the stack or within the variables. An example of this is the instructions performing conditional branch depending on stack values. As only the type of those values is known, it isn't possible to decide if the branch is taken. Instead, it is specified that, either the jump is performed, either the execution continues to the next instruction. The specification of the *ifle* instruction is given on figure 8.

```
op_ifle=
SELECT
        opcode(apc) = IFLE
THEN
        CHOICE
                IF
                        succ_pc(apc) ∈  opcode_locations ∧
                        can_update_stack(apc)
                THEN
                        apc := succ_pc(apc) ||
                        stack_type := tpushed(opcode(apc))
                            ^(stack_type↓size(tpopped(opcode(apc))))
                END
        OR
                IF
                        apc + parameter(apc, 1) ∈  opcode_locations ∧
                        can_update_stack(apc)
                THEN
                        apc := apc + parameter(apc, 1) ||
                        stack_type := tpushed(opcode(apc))
                            ^(stack_type↓size(tpopped(opcode(apc))))
                END
        END
END;
```

*Fig 8:Specification of the operation corresponding to the ifle opcode*

The B substitution CHOICE represents a non-deterministic choice. The first part of the clause represents the case where the execution continue to the next instruction, and the second to the case where the execution continue to the branch target.
Determinism will be added within the interpreter specification, since the values stored within the stack are not available before.

## 7. Replacement of runtime tests by static properties

### 7.1. Introduction of new variables

Replacing the runtime checks by static properties implies adding additional information about the method. Especially, we need to know the typing content of the stack, and the type of the potentially used local variables for each instruction. This information is provided by a type inference performed by the verifier. It is possible to infer this information, because a valid Java program has to be verifiable in a finite time [Lin-96]. The verifier would reject any program where this information could not be computed.

Two new variables are introduced (figure 9): *stack_type_s* and *frame_type_s,* representing the result of the type inference. For each instruction of the method, they define the expected content of the stack and the frame. These variables are linked to the state of the interpreter, by stating that the current state of the interpreter must match the expected state.

$$
\begin{array}{l}
frame\_type\_s \in \mathbf{seq}(0..max\_locals\text{-}1 \mapsto TYPE) \wedge \\
stack\_type\_s \in \mathbf{seq}(\mathbf{seq}(TYPE)) \wedge \\
stack\_type\_s(apc) = stack\_type \wedge \\
frame\_type\_s(apc) = frame\_type
\end{array}
$$

*Fig 9: Definition of the static variables*

### 7.2. Definition of the static properties

We currently consider three different static properties. These properties correspond to properties on the control flow (applet confinement), on the stack (correct typing and no underflow/overflow), and on the validity of local variables access (correct typing). These static properties are expressed as invariants of the machine, by predicates linking the state of the interpreter before execution of an instruction to its state after execution.
The confinement property is expressed by defining properties that must be enforced for opcodes of different groups.

$$
\begin{array}{l}
static\_flow\_checked == \\
\forall pc.((pc \in \mathbf{dom}(method) \wedge opcode(pc) \in OP\_NEXT) \\
\quad \Rightarrow \\
\quad \text{succ\_pc}(pc) \in opcode\_locations\ ) \wedge \\
\forall pc.((pc \in \mathbf{dom}(method) \wedge opcode(pc) \in OP\_BRANCH) \\
\quad \Rightarrow \\
\quad pc + BYTE\_to\_signed(method(pc+1)) \in opcode\_locations) \wedge \\
\forall pc.((pc \in \mathbf{dom}(method) \wedge opcode(pc) \in OP\_BRANCH\_W) \\
\quad \Rightarrow \\
\quad pc + BYTE2\_to\_signed(method(pc+1), method(pc+2)) \in opcode\_locations)
\end{array}
$$

*Fig 10: Static properties for confinement*

The stack properties are expressed in a similar way. They relate the content of the static typing stacks before the instruction to the content of those stacks after the instruction is executed. For example, in the case of branching opcode, it is stated that:

- the size of the stack after the execution of the instruction is less than *max_stack*,
- the stack does not underflow during the execution of the instruction,
- the resulting stack does not underflow,
- the static stack for the branch target matches with the resulting stack.

The property associated to the stack for branching opcodes are given on figure 11.

$$
\begin{array}{l}
static\_stack\_checked == \\
\ldots \\
\forall pc.((pc \in \mathbf{dom}(method) \wedge opcode(pc) \in OP\_BRANCH) \\
\Rightarrow \\
-max\_stack \leq \ \mathbf{size}(stack\_type\_s(pc)) + \mathbf{size}(tpopped(opcode\ (pc))) \\
\qquad\qquad - \mathbf{size}(tpushed(opcode(pc))) \wedge \\
\mathbf{size}(tpopped(opcode(pc))) \leq \mathbf{size}(stack\_type\_s(pc)) \wedge \\
0 \leq max\_stack - \mathbf{size}(stack\_types\_s(pc)) + \mathbf{size}(tpopped(opcode(pc))) \\
\quad - \mathbf{size}(tpushed(opcode(pc))) \ \wedge \\
stack\_type\_s(pc) \uparrow \mathbf{size}(tpopped(opcode(pc))) = tpopped(opcode(pc)) \wedge \\
stack\_type\_s(pc+1+BYTE\_to\_signed(method(pc+1))) = \\
\qquad tpushed(opcode(pc)) \ ^\wedge \ (stack\_type\_s(pc) \downarrow \mathbf{size}(tpopped(opcode(pc)))) \wedge \\
\ldots
\end{array}
$$

*Fig 11: Stack property for branching opcodes*

Note that the inequalities describing the size of the stack are written in such a way that they are suitable to the normalisation used by the prover. Although the specification is less straightforward to read, proving its correctness is far easier. For example, in some cases, the number of commands needed to achieve the proof can be divided by more than two.

The last set of properties ensures the consistency of the frame accesses. It uses functions similar to *tpopped* and *tpushed*: *frame_type_used* to get the expected type of the local variable used.

$$
\begin{array}{l}
static\_frame\_checked == \\
\forall pc.((pc \in \mathbf{dom}(method) \wedge opcode(pc) \in OP\_NEXT\_FRAME\_READ) \\
\Rightarrow \\
BYTE\_to\_unsigned(method(pc+1)) \in 0..max\_locals\text{-}1 \wedge \\
frame\_type\_s(pc)(BYTE\_to\_unsigned(method(pc+1))) = frame\_type\_used(opcode(pc)) \wedge \\
frame\_type\_s(pc+1+parameters\_size(opcode(pc))) \subseteq frame\_type\_s(pc))
\end{array}
$$

*Fig 12: Frame property for opcodes reading the frame*

Three boolean variables are defined: *flow_checked, stack_checked* and *frame_checked*. Those variables correspond to the result of the verifier, and are set to true only if the program has the corresponding property. Invariants are added to link those values to the static properties defined as shown on figure 13.

$flow\_checked \in \mathbf{BOOL} \wedge$
$(flow\_checked = \mathbf{TRUE} \Rightarrow static\_flow\_checked) \wedge$

$stack\_checked \in \mathbf{BOOL} \wedge$
$(stack\_checked = \mathbf{TRUE} \Rightarrow static\_stack\_checked) \wedge$

$frame\_checked \in \mathbf{BOOL} \wedge$
$(frame\_checked = \mathbf{TRUE} \Rightarrow static\_frame\_checked)$

*Fig 13: Invariant defining static properties*

The specification of the operations is nearly the same as previously. The difference is that tests against the values of the checks variable are placed within the guard, and that the dynamic tests are removed. For example, the specification of the iload operation is given on figure 14.

**op_iload=**
**SELECT**
$opcode(apc) = ILOAD \wedge flow\_checked \wedge stack\_checked \wedge frame\_checked$
**THEN**
$apc$ **:=** succ_pc($apc$) ||
$stack\_type$ :=
$tpushed(opcode(apc))^{\wedge}(stack\_type \downarrow \mathbf{size}(tpopped(opcode(apc))))$
**END**;

*Fig 14: Specification of the iload opcode*

The refinement mechanism ensures that every refined operation can occur only in a state corresponding to one in which the abstract operation could occur, and that the refined operation behaves as the abstract operation. So, proving that the new specification is a valid refinement of the defensive interpreter ensures the soundness of the byte code verifier and the interpreter.
The main difference between the defensive interpreter and the refined interpreter, apart the fact that no runtime tests are performed is that there is not a strict correspondence between the operations triggered by the defensive interpreter and the refined one. If the method can be checked, then the operations triggered will be the same as the abstract ones. However, if the method contain an error, the abstract operations will be called until the program counter reach the error, but no refined operation will be called at all.

## 8. Inclusion of the verifier and the interpreter

This refinement is mainly used to include both the verifier and a "real" interpreter. By real, we mean an interpreter that does not perform an abstract interpretation of the method based on the types of the values, but only uses values.

### 8.1. Verifier specification

The verifier specification contains only one operation, which performs the byte code verification, and returns a boolean value corresponding to the result of the verification. The specification of the *verify_method* corresponding to the previously described properties is given on figure 15.

```
flow_ok, stack_ok, frame_ok ← verify_method =
   ANY fl_ok, st_ok, fr_ok WHERE
         fl_ok ∈ BOOL ∧ st_ok ∈ BOOL ∧ fr_ok ∈ BOOL ∧
         (fl_ok = TRUE ⇒ static_flow_checked) ∧
         (st_ok = TRUE ⇒ static_stack_checked) ∧
         (fr_ok = TRUE ⇒ static_frame_checked)
   THEN
         flow_ok, stack_ok, frame_ok := fl_ok, st_ok, fr_ok
   END
```

*Fig 15: Specification of the verify_method operation*

The verifier machine is included in the refinement, and called during the initialisation to define the values of the variables *flow_checked*, *stack_checked* and *frame_checked* as shown on the following figure.

```
INITIALISATION
         flow_checked, stack_checked, frame_checked ← verify_method ||
         …
```

*Fig 16: Call of the verify_method operation*

The implementation of the verifier performs the type inference using a fixpoint computation as described in [Cas-99]. The presence of embedded loops increases the difficulty of the proof process. Splitting the implementation in several small operations allows the automatic prover to discharge up to 95% of the proof obligations. However, proving the remaining 5% proof obligations is still costly.

### 8.2. Interpreter specification

The interpreter is defined as a machine similar to the abstract interpreter, excepted that it is an aggressive interpreter, and that it operates on values instead of types. Its state consists of a pointer to the current instruction executed (*dpc*, for dynamic

program counter), the values stored in the stack (*stack_value*) and the values stored in the frame (*frame_value*).
To ensure the consistency between the abstract interpreter and the concrete interpreter, we have to glue the state of the abstract interpreter to the state of the concrete interpreter using additional invariants. For the stack, it is ensured that both the stack containing the values and the stack containing the types have the same size. That is, every defined value has a type, and every type has a value. The invariant relating the types frame to the values frame is not as simple: it is stated that the domain of the typing frame has to be included within the domain of the value frame. That is, every variable that may be used is defined. The domain *value_frame* can be larger than the domain of *type_frame*, since every local variable has a value even if its type is not defined. Last, the current instruction executed must be the same for both interpreters. Those three invariants, shown on figure 17 ensure that we have not specified two different and unrelated interpreters.

$apc = dpc \wedge$
$\mathbf{size}(stack\_type) = \mathbf{size}(stack\_value) \wedge$
$\mathbf{dom}(frame\_type) \subseteq \mathbf{dom}(frame\_value)$

*Fig 17: Gluing of the interpreter*

The guards corresponding to the operations are unchanged. However the body of the operation now only calls the associated operation of the interpreter. For example, figure 18 shows the operation *op_iload*, that calls the corresponding operation *int_iload* of the interpreter.

**op_iload=**
**SELECT**
$opcode(apc) = ILOAD \wedge flow\_checked \wedge stack\_checked \wedge frame\_checked$
**THEN**
*int_iload*
**END**;

*Fig 18: iload operation for the second refinement*

*int_iload* is the operation corresponding to the opcode iload within the interpreter machine (figure 19). It pushes the value contained in the specified local variable onto the stack. As this interpreter is implemented in a separate machine that has no knowledge of the constraints enforced on the byte code, the preconditions ensuring that the execution can be performed have to be provided. Preconditions are specification substitutions that specify the conditions that have to be true when the operation is called. They are used to generate proof obligations, and to achieve the proof.
The consistency between those preconditions and the byte code verification is ensured by the proof obligations generated when the operation *int_iload* is called from the operation *op_iload*: it will be needed to prove that the content of the *op_iload* guard implies the *int_iload* precondition.

```
int_iload=
PRE
        succ_pc(dpc) ∈ opcode_locations ∧
        size(stack_value) < max_stack ∧
        BYTE_to_unsigned(parameter(1)) ∈ dom(frame_value)
THEN
        dpc := succ_pc(dpc) ||
        LET var_value BE
                var_value = frame_value(BYTE_to_unsigned(parameter(1)))
        IN
                stack_value := var_value → stack_value
        END
END;
```

*Fig 19: iload operation for the interpreter*

Another point is that, instead of using a different machine, the interpreter could have been treated as a refinement of the abstract defensive machine, in a way similar to what has been done in [Lan-98]. However, separating the interpreter from the abstract specification seems to be a better solution, since less proof obligations will be generated: proofs are needed when the interpreter is included within the refinement, but not in later refinements of the interpreter, allowing to focus on the interpreter implementation. Moreover, implementing the interpreter as distinct machines allows to clearly separate the proof of consistency and the implementation.

## 9. Proof of the specification

The specification of the defensive virtual machine and its refinement is about 10000 lines of B specification. The *Atelier B* tool, that we used for this specification generates nearly 3000 proof obligations. It should be noted, however that the proofs are not complicated by themselves. The main difficulty lies in their number: proving the correctness of the specification corresponds to discharge a lot of simple proof obligations.

For this specification, the main goal is to limit the cost of the proof process. We focus on obtaining similar proof obligations, so that a single demonstration could be used to demonstrate several similar proof obligations. This is achieved by specifying opcodes properties and constraints in a generic way. This involves grouping opcodes by properties, but also using generic description. For example, using the functions *tpushed* and *tpopped* allows specifying nearly all operations that manipulate the stack the same way.

To illustrate the advantages of using generic specification, figure 20 presents two simplified proof obligations, the first corresponding to a specification that does not groups opcodes, and the second to the specification previously described.

Discharging the proof obligation without groups is quite straightforward: it involves using hypothesis *(1.2)* and *(1.3)* with hypothesis *(1.1)*. However, in the Java Card

case, there will be one hypothesis similar to *(1.1)* by opcode, and the automatic prover will not be able to choose the right one, requiring user interaction. Moreover, as the opcode considered is explicitly used, this interaction will be required for every opcodes. For the complete Java Card interpreter, this means that proving this property for each opcode will need approximately two hundred different, but very similar proofs with user interaction.

| *PO without groups* | *PO with groups* |
|---|---|
| *(1.1)* $\forall pc.((pc \in \mathbf{dom}(method) \wedge$<br>$opcode(pc) = ILOAD)$<br>$\Rightarrow$<br>$pc + 1 \in opcode\_locations) \wedge$<br>*(1.2)* $opcode(apc) = ILOAD \wedge$<br>*(1.3)* $apc \in \mathbf{dom}(method)$<br>$\Rightarrow$<br>$apc + 1 \in opcode\_locations$ | *(2.1)* $OP\_NEXT = \{ ..., ILOAD, ... \} \wedge$<br>*(2.2)* $\forall pc.((pc \in \mathbf{dom}(method) \wedge$<br>$opcode(pc) \in OP\_NEXT)$<br>$\Rightarrow$<br>$succ\_pc(pc) \in opcode\_locations) \wedge$<br>*(2.3)* $opcode(apc) = ILOAD \wedge$<br>*(2.4)* $apc \in \mathbf{dom}(method)$<br>$\Rightarrow$<br>$succ\_pc(apc) \in opcode\_locations$ |

*Fig 20: Comparison between proof obligations with and without using groups*

In the case where groups are used, the hypothesis $opcode(apc) \in OP\_NEXT$ *(2.5)* can be added. This hypothesis is automatically accepted by the prover thanks to *(2.1)* and *(2.3)*, and user interaction will not be needed. The new hypothesis *(2.5)* can then be used with hypothesis *(2.2)* to discharge the goal. The important point is that the commands used to demonstrate this goal does not consider the opcode names, and can be directly reused to prove similar proof obligations for opcodes that are elements of the set *OP_NEXT*. This means that there will be one user interaction for nearly two hundred proof obligations. This is all the more important, since the response times of the interactive prover can be very large for such a specification. However, those gains have to be balanced by the fact that the proof obligations are often more complicated to prove, and the initial proof can take some time to be carried out. Moreover, all the Java Card opcodes can not fit in a group, and some opcodes will still need to be treated as special cases.
Another important point with opcode groups is that it also reduces the number of predicates within the invariant. This reduction drastically increases the performance of the tool.

## 10. Conclusion

Proving the correctness and the soundness of the type system is a first step to a certification of Java Card. Other parts of the security policy are implemented by different functions such as the firewall, that controls access policies. As one of the common criteria requirements is to guarantee the coherence of all the security mechanisms, it is needed to integrate this model into a more generic model encompassing the whole security policies.

Future works will focus on integrating the firewall specification defined in [Mot-00] with the interpreter. Then, the model will be extended in order to model the complete Java Card interpreter. This will allow, not only to prove the soundness of the byte code verifier and of the interpreter, but also will ensure the correctness of their implementation.

**Acknowledgement:** Thanks to *G. Mornet* and *L.Casset* for their work on the model, discussions and feedback.

## References:

[Aba-98] M. Abadi, R. Stata, *A Type System for Byte Code Subroutines*, Proc. 25th ACM Symposium on Principle of Programming Languages, January 1998

[Cas-99] L. Casset, J.-L. Lanet, *A Formal Specification of the Java Byte Code Semantics using the B method*, Proceedings of the ECOOP'99 workshop on Formal Techniques for Java Programs, June 1999.

[Coh-96] Cohen, *Defensive Java Virtual Machine Specification*, http://www.cli.com/software/djvm

[Fre-99] S.N. Freund, J.C. Mitchell, *Specification and Verification of Java Bytecode Subroutines and Exceptions*, Stanford Computer Science Technical Note, August 1999

[Gol-97] A. Goldberg, *A Specification of Java Loading and Byte Code Verification*, Kestrel Institute, December 1997, http://www.kestrel.edu/HTML/people/goldberg

[Lan-98] J.L. Lanet, A. Requet, *Formal Proof of Smart Card Applets Correctness,* Proceedings of the Third Smart Card Research and Advanced Application Conference (CARDIS'98), Louvain-la-Neuve, Belgium, September 1998

[Lin-96] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, Addison Wesley, 1996

[Mot-00] S. Motré, *Formal Proof of the Applet Firewall*, AFADL 2000, Grenoble, France, February 2000.

[Nip-98] T. Nipkow, D. Oheimb, *Javalight is Type-Safe - Definitely*, 25th ACM Symposium on Principle of Programming Languages, January 1998.

[Pus-99] C. Pusch, *Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle / HOL*, TACAS 1999, http://www.in.tum.de/~pusch/

[Pos-98] J. Posegga, H. Vogt, *Byte Code Verification for Java Smart Cards Based on Model Checking*, 5th European Symposium on Research in Computer Security (ESORICS 98), Springer LNCS 1998.

[Qia-98] Z. Qian, *Least Types for Memory Locations in Java Byte Code,* Kestrel Institute, Technical Report, 1998.

[Sun-99] Sun Microsystems, *Java Card 2.1 Virtual Machine Specification*, March 1999.

[Sym-97] D. Syme, *Proving Java Type Soundness*, Technical Report, University of Cambridge, Computer Laboratory, 1997.

# Applying Formal Methods to Industrial Cases: The Language Approach

*(The Production-Cell and Mode-Automata)*[*]

Florence Maraninchi Yann Rémond

VERIMAG[†]– Centre Equation, 2 Av. de Vignate – F38610 GIERES
http://www-verimag.imag.fr/PEOPLE/Florence.Maraninchi
(Florence.Maraninchi|Yann.Remond)@imag.fr
Fax : (33) 4.76.63.48.50

**keywords**

Real-time systems, safety-critical, regulation systems, running-modes, language design and implementation, case-study, production cell

**Abstract**

In this paper we comment on the "language approach" to applying formal methods to real industrial problems. Our opinion is that it is always a good idea to let the user tell as much as he knows about the structure of a complex system. When he has a given structure in mind but needs to encode it into the available constructs of a language, the interesting information is likely to be lost somewhere on the way from the original design to the actual implementation. This may have consequences on the efficiency of the code produced, or even on the correctness of the design.

Following this idea, the family of *synchronous languages* [BB91] has been very successful in offering domain-specific, formally defined languages and programming environments for safety-critical systems. We are particularly interested in the data-flow language Lustre, well-suited for the description of regulation systems. These systems are often specified using the notion of *running modes*, which appears in informal designs. However, there seemed to exist no language in which the mode-structure of a complex system could be expressed directly. We proposed to extend Lustre with a new construct devoted to the description of these *running modes* of regulation systems.

The language extension is based upon the model of *mode-automata* [MR98]. We now have a running implementation of this extension [MRR00], which has been applied successfully to the industrial case-studies of the SYRF project [SYR99], proposed by SAAB M.A. (a temperature regulation system) and Schneider Electric (the control of the starting and shut-down phases in a nuclear plant).

We are now working on a case-study proposed by Aerospatiale (a piece of software of the Airbus A340-600, for the development of which Aerospatiale has chosen SCADE, the commercial version of Lustre), under a non-disclosure agreement. However, some of the ideas that this example already suggested to us can also be illustrated with a simpler example. In this paper we show how to program the production-cell case-study [LL95] using mode-automata (a pure Lustre version, written by Leszek Holenderski at GMD Birlinghoven, appeared in [LL95]). We used the environment simulator in TCL-TK provided by FZI Karlsruhe.

[*]This work has been partially supported by Esprit LTR Project SYRF 22703

[†]Verimag is a joint laboratory of Université Joseph Fourier, CNRS and INPG

# 1 Introduction

Real-time Systems, in particular regulation systems, are often specified using the notion of *running modes*. For instance, the commands of an aircraft may be specified by identifying take-off mode and landing mode; the commands for a robot arm are likely to be completely different when it moves right, and when it starts moving left because it has reached an obstacle, etc. This notion of a running mode appears frequently in informal designs, and we met it several times in the informal documentation of operational industrial critical systems from Schneider Electric, Aerospatiale, etc.

However, at least to our knowledge, there exist no language (be it a formal specification language, or a programming one), in which the mode-structure of a complex system can be expressed *directly*. Hence the mode-structure of the system is usually encoded in a variety of ways, depending on the language used, and on the kind of criteria one wants to improve (efficiency, size of the code for embedded systems, etc.). See [MR98, MRR00] for comments on the notion of mode and related work (Modecharts [JM88], the "state" Design pattern [GHJV95], real-time mode-machines [Pay96], SignalGTI [RM95], etc.)

The family of *synchronous languages* [BB91] has been very successful, over the ten past years, in offering formally defined languages and programming environments for safety-critical systems. We are particularly interested in the language Lustre [CHPP87], and in the industrial version of it, called SCADE and sold by Verilog S.A. Lustre is a data-flow language, well-suited for the description of regulation systems. We proposed to extend Lustre with a new construct devoted to modes in regulation systems. This language extension is based upon the mathematical model of *mode-automata* [MR98]. We now have a running implementation of this extension, by compilation into an intermediate format of the compilation chain from Lustre to imperative sequential code (C, Ada, Java) [MRR00]. The language extension allows flat mode-automata and composed ones. We use the composition operators from Argos [Mar92], which gives the language a hierarchic state-structure like in Statecharts [Har87].

The definition of mode-automata is a result of the task entitled "combination of formalisms" of the SYRF [SYR99] Esprit Project, in which various approaches have been studied. One of them was to describe complex systems partly in Lustre (data-flow declarative style) and partly in Esterel (parallel imperative style), and to perform link-editing at the level of an intermediate format of the compilation chains. To our opinion, this approach is too complex, and that is the reason why we chose to extend Lustre with a bit of imperative style, yet keeping the essential style and structure of the language, for the programming habits not to be modified deeply. An approach similar to ours — tight integration of styles, as opposed to full multi-language programming — is that of synchronousEifel (formerly "The Synchronie Workbench") [sE] developed at the GMD (Sankt Augustin).

In the family of synchronous languages, formal verification [HLR92] and automatic generation of test cases [RWNH98] are based upon the use of so-called *synchronous observers* [HLR93]. An observer $O$ is itself a synchronous program, which can be composed in parallel with a program $P$ to observe, without modifying the behavior of $P$. This is a consequence of the synchronous broadcast communication mechanism (which is asymmetrical), provided the outputs of $O$ are not connected back to the inputs of $P$. For

verification purposes, observers are used to describe the *safety properties* of a program to verify. For generating test sequences, observers are used for both the *oracle* and the *environment*. The environment-observer is used as a generator, for producing only sequences of inputs to $P$, that are *relevant* w.r.t. a model of the physical environment.

Numerous case studies have shown that, when the program is written in Lustre, it is often convenient to write the observers in a more imperative style. For instance, expressing the safety property: "*the outputs a and b alternate*" is easy with a two-states automaton, and a bit more difficult with a Lustre program. A language based on regular expressions has been used (via an efficient translation into Lustre [Ray96]). In this paper, we use mode-automata for both the controller and the model of the environment. We could use them for describing safety properties as well.

The rest of the paper is organized as follows: Section 2 is a brief introduction to data-flow synchronous languages and the mode-automaton model; Section 3 briefly recalls the production-cell case-study ; Section 4 describes the program written using composed mode-automata. Section 5 concludes and gives some directions for further work.

# 2 Data-flow Synchronous Languages and Mode-Automata

## 2.1 Data-flow Synchronous Languages

In a data-flow language for reactive systems, both the inputs and outputs of the system are described by their *flows* of values along time. Time is discrete and instants may be numbered by integers. If `x` is a flow, we will note $x_n$ its value at the $n$th reaction (or $n$th *instant*) of the program.

A program consumes *input* flows and computes *output* flows, possibly using *local* flows which are not visible from the environment. Local and output flows are defined by *equations*. An equation "`x = y + z`" defines the flow `x` from the flows `y` and `z` in such a way that, at each instant $n$, $x_n = y_n + z_n$.

A set of such equations, using arithmetic, Boolean, etc. operators, describes a network of operators, and is essentially equivalent to the description of a combinational circuit. The same constraints apply: one should not write sets of equations with instantaneous loops, like : $\{x = y + z, z = x + 1, ...\}$. This is a set of fix point equations that perhaps has solutions, but it is not accepted as a data-flow program. For referencing the *past*, the operator `pre` is introduced : $\forall n > 0, (\texttt{pre}X)_n = X_{n-1}$.

One typically writes `T = pre(T) + i ;` , where `T` is an output, and `i` is an input. It means that, at each instant, the value of the flow `T` is obtained by adding the value of the current input `i` to the previous value of `T`. Initialization of flows is provided by the `->` operator. The equation `X = 0 -> pre(X) + 1` defines the flow of integers; as a reactive program, it produces values on the *basic clock*.

The language is structured by the definition of reusable *nodes* that can be called anywhere in expressions defining variables, and programs usually input a library of small well-identified reactive behaviors, like a "two-states" with reset, a "bounded counter", etc.

## 2.2 Motivations for Mode-Automata

In a data-flow language, the notion of *running mode* corresponds to the fact that there may exist several definitions (equations) for the same output, that should be used in distinct periods of time. Faced with this kind of system, users usually write Lustre programs in which modes are encoded by Boolean flows, and the outputs that depend on modes are described by equations of the following form: *X = if (mode1) then ... else if (mode2) then ...* There was an obvious need for something more readable and modifiable than this encoding of modes by conditional structures.

Another important motivation has to do with code efficiency. In reactive systems, the base clock is imposed by the environment: it should be quick enough in order not to miss some relevant changes in the environment signals sensed by the system. For this base clock to be respected by the actual implementation of the system, the code of the reactive kernel should execute in less than a clock period. Hence there are strong constraints on the sequential code produced from a synchronous language.

The natural translation of a simple data-flow synchronous program into sequential code yields a program in which all nodes of the data-flow network do perform computations at each step of the base clock. In particular the `IF` is strict: in the program *X = if (mode1) then expr1 else if (mode2) then expr2 else ...* both `expr1` and `expr2` are computed at each step, before choosing one of them according to the mode. If `X` has different equations depending on the current mode, it is not a good idea to compute all equations at each step.

It appears that, in critical cases, users would like to put some of their knowledge about the running modes of the system, into the corresponding data-flow programs. Doing so, they hope that a compiler be able to generate more efficient code, namely some code in which not all of the data-flow network nodes work at each step. If they simply encode modes into conditionals, there is no hope to obtain better code. The only way of specifying that parts of the data-flow network should *not* perform computations, for some given steps, is to use the *clock* language feature, but it is not so easy to describe modes using clocks. That is the reason why we propose a new language feature for talking about exclusive modes in a data-flow language. It can be viewed as a high-level construct that offers part of the clock feature, but is easier to use when the system clearly has running modes.

## 2.3 Mode-Automata: A Proposal

### 2.3.1 Flat mode-automata

Suppose we need to program in Lustre the behavior of an output variable $X$ (the command to an actuator, for instance), as specified by the timing diagram of figure 1; in this simple case the behavior only depends on time (no explicit inputs are specified).

We identify three *running modes*: in the first one, $X$ increases by 1 at each step; in the second one, it increases by 2; in the third one, it decreases by 3, and then back to the first mode. We would like to give separately :
— The behavior of $X$ in the first mode : `X = pre (X) + 1 ;`
— The behavior of $X$ in the second mode : `X = pre (X) + 2 ;`

— The behavior of $X$ in the third mode : `X = pre (X) - 3 ;`
— The global initial value of $X$ (0 in the example)
— The way these three modes are organized into the global behavior of the system, in particular the conditions for changing modes.

This is exactly what you can do with a mode-automaton (see Figure 2). Notice that, in real cases, the behavior of the system in a given mode is likely to be a large system of equations, while the mode-structure remains relatively simple.

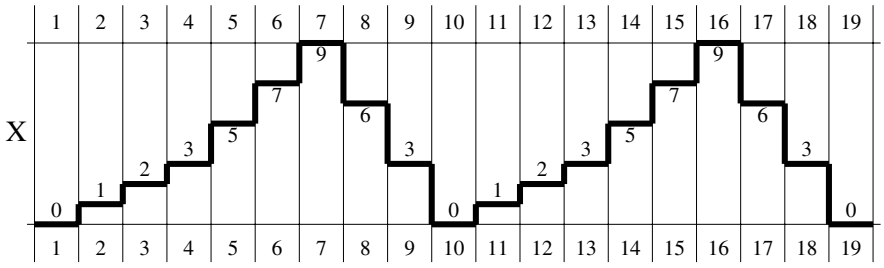



Figure 1: An example with modes: the timing behavior of an integer variable $X$

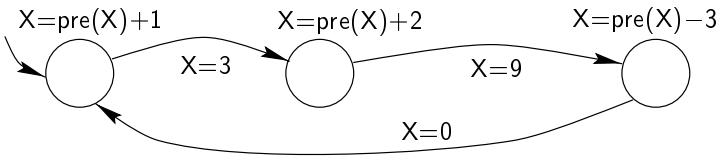



Figure 2: An example with modes: a mode-automaton for computing $X$

Mode-automata can be considered as a discrete version of hybrid automata [MMP91], in which the states are labeled by systems of differential equations that describe how the continuous environment evolves. In our model, states represent the running *modes* of a system, and the equations associated with the states could be obtained by discretizing the control laws. Mode-automata are designed as the basis of a *programming* (not only *specification*) language.

Note that, if we use no `pre` operator and do not mention the input variables in the equations attached to states, then the mode-automaton is merely a Moore machine. Testing inputs is limited to the conditions of the transitions, and equations of the form: `X = true` or `X = false` are attached to states, for defining an output `X`. It may be a bit more complex, because the set of outputs may be defined by a set of equations, like: `X = Y+1 ; Y = 0 ; Z = X − Y ;`, provided there is no dependency cycle. However, the behavior is essentially that of a Moore machine, with the usual one-instant delay between inputs and the actual influence on outputs. For sampled systems, the delay is not important.

### 2.3.2 Composing mode-automata

Mode-automata can be composed in parallel, with shared variables (in the graphical syntax we use the Statecharts notation, with a dashed line separating the components). When $n$ components are in parallel, sharing the variable $X$, at most one component may define $X$; all the other components may only read it. We also forbid instantaneous loops. The parallel composition with shared variables corresponds to the classical data-flow connection of Lustre nodes, where wires are identified according to their names: if the input of a component and the output of another one have the same name, they are connected together.

The semantics of parallel composition is easy to understand, by explaining how to *flatten* a composed mode-automaton into a flat one. Let us consider two mode-automata $M1$ and $M2$. The set of modes of their parallel composition is the Cartesian product of the sets of modes of $M1$ and $M2$. The set of equations attached to a composed mode $A1A2$ (where $A1$ is a mode in $M1$ and $A2$ is a mode in $M2$) is the union of the equations attached to $A1$ in $M1$ and those attached to $A2$ in $M2$. The guard of a composed transition is the conjunction of the guards of the component transitions. The parallel composition of two mode-automata is correct if all the Lustre programs attached to the flat modes are correct, i.e., there is no instantaneous dependency loop, and each variable has exactly one equation.

Mode-automata can also be *refined*: a composition of mode-automata may be put inside the state of a mode-automaton (see examples in the case-study). The flattening is as follows: the equations attached to the refined state are distributed to all the sub-states; the transitions sourced in a refined state also apply to all the states inside; a transition that enters a refined state should go to the initial state (among all the states inside); a transition between two states inside may happen only if no transition from the refined state can fire (the outermost transitions have priority). The correctness of a refined mode-automaton is also related to the correctness of all the Lustre programs attached to the flat states; for this reason, a variable may not be defined at two distinct levels, because the flattening will yield a flat state with which two equations defining the same variable are associated.

### 2.3.3 Implementation

Mode-automata are described using a textual syntax, and compiled into DC code by the tool MATOU [Rém99]. DC is then translated into C using the DRAC tool-set developed in the SYRF project [SYR99]. DC is a data-flow language with *activation conditions*, that allow to specify that some sub-networks are not always alive. This notion is exactly what we need for implementing modes.

The current implementation of mode-automata guarantees the following property: *The C code corresponding to the equations attached to a flat state $S$, and to the conditions of the transitions sourced in $S$, are computed exactly when this state $S$ is active* [MRR00].

In the code produced for the small 3-modes example, we find the following piece of code for computing the new value of `X` at each step, depending on the previous value `pX`: `if (mode==1) { X = pX+1 ; } else if (mode==2) { X=pX+2 ; } else { X=pX-3;}`.

When large pieces of code are attached to modes, this yields a significant improvement on the code obtained from a pure Lustre version of the system, which would have the following form: `{ X1 = pX+1 ; X2 = pX+2 ; X3 = pX-3; if (mode==1) { X = X1 ; } else if (mode==2) { X=X2 ; } else { X=X3 ; }`

When we have the mode-structure of the system in mind, the code produced by MATOU is clearly the best sequential code we can hope for.

# 3 The Production-Cell case-study

In this section, we quote the technical report on the production cell, for a brief presentation of the case-study.

In order to demonstrate the benefits of formal methods for industrial applications, and to evaluate and compare existing approaches for constructing and verifying control software for reactive systems, FZI launched the Case Study Production Cell in 1993 as an activity inside the German Korso Project. The architecture of the system is shown on Figure 3.

On the bottom left the feed belt is shown which conveys the blanks to an elevating rotary table. This table has to be between the feed belt and the robot to bring the blanks into the right position so that the robot can pick them up. To increase the utilization of the press, the robot is fitted with two arms — one always used for loading, the other one for unloading the press. The two belts are not at the same vertical position; both the press and the rotary table can move vertically.

In order to perform demonstrations of the graphic visualization of the toy model, the production sequence should be able to run without an operator. The "forged" metal plates — which the press in the model does not actually modify — are therefore taken from the deposit belt back to the feed belt by a traveling crane, thus making the entire sequence cyclical.

The production cell is composed of 14 sensors and 13 actuators. Actuators can switch motors on or off or change their directions. Sensors return Boolean or continuous values, though the latter can be made discrete to return a few interesting values. The table of Figure 5 gives the list of sensors and actuators, together with the variable names in the mode-automata programs.

In the simulation environment provided by FZI, the belt moves are managed by the TCL-TK part, as a reaction to the controller commands that switch the motors on and off. This simulated environment is intended to be physically relevant. (An example of irrelevant situation would occur with the sensors SBDB and SBFB being true at the same instant, while there is only one object in the plant).

# 4 The Production-Cell and Mode-Automata

The first interesting aspect is the need for a simulated environment. This is usually the case for reactive systems that are used as *controllers* of some physical activity. If

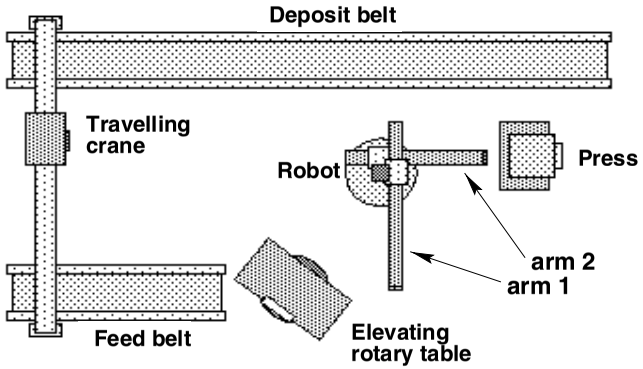



Figure 3: The Production Cell Architecture

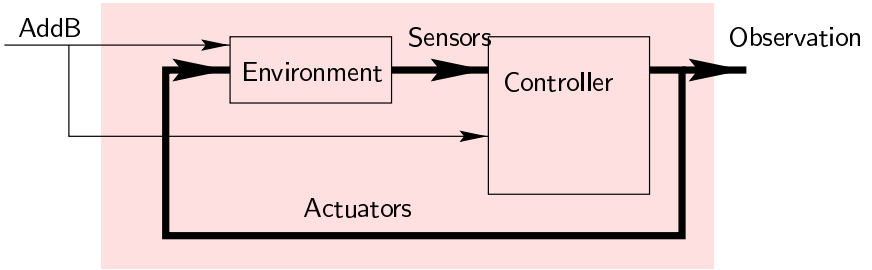



Figure 4: The controller and the simulated environment

| | | | |
|---|---|---|---|
| actH1 | extend and retract 1st robot arm | actH2 | extend and retract 2nd robot arm |
| actPR | move the lower part of the press | actA1 | pick up and drop a plate with 1st arm |
| actA2 | pick up and drop a plate with 2nd arm | actA | rotate robot |
| actRRT | rotate elevating rotary table | actVRT | move elevating rotary table vertically |
| HactC | move gripper of traveling crane horizontally | VactC | move gripper of traveling crane vertically |
| actFB | activate and deactivate feed belt | actDB | activate and deactivate deposit belt |
| actC | pick up and drop a plate with gripper of traveling crane | | |
| PB | Is the press in the lower position? | PM | Is the press in the middle position? |
| PH | Is the press in the upper position? | posH1 | How far has 1st arm been extended? |
| posH2 | How far has 2nd arm been extended? | posA | How far has the robot rotated? |
| TB | Is the elevating rotary table in its lower position? | TH | Is the elevating rotary table in its upper position? |
| posR | How far has the table rotated? | CDB | Is the traveling crane positioned over the deposit belt? |
| CFB | Is the traveling crane positioned over the feed belt? | posVC | What is the current vertical position of the gripper? |
| SBDB | Is there a plate at the extreme end of the deposit belt? | SBFB | Is there a plate at the extreme end of the feed belt? |

| | | |
|---|---|---|
| bool | TM | The rotary tables reaches the max rotation angle |
| bool | T0 | The rotary tables reaches the min rotation angle |
| int | NDB | Number of objects on the deposit belt |
| int | NFB | Number of objects on the feed belt |
| bool | pePR | An object is removed from the PRess |
| bool | ppPR | An object is put on the PRess |
| bool | peRT | An object is removed from the Rotary Table |
| bool | ppRT | An object is put on the Rotary Table |
| bool | peFB | An object is removed from the Feed Belt |
| bool | ppFB | An object is put on the Feed Belt |
| bool | peDB | An object is removed from the Deposit Belt |
| bool | ppDB | An object is put on the Deposit Belt |
| bool | CBas | The Crane is at its lower position |

Figure 5: Interface and Internal signals

we want to perform formal proofs, or to generate test sequences, we need to model the environment. The global picture is that of Figure 4. We built two distinct programs using mode-automata:

- A complete simulation program, comprising the simulation of the physical environment and the controller; in this case, the program we obtain has a single Boolean input AddB, telling it when an object is put on the deposit belt (it is always put at the same place; we should not put more than 5 objects). In this simulated environment, the speed of the belts is supposed to be constant. This program has a cyclic behavior. It can be run with an arbitrary sequence of inputs, and we can save the simulation results for observation or formal analysis purposes. On the other hand, the component that simulates the environment may be used by a tool like Lurette [RWNH98] that generates tests sequences relevant to a given specification of the environment.
- A controller that can be put in the TCL-TK simulated environment (the language of mode-automata is compiled into DC, which is then compiled into C, and the necessary interfacing is done at the C level). The controller written with mode-automata, and the environment simulated in TCL-TK, form a system that has a cyclical behavior. The controller is simply a part of the first specification, in which we removed the components representing the environment. Hence the interface is exactly the set of sensors and actuators of Figure 5, plus the AddB input. The piece of C code that interfaces our controller with the TCL-TK environment generates this input: it is true (meaning that an object is put in the plant) five times at the beginning, and then false forever. We could test other situations, of course. This little reactive behavior could also be described with a mode-automaton.

We cannot explain all the details of the programs in this paper. Our intension is only to show small pieces of programs, in order to illustrate the use of mode-automata. The program that simulates the environment makes use of full-featured mode-automata; the controller itself is almost a Moore machine (see comments on Moore machines being a special form of Mode-automata, in section 2.3.1 above). The automaton structures (and the parallel and hierarchic constructs) are well-suited for the description of the cyclical behavior of the plant.

### 4.1 The controller

The main structure of the data-flow program for the controller is given in Figure 6. The six modules are mode-automata composed in parallel with shared variables; this operation is exactly the data-flow connection as shown on the picture. The meaning of internal signals is shown Figure 5.

Figure 8 shows the rotary table component. Figure 9 shows the traveling crane component. Figure 10 shows the press component. The `Robot` component is the most complex one. It is given in Figure 7. It illustrates the cyclical behavior of the robot, which has two arms, sometimes moving together. The robot task is a cycle, as follows:

— State B : the robot extends first arm, then takes an object on the rotary table, then retracts first arm (necessary before rotating). In this state, the robot must wait for the

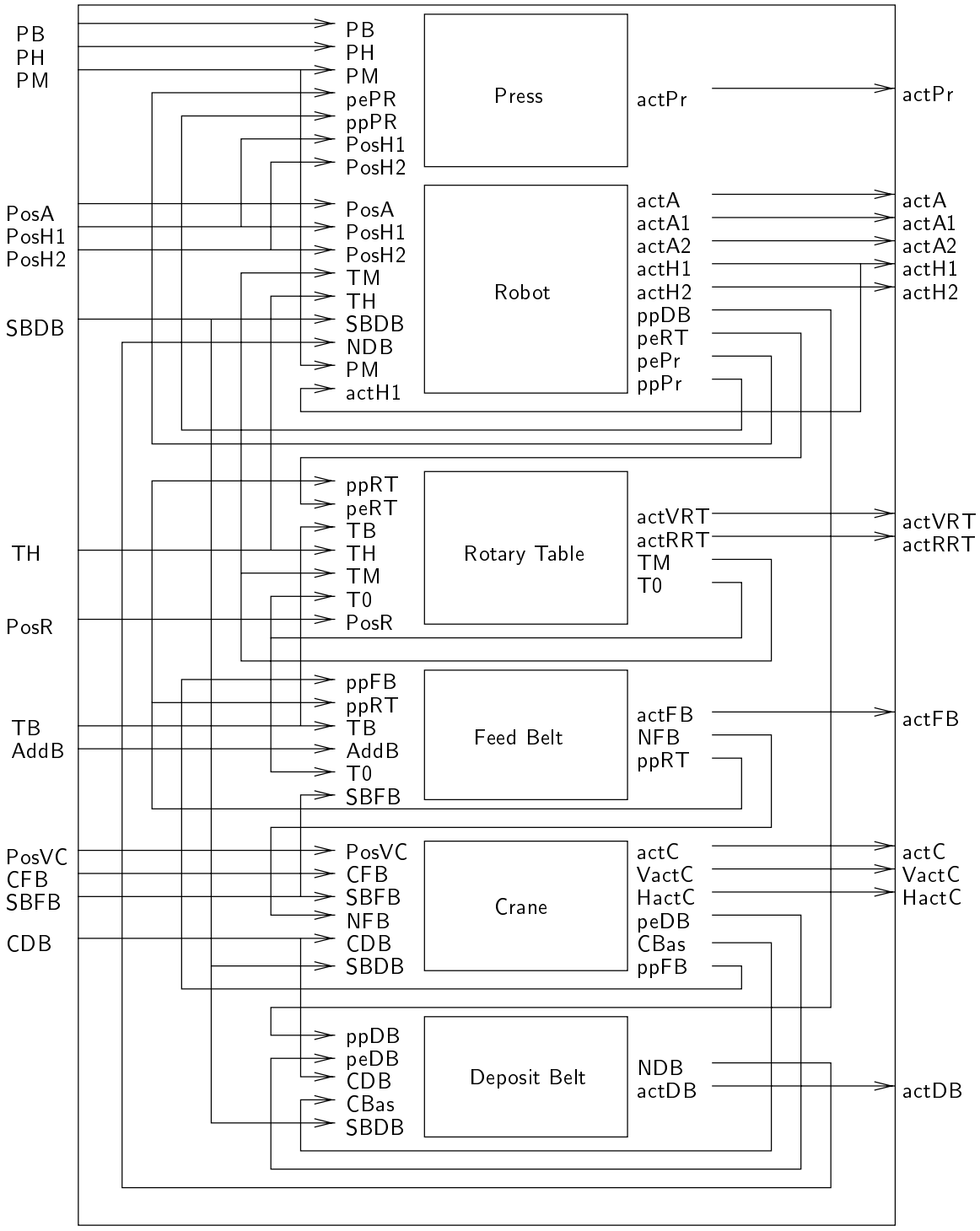



Figure 6: Architecture of the controller program

rotary table to be in the correct position, and for an object to be present on it.
— State C : the robot is rotating towards the press (actA=1) until the position is OK (PosA=A2P). It must wait for the press to be in the appropriate vertical position (PB) and not moving (actPR=0). If PB, actPR=0 and PosA=A2P happen exactly at the same time when in state C>A, the transition is to state D>A directly; otherwise the system may wait in state C>B for a while.
— State D : the robot extends its second arm towards the press (actH2=1), puts an object on it (actA2=1 while actH2=0), and then retracts (actH2=−1).
— State E : the robot is rotating until second arm is over the deposit belt
— State F : the robot extends its second arm towards the deposit belt, puts the object on the belt, and then retracts. It may wait for the belt to be free.
— State G : the robot rotates for the first arm to reach the press. It may wait for the press to be at the appropriate vertical position.
— State H : the robot extends its first arm towards the press, puts the object on it, and then retracts.
— State A : the robot rotates for the first arm to reach the rotary table.

### 4.2 The environment

When modeling the environment, all the signals that are *sensors* for the controller (PB, PM, PH, posH1, posH2, posA, TB, TH, posR, CDB, CFB, posVC, SBDB, and SBFB) are *computed*. Modeling the environment consists in defining how the *inputs* of the controller are influenced by its *outputs*. We model a very simple environment (all moving parts have constant speeds).

The first component is written in pure Lustre: there is no state. Actually, it is a particular case of a mode-automaton in which there is only one mode, and a set of Lustre equations attached to it.

We model an environment in which the motor that rotates the robot is supposed to work; hence, when actA=1 (rotate in one direction) or actA =-1 (rotate in the other direction), the position is given by the equation: posA = pre(posA) + (actA * DeltaA), where posA is an angle and DeltaA is a constant related to the rotation speed (via the base clock of the system, which defines the duration of one *instant*). When the motor is off (actA=0), the same equation holds, meaning posA=pre(posA), i.e. the robot does not rotate. The same holds for computing PosH1, PosH2, PosPr, PosV, PosR, PosC and PosVC, which gives:

```
PosA  = pre(PosA) +  (actA  *DeltaA) ;   PosVC = pre(PosVC) + (VactC *DeltaVC) ;
PosH1 = pre(PosH1) + (actH1 *DeltaH1) ;  PosH2 = pre(PosH2) + (actH2 *DeltaH2) ;
PosPr = pre(PosPr) + (actPr *DeltaPr) ;  PosV  = pre(PosV) +  (actVRT*DeltaV) ;
PosR  = pre(PosR) +  (actRRT*DeltaR) ;   PosC  = pre(PosC) +  (HactC *DeltaC) ;
```

Once the positions are available, computing the values of the sensors is simple: we just have to compare the positions to some constant values: PH=(PosPr=prH), etc. This is done for PH, PB, PM, TH, TB, CFB and CDB, which gives:

```
PH =  (PosPr=PrH)    ; PB  = (PosPr=PrB) ; PM = (PosPr=PrM)
TH =  (PosV=Vmax)    ; TB  = (PosV=Vmin)
CFB = (PosC=Cmin)    ; CDB = (PosC=Cmax)
```

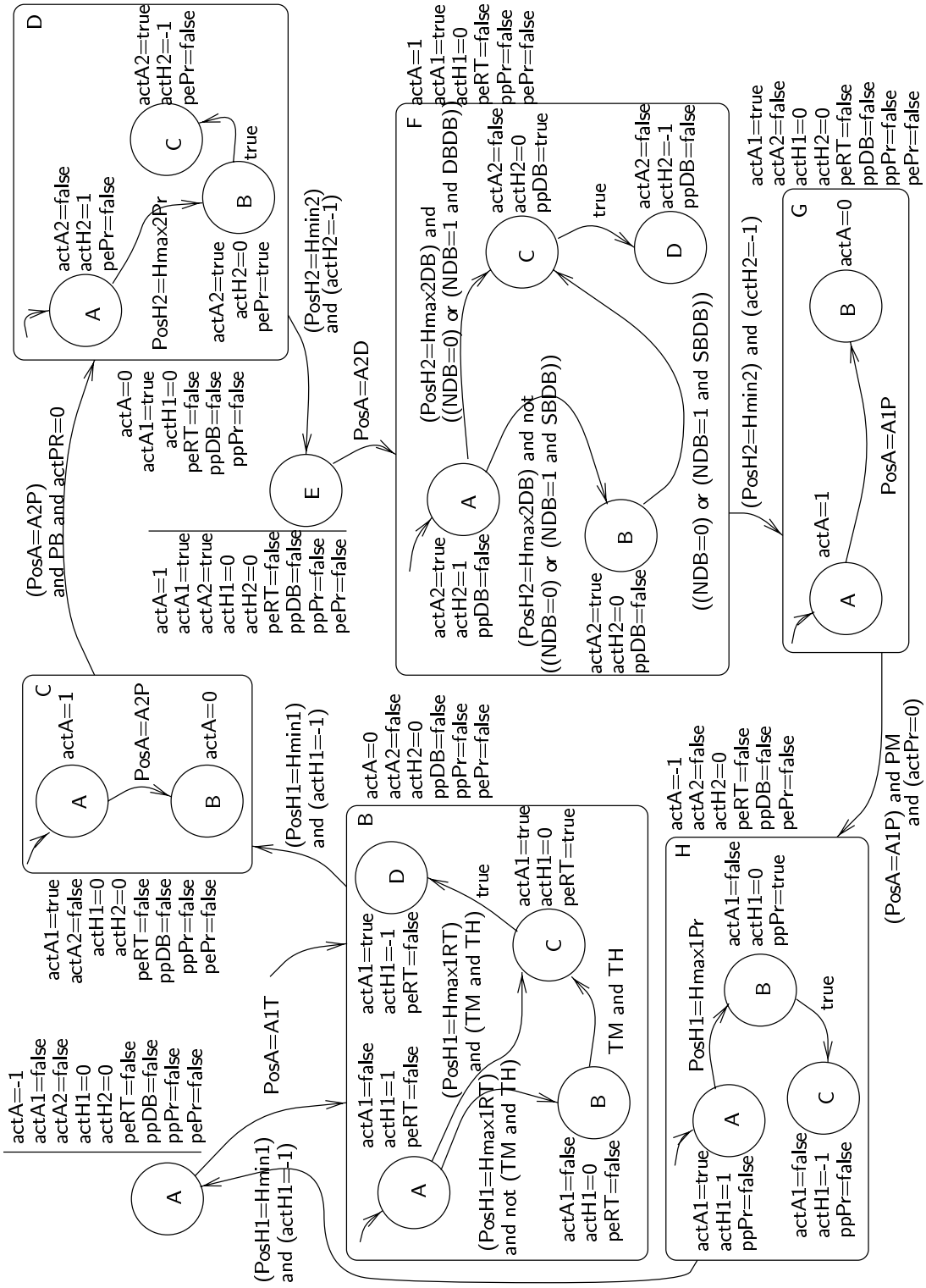



Figure 7: The `Robot` component

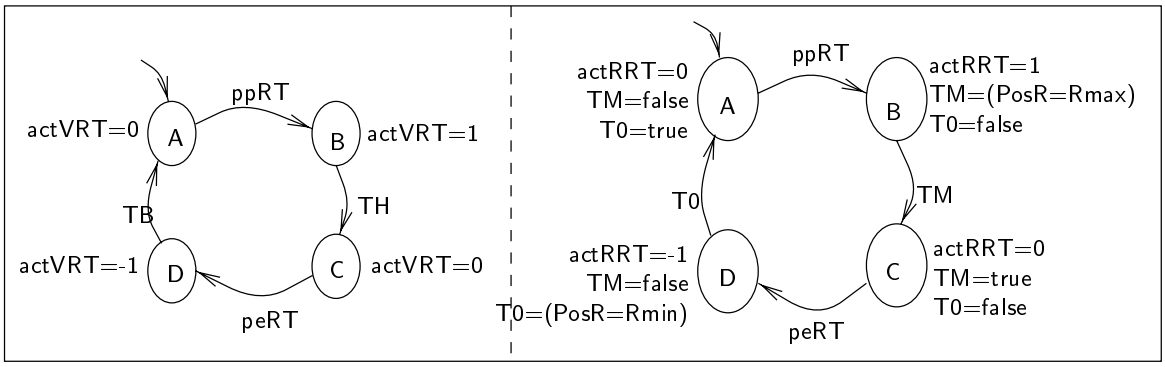



Figure 8: The `Rotary Table` component

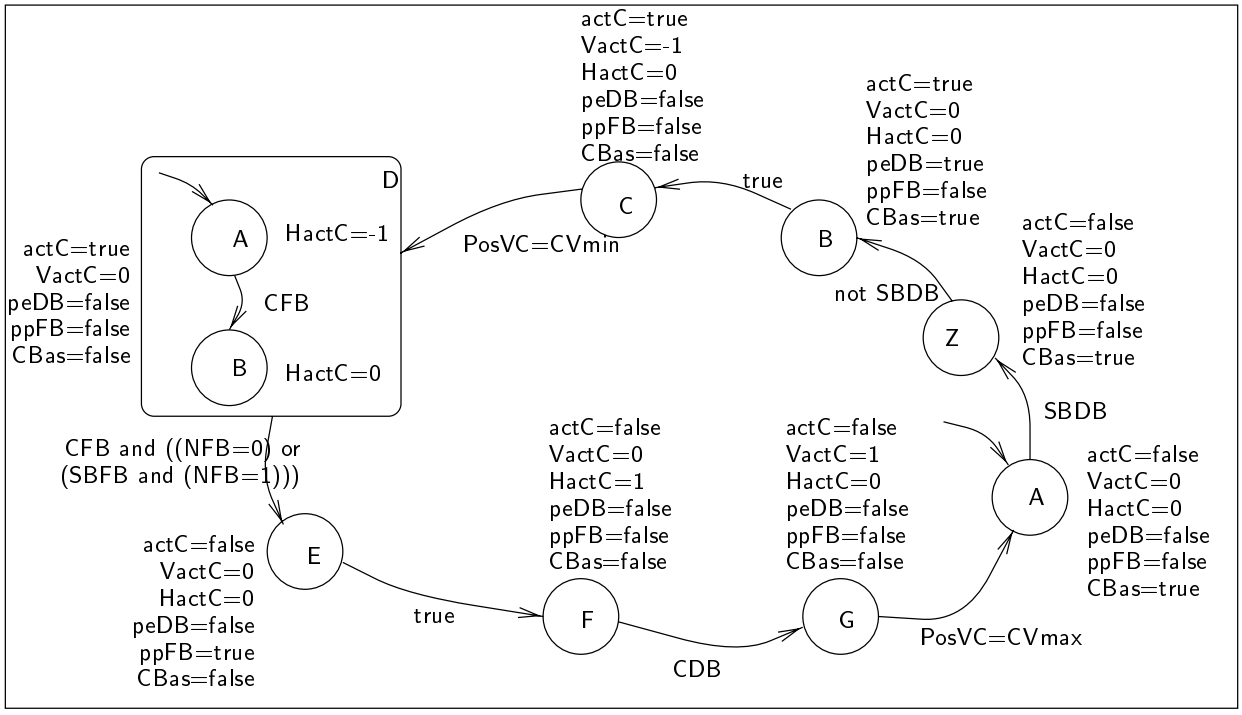



Figure 9: The `Crane` component

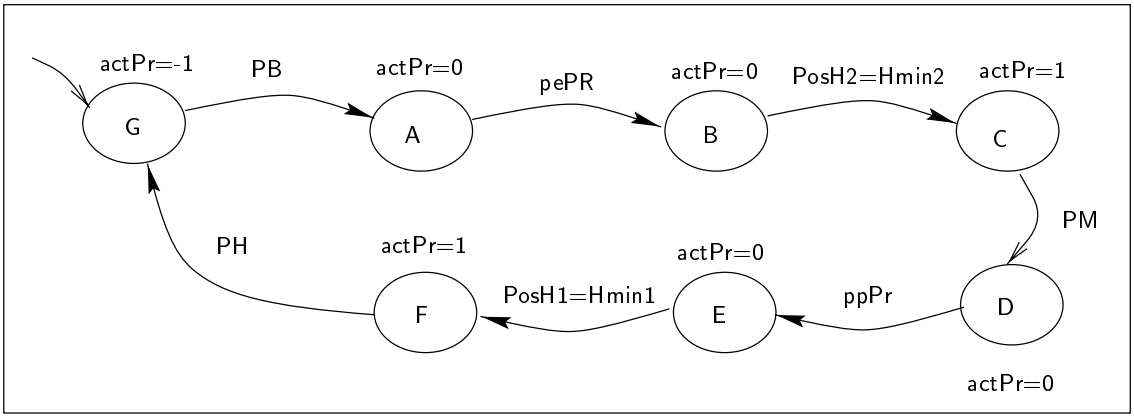



Figure 10: The `Press` component

This simple component is in parallel with two mode-automata, one for each belt.

Modeling the behavior of the belts is a bit more complex: in fact, we also have to model the behavior of the objects that travel on the belts. At least we need to specify that they do not vanish, and remain on the belt, moving with it, until they are taken by the crane or pushed to the rotary table.

The belt component in the controller and the belt component in the environment have the same automaton structure, with 6 states, as follows:

— State B : There is one object on the belt, and it is moving
— State D : Reached from B when the object reaches the sensor (SBFB) while the rotary table is at the appropriate rotation angle and vertical position (TB and T0); the object is being pushed to the rotary table; ppRT becomes true as soon as the value of the sensor is false.
— State C : Reached from B when the object reaches the sensor (SBFB) while the rotary table is not in the appropriate position; if it reaches it and no object has been put on the belt (ppFB or AddB), go to D; if an object is put on the belt, go to C or E, depending on the position of the rotary table;
— State E : Waiting state, like C, but with 2 objects
— State F : Waiting state, like D, but with 2 objects
— State A : Reached from D, when the object has left the belt and is on the rotary table, provided no other object is put on the belt.

In the global behavior of the system (controller+environment), the two mode-automata always evolve synchronously (they are always in corresponding states). We could have merged the two, but the separate version allows to deal with the controller alone, or with the complete system, just by adding one component (see the conclusion for a comment on reusing the same automaton structure in several places).

In the controller, we simply compute NFB (the number of objects on the belt); actFB (the command for the motor); ppRT (an object is put on the rotary table, which is detected by the belt when an object reaches its rightmost extremity). The complexity of the automaton is mainly due to the potential interleavings of events like addB (an object is put on the belt, by the external user) or ppFB (an object is put on the belt, by the rest of the system, namely the crane) or SBFB (an object reaches the end of the belt), etc.

In the component that models the environment, we also need to compute:

— the timer tF. It is an integer variable that counts instants, and is compared to a constant DeltaTF, representing the amount of time needed by an object to pass in front of the sensor. It depends on the speed of the belt and on the size of the object. It determines how long the value of the sensor is true.
— a memory MaPFB, used to store the current position of the belt when an object is put on it.
— the actual position of the belt PFB
— the actual value of the sensor SBFB: it starts being true when (PFB − MaPFB) = GammaPFB (where GammaPFB is yet another constant), and remains true until tF $\geq$ DeltaTF. In fact, this is not so simple, because the way SBFB is computed depends on the mode, but the idea is essentially that one.

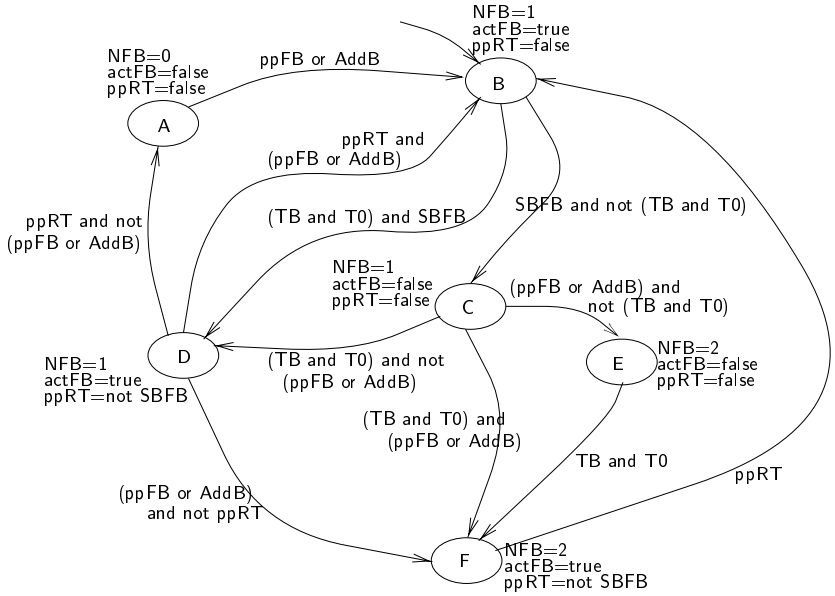



Figure 11: The Feed-Belt component of the controller.

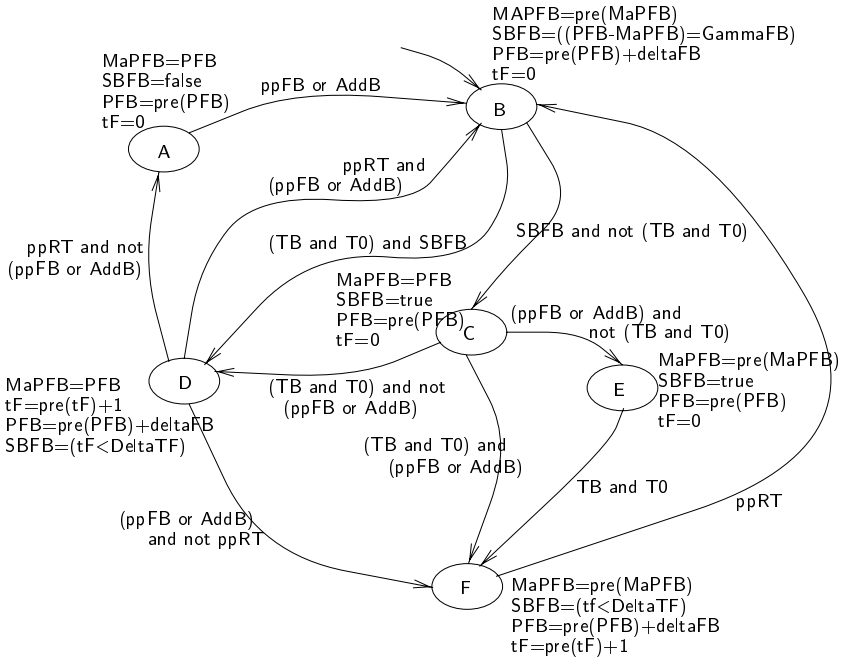



Figure 12: The Feed-Belt component in the environment.

# 5 Conclusion and further work

The aim of the case-study was to demonstrate the use of an imperative construct in a data-flow language for regulation systems, and the influence of this new construct in all stages of development (modeling the environment, programming the controller, simulating the behavior, etc.).

We think that the result is promising: when the mode-structure is part of the informal specification, the new construct is appropriate. Here is a list of benefits.

**Readability:** The controller is more readable than the Lustre version. When we observe the mode-structure of the mode-automaton, we clearly see where the modes differ, and the conditions for changing modes. The hierarchy of modes allows a set of states to share some equations, thus avoiding duplication of code. Explicit parallelism is used for almost independent behaviors, i.e., with a little interface. (see the mode-automaton on Figure 8).

**Size of the code:** The C code produced from the mode-automaton version (let us denote it by $C_m$) is a little smaller than the one produced from the pure Lustre version ($C_\ell$), because the conditionals are better structured; but this is not the more significant improvement. Intrinsically, the code for each mode has to be written somewhere.

**Execution time:** The real gain concerns execution time. The classical compilation techniques for Lustre are single-loop sequential programs of two kinds: 1) no control structure: each equation, as it is written in the source, is computed; 2) explicit control structure: an automaton is defined whose states correspond to all the valuations of the Boolean variables in the program; at each step, only a specialized version of the equations is computed (for a given value of all the Boolean variables, just rewrite the whole program, by propagating constants); this gives a better code, as far as the speed is concerned but, in practical cases, the control structure explodes. The size of the code is exponential in the size of the source. We would need a way to specify, for instance, which of the Boolean variables should be expanded into control states, and which of them should be considered as data, and therefore not expanded. Now, what is the picture for mode-automata? In the single-loop sequential program produced from a mode-automaton program, only the code corresponding to the current mode is computed at each step (and the transition from this mode). Hence the worst-case-execution-time (WCET) is the maximal execution time of the modes. Of course, if the system is described using a lot of trivial modes, and one complex one, the gain is low. But the key point is that, by defining *explicit running modes*, the user influences the control structure of the code produced, which is somewhere between the one-state program and the full-automaton program.

Now that the benefits for readability and code efficiency have been established, we are working on the definition of a kind of *assume/guarantee* scheme for modes, mimicking the assume-guarantee schemes that already exist for proving properties of parallel systems in a compositional way. This would show that offering the appropriate language construct to the users can also allow them to give hints that simplify proofs.

As far as language design is concerned, we are also working on a less restrictive definition of the hierarchic composition of mode-automata. With the present definition of the

language, a variable is defined at one level only. We would like to allow several definitions of the same variable at different levels of the hierarchy. This makes sense if refinement is thought of as a kind of inheritance mechanism (which is the case in UML behavioral models, where the innermost transitions have priority over the outermost ones).

Finally, the example of the two feed belt components (Figures 11 and 12) having the same automaton structure suggests an extension of the language in which (flat or even composed) automaton structures can be defined once and reused in different contexts, with different sets of equations attached to states.

## References


[BB91] A. Benveniste and G. Berry. Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9), September 1991.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*, Munich, January 1987.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995.

[Har87] D. Harel. Statecharts : A visual approach to complex systems. *Science of Computer Programming*, 8:231–275, 1987.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Ingeneering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[JM88] Farnam Jahanian and Aloysius Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 14, 1988.

[LL95] Claus Lewerentz and Thomas Lindner. *Formal Development of Reactive Systems: Case Study Production Cell.* Number 891 in Lecture Notes in Computer Science. Springer Verlag, January 1995.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*. LNCS 630, Springer Verlag, August 1992.

[MMP91] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In REX *Workshop on Real-Time: Theory in Practice*, DePlasmolen (Netherlands), June 1991. LNCS 600, Springer Verlag.

[MR98] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer Verlag, LNCS 1381.

[MRR00] F. Maraninchi, Y. Rémond, and Y. Raoul. Matou: An implementation of mode-automata. In *International Conference on Compiler Construction*, Berlin (Germany), March 2000. Springer Verlag.

[Pay96] S. Paynter. Real-time mode-machines. In *Formal Techniques for Real-Time and Fault Tolerance (FTRTFT)*, pages 90–109. LNCS 1135, Springer Verlag, 1996.

[Ray96] P. Raymond. Recognizing regular expressions by means of dataflows networks. In *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)* Paderborn, Germany. Springer Verlag, July 1996.

[Rém99] Yann Rémond. Matou home page. Technical report, VERIMAG, June 1999. `http://www-verimag.imag.fr/PEOPLE/Florence.Maraninchi/MATOU`.

[RM95] E. Rutten and F. Martinez. SIGNALGTI, implementing task preemption and time interval in the synchronous data-flow language SIGNAL. In *7th Euromicro Workshop on Real Time Systems*, Odense (Denmark), June 1995.

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[sE] synchronousEifel. `http://ais.gmd.de/~budde/` – GMD SET-EES, Schloss Birlinghoven, 53754 Sankt Augustin, Germany.

[SYR99] SYRF. Esprit LTR 22703, "synchronous reactive formalisms". Technical report, 1996-1999. `http://www-verimag.imag.fr/SYNCHRONE/SYRF/syrf.html`.

# Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus

Radu Mateescu[1] and Mihaela Sighireanu[2]

[1] Inria Rhône-Alpes / Vasy, 655, avenue de l'Europe
F-38330 Montbonnot Saint Martin, France
Radu.Mateescu@inria.fr
[2] Université Paris 7 / Liafa, 2, place Jussieu
F-75251 Paris, France
Mihaela.Sighireanu@liafa.jussieu.fr

**Abstract.** Model-checking is a successful technique for automatically verifying concurrent finite-state systems. When building a model-checker, a good compromise must be made between the expressive power of the property description formalism, the complexity of the model-checking problem, and the user-friendliness of the interface. We present a temporal logic and an associated model-checking method that attempt to fulfill these criteria. The logic is an extension of the alternation-free $\mu$-calculus with Actl-like action formulas and Pdl-like regular expressions, allowing a concise and intuitive description of safety, liveness, and fairness properties over labeled transition systems. The model-checking method is based upon a succinct translation of the verification problem into a boolean equation system, which is solved by means of an efficient local algorithm having a good average complexity. The algorithm also allows to generate full diagnostic information (examples and counterexamples) for temporal formulas. This method is at the heart of the Evaluator 3.0 model-checker that we implemented within the Cadp toolset using the generic Open/Cæsar environment for on-the-fly verification.

**Key-words:** boolean equation system, diagnostic, model-checking, $\mu$-calculus, specification, temporal logic, verification

## 1 Introduction

Formal verification is essential in order to improve the reliability of complex, critical applications such as communication protocols and distributed systems. A state-of-the-art technique for automatic verification of concurrent finite-state systems is called *model-checking*. In this approach, the application under design is first translated into a finite labeled transition system (Lts) model, on which the desired correctness properties (expressed e.g., as temporal logic formulas) are verified using appropriate model-checking algorithms.

When designing and building a model-checker, several important criteria must be considered. Firstly, the specification formalism should be sufficiently powerful to describe the main temporal property classes usually encountered

(safety, liveness, fairness). Among the wide range of temporal logics proposed in the literature, the modal $\mu$-calculus [18] is particularly powerful, subsuming linear-time logics as LTL [22], branching-time logics as CTL [4] or ACTL [25], and regular logics as PDL [12] or PDL-$\Delta$ [27].

Secondly, the underlying model-checking problem should have a sufficiently low complexity, in order to offer reasonable response times on practical applications. Optimizing this is often contradictory with the first criterion above, because the model-checking complexity of temporal logics usually increases with their expressive power. Since the model-checking problem of the full $\mu$-calculus is exponential-time, various sublogics of lower complexity have been defined. Among these, the *alternation-free* fragment [7] makes a good compromise between expressiveness (allowing direct encodings of CTL and ACTL) and efficiency of model-checking (several linear-time algorithms being available [5, 1, 29, 20]).

Thirdly, the model-checker interface should allow an intuitive, concise, and flexible description of properties, in order to avoid specification errors and to facilitate the verification task for non-expert users. Moreover, the model-checker must provide enough feedback information to make the debugging of the applications feasible; in practice, this means to provide a precise diagnostic in addition to a simple yes/no answer for a temporal property.

In this paper, we present a temporal logic and an associated model-checking method attempting to fulfill the aforementioned criteria. The temporal logic adopted is an extension of the alternation-free $\mu$-calculus with ACTL-like action formulas and PDL-like regular expressions, allowing a concise and intuitive description of safety, liveness, and (some) fairness properties without sacrificing the efficiency of verification. The method proposed for verifying a temporal formula over an LTS has a linear-time worst-case complexity (both in LTS size and formula size) and is based upon a succinct translation of the verification problem into a boolean equation system (BES). The method works on-the-fly, by exploring the LTS in a demand-driven way during the verification of the formula. The resulting BES is solved using a new linear-time local algorithm based on a depth-first search of the corresponding boolean graph. Compared to other linear-time local algorithms [1, 29], our algorithm is simpler to understand and has a good average complexity, achieved by a careful bookkeeping of the information in the portion of boolean graph visited during the search. Moreover, our algorithm is easily connected to the diagnostic generation algorithms given in [24], allowing to produce examples and counterexamples (subgraphs of the LTS) fully explaining the truth values of the formulas. This verification method has been used as a basis for the EVALUATOR 3.0 model-checker that we developed within the CADP (CÆSAR/ALDÉBARAN) toolset [9] using the generic OPEN/CÆSAR environment for on-the-fly verification [13].

The paper is organized as follows. Section 2 defines the syntax and semantics of the temporal logic proposed and illustrates its use by means of various examples of properties. Section 3 presents in detail the model-checking method and Section 4 discusses its implementation within the CADP toolset. Finally, Section 5 gives some concluding remarks and directions for future work.

## 2 Regular alternation-free $\mu$-calculus

The logic that we propose, called regular alternation-free $\mu$-calculus, is an extension of the alternation-free fragment of the modal $\mu$-calculus [18, 7] with action formulas as in ACTL [25] and with regular expressions over action sequences as in PDL [12]. It allows direct encodings of "pure" branching-time logics like ACTL or CTL [4], as well as of regular logics like PDL or PDL-$\Delta$ [27]. We first define its syntax and semantics, and then we show its usefulness by means of several examples of commonly encountered temporal properties.

### 2.1 Syntax and semantics

We consider as interpretation models finite labeled transition systems (LTSs), which are particularly suitable for action-based description formalisms such as process algebras. An LTS is a tuple $L = (S, A, T, s_0)$, where: $S$ is a finite set of *states*, $A$ is a finite set of *actions*, $T \subseteq S \times A \times S$ is the *transition relation*, and $s_0 \in S$ is the *initial state*. A transition $(s, a, s') \in T$, also noted $s \xrightarrow{a} s'$, indicates that the system can move from state $s$ to state $s'$ by performing action $a$.

The regular alternation-free $\mu$-calculus is built from three types of formulas, according to the syntax given on Figure 1.

| | |
|---|---|
| Action formulas | $\alpha ::= a \mid \neg\alpha \mid \alpha_1 \wedge \alpha_2$ |
| Regular formulas | $\beta ::= \alpha \mid \beta_1.\beta_2 \mid \beta_1 \vert \beta_2 \mid \beta^*$ |
| State formulas | $\varphi ::= \mathsf{F} \mid \mathsf{T} \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \langle\beta\rangle\,\varphi \mid [\beta]\,\varphi \mid Y \mid \mu Y.\varphi \mid \nu Y.\varphi$ |

**Fig. 1.** Syntax of regular alternation-free $\mu$-calculus

Action formulas $\alpha$ are built from action names $a \in A$ by using the standard boolean operators. Derived boolean connectives are defined as usual: $\mathsf{F} = a \wedge \neg a$ for some $a$, $\mathsf{T} = \neg\mathsf{F}$, $\alpha_1 \vee \alpha_2 = \neg(\neg\alpha_1 \wedge \neg\alpha_2)$, etc. Regular formulas $\beta$ are built from action formulas $\alpha$ by using the standard regular expression operators: concatenation (.), choice (|), and transitive-reflexive closure ($*$). The empty sequence operator $\varepsilon$ and the transitive closure operator $+$ are defined as $\varepsilon = \mathsf{F}^*$ and $\beta^+ = \beta.\beta^*$. State formulas $\varphi$ are built from propositional variables $Y \in \mathcal{Y}$ by using the standard boolean operators, the possibility and necessity operators $\langle\beta\rangle\,\varphi$ and $[\beta]\,\varphi$, and the minimal and maximal fixed point operators $\mu Y.\varphi$ and $\nu Y.\varphi$. The $\mu$ and $\nu$ operators act as binders for $Y$ variables in a way similar to quantifiers in first-order logic. A formula $\varphi$ without free occurrences of $Y$ variables is *closed*. Formulas $\varphi$ are assumed to be *alternation-free*, i.e., without mutually recursive minimal and maximal fixed point subformulas ($\langle\beta\rangle\,\varphi'$ and $[\beta]\,\varphi'$ modalities, where $\beta$ contains $*$ operators, must be considered as "hidden" minimal and maximal fixed point subformulas, respectively).

The semantics of the logic is shown on Figure 2. The interpretation $[\![\alpha]\!] \subseteq A$ of action formulas gives the set of LTS actions satisfying $\alpha$. The interpretation $\|\beta\| \subseteq S \times S$ of regular formulas gives a binary relation between the source and target states of transition sequences satisfying $\beta$ ($\circ$, $\cup$, and $*$ denote composition, union, and transitive-reflexive closure of binary relations). The $\alpha$ regular formula characterizes one-step sequences $s \xrightarrow{a} s'$ such that $a$ satisfies $\alpha$. The $\beta_1.\beta_2$ formula states that a sequence is the concatenation of two sequences satisfying $\beta_1$ and $\beta_2$; $\beta_1|\beta_2$ states that a sequence can satisfy $\beta_1$ or $\beta_2$; and $\beta^*$ states that a sequence is the concatenation of (zero or more) sequences satisfying $\beta$. The interpretation $[\![\varphi]\!]\,\rho \subseteq S$ of state formulas, where the propositional context $\rho : \mathcal{Y} \to 2^S$ assigns state sets to propositional variables, gives the set of LTS states satisfying $\varphi$ in the context of $\rho$ ($\oslash$ denotes context overriding). The modalities $\langle\beta\rangle\,\varphi$ and $[\beta]\,\varphi$ characterize the states for which some (all) outgoing transition sequences satisfying $\beta$ lead to states satisfying $\varphi$. The formulas $\mu Y.\varphi$ and $\nu Y.\varphi$ denote the minimal and maximal solutions (over $2^S$) of the fixed point equation $Y = \varphi$.

| | |
|---|---|
| Action formulas | $[\![a]\!] = \{a\}$ |
| | $[\![\neg\alpha]\!] = A \setminus [\![\alpha]\!]$ |
| | $[\![\alpha_1 \wedge \alpha_2]\!] = [\![\alpha_1]\!] \cap [\![\alpha_2]\!]$ |
| Regular formulas | $\|\alpha\| = \{(s,s') \in S \times S \mid \exists a \in A.s \xrightarrow{a} s' \wedge a \in [\![\alpha]\!]\}$ |
| | $\|\beta_1.\beta_2\| = \|\beta_1\| \circ \|\beta_2\|$ |
| | $\|\beta_1|\beta_2\| = \|\beta_1\| \cup \|\beta_2\|$ |
| | $\|\beta^*\| = \|\beta\|^*$ |
| State formulas | $[\![\mathsf{F}]\!]\,\rho = \emptyset$ |
| | $[\![\mathsf{T}]\!]\,\rho = S$ |
| | $[\![\varphi_1 \vee \varphi_2]\!]\,\rho = [\![\varphi_1]\!]\,\rho \cup [\![\varphi_2]\!]\,\rho$ |
| | $[\![\varphi_1 \wedge \varphi_2]\!]\,\rho = [\![\varphi_1]\!]\,\rho \cap [\![\varphi_2]\!]\,\rho$ |
| | $[\![\langle\beta\rangle\,\varphi]\!]\,\rho = \{s \in S \mid \exists s' \in S.(s,s') \in \|\beta\| \wedge s' \in [\![\varphi]\!]\,\rho\}$ |
| | $[\![[\beta]\,\varphi]\!]\,\rho = \{s \in S \mid \forall s' \in S.(s,s') \in \|\beta\| \Rightarrow s' \in [\![\varphi]\!]\,\rho\}$ |
| | $[\![Y]\!]\,\rho = \rho(Y)$ |
| | $[\![\mu Y.\varphi]\!]\,\rho = \bigcap\{S' \subseteq S \mid \Phi_\rho(S') \subseteq S'\}$ |
| | $[\![\nu Y.\varphi]\!]\,\rho = \bigcup\{S' \subseteq S \mid S' \subseteq \Phi_\rho(S')\}$ |
| | where $\Phi_\rho : 2^S \to 2^S$, $\Phi_\rho(S') = [\![\varphi]\!]\,(\rho \oslash [S'/Y])$ |

**Fig. 2.** Semantics of regular alternation-free $\mu$-calculus

Let $L = (S, A, T, s_0)$ be an LTS. An action $a \in A$ satisfies a formula $\alpha$ (written as $a \models \alpha$) iff $a \in [\![\alpha]\!]$. A state $s \in S$ satisfies a closed formula $\varphi$ (written $s \models \varphi$) iff $s \in [\![\varphi]\!]$. $L$ is a $\varphi$-model (written $L \models \varphi$) iff $[\![\varphi]\!] = S$. Since an on-the-fly model-checker only decides whether $s_0 \models \varphi$, the user should be aware that verifying $L \models \varphi$ amounts to check on-the-fly the formula $[\mathsf{T}^*]\,\varphi$ (equivalent to the ACTL formula $\mathsf{AG}_\mathsf{T}\varphi$), stating that $\varphi$ holds on every state reachable from $s_0$.

### 2.2 Examples

The regular alternation-free $\mu$-calculus allows to express intuitively and concisely various useful properties of LTSs. Table 1 shows several examples of typical formulas representing safety, liveness, and fairness properties.

**Table 1.** Examples of properties in regular alternation-free $\mu$-calculus

| CLASS | PROPERTY | FORMULA |
|---|---|---|
| Safety | Absence of `Error` actions | $[\mathsf{T}^*.\texttt{Error}]\,\mathsf{F}$ |
| | Unreachability of a `Recv` action before a `Send` | $[(\neg\texttt{Send})^*.\texttt{Recv}]\,\mathsf{F}$ |
| | Mutual exclusion of sections delimited by `Open` and `Close` | $[\mathsf{T}^*.\texttt{Open1}.(\neg\texttt{Close1})^*.\texttt{Open2}]\,\mathsf{F}$ |
| Liveness | Deadlock freedom: absence of states without successors | $[\mathsf{T}^*]\,\langle\mathsf{T}\rangle\,\mathsf{T}$ |
| | Potential reachability (via some `Errors`) of a `Recv` after a `Send` | $\langle\mathsf{T}^*.\texttt{Send}.(\mathsf{T}^*.\texttt{Error})^*.\texttt{Recv}\rangle\,\mathsf{T}$ |
| | Inevitable reachability of a `Grant` action after a `Request` | $[\mathsf{T}^*.\texttt{Request}]\,\mu Y.\,\langle\mathsf{T}\rangle\,\mathsf{T}\wedge[\neg\texttt{Grant}]\,Y$ |
| Fairness | Livelock freedom: absence of `tau`-circuits | $[\mathsf{T}^*]\,\mu Y.\,[\texttt{tau}]\,Y$ |
| | Fair reachability (by skipping circuits) of a `Recv` after a `Send` | $[\mathsf{T}^*.\texttt{Send}.(\neg\texttt{Recv})^*]\,\langle(\neg\texttt{Recv})^*.\texttt{Recv}\rangle\,\mathsf{T}$ |

Note that boolean connectives (negation in particular) over actions improve the conciseness of formulas: without these operators, it would be impossible to express the inevitable reachability of an action without referring to other actions in the LTS. Also, regular operators (although theoretically they do not increase the expressive power of the alternation-free modal $\mu$-calculus) improve the readability of formulas: without these operators, the second liveness property given in Table 1 would be described by the equivalent fixed point formula $\mu Y_1.(\langle\texttt{Send}\rangle\,\mu Y_2.(\langle\texttt{Recv}\rangle\,\mathsf{T}\vee\mu Y_3.(\langle\texttt{Error}\rangle\,Y_2\vee\langle\mathsf{T}\rangle\,Y_3))\vee\langle\mathsf{T}\rangle\,Y_1)$.

Other, more elaborate examples of generic temporal properties encoded in regular alternation-free $\mu$-calculus can be found in Section 4.

## 3 On-the-fly model-checking

We present in this section a method for on-the-fly model-checking of regular alternation-free $\mu$-calculus formulas over finite LTSs. The method works by translating the verification problem into a boolean equation system, which is simultaneously solved using an efficient local algorithm.

### 3.1 Translation into boolean equation systems

Consider an LTS $L = (S, A, T, s_0)$ and a closed formula $\varphi$ in normal form (i.e., in which all propositional variables are unique). The verification problem we are interested in consists of deciding whether $s_0 \models \varphi$. An efficient method used for the ACTL logic [8] and for the alternation-free $\mu$-calculus [5,1] is to translate the problem into a boolean equation system (BES) [1,21], which is solved using specific local algorithms [1,29,28]. For the regular alternation-free $\mu$-calculus, one way to proceed could be first to translate a state formula $\varphi$ in plain alternation-free $\mu$-calculus and then to apply the above procedure. This means to encode the regular modalities of $\varphi$ using fixed point operators, e.g., by applying the Emerson-Lei translation from PDL to alternation-free $\mu$-calculus [7]. This translation is succinct (it produces at most a linear blow-up in the size of $\varphi$), but requires the identification and sharing of common subformulas.

However, we can also devise a succinct translation of the verification problem $s_0 \models \varphi$ into a BES resolution without computing common subformulas, but using instead an equational intermediate representation. The translation that we propose involves three steps, described below.

**Translation into PDL with recursion.** The first step is to translate a regular alternation-free $\mu$-calculus formula $\varphi$ into PDL *with recursion* (PDLR), which is a generalization of the Hennessy-Milner logic with recursion HMLR [19]. A PDLR specification (see Figure 3) consists of a propositional variable $Y$ and a fixed point equation system with propositional variables in left-hand sides and PDL formulas in right-hand sides. The equation system is given as a list $M_1 . \ldots . M_p$ of $\sigma$-blocks ($.$ denotes concatenation), i.e., subsystems of equations with the same sign $\sigma \in \{\mu, \nu\}$. We consider here only alternation-free PDLR specifications, in which every $\sigma$-block $M_j$ (for $1 \leq j < p$) depends only upon (has free variables that may be bound in) $M_{j+1}, \ldots, M_p$. The $Y$ variable must be defined in one of the $\sigma$-blocks $M_1, \ldots, M_p$ (usually in $M_1$). A PDLR specification is *closed* if all variables occurring in it are bound in the equation system.

Syntax of a PDLR specification:

$P = (Y, M_1 . \ldots . M_p)$

where $M_j = \{Y_{j_i} \stackrel{\sigma_j}{=} \varphi_{j_i}\}_{1 \leq i \leq n_j}$ for all $1 \leq j \leq p$

Semantics w.r.t. an LTS $(S, A, T, s_0)$ and a context $\rho : \mathcal{Y} \rightarrow 2^S$:

$$[\![(Y, M_1 . \ldots . M_p)]\!]\,\rho = (\rho \oslash [\![M_1 . \ldots . M_p]\!]\,\rho)(Y)$$

$$[\![M_j . \ldots . M_p]\!]\,\rho = ([\![M_j]\!]\,(\rho \oslash [\![M_{j+1} . \ldots . M_p]\!]\,\rho)) . [\![M_{j+1} . \ldots . M_p]\!]\,\rho$$

$$[\![\{Y_{j_i} \stackrel{\sigma_j}{=} \varphi_{j_i}\}_{1 \leq i \leq n_j}]\!]\,\rho = [\sigma_j \overline{\Phi}_{j\rho}/(Y_{j_1}, \ldots, Y_{j_{n_j}})]$$

where $\overline{\Phi}_{j\rho} : (2^S)^{n_j} \rightarrow (2^S)^{n_j}$, $\overline{\Phi}_{j\rho}(U_1, \ldots, U_{n_j}) = ([\![\varphi_i]\!]\,(\rho \oslash [U_1/Y_1, \ldots, U_{n_j}/Y_{n_j}]))_{1 \leq i \leq n_j}$

**Fig. 3.** Syntax and semantics of PDLR

A PDLR specification $(Y, M_1.\ldots.M_p)$ interpreted over an LTS yields the set of states associated to $Y$ in the solution of $M_1.\ldots.M_p$. The solution of $M_1.\ldots.M_p$ is a propositional context in $\mathcal{Y} \rightarrow 2^S$ obtained by concatenating the solutions of all $\sigma$-blocks $M_j$ $(1 \leq j < p)$, each one being calculated in the context of the subsystem $M_{j+1}.\ldots.M_p$. The solution of a $\sigma$-block $M_j$ with $n_j$ variables is a context mapping $M_j$'s variables to the $\sigma_j$ fixed point of a functional defined over $(2^S)^{n_j}$. The semantics of an empty system $\{\,\}$ is the empty context $[\,]$.

Before translating a closed regular alternation-free $\mu$-calculus formula $\varphi$ in PDLR, we must convert $\varphi$ into *expanded* form, by performing two actions: (a) add a new $\mu Y$ $(\nu Y)$ operator, where $Y$ is a "fresh" variable, in front of every $\langle\beta\rangle\,\varphi_1$ $([\beta]\,\varphi_1)$ subformula of $\varphi$ in which $\beta$ contains a $*$ operator (recall from Section 2.1 that these modalities are considered as "hidden" fixed point operators); (b) if the resulting formula $\varphi_0$ is not a fixed point one, add in front of $\varphi_0$ a $\sigma Y_0$ operator, where $\sigma \in \{\mu, \nu\}$ and $Y_0$ is another "fresh" variable.

The translation of an expanded formula $\sigma Y_0.\varphi_0$ into a PDLR specification $(\mathbf{T_1}(\sigma Y_0.\varphi_0, \sigma), \mathbf{T_2}(\sigma Y_0.\varphi_0, \sigma))$ is obtained using two syntactic functions $\mathbf{T_1}$ and $\mathbf{T_2}$, defined inductively in Figure 4. $\mathbf{T_1}(\varphi, \sigma)$ yields a formula obtained from $\varphi$ by substituting each fixed point subformula by its corresponding variable. $\mathbf{T_2}(\varphi, \sigma)$ yields a system containing, for each fixed point subformula of $\varphi$, an equation with the corresponding variable in the left-hand side and a PDL formula in the right-hand side. The first $\sigma$-block, denoted by $hd(\mathbf{T_2}(\varphi, \sigma))$, contains the equations of sign $\sigma$ associated to the topmost fixed point subformulas of $\varphi$. The remainder of the system, denoted by $tl(\mathbf{T_2}(\varphi, \sigma))$, contains the $\sigma$-blocks already constructed from subformulas of $\varphi$. A new $\sigma$-block is created every time that a fixed point subformula with a sign $\tilde{\sigma}$ dual to $\sigma$ is encountered ($\tilde{\mu} = \nu$ and $\tilde{\nu} = \mu$).

| $\varphi$ | $\mathbf{T_1}(\varphi, \sigma)$ | $\mathbf{T_2}(\varphi, \sigma)$ |
|---|---|---|
| $\mathsf{F}$ | $\mathsf{F}$ | $\{\,\}$ |
| $\mathsf{T}$ | $\mathsf{T}$ | |
| $\langle\beta\rangle\,\varphi_1$ | $\langle\beta\rangle\,\mathbf{T_1}(\varphi_1, \sigma)$ | $\mathbf{T_2}(\varphi_1, \sigma)$ |
| $[\beta]\,\varphi_1$ | $[\beta]\,\mathbf{T_1}(\varphi_1, \sigma)$ | |
| $\varphi_1 \vee \varphi_2$ | $\mathbf{T_1}(\varphi_1, \sigma) \vee \mathbf{T_1}(\varphi_2, \sigma)$ | $(hd(\mathbf{T_2}(\varphi_1, \sigma)) \cup hd(\mathbf{T_2}(\varphi_2, \sigma)))\,.$ $tl(\mathbf{T_2}(\varphi_1, \sigma)).tl(\mathbf{T_2}(\varphi_2, \sigma))$ |
| $\varphi_1 \wedge \varphi_2$ | $\mathbf{T_1}(\varphi_1, \sigma) \wedge \mathbf{T_1}(\varphi_2, \sigma)$ | |
| $Y$ | $Y$ | $\{\,\}$ |
| $\sigma Y.\varphi_1$ | | $(\{Y \stackrel{\sigma}{=} \mathbf{T_1}(\varphi_1, \sigma)\} \cup hd(\mathbf{T_2}(\varphi_1, \sigma))).tl(\mathbf{T_2}(\varphi_1, \sigma))$ |
| $\tilde{\sigma} Y.\varphi_1$ | | $\{\,\}.(\{Y \stackrel{\tilde{\sigma}}{=} \mathbf{T_1}(\varphi_1, \tilde{\sigma})\} \cup hd(\mathbf{T_2}(\varphi_1, \tilde{\sigma}))).tl(\mathbf{T_2}(\varphi_1, \tilde{\sigma}))$ |

**Fig. 4.** Translation of state formulas in PDLR

We illustrate this translation by an example. Consider the following formula (already written in expanded form), stating that every `Send` action in the LTS will be eventually followed by a `Recv`:

$$\varphi = \nu Y_0.\ [\mathsf{T}^*.\mathtt{Send}]\ \mu Y_1.\ \langle\mathsf{T}\rangle\ \mathsf{T} \wedge [\neg\mathtt{Recv}]\ Y_1$$

The translation $(\mathbf{T_1}(\varphi,\nu), \mathbf{T_2}(\varphi,\nu))$ yields the PDLR specification below:

$$\big(Y_0, \{Y_0 \stackrel{\nu}{=} [\mathsf{T}^*.\mathtt{Send}]\, Y_1\}.\{Y_1 \stackrel{\mu}{=} \langle\mathsf{T}\rangle\,\mathsf{T} \wedge [\neg\mathtt{Recv}]\, Y_1\}\big)$$

Using Bekić's theorem [3], we can show that the translation from regular alternation-free $\mu$-calculus to PDLR preserves the semantics of formulas: $[\![\sigma Y.\varphi]\!]\,\rho = [\![(\mathbf{T_1}(\sigma Y.\varphi,\sigma), \mathbf{T_2}(\sigma Y.\varphi,\sigma))]\!]\,\rho$ for any context $\rho : \mathcal{Y} \to 2^S$ and $\sigma \in \{\mu,\nu\}$. Note also that the size of the PDLR specification obtained is linear in the size of $\varphi$: there are as many equations in the system as variables in (the expanded form of) $\varphi$ and as many operators in the right-hand sides as operators in $\varphi$. However, in order to obtain a succinct translation into BESs, we need *simple* PDLR specifications, i.e., in which all PDL formulas in right-hand sides contain at most one boolean or modal operator. This is easily done by splitting the PDL formulas and introducing new variables, and may cause at most a linear blow-up in the size of the equation system. For the example above, we obtain the following equivalent simple PDLR specification:

$$\big(Y_0, \{Y_0 \stackrel{\nu}{=} [\mathsf{T}^*.\mathtt{Send}]\, Y_1\}.\{Y_1 \stackrel{\mu}{=} Y_2 \wedge Y_3, Y_2 \stackrel{\mu}{=} \langle\mathsf{T}\rangle\,\mathsf{T}, Y_3 \stackrel{\mu}{=} [\neg\mathtt{Recv}]\, Y_1\}\big)$$

**Translation into HML with recursion.** The second step is to translate a simple PDLR specification into HMLR, which amounts to eliminate all regular operators inside the modal formulas present in the right-hand sides of the equation system. This translation is performed by the syntactic function $\mathbf{R}$ defined in Figure 5. Every equation containing a modality with a regular expression is translated into (one or more) equations of the same sign that contain modalities with simpler regular formulas (having less regular operators). This process continues recursively until all resulting modalities in the right-hand sides belong to HML, i.e., they contain only pure action formulas.

$$\begin{array}{rl}
\mathbf{R}(Y, M_1.\ldots.M_p) & = (Y, \mathbf{R}(M_1).\ldots.\mathbf{R}(M_p)) \\[2ex]
\mathbf{R}(\{Y_i \stackrel{\sigma}{=} \varphi_i\}_{1\le i\le n}) & = \bigcup_{i=1}^{n} \mathbf{R}(Y_i \stackrel{\sigma}{=} \varphi_i) \\[2ex]
\mathbf{R}(Y \stackrel{\sigma}{=} \langle\alpha\rangle\,\varphi) & = \{Y \stackrel{\sigma}{=} \langle\alpha\rangle\,\varphi\} \\
\mathbf{R}(Y \stackrel{\sigma}{=} [\alpha]\,\varphi) & = \{Y \stackrel{\sigma}{=} [\alpha]\,\varphi\} \\
\mathbf{R}(Y \stackrel{\sigma}{=} \langle\beta_1.\beta_2\rangle\,\varphi) & = \mathbf{R}(Y \stackrel{\sigma}{=} \langle\beta_1\rangle\, Y_1) \cup \mathbf{R}(Y_1 \stackrel{\sigma}{=} \langle\beta_2\rangle\,\varphi) \\
\mathbf{R}(Y \stackrel{\sigma}{=} [\beta_1.\beta_2]\,\varphi) & = \mathbf{R}(Y \stackrel{\sigma}{=} [\beta_1]\, Y_1) \cup \mathbf{R}(Y_1 \stackrel{\sigma}{=} [\beta_2]\,\varphi) \\
\mathbf{R}(Y \stackrel{\sigma}{=} \langle\beta_1|\beta_2\rangle\,\varphi) & = \{Y \stackrel{\sigma}{=} Y_1 \vee Y_2\} \cup \mathbf{R}(Y_1 \stackrel{\sigma}{=} \langle\beta_1\rangle\,\varphi) \cup \mathbf{R}(Y_2 \stackrel{\sigma}{=} \langle\beta_2\rangle\,\varphi) \\
\mathbf{R}(Y \stackrel{\sigma}{=} [\beta_1|\beta_2]\,\varphi) & = \{Y \stackrel{\sigma}{=} Y_1 \wedge Y_2\} \cup \mathbf{R}(Y_1 \stackrel{\sigma}{=} [\beta_1]\,\varphi) \cup \mathbf{R}(Y_2 \stackrel{\sigma}{=} [\beta_2]\,\varphi) \\
\mathbf{R}(Y \stackrel{\sigma}{=} \langle\beta^*\rangle\,\varphi) & = \{Y \stackrel{\sigma}{=} \varphi \vee Y_1\} \cup \mathbf{R}(Y_1 \stackrel{\sigma}{=} \langle\beta\rangle\, Y) \\
\mathbf{R}(Y \stackrel{\sigma}{=} [\beta^*]\,\varphi) & = \{Y \stackrel{\sigma}{=} \varphi \wedge Y_1\} \cup \mathbf{R}(Y_1 \stackrel{\sigma}{=} [\beta]\, Y)
\end{array}$$

**Fig. 5.** Translation of simple PDLR specifications in HMLR

For the simple PDLR specification obtained in the previous example, the translation $\mathbf{R}$ yields the following (simple) HMLR specification:

$$\begin{array}{l}(Y_0, \{Y_0 \stackrel{\nu}{=} Y_4 \wedge Y_5, Y_4 \stackrel{\nu}{=} [\mathtt{Send}]\, Y_1, Y_5 \stackrel{\nu}{=} [\mathsf{T}]\, Y_0\}. \\ \qquad \{Y_1 \stackrel{\mu}{=} Y_2 \wedge Y_3, Y_2 \stackrel{\mu}{=} \langle \mathsf{T} \rangle\, \mathsf{T}, Y_3 \stackrel{\mu}{=} [\neg \mathtt{Recv}]\, Y_1\})\end{array}$$

The translation from PDLR to HMLR preserves the semantics of specifications: $[\![(Y, M_1.\ldots.M_p)]\!]\,\rho = [\![\mathbf{R}(Y, M_1.\ldots.M_p)]\!]\,\rho$ for any context $\rho : \mathcal{Y} \to 2^S$. Moreover, it is easy to see that $\mathbf{R}$ may cause at most a linear blow-up in the size of the equation system.

**Translation into BESs.** The third step is to translate a simple HMLR specification into an (alternation-free) boolean equation system. A BES (see Figure 6) consists of a boolean variable $x$ and a fixed point equation system $B_1.\ldots.B_p$ with boolean variables in left-hand sides and boolean formulas in right-hand sides. For simplicity, we consider only pure disjunctive or conjunctive boolean formulas. An empty disjunction is equivalent to $\mathsf{F}$ and an empty conjunction is equivalent to $\mathsf{T}$. The semantics of a BES is defined in a way similar to a PDLR specification, except that it produces the boolean value associated to $x$ in the solution of $B_1.\ldots.B_p$.

Syntax of a BES:

$E = (x, B_1.\ldots.B_p)$

where $B_j = \{x_{j_i} \stackrel{\sigma_j}{=} op_{j_i} X_{j_i}\}_{1 \le i \le n_j}$, $x_{j_i} \in \mathcal{X}$, $op_{j_i} \in \{\vee, \wedge\}$, and $X_{j_i} \subseteq \mathcal{X}$ for all $1 \le j \le p, 1 \le i \le n_j$

Semantics w.r.t. $\mathbf{Bool} = \{\mathsf{F}, \mathsf{T}\}$ and a context $\delta : \mathcal{X} \to \mathbf{Bool}$:

$$[\![(x, B_1.\ldots.B_p)]\!]\,\delta = (\delta \oslash [\![B_1.\ldots.B_p]\!]\,\delta)(x)$$

$$[\![B_j.\ldots.B_p]\!]\,\delta = ([\![B_j]\!]\,(\delta \oslash [\![B_{j+1}.\ldots.B_p]\!]\,\delta)).\,[\![B_{j+1}.\ldots.B_p]\!]\,\delta$$

$$[\![\{x_{j_i} \stackrel{\sigma_j}{=} op_{j_i} X_{j_i}\}_{1 \le i \le n_j}]\!]\,\delta = [\sigma_j \overline{\Psi}_{j\delta}/(x_{j_1}, \ldots, x_{j_{n_j}})]$$

where $[\![op\{x_1, ..., x_k\}]\!]\,\delta = \delta(x_1)\ op \ldots op\ \delta(x_k)$ and $\overline{\Psi}_{j\delta}$:$\mathbf{Bool}^{n_j} \to \mathbf{Bool}^{n_j}$,
$\overline{\Psi}_{j\delta}(b_1, ..., b_{n_j}) = ([\![op_{j_i} X_{j_i}]\!](\delta \oslash [b_1/x_1, ..., b_{n_j}/x_{n_j}]))_{1 \le i \le n_j}$

**Fig. 6.** Syntax and semantics of boolean equation systems

The local model-checking of a (simple) HMLR specification $(Y, M_1.\ldots.M_p)$ on the initial state $s_0$ of an LTS $L = (S, A, T, s_0)$ means to decide whether the set of states denoted by $Y$ contains $s_0$. This is translated into a BES by the semantic function $\mathbf{B}$ defined inductively in Figure 7. To every propositional variable $Y$ in the left-hand side of an equation and to every state $s \in S$ is associated a boolean variable $Y_s$ encoding the fact that $s$ belongs to the set of states denoted by $Y$. To every HML formula $\varphi$ in a right-hand side and to every state $s$ is associated a boolean formula $\mathbf{B}(\varphi, s)$ encoding the fact that $s$ satisfies $\varphi$.

$$\mathbf{B}(Y, M_1.\ldots.M_p) = (Y_{s_0}, \mathbf{B}(M_1).\ldots.\mathbf{B}(M_p))$$

$$\mathbf{B}(\{Y_i \stackrel{\sigma}{=} \varphi_i\}_{1\leq i\leq n}) = \{Y_{i,s} \stackrel{\sigma}{=} \mathbf{B}(\varphi_i, s)\}_{1\leq i\leq n, s\in S}$$

$$\begin{aligned}
\mathbf{B}(\mathsf{F}, s) &= \mathsf{F} \\
\mathbf{B}(\mathsf{T}, s) &= \mathsf{T} \\
\mathbf{B}(\varphi_1 \vee \varphi_2, s) &= \mathbf{B}(\varphi_1, s) \vee \mathbf{B}(\varphi_2, s) \\
\mathbf{B}(\varphi_1 \wedge \varphi_2, s) &= \mathbf{B}(\varphi_1, s) \wedge \mathbf{B}(\varphi_2, s) \\
\mathbf{B}(\langle\alpha\rangle\, \varphi, s) &= \bigvee\nolimits_{\{s \stackrel{a}{\rightarrow} s' \mid a\models\alpha\}} \mathbf{B}(\varphi, s') \\
\mathbf{B}([\alpha]\, \varphi, s) &= \bigwedge\nolimits_{\{s \stackrel{a}{\rightarrow} s' \mid a\models\alpha\}} \mathbf{B}(\varphi, s') \\
\mathbf{B}(Y_i, s) &= Y_{i,s}
\end{aligned}$$

**Fig. 7.** Translation of simple HmlR specifications into Bess

The **B** function is similar to other translations from modal equation systems to Bess [2, 5, 1, 29, 21]. **B** produces a Bes whose size is linear in the size of the HmlR specification (which in turn is linear in the size of the initial state formula) and the size of the Lts (number of states and transitions). It is important to note that during the translation of modal formulas (see Figure 7), the transitions in the Lts are traversed forwards, which enables to construct the Lts in a demand-driven way during the verification.

### 3.2 Local resolution of BESs

The final step of the model-checking procedure is the local resolution of the alternation-free Bes obtained by translating the local verification of a formula $\varphi$ on an Lts $(S, A, T, s_0)$. As we saw in Section 3.1, the verification of a fixed point formula $\sigma Y.\varphi$ on the initial state $s_0$ amounts to compute the value of the boolean variable $Y_{s_0}$ contained in the first $\sigma$-block of the Bes.

For simplicity, we consider here the resolution of Bess containing a single $\mu$-block (the solving routine for $\nu$-blocks is completely dual). Multiple-block alternation-free Bess can be handled by associating to each $\sigma$-block in the Bes its corresponding solving routine. Every time a variable $x_j$ bound in a $\sigma$-block $B_j$ is required in another block $B_i$ that depends on $B_j$, the solving routine of $B_j$ is called to compute $x_j$. The computation of $x_j$ may require in turn the values of other variables that are free in $B_j$ and defined in other blocks, leading to calls of the routines corresponding to those blocks, and so on. This process will eventually stop, because the Bes being alternation-free, there are no cyclic dependencies between blocks. During the resolution, the same variable of a block may be required several times in other blocks; therefore, the computation results must be persistent between subsequent calls of the same solving routine[1].

[1] This resolution scheme could be naturally implemented using coroutines.

**Extended Boolean Graphs.** Our resolution algorithm is easier to develop using a representation of BESs as *extended boolean graphs* [24], which are a slight generalization of the boolean graphs proposed in [1]. An extended boolean graph (EBG) is a tuple $G = (V, E, L, F)$, where: $V$ is the set of vertices; $E \subseteq V \times V$ is the set of edges; $L : V \rightarrow \{\vee, \wedge\}$ is the vertex labeling; and $F \subseteq V$ is the *frontier* of $G$. Intuitively, the frontier of an EBG $G$ contains the only vertices of $G$ starting at which new edges can be added when $G$ is embedded in another EBG. The set of successors of a vertex $x \in V$ is noted $E(x)$.

A closed BES can be represented by an EBG $G = (V, E, L, \emptyset)$, where $V$ denotes the set of boolean variables, $E$ denotes the dependencies between variables, and $L$ labels the vertices as disjunctive or conjunctive according to the operator in the corresponding equation of the BES (the frontier set is empty since $G$ is not meant to be embedded in another graph). Figure 8 shows a closed BES and its associated EBG, where black (white) vertices denote variables that are true (false) in the BES solution. The grey area delimits a subgraph containing the vertices $\{x_0, x_3, x_4, x_5, x_8\}$ and having the frontier $\{x_0, x_5, x_8\}$.

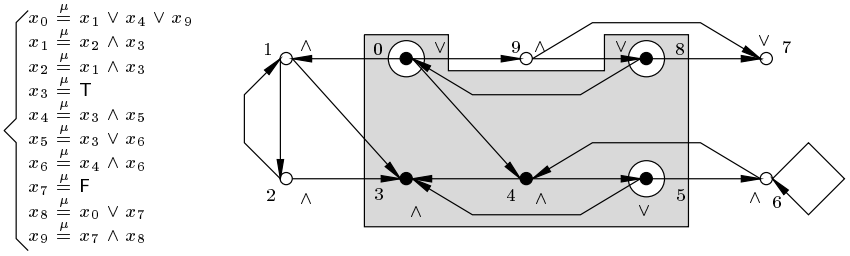



**Fig. 8.** A BES, its associated EBG, and a subgraph

Every EBG $G = (V, E, L, F)$ induces a Kripke structure $\mathbf{G} = (V, E, L)$. Such a Kripke structure is represented in an *implicit* manner when the "successor" function $E(x)$ can be computed for every vertex $x \in V$ without knowing the whole set $V$ (this is the case for the successor function implemented by the translation **B** given in Figure 7).

Let $P_\vee$ and $P_\wedge$ be two atomic propositions denoting the $\vee$- and $\wedge$-vertices of a Kripke structure $\mathbf{G}$ induced by a BES. The BES solution can be characterized by the following $\mu$-calculus formula interpreted over $\mathbf{G}$ [24]:

$$\text{EX} = \mu Y.(P_\vee \wedge \langle \mathsf{T} \rangle\, Y) \vee (P_\wedge \wedge [\mathsf{T}]\, Y)$$

A variable $x$ of the BES is true iff the vertex $x$ satisfies EX in $\mathbf{G}$, noted $x \models_{\mathbf{G}} \text{EX}$. Intuitively, EX expresses that some (all) successors of a $\vee$-vertex ($\wedge$-vertex) lead, in a finite number of steps, to vertices corresponding to $\mathsf{T}$ variables of the BES (these are $\wedge$-vertices without successors, characterized by the formula $P_\wedge \wedge [\mathsf{T}]\, \mathsf{F}$). For the EBG in Figure 8, it is easy to check that the set $\{x_0, x_3, x_4, x_5, x_8\}$ of

black vertices is equal to the interpretation of EX on $\mathbf{G}$, noted $[\![\text{EX}]\!]_{\mathbf{G}}$. Thus, the local resolution of a BES amounts to the local model-checking of the EX formula on the corresponding Kripke structure.

Consider an EBG $G = (V, E, L, \emptyset)$, its associated Kripke structure $\mathbf{G} = (V, E, L)$, and $x \in V$. The local model-checking of EX on $x$ does not always require to entirely explore $\mathbf{G}$ (e.g., on Figure 8, one could explore only the outlined subgraph in order to check EX on $x_0$), but rather to explore a part $\mathbf{G}'$ of $\mathbf{G}$ such that the value of $x$ can be computed based only on the information in $\mathbf{G}'$. Formally, this means to compute a subgraph $G' = (V', E', L', F')$ of $G$ that contains $x$ and is *solution-closed* [24], i.e., the satisfaction of EX by $x$ is the same in $\mathbf{G}'$ and $\mathbf{G}$: $[\![\text{EX}]\!]_{\mathbf{G}'} = [\![\text{EX}]\!]_{\mathbf{G}} \cap V'$. A subgraph $G'$ is solution-closed iff the satisfaction of EX on its frontier $F'$ can be decided using only the information in $G'$: $F' \subseteq [\![(P_\vee \wedge \text{EX}) \vee (P_\wedge \wedge \neg\text{EX})]\!]_{\mathbf{G}'}$. For the EBG on Figure 8, it is easy to see that the subgraph outlined is solution-closed: its frontier $\{x_0, x_5, x_8\}$ contains only $\vee$-vertices satisfying EX.

**Local resolution algorithm.** The SOLVE algorithm that we propose (see Figure 9) takes as input an implicit Kripke structure $\mathbf{G} = (V, E, L)$ induced by an EBG $G$ and a vertex $x \in V$ on which the EX formula must be checked. Starting from $x$, SOLVE performs a depth-first search (DFS) of $\mathbf{G}$ and simultaneously checks EX on all visited vertices, which are stored in a set $A \subseteq V$. Upon termination, the subgraph $G_A$ of $G$ containing all vertices in $A$ and all edges traversed during the DFS is solution-closed ($[\![\text{EX}]\!]_{\mathbf{G_A}} = [\![\text{EX}]\!]_{\mathbf{G}} \cap A$), meaning that the truth value of EX on $x$ computed in $G_A$ is the same as that in $G$.

SOLVE is similar in spirit with other graph-based local resolution algorithms like those of Andersen [1] and Vergauwen-Lewi [29]. However, since it implements the DFS iteratively, using an explicit *stack* and two nested while-loops, we believe that SOLVE is easier to understand than e.g., Andersen's algorithm, which uses a while-loop and two mutually recursive functions.

The successors $E(y)$ of every vertex $y \in V$ are assumed to be ordered from $(E(y))_0$ to $(E(y))_{|E(y)|-1}$. For every vertex $y \in A$, a counter $p(y)$ denotes the current successor of $y$ that must be explored. Every time a vertex $y$ such that $y \models_{\mathbf{G}} \text{EX}$ is encountered on top of the stack (this can be either a "new" $\wedge$-sink vertex, or an already visited vertex), the EX formula is reevaluated in $G_A$.

This reevaluation is carried out by the inner while-loop by keeping a work set $B \subseteq A$ containing the vertices $u$ such that $u \models_{\mathbf{G_A}} \text{EX}$ and EX has not yet been reevaluated on the nodes that depend upon $u$. To keep track of these backward dependencies, to each vertex $y \in A$ we associate the set $d(y) \subseteq A$ containing the currently visited predecessor vertices of $y$ (these vertices directly depend upon $y$ and EX must be reevaluated on them when EX becomes true on $y$). To efficiently perform the reevaluation of EX, we use the counter-based technique introduced in [2, 5]: to every vertex $y \in A$, we associate a counter $c(y)$ denoting the least number of successors of $y$ that currently have to satisfy EX in order to ensure $y \models_{\mathbf{G_A}} \text{EX}$ ($c(y)$ is initialized to 1 for $\vee$-vertices and to $|E(y)|$ for $\wedge$-vertices). Thus, for every $y \in A$, $y \models_{\mathbf{G_A}} \text{EX}$ iff $c(y) = 0$.

```
procedure SOLVE (x, (V, E, L)) is
  var A, B : 2^V; d : V → 2^V; c, p : V → Nat;
      y, z, u, w : V; stack : V*;
  c(x) := if L(x) = ∧ then |E(x)| else 1;
  p(x) := 0; d(x) := ∅;
  A := {x}; stack := push(x, nil);
  while stack ≠ nil do
    y := top(stack);
    if c(y) = 0 then
      if d(y) ≠ ∅ then
        B := {y};
        while B ≠ ∅ do
          let u ∈ B; B := B \ {u};
          forall w ∈ d(u) do
            if c(w) > 0 then
              c(w) := c(w) − 1;
              if c(w) = 0 then
                B := B ∪ {w}
              endif
            endif
          end;
          d(u) := ∅
        end
      else
        stack := pop(stack)
      endif
    elsif p(y) ≤ |E(y)| − 1 then
      z := (E(y))_{p(y)}; p(y) := p(y) + 1;
      if z ∈ A then
        d(z) := d(z) ∪ {y}
        if c(z) = 0 then
          stack := push(z, stack)
        endif
      else
        c(z) := if L(z) = ∧ then |E(z)| else 1
        p(z) := 0; d(z) := {y};
        A := A ∪ {z}; stack := push(z, stack)
      endif
    else
        stack := pop(stack)
    endif
  end
end
```

**Fig. 9.** Graph-based local resolution of a BES with sign $\mu$

Figure 10 shows the result of executing SOLVE for the variable $x_0$ and the EBG in Figure 8 (during the DFS, the successors of each vertex are visited as if the right-hand side of the corresponding equation was evaluated from left to right). The subgraph $G_A$ computed by SOLVE, containing the vertices $\{x_0, x_1, x_2, x_3, x_4, x_5\}$, is solution-closed, because its frontier $\{x_0, x_5\}$ contains only $\vee$-vertices satisfying EX in $\mathbf{G_A}$.

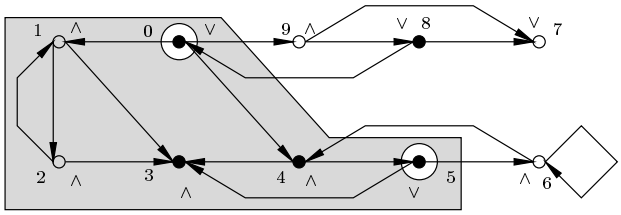



**Fig. 10.** A solution-closed subgraph computed by SOLVE

During the execution of SOLVE, the DFS stack repeatedly takes one of the three forms outlined on Figure 11.

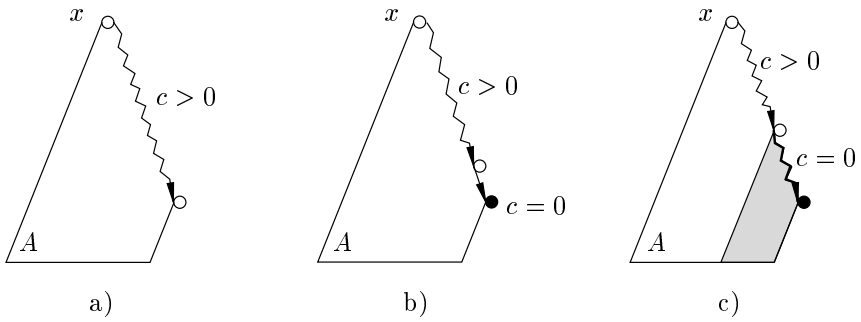



**Fig. 11.** Structure of the DFS stack during the execution of SOLVE

In form a), all vertices $y$ pushed on the stack are "unstable" ($c(y) > 0$), meaning that the truth of EX on $y$ depends on the portion $V \setminus A$ of $\mathbf{G}$ that has not been explored yet: so, the DFS must continue. In form b), a vertex $y$ that is "stable" ($c(y) = 0$) has been encountered and pushed on top of the stack, meaning that some vertices depending on $y$ may also become stable: therefore, EX must be reevaluated in $G_A$. In form c), this reevaluation has been finished, possibly leading to stabilization of some vertices in $A$: then, all stable vertices present on the stack will be popped, since no further information is needed for

them. The DFS properties ensure that all stable vertices on the stack are adjacent to the top[2], and thus after they are popped the stack takes again the form a).

SOLVE has a linear-time worst-case complexity, since every edge in $G_A$ is traversed at most twice: forwards (when its target vertex is visited by the DFS) and backwards (when EX is reevaluated on its source vertex). Moreover, SOLVE has also a good average-case complexity, improving on Andersen and Vergauwen-Lewi's algorithms, since it stops as soon as $x \models_{\mathbf{G_A}} \text{EX}$ and explores only vertices that are likely to influence $x$. Also, backward dependencies $d(u)$ of stable vertices $u$ are freed during the inner while-loop, thus reducing memory consumption.

**Diagnostic generation.** Practical applications of BES resolution, such as temporal logic model-checking, often require a more detailed feedback than a simple yes/no answer. To allow an efficient debugging of the temporal formulas, it is desirable to have also *diagnostic* information explaining the truth value obtained for the boolean variable of interest. Both positive diagnostics (examples) and negative diagnostics (counterexamples) are needed in order to have a full explanation of a temporal formula.

Let $G = (V, E, L, F)$ be an EBG and $x \in V$ the variable of interest. A diagnostic for $x$ is a solution-closed subgraph $G'$ of $G$ that contains $x$ and is minimal w.r.t. to subgraph inclusion, i.e., it contains the minimal amount of information needed in order to decide the satisfaction of EX by $x$. A diagnostic $G'$ is called *example* if $x \models_{\mathbf{G'}} \text{EX}$ and *counterexample* if $x \not\models_{\mathbf{G'}} \text{EX}$.

The SOLVE algorithm does not directly produce diagnostics; however, it can be easily coupled with the diagnostic generation algorithms proposed in [24]. These algorithms take as input a solution-closed subgraph (in which the semantics of EX has been already computed) and construct a diagnostic for a given variable by performing efficient traversals of the subgraph. Figure 12 shows an example for the variable $x_0$ obtained by traversing again the solution-closed subgraph on Figure 10 previously computed by SOLVE.

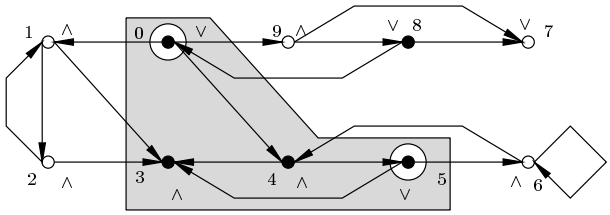



**Fig. 12.** An example for $x_0$

[2] The reevaluation of EX, which involves a backwards traversal of edges in $G_A$, can affect only those vertices in the DFS tree that are descendants of stable vertices present on the stack, outlined by the grey portion on Figure 11 c).

Since these diagnostic generation algorithms have a linear complexity in the size of the solution-closed subgraph they are executed upon, they affect neither the worst-case, nor the average-case complexity of SOLVE.

## 4 Implementation and use

We used the model-checking method presented in Section 3 as a basis for developing the EVALUATOR 3.0 model-checker within the CADP (CÆSAR/ALDÉBARAN) toolset [9]. The tool has been built using the OPEN/CÆSAR environment [13], which provides a generic API for on-the-fly exploration of transition systems. As a consequence, EVALUATOR 3.0 can be used in conjunction with every compiler that is OPEN/CÆSAR–compliant (i.e., that implements a translation from its input language to the OPEN/CÆSAR API), and particularly with the CÆSAR compiler [14] for LOTOS.

### 4.1 Additional operators and property patterns

Practical experience in using model-checking has shown the need for abstraction mechanisms enabling the specifier to define and use his own temporal operators in addition to those predefined in the model-checker. The input language of EVALUATOR 3.0 offers a macro-expansion mechanism allowing to define parameterized formulas and an inclusion mechanism allowing to group these definitions into separate libraries that can be reused in temporal specifications.

An immediate application was to build libraries for particular logics like CTL or ACTL by translating their temporal operators as fixed point formulas in regular alternation-free $\mu$-calculus. For example, the $\mathsf{E}\left[\varphi_1 {}_{\alpha_1}\mathsf{U}_{\alpha_2}\varphi_2\right]$ operator of ACTL (stating the existence of a sequence $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots s_k \xrightarrow{a_k} s_{k+1}$ such that $s_i \models \varphi_1$ for all $1 \leq i \leq k$, $a_j \models \alpha_1$ for all $1 \leq j < k$, $a_k \models \alpha_2$, and $s_{k+1} \models \varphi_2$) can be encoded as a macro $\mathsf{EU\_A\_A}(\varphi_1, \alpha_1, \alpha_2, \varphi_2) = \mu Y.(\varphi_1 \wedge (\langle\alpha_2\rangle\,\varphi_2 \vee \langle\alpha_1\rangle\,Y))$. Of course, these particular operators can be freely mixed with the built-in ones in temporal formulas, thus providing added flexibility to advanced users.

Another source of flexibility is provided by the use of *wildcards* (regular expressions on character strings) instead of action names in the formulas. If transition labels are represented as character strings (as it is currently the case with the OPEN/CÆSAR API), this allows to specify a set of labels using a single action predicate. For example, the wildcard `'SEND.*'` represents all transition labels denoting communication of 0 or more values on gate `SEND`.

In practice, it appears that in many cases, temporal properties tend to belong to particular classes of high-level "property patterns", such as *absence*, *existence*, *universality*, *precedence*, and *response*. These patterns have been identified in [6] after an important statistical study concerning over 500 applications of temporal logic model-checking. The knowledge embedded in this pattern system is important for both expert and non-expert users, since it reduces the risk of specification errors and facilitates the learning of temporal logic-based formalisms.

These property patterns have been expressed in [6] using several specification formalisms (CTL, LTL, regular expressions, etc.) but none of them was directly applicable to description languages with action-based semantics such as process algebras. Therefore, we developed in EVALUATOR 3.0 a library of parameterized formulas implementing the property patterns in regular alternation-free $\mu$-calculus. It turned out that many of them could be expressed in a much more concise and readable form than with the other formalisms used in [6]. Table 2 shows the first three patterns contained in the library.

**Table 2.** Property patterns in regular alternation-free $\mu$-calculus

| PATTERN | SCOPE | FORMULA |
|---|---|---|
| Absence ($\alpha_1$ is false) | Globally | $[\mathsf{T}^*.\alpha_1]\,\mathsf{F}$ |
| | Before $\alpha_2$ | $[(\neg\alpha_2)^*.\alpha_1.\mathsf{T}^*.\alpha_2]\,\mathsf{F}$ |
| | After $\alpha_2$ | $[(\neg\alpha_2)^*.\alpha_2.\mathsf{T}^*.\alpha_1]\,\mathsf{F}$ |
| | Between $\alpha_2$ and $\alpha_3$ | $[\mathsf{T}^*.\alpha_2.(\neg\alpha_3)^*.\alpha_1.\mathsf{T}^*.\alpha_3]\,\mathsf{F}$ |
| | After $\alpha_2$ until $\alpha_3$ | $[\mathsf{T}^*.\alpha_2.(\neg\alpha_3)^*.\alpha_1]\,\mathsf{F}$ |
| Existence ($\alpha_1$ becomes true) | Globally | $\mu Y.\,\langle\mathsf{T}\rangle\,\mathsf{T} \wedge [\neg\alpha_1]\,Y$ |
| | Before $\alpha_2$ | $[(\neg\alpha_1)^*.\alpha_2]\,\mathsf{F}$ |
| | After $\alpha_2$ | $[(\neg\alpha_2)^*.\alpha_2]\,\mu Y.\,\langle\mathsf{T}\rangle\,\mathsf{T} \wedge [\neg\alpha_1]\,Y$ |
| | Between $\alpha_2$ and $\alpha_3$ | $[\mathsf{T}^*.\alpha_2.(\neg\alpha_1)^*.\alpha_3]\,\mathsf{F}$ |
| | After $\alpha_2$ until $\alpha_3$ | $[\mathsf{T}^*.\alpha_2]\,([(\neg\alpha_1)^*.\alpha_3]\,\mathsf{F} \wedge \mu Y.\,\langle\mathsf{T}\rangle\,\mathsf{T} \wedge [\neg\alpha_1]\,Y)$ |
| Universality ($\alpha_1$ is true) | Globally | $[\mathsf{T}^*.\neg\alpha_1]\,\mathsf{F}$ |
| | Before $\alpha_2$ | $[(\neg\alpha_2)^*.\neg(\alpha_1 \vee \alpha_2).(\neg\alpha_2)^*.\alpha_2]\,\mathsf{F}$ |
| | After $\alpha_2$ | $[(\neg\alpha_2)^*.\alpha_2.\mathsf{T}^*.\neg\alpha_1]\,\mathsf{F}$ |
| | Between $\alpha_2$ and $\alpha_3$ | $[\mathsf{T}^*.\alpha_2.(\neg\alpha_3)^*.\neg(\alpha_1 \vee \alpha_3).\mathsf{T}^*.\alpha_3]\,\mathsf{F}$ |
| | After $\alpha_2$ until $\alpha_3$ | $[\mathsf{T}^*.\alpha_2.(\neg\alpha_3)^*.\neg(\alpha_1 \vee \alpha_3)]\,\mathsf{F}$ |

Besides facilitating the user task at the specification level, it is also important to offer enough feedback on the verification results to allow an easy debugging of the applications. This is achieved through the diagnostic generation facilities provided by EVALUATOR 3.0, which allows to produce examples and counterex-

amples explaining the truth value of regular alternation-free $\mu$-calculus formulas. As a side effect, this enables the user to get full diagnostics for particular temporal logics implemented as libraries, such as CTL and ACTL. Moreover, EVALUATOR 3.0 can be used to search regular execution sequences in LTSs, by checking PDL basic modalities: a transition sequence starting at the initial state and satisfying a regular formula $\beta$ can be obtained either as an example for the $\langle \beta \rangle \mathsf{T}$ formula, or as a counterexample for the $[\beta]\,\mathsf{F}$ formula.

### 4.2 Experimental results

We illustrate below the behaviour of EVALUATOR 3.0 by means of a simple benchmark example: the Alternating Bit Protocol (ABP for short) described in LOTOS. The protocol specification (available in the CADP release) contains four parallel processes: a sender entity, a receiver entity, and two channels modelling the communication of messages and acknowledgements, respectively. The sender accepts messages from a local user through a gate `Get` and the receiver delivers the messages to a remote user through a gate `Put`. Messages are represented by natural numbers between 0 and $n$, where $n$ is a parameter of the specification.

We formulated and verified several safety, liveness, and fairness properties of the ABP (see Table 3). For each property, the table gives its informal meaning, its corresponding regular alternation-free $\mu$-calculus formula, and its truth value on the LOTOS specification. Action predicates $\texttt{Put}_i$ and $\texttt{Get}_i$ denote the communication of message $i$ on gates `Put` and `Get`, respectively. Predicates $\texttt{Put}_{any}$ and $\texttt{Get}_{any}$ (wildcards) denote the communication of arbitrary messages on gates `Put` and `Get`. Every property containing an occurrence of $\texttt{Put}_i$ and/or $\texttt{Get}_i$ has been checked for all values of $i$ between 0 and $n$.

**Table 3.** Properties of the Alternating Bit Protocol

| NO. | PROPERTY | FORMULA | VALUE |
|---|---|---|---|
| $P_1$ | Initially, a `Put` will be eventually reached | $\mu Y.\, \langle \mathsf{T} \rangle\, \mathsf{T} \wedge [\neg \texttt{Put}_{any}]\, Y$ | false |
| $P_2$ | Initially, a `Put` will be fairly reached | $[(\neg \texttt{Put}_{any})^*]\, \langle \mathsf{T}^*.\texttt{Put}_{any} \rangle\, \mathsf{T}$ | true |
| $P_3$ | Initially, no `Get` can be reached before the corresponding `Put` | $[(\neg \texttt{Put}_i)^*.\texttt{Get}_i]\, \mathsf{F}$ | true |
| $P_4$ | Between two consecutive `Put`, there is a corresponding `Get` | $[\mathsf{T}^*.\texttt{Put}_i.(\neg \texttt{Get}_i)^*.\texttt{Put}_{any}]\, \mathsf{F}$ | true |
| $P_5$ | Between two consecutive `Get`, there is a corresponding `Put` | $[\mathsf{T}^*.\texttt{Get}_{any}.(\neg \texttt{Put}_i)^*.\texttt{Get}_i]\, \mathsf{F}$ | true |
| $P_6$ | After a `Put`, the corresponding `Get` is eventually reachable | $[\mathsf{T}^*.\texttt{Put}_i]\, \mu Y.\, \langle \mathsf{T} \rangle\, \mathsf{T} \wedge [\neg \texttt{Get}_i]\, Y$ | false |
| $P_7$ | After a `Put`, the corresponding `Get` is fairly reachable | $[\mathsf{T}^*.\texttt{Put}_i.(\neg \texttt{Get}_i)^*]\, \langle (\neg \texttt{Get}_i)^*.\texttt{Get}_i \rangle\, \mathsf{T}$ | true |

Properties $P_1$ and $P_6$, which express the inevitable reachability of `Put` and `Get` actions, are false because of the livelocks ($\tau$-loops) present in the LOTOS description. These two properties can be reformulated — as $P_2$ and $P_7$, respectively — in order to state the inevitable reachability only over fair execution sequences (i.e., by skipping loops).

We performed several experiments with EVALUATOR 3.0, by checking all properties on the ABP specification for different values of $n$. For comparison, we also used the EVALUATOR 2.0 model-checker developed at VERIMAG, which accepts as input plain alternation-free $\mu$-calculus formulas and implements the Fernandez-Mounier local boolean resolution algorithm [11]. All experiments have been performed on a Sparc Ultra 1 machine with 256 Mbytes of memory.

The results are shown in Table 4. For each experiment, the table gives the number of states of the LTS, the time (in minutes) required for the local model-checking of each property, and the percentage of states explored by each tool. The SOLVE algorithm performs uniformly better than the Fernandez-Mounier algorithm, the time needed being at least 50% smaller and the percentage of LTS states explored being always smaller or equal. For properties $P_1$, $P_2$, and $P_6$, which require to explore only a very small part of the LTS in order to decide their truth value, EVALUATOR 3.0 stops almost instantaneously (less than a second) in all cases, while EVALUATOR 2.0 takes up to one hour for $n = 100$.

**Table 4.** Local model-checking statistics

| No. | | $n=20$ $\lvert S\rvert = 39\,800$ | | $n=40$ $\lvert S\rvert = 153\,200$ | | $n=60$ $\lvert S\rvert = 340\,200$ | | $n=80$ $\lvert S\rvert = 600\,800$ | | $n=100$ $\lvert S\rvert = 935\,000$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *time* | *expl.%* | *time* | *expl.%* | *time* | *expl.%* | *time* | *expl.%* | *time* | *expl.%* |
| **$P_1$** | $a$ | **0″** | **0.01** | **0″** | **0.00** | **0″** | **0.00** | **0″** | **0.00** | **0″** | **0.00** |
| | $b$ | **20″** | **93.1** | **1′42″** | **96.4** | **4′49″** | **97.6** | **10′04″** | **98.2** | **18′23″** | **98.5** |
| **$P_2$** | $a$ | **0″** | **0.01** | **0″** | **0.00** | **0″** | **0.00** | **0″** | **0.00** | **0″** | **0.00** |
| | $b$ | **1′02″** | **100** | **5′11″** | **100** | **14′29″** | **100** | **30′59″** | **100** | **56′28″** | **100** |
| $P_3$ | $a$ | 8″ | 91.7 | 35″ | 95.7 | 1′20″ | 97.1 | 2′28″ | 97.8 | 4′03″ | 98.2 |
| | $b$ | 16″ | 91.7 | 1′09″ | 95.7 | 2′53″ | 97.1 | 5′49″ | 97.8 | 9′57″ | 98.2 |
| $P_4$ | $a$ | 9″ | 100 | 37″ | 100 | 1′25″ | 100 | 2′35″ | 100 | 4′13″ | 100 |
| | $b$ | 19″ | 100 | 1′14″ | 100 | 3′05″ | 100 | 6′05″ | 100 | 10′17″ | 100 |
| $P_5$ | $a$ | 18″ | 100 | 1′15″ | 100 | 2′58″ | 100 | 5′48″ | 100 | 10′07″ | 100 |
| | $b$ | 38″ | 100 | 3′01″ | 100 | 8′20″ | 100 | 17′40″ | 100 | 31′53″ | 100 |
| **$P_6$** | $a$ | **0″** | **0.02** | **0″** | **0.00** | **0″** | **0.00** | **0″** | **0.00** | **0″** | **0.00** |
| | $b$ | **48″** | **100** | **3′34″** | **100** | **9′16″** | **100** | **18′54″** | **100** | **33′26″** | **100** |
| $P_7$ | $a$ | 10″ | 100 | 38″ | 100 | 1′26″ | 100 | 2′36″ | 100 | 4′15″ | 100 |
| | $b$ | 20″ | 100 | 1′18″ | 100 | 3′06″ | 100 | 6′08″ | 100 | 10′23″ | 100 |

($a$) EVALUATOR 3.0 (SOLVE algorithm)
($b$) EVALUATOR 2.0 (Fernandez-Mounier algorithm)

## 5 Conclusion and future work

We presented an efficient method for on-the-fly model-checking of regular alternation-free $\mu$-calculus formulas over finite labeled transition systems. The method is based on a succinct reduction of the verification problem to a boolean equation system, which is solved using an efficient local algorithm. Used in conjunction with specialized diagnostic generation algorithms [24], the method also allows to produce examples and counterexamples fully explaining the truth values of the formulas. The method has been implemented in the model-checker EVALUATOR 3.0 that we developed as part of the CADP (CÆSAR/ALDÉBARAN) protocol engineering toolset [9] using the OPEN/CÆSAR environment [13].

The input language of EVALUATOR 3.0 allows to define reusable libraries containing new temporal logic operators expressed in regular alternation-free $\mu$-calculus. At the present time, we developed libraries encoding the operators of CTL [4], ACTL [25], and a collection of generic property patterns proposed in [6] intended to facilitate the temporal logic specification activity.

EVALUATOR 3.0 has been successfully experimented on various specifications of communication protocols and distributed applications (see for instance the examples in the CADP release). The diagnostic generation features and the possibility of defining separate libraries of temporal operators appeared to be extremely useful in practice. Moreover, a connection between EVALUATOR 3.0 and the ORCCAD environment for robot controller design [26], including a graphical interface for the property pattern system, is currently under development.

In the future, we plan to apply EVALUATOR 3.0 also for bisimulation/preorder checking, by using the characteristic formula approach [16] that allows to compare two labeled transition systems $M_1$ and $M_2$ by constructing a characteristic formula of $M_1$ and verifying it on $M_2$. Also, the diagnostic generation features could be useful in the framework of test generation based on verification [10]. Using again the characteristic formula approach, test purposes could be described as temporal formulas and the corresponding test cases would be obtained as diagnostics for these formulas.

Finally, we plan to extend the logic of EVALUATOR 3.0 with data variables, which allow to reason more naturally about systems described in value-passing process algebras such as $\mu$CRL [15] and full LOTOS [17]. This can be done by translating data-based temporal logic formulas into parameterized boolean equation systems, which can be solved on-the-fly [23]. The implementation of these algorithms within the CADP toolset will require the extension of the OPEN/CÆSAR environment with data-handling facilities.

## Acknowledgements

This work was partially supported by the INRIA Cooperative Research Action TOLERE directed by Alain Girault. We are also grateful to Hubert Garavel for his useful comments and for providing valuable assistance during the development of the EVALUATOR 3.0 model-checker. Versions 1.0 and 2.0 of EVALUATOR [11] were developed by Marius Bozga and Laurent Mounier from VERIMAG.

## References


1. H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, April 1994.
2. A. Arnold and P. Crubillé. A linear algorithm to solve fixed-point equations on transition systems. *Information Processing Letters*, 29:57–66, 1988.
3. H. Bekić. *Definable operations in general algebras, and the theory of automata and flowcharts.* volume 177 of *Lecture Notes in Computer Science*, pages 30–55. Springer Verlag, Berlin, 1984.
4. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
5. R. Cleaveland and B. Steffen. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. In K. G. Larsen and A. Skou, editors, *Proceedings of 3rd Workshop on Computer Aided Verification CAV '91 (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, pages 48–58, Berlin, July 1991. Springer Verlag.
6. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering ICSE'99 (Los Angeles, CA, USA)*, May 1999. Full information available at the URL `http://www.cis.ksu.edu/santos/spec-patterns`.
7. E. A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proceedings of the 1st LICS*, pages 267–278, 1986.
8. A. Fantechi, S. Gnesi, F. Mazzanti, R. Pugliese, and E. Tronci. A Symbolic Model Checker for ACTL. In *Proceedings of the International Workshop on Current Trends in Applied Formal Methods FM-Trends'98 (Boppard, Germany)*, volume 1641 of *Lecture Notes in Computer Science*. Springer Verlag, October 1998.
9. Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.
10. Jean-Claude Fernandez, Claude Jard, Thierry Jéron, Laurence Nedelka, and César Viho. Using On-the-Fly Verification Techniques for the Generation of Test Suites. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification (Rutgers University, New Brunswick, NJ, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer Verlag, August 1996. Also available as INRIA Research Report RR-2987.
11. Jean-Claude Fernandez and Laurent Mounier. A Local Checking Algorithm for Boolean Equation Systems. Rapport SPECTRE 95-07, VERIMAG, Grenoble, March 1995.
12. M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and System Sciences*, (18):194–211, 1979.
13. Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.

14. Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.
15. J-F. Groote and A. Ponse. The syntax and semantics of $\mu$CRL. Technical Report CS-R9076, CWI, Amsterdam, December 1990.
16. A. Ingolfsdottir and B. Steffen. Characteristic Formulae for Processes with Divergence. *Information and Computation*, 110(1):149–163, June 1994.
17. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.
18. D. Kozen. Results on the Propositional $\mu$-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
19. K. G. Larsen. Proof Systems for Hennessy-Milner logic with Recursion. In *Proceedings of the 13th Colloquium on Trees in Algebra and Programming CAAP '88 (Nancy, France)*, volume 299 of *Lecture Notes in Computer Science*, pages 215–230, Berlin, March 1988. Springer Verlag.
20. X. Liu, C. R. Ramakrishnan, and S. A. Smolka. Fully Local and Efficient Evaluation of Alternating Fixed Points. In Bernhard Steffen, editor, *Proceedings of 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 5–19, Berlin, March 1998. Springer Verlag.
21. Angelika Mader. *Verification of Modal Properties Using Boolean Equation Systems.* VERSAL 8, Bertz Verlag, Berlin, 1997.
22. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume I (Specification). Springer Verlag, 1992.
23. R. Mateescu. Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In Annalisa Bossi, Agostino Cortesi, and Francesca Levi, editors, *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation VMCAI'98 (Pisa, Italy)*. University Ca' Foscari of Venice, September 1998.
24. Radu Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In *Proceedings of 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2000 (Berlin, Germany)*, Lecture Notes in Computer Science. Springer Verlag, March 2000.
25. R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems.* In *Proceedings Ecole de Printemps on Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, 1990.
26. D. Simon, B. Espiau, K. Kapellos, R. Pissard-Gibollet, and al. The Orccad Architecture. *International Journal of Robotics Research*, 17(4):338–359, April 1998.
27. R. Streett. Propositional Dynamic Logic of Looping and Converse. *Information and Control*, (54):121–141, 1982.
28. E. Tronci. Hardware Verification, Boolean Logic Programming, Boolean Functional Programming. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science LICS'95 (San Diego, California)*, pages 408–418. IEEE Computer Society Press, June 1995.
29. B. Vergauwen and J. Lewi. Efficient Local Correctness Checking for Single and Alternating Boolean Equation Systems. In S. Abiteboul and E. Shamir, editors, *Proceedings of the 21st ICALP (Vienna)*, volume 820 of *Lecture Notes in Computer Science*, pages 304–315, Berlin, July 1994. Springer Verlag.

# Verification in the Codesign process by means of LOTOS based model-checking

Fabrice Baray and Pierre Wodey

ISIMA/LIMOS Laboratory,
Blaise Pascal University Clermont Ferrand II,
BP 10125 F63173 Aubière, France,
fabrice.baray@isima.fr,pierre.wodey@isima.fr

**Abstract.** When considering the design of complex systems, the designers use ever more synthesis tools transform formal specifications into an implementation of the system. Such tools are based on a given description of the system. The description is based on a model of computation including behaviour and communication mechanisms. The model of computation depends on the level of representation of the system and varies from the specification to the implementation. There exist generally verification tools associated with the specification level but, for a more implementation oriented model the verification is often inexistent. But according to the semantic transformations in the design process (mainly at the communication level), this verification is needed. As generally implementation model of computation are composed of communicating finite state machines with datapath (CFSMD) in which the communications are performed by mean of "hardware" signals (physical connections), we propose to allow model checking verification on this model of computation. This paper presents the translation between the CFSMD and an equivalent LOTOS description in which the communication basic mechanism is the rendezvous. Based on process algebra a LOTOS description is easily translated into a labelled transition system by existing model checkers such as the CADP toolbox which we use in our experiment. We apply this technique on the COSMOS Codesign environment in which the deadlock free property has to be verified at each step of transformation in the design, the equivalence of communication semantics being not assured by the transformations. The deadlock free property is described by temporal logic formulas handled by the XTL model checker included in the CADP toolbox.

**Keywords.** Verification, Model-Checking, LOTOS, Communicating Finite State Machine, Codesign.

## 1 Introduction

For the design of complex systems the designers use ever more CAD tools working at the system level [GM93,Wol94,GV95,ELLSV97]. Such tools offer generally the following capabilities :

– formal or abstract specification of the system,

- verification at the specification level,
- architecture exploration linked with performance analysis,
- automatic synthesis of behaviour and communication,
- automatic code generation,
- simulation of the generated model.

Thus, such tools handle descriptions of the system based on model of computation. A model is composed of a behavioural (control, action) model of individual components and a communication model (communication mechanisms) among components [LSVS98].

The abstraction level of the specification formalism and its model of computation offer the ability to perform easily formal verification by a model checking technique for instance.

During the design, starting from the specification to the code generation, the description evolves together with the model of computation. Actually, the communication mechanisms at the specification level are quiet different from those at the implementation level.

Furthermore, at the communication point of view, the generated system does not implement a semantically equivalent mechanism as the one at the specification level (which is too abstract for implementation). This induces that the implemented system has not an equivalent global behaviour as the specified one. So, the properties verified at the specification level are no more guaranteed at the implementation level.

Thus, there is a need to be able to verify properties by applying model checking at the implementation level or at any level in the design process [WB99]. This needs to be able to generate a verifiable model from the model of computation at the implementation level. This is the purpose of our paper.

The considered model of computation is the communicating finite state machines with datapath (CFSMD) where the only communication mechanism is the "hardware" signal (connection net between components). This communication mechanism is at a very low level of abstraction.

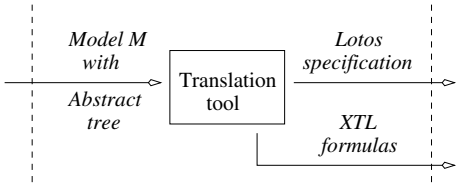



**Fig. 1.** Tool architecture

From such a description our tool generates (Fig. 1) :

- a semantically equivalent description in LOTOS language [ISO88,GLO91] in order to use model checking tools,

– temporal logic formulas allowing to verify the deadlock free property which is at this time the only one considered.

The choice of LOTOS is motivated by :

– LOTOS is an ISO standard [ISO88],
– LOTOS is based on process algebra and induces clearly a Label Transition System (LTS) needed for model checking,
– several verification tools accept LOTOS as entry.

The proposed translation has been applied considering the CADP toolbox developed at INRIA [GS90,Gar89,FGM+91,Gar98], on one hand, and on the Codesign environment COSMOS developed at TIMA Laboratory [IJ95,VCJ96], on another hand.

The CADP toolbox accepts LOTOS as entry and performs model checking on a generated LTS. It includes also logic formula checking described in the XTL language [SM98,Mat98].

The COSMOS tool is a good representative of a complete and realistic Codesign tool.

This paper is structured as follow :

– introduction of an example used to illustrate the different part of the paper and also pointing out a deadlock introduction in the COSMOS tool,
– formal presentation of the implementation oriented model of computation including behaviour and communications (CFSMD),
– the translation of the communications which are based on different mechanism in the implementation model and in the LOTOS model (rendezvous),
– the translation of the behaviour into LOTOS,
– XTL formulas automatically generated for deadlock free checking,
– the application on COSMOS Codesign tool.

## 2 Illustrating example

As example, we propose a system composed of three processes : two producers, and one consumer. The consumer accepts data from the two producers, but in some states, it limits to one specific producer. The structural representation of this simple example is given in figure 2.

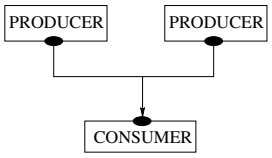



**Fig. 2.** Structural representation

The behavioural representation of the consumer is given in figure 3. The system is described in SDL [SDL88] specification language. Each component behaviour is described by a finite state machine. At this high level of specification, the consumer has three possible states. In one state, it waits independently a data from the two producers and in others states, it waits data only from one producer. In the communication point of view, in SDL, the components have gates and are communicating by asynchronous signal exchange through buffer. In the example, *Prod1_Cons_Value* are *Prod2_Cons_Value* two gates on which consumer read data. When a component send a data, it is not blocked and the data is placed in the buffer. When a component received a data, it is blocked until a data is in the buffer.

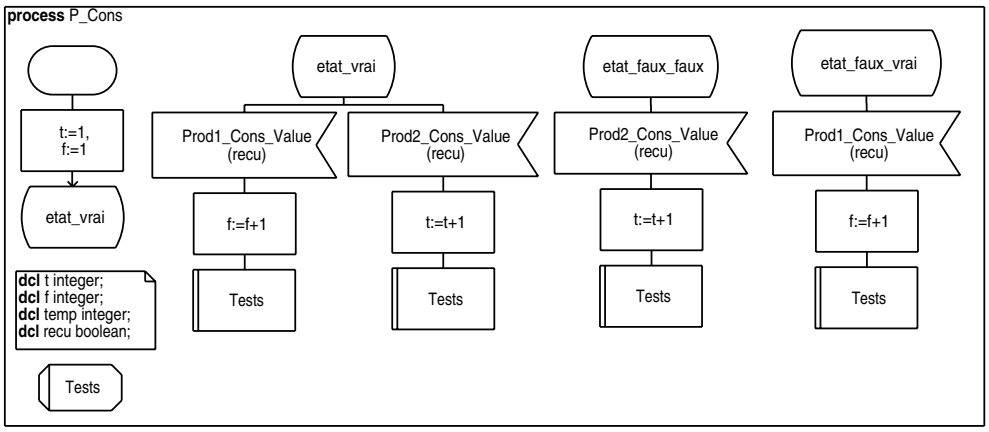



**Fig. 3.** Consumer behavioural representation

After one step of synthesis, a new description in a different model of computation is generated with a FIFO queue between the components (Fig. 4). There are three differences between the two models :

- The structural view grows with a new component and new connections between all components.
- The producers and consumer behaviours change in terms of communication protocol. The new communication principle implements hardware signals in the computational model which becomes more concrete.
- The initial and generated descriptions are semantically different in term of communication principle with the FIFO queue insertion.

## 3 Abstract model of communicating state machines

This section provides an abstract syntactic model definition of the considered CFSMD. This model is presented by inference rules which are described with

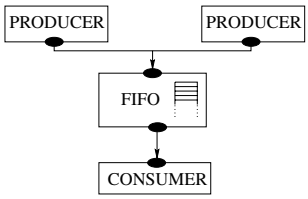



**Fig. 4.** Example after one step of communication synthesis

Backus-Naur syntax style. A system is a set of components communicating through signals. The behaviours of these components are finite state machines. Let $\mathcal{M} = \{M_0, M_1, \ldots, M_{n-1}\}$ be a model composed of $n = |\mathcal{M}|$ components. A signal $e$, used for the communication between components of the system, forms a part of the global signal set $\mathcal{E}$.

A signal communication is a non-blocking communication. Two components connected with a hardware signal are communicating by using only two actions :

- a new value can be written on the signal, which contains only one value at a time ;
- the current signal value can be read.

Hence to implement a more complicated communication protocol, many signals and many series of actions (read or write actions) are necessary. The blocking communication is implemented with loop on a state until the expected value is written by another component.

Each component $M_i$ is a tuple $(\pi, V, E, S, \lambda, \delta)$ where :

- $\pi$ is the component name (identifier) ;
- $V$ is a local variable definition list. A variable, denoted by $v$ (identifier), has an initial value $vi$ ;
- $E$ is a signal definition list. Each signal $e \in E$ is used for communication between $M$ and other components of the system. Furthermore, we have $E \subset \mathcal{E}$ ;
- $S$ is the set of states ($S = \{s_0, \ldots, s_{|S|-1}\}$). Let $s$ be a state, and $s_0$ be the initial state ;
- $\lambda$ is the "state-action" function ;
- $\delta$ is the transition function.

With each variable $v$ (respectively signal $e$) is associated its type $t(v)$ (respectively $t(e)$). The two functions (state-action and transition function) are presented by an abstract syntax. The set of terminal symbols is composed of $c$ for a constant value, $v$ for a variable identifier and $e$ for a signal identifier. And the set of nonterminal symbols is composed of $ex$ for the expressions, $ai$ for the internal actions (associated to a state), $a$ for the action associated to a transition in the model, and $\delta_a$ for the transition function definition.

– **Expressions ($ex$)**

$$ex ::= c \mid v \mid e \mid uop\, ex_0 \mid ex_0\, bop\, ex_1 \tag{3.1}$$

The $uop$ and $bop$ are unary and binary operators.

– **Internal actions ($ai$)**

$$ai ::= \varepsilon \mid v := ex\ ;\ ai_0 \mid if\ ex\ then\ ai_0\ else\ ai_1 \tag{3.2}$$

- $\varepsilon$ corresponds to no internal action ;
- ; is the sequential operator ;
- $v := ex$ is the assignment of a variable $v$ with the value $ex$ ;
- $if$ is a conditional statement.

We denote by $\mathcal{A}i$ the set of all internal actions. The $\lambda$ function of the machine $M$ is defined by $\lambda : S \rightarrow \mathcal{A}i$.

– **Actions associated to transitions ($a$)**

$$a ::= \varepsilon \mid a_0\ ;\ a_1 \mid e := ex \mid if\ ex\ then\ a_0\ [else\ a_1] \tag{3.3}$$

- in action associated to transitions, assignment operation := is only applied to signals ;
- in the $if$ statement, the expression $ex$ is a boolean expression.

– **Transition expressions ($\delta_a$)**

$$\delta_a ::= (s, a) \mid if\ ex\ then\ \delta_a^0\ [else\ \delta_a^1] \tag{3.4}$$

We denote by $\Delta_a$ the set of all transition expressions. The function $\delta$ is defined by $\delta : S \rightarrow \Delta_a$.

The figure 5 shows a part of consumer behaviour presented in figure 3 with the finite state machine model.

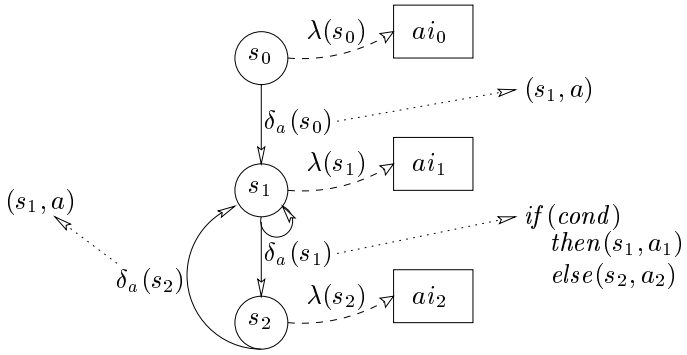



**Fig. 5.** Statemachine example on behaviour example

As example, one producer component (producer1) is defined in our syntactic model by $M = (producer1, V_{p1}, E_{p1}, S_{p1}, \lambda_{p1}, \delta_{p1})$ with $V_{p1} = \{\}$, $E_{p1} = \{bus\_req, wr\_req, wr\_ack, data\}$ and $S_{p1} = \{s_0, s_1, s_2, s_3, s_4, s_5\}$. In order to illustrate clearly the translation procedure, only two states $s_1$ and $s_2$ are detailed. For these two states, the functions $\lambda_{p1}$ and $\delta_{p1}$ are defined in box example 1.

$$\lambda_{p1}(s_1) = ai_1 \quad \text{with } ai_1 = \varepsilon$$
$$\lambda_{p1}(s_2) = ai_2 \quad \text{with } ai_2 = \varepsilon$$
$$\delta_{p1}(s_1) = \delta_{a_1} \quad \text{with} \begin{cases} \delta_{a_1} = (s_2, a_1) \\ \text{and } a_1 = bus\_req := true \end{cases}$$
$$\delta_{p1}(s_2) = \delta_{a_2} \quad \text{with} \begin{cases} \delta_{a_2} = if\ (wr\_ack = true)\ then\ \delta_{a'_2}\ else\ \delta_{a'_{2b}} \\ \text{and } \delta_{a'_2} = (s_2, \varepsilon) \\ \text{and } \delta_{a'_{2b}} = (s_3, data := 0; wr\_req := true) \end{cases}$$

EXAMPLE **1.** CFSMD example

## 4 Translation rules

LOTOS is a high level specification language based on algebraic models CSP (Communicating Sequential Processes) and CCS (Calculus of Communicating Systems). A LOTOS model of a system is composed of interconnected processes via gates. Each process communicates through gates with rendezvous communication protocol. For instance, if we consider a gate $G$, a general rendezvous in LOTOS is written by $G\ O_0 \ldots O_n$ where $O_0 \ldots O_n$ are offers defined by $O ::= !V \mid ?X_0, \ldots X_n : S$. One offer like $!V$ is the $V$ value emission on gate $G$, and one offer like $?X_0, \ldots X_n : S$ is $n+1$ receptions of values of type $S$ on gate $G$.

The translated model of CFSMD is composed of interconnected processes. With each state machine is associated one LOTOS process. This section presents :

- the structure of the LOTOS generated model and the communication principles between these processes ;
- the translation of the internal behaviour of CFSMD into the LOTOS process behaviour ;

### 4.1 Structure and communication principles

In order to reproduce the semantic of hardware signal in our communicating finite state machine, one LOTOS process named *signal* is introduced between the LOTOS processes for each hardware signal. In figure 6, the structural view of the generated LOTOS model of our example is presented.

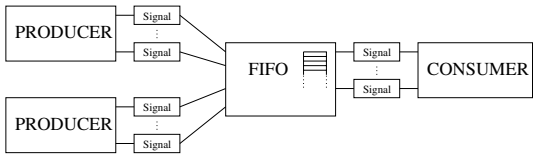



**Fig. 6.** Structural view of LOTOS generated model

The behaviour of the signal process, which is defined in a LOTOS library, is shown below in the example 2. It reproduces the semantic of the hardware signal communication. A signal is a physical link which take only one value at a time. A component can write a new value, or read the current value. The LOTOS process is based on a choice statement ([]). In the read signal action the current value is one offer of the synchronization. In the write signal action, the new value is received and memorized. Hence this is a description of a non-blocking communication with the rendezvous communication principle.

**process** $signal[s](value : signaltype)$ : **noexit** :=
    $(s!cread!value\ ;\ signal[s](value))$
    []
    $(s!cwrite?v : signaltype\ ;\ signal[s](v))$
**endproc** (* signal *)

EXAMPLE **2.** LOTOS description of a signal process

### 4.2 Translation of internal behaviour of CFSMD

Each CFSMD is translated into one process. The state variable of the state machine is defined as a LOTOS process parameter. The transition between two states is reproduced with a final recursive call of the process, with the new state value in the state parameter.

The translation of the state machine behaviour is described by a set of inference rules. Only a commented subset of rules is presented in this paper. However, this section presents some significant rules, in order to give a precise idea of our translation method[1]. After three basic definitions, some notations and environments are defined. Then the global definition of the inference rules is described for different syntactic parts of the model.

#### 4.2.1 Basic definitions

[1] It is possible to contact the authors to obtain the global set of rules

**Definition 1 (Inference rule).** *Let* $\frac{conditions}{a \rightarrow b}$ *be a rule where conditions* $= c_1, c_2, \ldots, c_n$ *must be satisfied to validate the transformation rule of a into b.*

□

**Definition 2 (Partially defined functions).** *Let* $D_1$ *and* $D_2$ *be two discrete domains. We consider the function f defined on these domains :*

$$f : D_1 \rightarrow D_2$$

*We define :*

- $\perp$ *the undefined value.* $x \in D_1$ ; $f(x) = \perp$ *means that f is undefined for the value x ;*
- *let* $x \in D_1$ *and* $y \in D_2$*, we denote by* $[y/x]$ *the function f defined only for the value x and such that* $[y/x](x) = y$ *:*

$$\forall x_i \in D_1 \quad [y/x](x_i) = \begin{cases} y \ \textit{if} \ x_i = x \\ \perp \ \ \textit{otherwise} \end{cases}$$

□

**Definition 3 (function increase).** *Let f and g be two functions defined on domains* $D_1$ *and* $D_2$*. We denote by* $f \triangleleft g$ *the function defined by :*

$$f \triangleleft g : D_1 \rightarrow D_2$$

$$\forall x \in D_1 \quad (f \triangleleft g)(x) = \begin{cases} g(x) \ \textit{if} \ g(x) \neq \perp \\ f(x) \ \textit{otherwise} \end{cases}$$

□

#### 4.2.2 Environment definitions

Let $LC$ be a LOTOS construction, and $LC = \perp$ the empty LOTOS construction. The set of all LOTOS constructions which can be written is denoted by $\mathcal{LC}$. Let $Id$ be the set of all identifiers. Two environments are defined :

- $\alpha = (\pi, id_s, E, V)$ is a tuple constructed for each component of the system. Let $\pi$ be the component name, $id_s$ the state variable name of the component behaviour, $E$ and $V$ the signal and variable sets. The LOTOS description of one component consists of one recursive process, with parameters like the state variable name, component variables and gates for the signal communication. The environment $\alpha$ is used in order to translate the recursive call of this LOTOS process ;
- $\beta$ is a second environment used to translate the signals contained in the expressions. The function $\beta$ associates with each signal a local variable name to contain the signal value :

$$\begin{array}{rl} \beta : Id & \rightarrow Id \\ e & \rightarrow v \end{array}$$

Let $Env_\beta$ be the set of all possible environments $\beta$. In order to translate the signal communications, we define the function $L$ used to generate a LOTOS construct for all signals used in $\beta$.

$$\begin{array}{c} L : Env_\beta \rightarrow \mathcal{LC} \\ \beta \quad\quad \rightarrow LC \\ \text{with } \forall e_i \text{ such that } \beta(e_i) = v_i \neq \perp \;\; LC_i = e_i!cread?v_i : t_i \\ \text{and } LC = LC_0; LC_1; \ldots \end{array} \tag{4.1}$$

### 4.2.3 Global definition of transformation rules

Five rule types are necessary for the translation. They correspond to the expressions translation, internal actions translation, the action translation, the transition translation and finally one component and the whole model translation.

- let $\beta \vdash ex \longrightarrow \langle \beta', LC \rangle$ be the rule type for the expression $ex$ translation. In environment $\beta$ the expression $ex$ is translated into LOTOS construction $LC$ and returns a micro environment $\beta'$ which contains the variable names associated to signals used in the expression ;
- let $(\alpha, \delta_a, \beta) \vdash ai \longrightarrow \langle \beta', LC \rangle$ be the rule type for the internal action $ai$ translation. In the couple of environment $\alpha$, $\beta$, and considering the $\delta_a$ transition expression, the internal action $ai$ is translated into LOTOS construction $LC$ and returns the micro environment $\beta'$ ;
- let $\beta \vdash a \longrightarrow \langle \beta', LC \rangle$ be the rule type for the action $a$ translation. In environment $\beta$, action $a$ is translated into LOTOS construction $LC$ and returns the micro environment $\beta'$ ;
- let $(\alpha, \beta) \vdash \delta_a \longrightarrow \langle \beta', LC \rangle$ be the rule type for the $\delta_a$ transition translation. In the couple of environment $\alpha$, $\beta$, the transition expression $\delta_a$ is translated into the LOTOS expression $LC$ and returns a micro environment $\beta'$ ;
- let $M \longrightarrow LC$ and $\mathcal{M} \longrightarrow LC$ be the rules for component and model translation.

### 4.2.4 Inference rules for the translation

**Signal identifier in expressions $ex$ :** according to the environment $\beta$, the translation of a signal identifier $e$ is defined with the help of two rules. If the signal has been used before, we just have to reuse its associated local variable. Else we assume that the function "newid" gives a new variable identifier for the signal $e$, and a new environment is constructed.

$$\frac{\beta(e) = ve}{\beta \vdash e \longrightarrow \langle \perp, ve \rangle} \tag{4.2a}$$

$$\frac{\beta(e) = \perp \;, \;\; ve = newid()}{\beta \vdash e \longrightarrow \langle [ve/e], ve \rangle} \tag{4.2b}$$

**Binary operator in expressions :** assuming that *bop* operator is declared for LOTOS language, a constructed environment for the binary operator in expressions with the addition of environments is defined as follows :

$$\frac{\begin{array}{c}\beta \vdash ex_0 \longrightarrow \langle \beta', LC_0 \rangle, \\ \beta \triangleleft \beta' \vdash ex_1 \longrightarrow \langle \beta'', LC_1 \rangle\end{array}}{\beta \vdash ex_0 \; bop \; ex_1 \longrightarrow \langle \beta' \triangleleft \beta'', LC_0 \; bop \; LC_1 \rangle} \tag{4.3}$$

**Variable assignment for internal actions** $ai$ must be translated with the **let** LOTOS operator such that :

$$\frac{\begin{array}{c}\beta \vdash ex \longrightarrow \langle \beta', LC_1 \rangle, \\ (\alpha, \delta_a, \beta \triangleleft \beta') \vdash ai_0 \longrightarrow \langle \beta'', LC_2 \rangle\end{array}}{(\alpha, \delta_a, \beta) \vdash v := ex \; ; \; ai_0 \longrightarrow \langle \beta' \triangleleft \beta'', (\textbf{let } v : t = LC_1 \textbf{ in } (LC_2)\;) \rangle} \tag{4.4}$$

**Signal assignment for actions** $a$ are translated into a LOTOS communication on gate denoted by $e$. This communication is prefixed by the word *cwrite* defined in the LOTOS model. It means that this is an assignment on the signal. The LOTOS communication is a rendezvous communication. In order to reproduce the signal semantic communication, this rendezvous is not implemented directly between the signal interconnected components in the model, but with a "signal LOTOS component" (see rule 4.9) :

$$\frac{\beta \vdash ex \longrightarrow \langle \beta', LC \rangle}{\beta \vdash e := ex \longrightarrow \langle \beta', e!cwrite!(LC) \rangle} \tag{4.5}$$

For instance, in example 1, the $a_1$ signal assignment on *bus_req* signal in $\delta_{a_1}$ action is translated with this rules and gives the LOTOS expression : *bus_req*!*cwrite*!*true*.

**Conditional statement for transition function** $\delta_a$ **:** three rules are required to translate the conditional statement. The first one has a restrictive condition such that it is applied when no else condition is present, and when the condition is dependent only on one signal. Then a LOTOS communication is derived with a predicate corresponding to the condition. The second rule is applied when the condition depends on more than one signal. In this case, it is not possible to create a LOTOS communication directly, thus a guarded LOTOS statement is used. The third rule is like the second one, with a else statement and the use of the $L$ function defined in 4.1 to generate the LOTOS synchronization operator :

$$\frac{\begin{array}{c}\beta \vdash ex \longrightarrow \langle \beta'', LC_1 \rangle \; , \; \beta'' = [v/s], \\ (\alpha, \beta \triangleleft \beta'') \vdash \delta_a^0 \longrightarrow \langle \beta', LC_0 \rangle\end{array}}{(\alpha, \beta) \vdash if \; ex \; then \; \delta_a^0 \longrightarrow \langle \beta', s!cread?v : \tau[LC_1]; LC_0 \rangle} \tag{4.6a}$$

$$\frac{\begin{array}{c}\beta \vdash ex \longrightarrow \langle \beta'', LC_1 \rangle \; , \; \beta'' \neq [v/s], \\ (\alpha, \beta \triangleleft \beta'') \vdash \delta_a^0 \longrightarrow \langle \beta', LC_0 \rangle\end{array}}{(\alpha, \beta) \vdash if \; ex \; then \; \delta_a^0 \longrightarrow \langle \beta' \triangleleft \beta'', ([LC_1] \rightarrow (LC_0)) \rangle} \tag{4.6b}$$

$$\frac{\begin{array}{c}\beta \vdash ex \longrightarrow \langle \beta''', LC_2\rangle,\\ (\alpha, \beta \lhd \beta''') \vdash \delta_a^0 \longrightarrow \langle \beta', LC_0\rangle \ ,\ LC_0' = L(\beta'),\\ (\alpha, \beta \lhd \beta''') \vdash \delta_a^1 \longrightarrow \langle \beta'', LC_1\rangle \ ,\ LC_1' = L(\beta'')\end{array}}{(\alpha,\beta) \vdash if\ ex\ then\ \delta_a^0\ else\ \delta_a^1 \longrightarrow \left\langle \beta''', \begin{array}{l}[LC_2] \rightarrow (LC_0' \ ;\ LC_0)\\ [] \ [\mathbf{not}(LC_2)] \rightarrow (LC_1' \ ;\ LC_1))\end{array} \right\rangle} \quad (4.6c)$$

For instance, the rule 4.6c is used to translate the condition statement of $\delta_{a_2}$ in the model example 1. In the LOTOS generated expression presented below, the value of *wr_ack* is saved in a LOTOS variable $v$ defined with the 4.2b rule :

$$\begin{array}{ll}([v] \rightarrow & producer1[\ldots](s_2)\\ [] \ [not(v)] \rightarrow & (LC_1'; producer1[\ldots](s_3)))\end{array}$$

**Next state, action in transition function $\delta_a$ :** this rule generates a recursive call for the process $\pi$, with the new values for all variables and the next state of the component :

$$\frac{\begin{array}{c}\beta \vdash a \longrightarrow \langle \beta', LC\rangle \ ,\ \alpha = (\pi, id_s, E, V),\\ E = \{e_0, \ldots, e_{|E|-1}\} \quad V = \{v_0, \ldots, v_{|V|-1}\}\end{array}}{(\alpha,\beta) \vdash (s,a) \longrightarrow \langle \beta', LC; \pi[e_0, \ldots, e_{|E|-1}](id_s, v_0, \ldots, v_{|V|-1})\rangle} \quad (4.7)$$

For instance, this rule is used to generate the recursive call of LOTOS process in $\delta_{a_1}$, $\delta_{a_2'}$ and $\delta_{a_{2b}'}$ actions in the model example 1.

**Component $M$ :** a LOTOS process construction is defined for one component. The process gates are derived from the signal set $E$ such that one signal corresponds to one gate. A variable in the component implies a parameter in the LOTOS process. For each state of component $M$, the following LOTOS construct is used in a choice statement based on the sate variable value of the process $\pi$. :

$$\frac{\begin{array}{c}V \longrightarrow LC_1 \ ,\ E \longrightarrow LC_2,\\ \forall s_i \in S \quad ai^i = \lambda(s_i) \ ,\ \delta_a^i = \delta(s_i) \ ,\ ((\pi_{tr}, id_s, E, V), \delta_a^i, \perp) \vdash ai^i \longrightarrow \langle \beta_i, LC_i'\rangle\\ \forall \beta_i \ LC_i'' = L(\beta_i)\end{array}}{\begin{array}{ll}& \mathbf{process}\ \pi\ [LC_2](id_s : state, LC_1) : \mathbf{noexit} :=\\ & \quad [id_s\ eq\ s_0] \rightarrow LC_0'' \ ;\ LC_0'\\ (\pi, V, E, S, \lambda, \delta) \longrightarrow & \quad \ldots\\ & \quad [] \ [id_s\ eq\ s_{|S|-1}] \rightarrow LC_{|S|-1}'' \ ;\ LC_{|S|-1}'\\ & \mathbf{endproc}\end{array}} \quad (4.8)$$

This rule can be used to generate the whole process statement, with the gate parameters derived from $E_{p1}$, the parameters obtained from the state variable and $V_{p1}$, and with all the LOTOS constructions for all the states in $S_{p1}$. The global LOTOS statement for this component is :

```
process producer1 [bus_req, wr_req, wr_ack, data]
                  (id_s : state) : noexit :=
   [id_s eq s_0] → ...
   [] [id_s eq s_1] → ( bus_req!cwrite!true ;
                        producer1[bus_req, ... , data](s_2))
   [] [id_s eq s_2] → ( wr_ack!cread?v : bool ;
                        ([v] →          producer1[...](s_2)
                        [] [not(v)] → (data!cwrite!0 of int ; wr_req!cwrite!true ;
                                       producer1[...](s_3))))
   ...
   [] [id_s eq s_5] → ...
endproc
```

EXAMPLE **3.** LOTOS description of producer process

**System $\mathcal{M}$ :** whole the system is described in a LOTOS specification and a library (signallib) which contains signal process definition. The specification is made up of all the instantiations of the processes associated to the components and a LOTOS synchronization to the signal process instantiation. *initvalue* is a function which associates an initial value at each variable.

$$
\begin{array}{l}
\mathcal{M} = \{M_i | 0 \leq i < n = |\mathcal{M}|\}, \\
\forall i \in [0 \ldots n-1] \left\{ \begin{array}{l} M_i = (\pi_i, V_i, E_i, S_i, \lambda_i, \delta_i), \\ M_i \longrightarrow LC_i, \\ V_i = \{v_j^i\} \quad \forall j \in [0 \ldots |V_i| - 1]\, initvalue(v_j^i) = vi_j^i, \\ E_i = \{e_0^i, \ldots, e_{|E_i|-1}^i\} \end{array} \right. \\
\mathcal{E} = \cup_{i=0}^{i=|\mathcal{E}|-1} E_i \ , \ \mathcal{E} = \{e_0, \ldots, e_{|\mathcal{E}|-1}\} \\
\hline \\
\mathcal{M} \longrightarrow \begin{array}{l}
\textbf{specification } \pi_s \ \left[e_0, \ldots, e_{|\mathcal{E}|-1}\right] : \textbf{noexit} \\
\quad \textbf{library } \textit{signallib} \textbf{ endlib} \\
\quad \textbf{behaviour} \\
\quad\quad (\pi_0 \left[e_0^0, \ldots, e_{|E_0|-1}^0\right] (s_0^0, vi_0^0, \ldots, vi_{|V_0|-1}^0) \ ||| \\
\quad\quad \ldots \\
\quad\quad \pi_{n-1} \left[e_0^{n-1}, \ldots, e_{|E_{n-1}|-1}^{n-1}\right] (s_0^{n-1}, vi_0^{n-1}, \ldots, vi_{|V_{n-1}|-1}^{n-1})) \\
\quad\quad |[e_0, \ldots, e_{|\mathcal{E}|-1}]| \\
\quad\quad (signal[e_0](Z) \ ||| \\
\quad\quad \ldots \\
\quad\quad signal[e_{|\mathcal{E}|}](Z)) \\
\quad \textbf{where} \\
\quad\quad LC_0 \ LC_1 \ \ldots \ LC_{n-1} \\
\textbf{endspec}
\end{array}
\end{array}
\tag{4.9}
$$

The simple system shown in figure 4 has been translated in LOTOS by applying this rule. The generated LOTOS description has about 400 lines. It contains

some processes : two producers, one consumer, the FIFO queue and some signals components. The structural view of the LOTOS description is given in figure 6. A part of the global LOTOS statement for this system is in example 4

```
specification ProdCons[rd_req, rd_ack, bus_req1, wr_req1, wr_ack1,
        bus_req2, wr_req2, wr_ack2, data] : noexit
  ...
behaviour
  (
    FIFO2[rd_req, rd_ack, bus_req1, wr_req1, wr_ack1,
      bus_req2, wr_req2, wr_ack2, data](q0, nil, 2ofint) |||
    P1[bus_req1, wr_req1, wr_ack1, data](q0) |||
    P2[bus_req2, wr_req2, wr_ack2, data](q0) |||
    C[rd_req, rd_ack, data](q0, 0ofint, 0ofint, 0ofint)
  )
  |[rd_req, rd_ack, bus_req1, wr_req1, wr_ack1,
        bus_req2, wr_req2, wr_ack2, data]|
  (
    signal[rd_req](zvalue) |||
    signal[rd_ack](zvalue) |||
    signal[bus_req1](zvalue) |||
    signal[wr_req1](zvalue) |||
    signal[wr_ack1](zvalue) |||
    signal[bus_req2](zvalue) |||
    signal[wr_req2](zvalue) |||
    signal[wr_ack2](zvalue) |||
    signal_int[data](0ofint)
  )
  where
    ...
endspec
```

EXAMPLE **4.** LOTOS description of producer process

## 5 Deadlock free property verification

The CADP toolbox is used for deadlock free property verification. According to the operational semantics of LOTOS, the LOTOS system specification is translated into a (possibly infinite) Labelled Transition System (LTS for short), which encodes all its possible execution sequences [SM98]. Only finite LTS can be generated with the CADP tool. An LTS is formally defined by :

**Definition 4 (LTS).** *Let* $L = \langle \mathcal{Q}, A, T, q_{init} \rangle$ *be a* LTS *such that :*

- $\mathcal{Q}$ *is the set of states of the program ;*
- $A$ *is a set of actions performed by the program. An action* $a \in A$ *is a tuple* $G\,V_1, \ldots V_n$ *where* $G$ *is a gate, and* $V_1, \ldots V_n$ *are the values exchanged (i.e., sent or received) during the rendezvous at* $G$ *;*
- $T \subseteq \mathcal{Q} \times A \times \mathcal{Q}$ *is the transition relation. A transition* $\langle q_1, a, q_2 \rangle \in T$ *(written also* $q_1 \xrightarrow{a} q_2$*) means that the program can move from state* $q_1$ *to state* $q_2$ *by performing action* $a$*;*
- $q_{init} \in \mathcal{Q}$ *is the initial state of the program.*

□

For each state $q \in \mathcal{Q}$, we denote by $Pathd(q)$ the set of all distinct paths $q(= q_0) \xrightarrow{a_1} q_1 \xrightarrow{a_1} q_2 \ldots$ issued from $q$ (such that $\forall i, j\ q_i \neq q_j$).

Typically, when an expert designs a LOTOS specification, the graph is analyzed by searching deadlocks which appear as states $q$ in the LTS such that $\nexists \langle q, a, q' \rangle$. No more communication can be done in the whole system if the behaviour reaches this sink state $q$. This technique is efficient when two conditions are satisfied :

- the specification is written assuming this search of deadlocks. In other words, it contains "true" blocking communications with the rendezvous semantics ;
- the deadlocks found are global in the system, meaning that no more communication can be done in the **whole** system. With this technique, *local deadlocks* in some processes are not detected. In a state, if one or more processes never have communication, it is possible that they are waiting for specific signal values. However other processes in the system continue their communications. This is our local deadlock definition.

In our LTS, a transition corresponds to one signal utilization. A signal utilization can be a reading (labelled cread) or writing (labelled cwrite) task. The signal processes introduced in the translated specification are designed in order to respect the signal semantics. Hence, sink states do not appear in our LTS. Furthermore, local deadlocks detection is an important issue in the context of systems derived from Codesign design.

Considering these aspects, correctness properties can be expressed with formulas inspired from ACTL temporal logic, and verified on the LTS model using the XTL model-checker [Mat98]. First, some notations (described in [SM98]) are presented, and then our deadlock correctness property are discussed.

### 5.1 Preliminary notations

Definition 5 extracted from [SM98] is presented for comprehension.

**Definition 5 (Action Formulas).** *Let* $\alpha$ *be an action formula as specified by the following context-free grammar :*

$$\begin{array}{rl} \alpha ::= & true \\ | & \{G\,V_1, \ldots V_n\} \\ | & \neg\alpha \\ | & \alpha \wedge \alpha' \end{array}$$

*where $\{G\,V_1, \ldots V_n\}$ denotes an "action pattern", $G$ a gate and all the values $V_i$ match with the corresponding values exchanged when the action is performed.*

*An action formula $\alpha$ is interpreted over an action $a \in A$. $\alpha$ satisfaction by an action $a$ of the model (*LTS*) $L$, written with $a \models_L \alpha$ (or simply $a \models \alpha$ when the model $L$ is understood), is defined by :*

$$\begin{array}{ll} a \models true & always; \\ a \models \{G\,V_1, \ldots V_n\} & iff\ a = G\,V_1, \ldots V_n; \\ a \models \neg\alpha & iff\ a \not\models \alpha; \\ a \models \alpha \wedge \alpha' & iff\ a \models \alpha\ and\ a \models \alpha'. \end{array}$$

□

The satisfaction of a formula $\varphi$ by a state $q \in \mathcal{Q}$ of a LTS $L$ is written with $q \models_L \varphi$ (or simply $q \models \varphi$ when the model $L$ is understood).

### 5.2 The deadlock free property

In order to clearly present our verification, we introduce some formulas and their semantics. First, consider a process which is waiting for a specific value of a signal. The signal is read until it takes the expected value. This classical behaviour induces in the generated LTS some state like $q$ in figure 7.

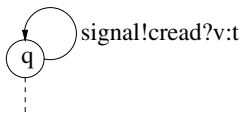



**Fig. 7.** One loop on a state

The $\varphi$ formula $EB_\alpha$ is defined by the equation 5.1. It detects the loop on a state, with a specific label $\alpha$. The global deadlocks in our system do not appear as sink states, the $AB_\alpha$ formula (equation 5.2) can be used to characterize a global deadlock on a state by evaluating $q \models AB_{true}$. This formula is almost the same as $EB_\alpha$, with a *forall* quantifier.

$$q \models EB_\alpha \text{ iff } \exists q \xrightarrow{a} q' \text{ such that } q' = q \text{ and } a \models \alpha \tag{5.1}$$

$$q \models AB_\alpha \text{ iff } \forall q \xrightarrow{a} q',\ q' = q \text{ and } a \models \alpha \tag{5.2}$$

The XTL implementation of the $EB_\alpha$ formula is given as follows :

```
def EB(LS:labelset) : stateset =
    { S : state where
        exists T : edge among out(S) in
            ((target(T)=S) and (label(T) among LS))
        end_exists
    }
end_def
```

In order to detect local deadlocks, we define a $\varphi$ formula $F_\alpha$ by the equation 5.3. A state $q$ satisfies $F_\alpha$ if and only if all the reachable states from $q$ satisfy $EB_\alpha$. The second condition in 5.3 verifies that the transitions between two distinct reachable states have actions not satisfying $\alpha$. This is not useful in our translated model because this is a deterministic model, and this condition is always true for a state $q$ of a deterministic model which satisfy the first part of $F_\alpha$. Figure 8 illustrates the equation 5.3

$$\begin{aligned} q \models F_\alpha \text{ iff } \forall P = (q \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{k-1}} q_k) \in Pathd(q), \\ \forall i \in [0;k]\ q_i \models EB_\alpha \text{ and } \forall i \in [0;k-1]\ a_i \not\models \alpha \end{aligned} \tag{5.3}$$

Hence, if $\nexists q \in \mathcal{Q}$ such that $\exists \alpha \in A$ such that $q \models F_\alpha$, then the model does not contain any local deadlock.

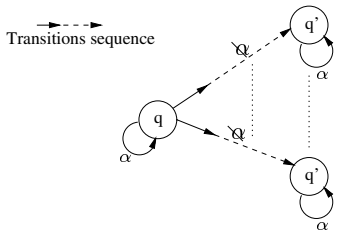



**Fig. 8.** Temporal formula illustration

Let function *succset* be a transitive closure of the successor relation, which can be achieved with a least fixed point function. Assuming that we have implemented the *succset* function in XTL, the $\varphi$ formula $F_\alpha$ is defined with :

```
def F(LS:labelset) : stateset =
    let Bs : stateset = EB(LS) in
        { S : state among Bs where Bs includes succset(S) }
    end_let
end_def
```

This formula must be evaluated with all labels contained in $A$. These labels can be automatically obtained with the analysis of the communication in the first model. In the translation, only the labels used in the XTL formulas are generated. The verification with this technique is thus a push button like function.

# 6 Application on COSMOS Codesign tool

## 6.1 COSMOS presentation

This work has been applied in the scope of the Codesign domain. In our study we consider the COSMOS tool developed at TIMA laboratory [IJ95]. The main characteristics of the COSMOS method and tool are the following :

- the specification of the system is independent of the implementation technology of the different parts of the system. This high level of abstraction description is written in SDL (Specification and Description Language [SDL88]) ;
- the use of an intermediate format SOLAR, describing the system and the communication channels among processes (CFSMD like);
- the implementation of processes in hardware or software and the choices of communications implementation (for instance a choice of a communication protocol between two or more components) are performed by several iterations of refinement steps decided manually by the designer ;
- automatic generation of the C-VHDL virtual prototype from the completely refined SOLAR description of the system ;
- cosimulation environment of the virtual prototype.

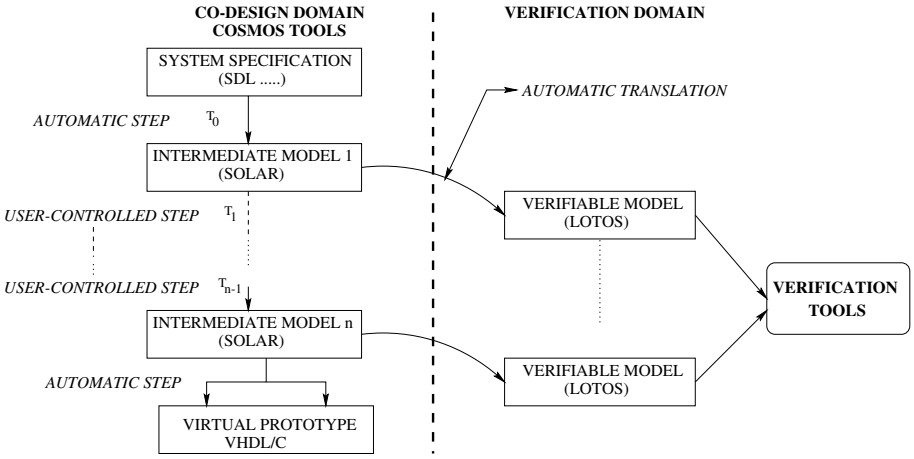



**Fig. 9.** Linking Codesign and Verification Domains

The design flow of the COSMOS tool is a sequence of refinement steps. At each step, a decision is taken by the designer, and the tool automatically integrates this decision by transforming the SOLAR description of the system. Typically this decision can be a communication synthesis among two components.

In such tools, the verification process is performed either by formal verification on the entry specification level or by cosimulation at the virtual prototype

level [VCR+95,LNV+97]. But, as the refinement is decided by the designer, and as the choice concerns communication synthesis, deadlocks can be introduced inadvertently in the system at each refinement step. The detection of such deadlocks is performed at the virtual prototype level. However this task is difficult and uncertain because :

- deadlocks generally induce active loops in the generated model,
- the link between the generated code and the initial description is not easy to establish,
- there can be several errors in the design at the virtual prototype level, yielding several decisions to modify, but these are difficult to identify,
- the virtual prototype describes the system at a low level of abstraction, the description is thus complex.

Our work on the verification of CFSMD can be used to verify the SOLAR description at each step of Codesign (Fig. 9). Our tool architecture is completed with a front end analyzer of SOLAR description like shown in figure 10.

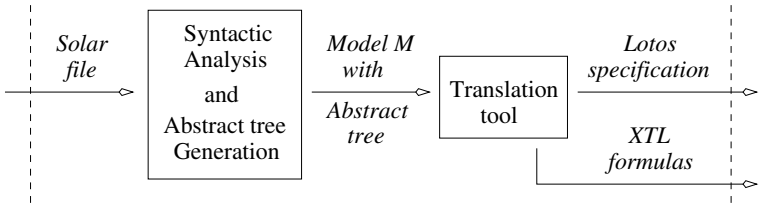



**Fig. 10.** Tool architecture

### 6.2 Example results

Considering the consumer behaviour and the protocol communication choiced, a deadlock appears in the system. This occurs when a non expected data is in front of the FIFO queue. Table 5 presents the results of the verification tools on the example. The graph generated by the CADP tool is minimized modulo strong bisimulation. The time to compute the graph is calculated on a SUN ULTRA 30. The last column presents the number of states satisfying the XTL formula $F_\alpha$, with $\alpha = (wr\_ack2!cread!false)$. The signal $wr\_ack2$ is an acknowledgement for the communication between the FIFO queue and the second producer. The $F_\alpha$ property verifies the correctness of the protocol in all execution cases.

On the reduced graphs, the XTL $AB_{true}$ property gives us one global deadlock. But, if the system is considered in an environment of other communications, this type of sink state disappears. In fact, no more communication is done between the producers, consumer and FIFO queue, but communications are made between the environment processes. This is a local deadlock. Hence the XTL $F_\alpha$ formula is performed on the LOTOS program to successfully find this local deadlock.

| Description | Reduced graph | | Time to compute | Number of states satisfying $Ba_{true}$ | Number of states satisfying $F_\alpha$ |
|---|---|---|---|---|---|
| | states | trans. | | | |
| FIFO size 1 | 322 | 1288 | 07' | 1 | 97 |
| FIFO size 2 | 652 | 2608 | 13' | 1 | 179 |
| FIFO size 2 with environment | 1304 | 6520 | 27' | 0 | 358 |

EXAMPLE **5.** CADP graph generation - XTL verification

## 7 Conclusion

In this paper, we have proposed an approach of verification of communicating finite state machines with datapath (CFSMD). This abstract model of computation with a communication principle based on hardware signal has been translated into an equivalent LOTOS description in which the communication basic mechanism is the rendezvous. Model checking verification techniques are applied on the system in order to verify deadlock property.

Then, by using this translation, we propose an approach to link the Codesign tool COSMOS with the CADP validation/verification toolbox and the XTL model-checker. COSMOS is based on refinements of the system and verification is needed when the designer chooses the implementation of communications. We intend to implement the verification as a push button function of the system. The results show the usefulness and efficiency of our deadlock verification with temporal logical formulas.

In order to apply this work with other tools, the extension of the CFSMD model to different models of communication is to study. Future work will focus on some abstractions of the communications. The goal is to study the influences of abstractions on the size of the generated LTS, and on the deadlocks search. Furthermore, we will work on larger case studies, in order to examine the complexity limits of this approach.

The study of other kind of properties which could be verified on such system will lead to a more powerful tool. Perhaps it will interesting to generate XTL formula which depend on the step of communication synthesis.

A other aspect is the study of verification based on IF language currently developed at VERIMAG laboratory [BFG+99]. This language integrates principles of communication which are different from the LOTOS rendezvous, and it is introduced in a complete open validation environment. Then we will work on the feasibility study of the system modeling with IF, and on the comparison between the verifications on a LOTOS generated model and the verifications on a generated IF model.

## References


[BFG+99] M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis. If: An Intermediate Representation for SDL and its Applications. In *Proceedings of SDL-FORUM'99, Montreal, Canada*, June 1999.

[ELLSV97] S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. In Giovanni De Micheli, editor, *Proceedings of the IEEE, Special issue on Hardware/Software Co-design*, volume 85, pages 366–390. The institute of electrical and electronics engineers, inc., March 1997.

[FGM+91] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodriguez, and Joseph Sifakis. Une boîte à outils pour la vérification de programme LOTOS. In *Actes du Colloque Francophone pour l'Ingénierie des Protocoles*, pages 479–500, September 1991.

[Gar89] H. Garavel. *Compilation et vérification de programmes LOTOS*. PhD thesis, Université Joseph Fourier, Grenoble, 1989.

[Gar98] Hubert Garavel. OPEN/CAESAR : An Open Software Architecture for Verification, Simulation and Testing. In *TACAS'98, Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*, 1998.

[GLO91] S. Gallouzi, L. Logrippo, and A. Obaid. Le LOTOS, Théorie, Outils, Applications. In O. Rafiq, editor, *CFIP'91 - Ingénierie des Protocoles*, pages 385–404. Hermes, 1991.

[GM93] R.K. Gupta and G. de Micheli. Hardware-Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers*, 10(3):29–41, September 1993.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In R.L. Probert L. Logrippo and H. Ural, editors, *10th International Symposium on Protocol Specification, Testing and Verification*, pages 379–394. IFIP, North-Holland, June 1990.

[GV95] D.D. Gajski and F. Vahid. Specification and Design of Embedded Hardware-Software Systems. *IEEE Design & Test of Computers*, 1995.

[IJ95] T.B. Ismail and A.A. Jerraya. Synthesis Steps and Design Models for Codesign. *IEEE Computer*, February 1995.

[ISO88] ISO-8807. LOTOS, a formal description technic based on the temporal ordering of observational behaviour. 1988.

[LNV+97] C. Liem, F. Nacabal, C. Valderrama, P. Paulin, and A. Jerraya. Co-simulation and Software Compilation Methodologies for the System-on-a-Chip in Multimedia. *IEEE Design & Test of Computers*, 1997. special issue on "Design, Test & ECAD in Europe".

[LSVS98] Luciano Lavagno, Alberto Sangiovanni-Vincentelli, and Ellen Sentovich. Models of computation for embedded system design. In A.A. Jerraya and J. Mermet, editors, *System-Level Synthesis*, chapter Models for system-level synthesis, pages 45–102. Kluwer Academic Publishers, 1998.

[Mat98] Radu Mateescu. *Vérification des propriétés temporelles des programmes parallèles*. PhD thesis, Institut National Politecnique de Grenoble, 1998.

[SDL88] *CCITT. Recommendation Z.100: Specification and Description Language, volume X.1-X.5*, 1988.

[SM98] M. Sighireanu and R. Mateescu. Verification of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2(1), 1998.

[VCJ96] C.A. Valderrama, A. Changuel, and A.A. Jerraya. Virtual Prototyping For Modular And Flexible Hardware-Software Systems. *Journal of Design Automation for Embedded Systems*, 1996.

[VCR+95] C.A. Valderrama, A. Changuel, P.V. Raghavan, M. Abid, T. Ben Ismail, and A.A. Jerraya. A Unified Model for Co-simulation and Co-synhesis of Mixed Hardware/Software Systems. In *The European Design and Test Conference ED& TC'95, Paris (France)*, March 1995.

[WB99] Pierre Wodey and Fabrice Baray. Linking Codesign and verification by mean of E-LOTOS FDT. In Bob Werner, editor, *Euromicro 99, Digital Systems Design*, volume 1. IEEE Computer Society, September 1999.

[Wol94] W.H. Wolf. Hardware-Software Co-Design of Embedded Systems. *Proceedings of the IEEE*, 82(7), July 1994.

# Verification of Erlang Programs: Factoring out the Side-effect-free Fragment*

Dilian Gurov Gennady Chugunov

Swedish Institute of Computer Science,
Box 1263, SE-164 29 Kista, Sweden,
dilian|gena@sics.se

**Abstract**

Erlang is a functional programming language developed at Ericsson for writing economical and yet powerful and efficient telecommunication applications. Correctness is of major importance in such applications, and since they usually exhibit a high degree of concurrency, testing is often not sufficient. Verification, namely formally proving that a system is correct, is becoming a more and more widespread practice. Due to the complexity of Erlang, there is no general method for verification of arbitrary Erlang programs which is effective and at the same time leads to economic proofs. However, one can do much better in specialised cases which are well-understood. A main direction of research is the identification of fragments of Erlang for which efficient verification methods exist. One such fragment is the side-effect-free one, in which an Erlang expression is evaluated purely for its value, and is not affecting the environment in which it is evaluated in terms of sending messages, reading from the message queue, or process spawning. This is furthermore a very common situation, given the number of libraries of side-effect-free functions used extensively in practice. The present paper presents work in progress and outlines an idea for compositional reasoning about the behaviour of an Erlang system modulo replacement of side-effect-free Erlang expressions with the result of their evaluation.

**Keywords:** software verification, Erlang, side-effect-free evaluation, compositional reasoning.

*Work partially supported by the Computer Science Laboratory of Ericsson Utvecklings AB, Stockholm, the Swedish National Board for Technical and Industrial Development (NUTEK) through the ASTEC competence centre, and a Swedish Foundation for Strategic Research Junior Individual Grant.

# 1 Introduction

Software written for telecommunication applications has to meet high quality demands. *Correctness* is one major concern; the activity of proving formally that a system is correct is called *verification*. Telecommunications software is highly concurrent, and *testing* is often not capable of guaranteeing correctness to a satisfactory degree. The software we are faced with consists of many, relatively small modules, written in the functional language Erlang [1]. These modules define the behaviour of a number of processes operating in parallel and communicating through asynchronous message-passing. New processes can be generated during execution. Because of the complexity of such software, our approach to verification is to prove that the software satisfies a set of properties formalized in a suitable logic language. The specification language we use is based on Park's $\mu$-calculus [12, 11], extended with Erlang-specific features. This is a very powerful logic, due to the presence of least and greatest fixed point recursion, allowing the formalization of a wide range of behavioural properties. Verification in this context is not decidable, but can be automated to a large extent, requiring human intervention in a few, but critical points.

For a few years now, the Formal Design Techniques group at the Swedish Institute of Computer Science has pursued a programme aimed at enabling formal verification of complex open distributed systems (ODSs) through program code verification. Previous work by the group has been directed towards establishing the mathematical machinery [5, 6], providing basic tool support [3], performing case studies [2], and motivating the chosen verification framework [8].

Due to the complexity of Erlang, there is obviously no general method for verification of arbitrary Erlang programs which is effective and at the same time leads to economic proofs. However, one can do much better in specialised cases which are well-understood. A main direction of research is the identification of fragments of Erlang for which efficient verification methods exist. One such fragment is the side-effect-free one, in which an Erlang expression is evaluated purely for its value, and is not affecting the environment in which it is evaluated in terms of sending messages, reading from the message queue, or process spawning. A very common situation are function calls to functions, the body of the definition of which are side-effect-free expressions. A large number of libraries of such functions are used extensively in practice. In all the case studies we have performed so far we had repeatedly to deal with library functions for manipulation of lists, numbers etc. Following the *compositional reasoning* paradigm used for reasoning about large component-based software, one would like in such cases is to be able to reason modulo replacement of side-effect-free function calls with the result of their evaluation. The present paper is dedicated to technically achieving this goal in a systematic fashion within our verification framework by factoring out the reasoning about the behaviour of side-effect-free Erlang expressions from the reasoning about general Erlang systems.

The paper is organised as follows. The next section summarises our ver-

ification framework. Section 3 describes the general problem of factoring out the reasoning about side-effect-free Erlang expressions from the reasoning about general Erlang systems, and presents a systematic way of performing this within our verification framework. The following section focuses on the subtask of verifying side-effect-free Erlang expressions, which is illustrated on a concrete example in section 5. The last section gives a summary and concluding remarks.

## 2 Verification of Erlang Programs

In this section we summarise our verification framework as presented in [3, 8].

**The Erlang Programming Language.** We consider a core fragment of the Erlang programming language with dynamic networks of processes operating on data types such as natural numbers, lists, tuples, or process identifiers (pid's), using asynchronous, first-order call-by-value communication via unbounded ordered message queues called mailboxes. Real Erlang has several additional features such as communication guards, exception handling, modules, distribution extensions, and a host of built-in functions.

Besides Erlang *expressions* $e$ we operate with the syntactical categories of *matches* $m$, *patterns* $p$, and *values* $v$. The abstract syntax of Core Erlang expressions is summarised as follows:

$$
\begin{array}{rcl}
e & ::= & V \mid \texttt{self} \mid op(e_1, \ldots, e_n) \mid \\
 & & e_1\ e_2 \mid e_1, e_2 \mid \texttt{case}\ e\ \texttt{of}\ m \mid \texttt{spawn}(e_1, e_2) \mid \\
 & & \texttt{receive}\ m\ \texttt{end} \mid e_1 ! e_2 \\
m & ::= & p_1 \to e_1; \cdots ; p_n \to e_n \\
p & ::= & op(p_1, ..., p_n) \mid V \\
v & ::= & op(v_1, ..., v_n)
\end{array}
$$

Here $op$ ranges over a set of primitive constants and operations including zero 0, successor $e + 1$, tupling $\{e_1, e_2\}$, the empty list $[]$, list prefix $[e_1|e_2]$, pid constants ranged over by $pid$, and atom constants ranged over by $a$, $f$, and $g$. The constructs involving side effects are: `self`, evaluating to the pid of the process evaluating this expression; `spawn`, resulting in a new process being generated; `receive` for reading from the mailbox which is assotiated with the process evaluating the expression; and "!" for sending a value to a process identified by its pid. These constructs will not be discussed in further detail here, since we focus on the side-effect-free part of the language.

To reason in a formal fashion about the behaviour of an Erlang program, a suitable formal semantics of the Erlang language is needed. This can be done in different styles, depending on the intended style of reasoning. Our approach is tailored to *small-step operational semantics*, although other formal notions

of behaviour are derivable in our framework, supporting reasoning in different flavours. Operational semantics are usually presented by transition rules involving labelled transitions between structured states [13]. A natural approach to handling the different conceptual layers of entities in the Erlang language, namely expressions, processes, and systems, is to organise the semantics hierarchically, in layers, using different sets of transition labels at each layer, and extending at each layer the structure of the state with new components as needed. A suitable formal semantics for Erlang has been recently developed, and can be found in a (yet unpublished) manuscript by Fredlund [7].

**The Property Specification Language.** Reasoning about complex systems requires *compositional reasoning*, i.e. the capability to reduce arguments about the behaviour of compound entities to arguments about the behaviours of its parts. To support compositional reasoning, a specification language should capture the labelled transitions at each layer of the transitional semantics. Polymodal logic is particularly suitable for the task, employing box and diamond *modalities* labelled by the transition labels: a structured state $s$ satisfies formula $\langle\alpha\rangle\,\Phi$ if there is an $\alpha$-derivative of $s$ (i.e. a state $s'$ such that $s \xrightarrow{\alpha} s'$ is a valid labelled transition) satisfying $\Phi$, while $s$ satisfies $[\alpha]\,\Phi$ if all $\alpha$-derivatives of $s$ (if any) satisfy $\Phi$. Additionally, *state predicates* are needed to capture the "local", unobservable characteristics of structured states, such as e.g. the value of a local variable. The presence of recursion on different layers requires also the specification language to be recursive. Adding recursion in the form of least and greatest fixed-points to the modalities described above results in a powerful specification language, broadly known as the *$\mu$-calculus* [12, 11]. Roughly speaking, least fixed-point formulas $\mu X.\phi$ express eventuality properties, while greatest fixed-point formulas $\nu X.\phi$ express invariant properties. Nesting of fixed points allows complicated reactivity and fairness properties to be expressed.

This powerful logic is capable of expressing a wide range of important system properties, ranging from type-like assertions to complex reactivity properties of the interaction behaviour of a telecommunication system. For instance, the type of natural numbers is the least set containing zero and closed under successor. The property of being a natural number can hence be defined recursively as a least fixed-point:

$$
\begin{aligned}
&N : nat \Leftarrow \\
&\quad N = 0 \\
&\vee\, \exists V.\, (V : nat \,\wedge\, N = V + 1)
\end{aligned}
$$

where $\Leftarrow$ is used for least-fixed-point definitions (i.e., for denoting the least solution of a recursive definition), while $\Rightarrow$ is used for greatest-fixed-point definitions (overloaded with implication).

**The Proof System.** Reasoning about open distributed systems written in Erlang requires reasoning about their interface behaviour relativised by assumptions about certain system parameters. Technically, this can be achieved by using Gentzen-style proof systems, allowing free parameters to occur within the *proof judgments* of the proof system. The judgments are of the form , $\vdash \Delta$, where , and $\Delta$ are sets of assertions. A judgment is deemed *valid* if, for any interpretation of the free variables, some assertion in $\Delta$ is valid whenever all assertions in , are v alid. Parameters are simply variables ranging over specific types of entities, such as messages, functions, or processes. For example, the proof judgment $x : \Psi \vdash P(x) : \Phi$ states that object $P$ has property $\Phi$ provided the parameter $x$ of $P$ satisfies property $\Psi$.

This idea of open correctness assertions gave rise to the development of a Gentzen-style proof system [6] that serves as the basis for the implementation of a verification tool. On top of a fairly standard proof system we added two rules: the first a "term-cut" rule for decomposing proofs about a compound system to proofs about the components, the second a discharge rule based on detecting loops in the proof. Roughly, the goal is to identify situations where a latter proof node is an instance of an earlier one on the same proof branch, and where appropriate fixed points have been safely unfolded. The discharge rule thus takes into account the history of assertions in the proof tree. In terms of the implementation this requires the preservation of the proof tree during proof construction. Combined, the term-cut rule and the discharge rule allow general and powerful induction and co-induction principles to be applied, ranging from induction on the dynamically evolving architecture of a system, to induction on finitary and co-induction on infinitary datatypes.

**The Erlang Verification Tool.** From a user's point of view, proving a property of an Erlang program using the verification tool involves "backward" (i.e., goal-directed) construction of a proof tree (tableau). The user is provided with commands for defining the initial node of the proof tree, for expanding a proof tree node ('the current proof node can be considered proved if the following nodes are proved instead'), for navigating through the proof tree, for checking whether the discharge rule is applicable, and for visualizing the current state of the proof tree using the daVinci graph visualization tool. Since the whole proof tree is maintained, proof reuse and sharing is greatly facilitated. Automation is achieved through a set of proof tactics and tacticals.

At the present point in time a prototype tool has been completed with the functionality described above. The largest case study performed so far is the verification of a distributed database lookup manager written in Erlang [2].

A high degree of mechanization of the low-level reasoning steps is crucial for making our verification method industrially applicable. This is the primary motivation for the ideas presented below, since the side-effect-free fragment of Erlang is well understood and classic methods exist to allow the treatment of

this fragment to be mechanized to a satisfactory degree.

# 3 Factoring out the Side-effect-free Fragment

Due to the complexity of Erlang, there is obviously no general method for verification of arbitrary Erlang programs, which is effective and at the same time leads to economic proofs. However, one can do much better in specialised cases which are well-understood. One such fragment is the side-effect-free one, in which a (possibly recursively defined) Erlang expression is evaluated purely for its value, and does not affect the environment in which it is evaluated in terms of sending messages, reading from the message queue, or process spawning.

**Compositional Verification.** The essence of compositional verification is the reduction of an argument about the behaviour of a compound system to arguments about the behaviour of its components. A system $P$ containing component $Q$ can be represented through term substitution as $P[Q/X]$, where $X$ is a variable ranging over entities of the type of $Q$. We can relativize an assertion $P[Q/X] : \Phi$ about the compound object $P[Q/X]$ to a certain property $\Psi$ of its component $Q$ by considering $Q$ as a parameter for which property $\Psi$ is assumed, provided we can show that $Q$ indeed satisfies the assumed property $\Psi$. Technically, we achieve this through a *term-cut* proof rule of the shape:

$$(\mathsf{TermCut})\ \frac{\Gamma \vdash Q : \Psi, \Delta \qquad \Gamma, X : \Psi \vdash P : \Phi, \Delta}{\Gamma \vdash P[Q/X] : \Phi, \Delta}$$

We consider the case when component $Q$ is a function call to a function the evaluation of which involves no side-effects. In this case we can offer a more powerful (de)composition principle than the one explained above.

**Side-effect-free Function Calls.** Let us consider a common situation when verifying Erlang programs. Let `proc<e, pid, q>` be an Erlang process with process identifier `pid` and message queue `q` evaluating expression `e`. Assume now that the redex (i.e., the current control point) of `e` is a function call of the shape $\mathtt{f}(Y)$, where $Y$ is a value variable. Then `e` is equal to $\mathtt{e'}[\mathtt{f}(Y)/X]$ for some Erlang expression $\mathtt{e'}(X)$ having a single occurrence of expression variable $X$ (at redex point). A proof goal involving this (open) process would generally have the form:

$$\Gamma \vdash \mathtt{proc<e,\ pid,\ q>} : \phi, \Delta \tag{1}$$

where $\phi$ is a desired property of the process, and where $\Gamma$ might put some constraints on the free variables in the process, such as $Y$. If function $\mathtt{f}(Y)$ is side-effect-free, it is evaluated only for its value. The specification of the process, and hence $\phi$, should not depend on the actual number of internal steps

which the evaluation of the function call requires. Therefore, we should be able to replace the above proof goal with:

$$, , V : \theta(Y) \vdash \texttt{proc<e'(V), pid, q>} : \phi, \Delta \tag{2}$$

where $V$ is a value variable and $\theta$ states the relation of $V$ to the input parameter(s) $Y$ of function `f`. This reduction of goal (1) to goal (2) is what we call *factoring out the side-effect-free fragment*, since it enables us to reason modulo replacement of (open) side-effect-free function calls with the result of their evaluation.

**The Reduction Steps.** There is a systematic way of performing this reduction within our proof system. It is based on the following assumptions:

- the redex of expression `e` is a function call of the shape $\texttt{f}(Y_1, ..., Y_n)$;
- the body of the definition of `f` is side-effect-free;
- property $\phi$ is insensitive to the number of side-effect-free actions (usually denoted by $\tau$), such as "eventually the process sends out a reply ...", e.g. of the shape $\mu Z.(\phi' \vee \langle\tau\rangle\, Z)$.

We explain the method on the example given above, i.e. starting with proof goal (1). First, we relativize the goal w.r.t. the specification of $\texttt{f}(Y)$, which is given as a formula of the shape $\texttt{prepost}(\psi, \theta)$ relating input values to output values of `f`. Intuitively, $\texttt{prepost}(\psi, \theta)$ states that given pre-condition $\psi$ holds of the input value $Y$, evaluation of $\texttt{f}(Y)$ terminates with a value satisfying post-condition $\theta$. Formalising `prepost` in our logic and verifying side-effect-free expressions is the topic of the next section. This relativization can be achieved through applying the term-cut rule, resulting in two new proof goals replacing goal (1):

$$, \vdash \texttt{f}(Y) : \texttt{prepost}(\psi, \theta), \Delta \tag{3}$$

$$, , X : \texttt{prepost}(\psi, \theta) \vdash \texttt{proc<e'(X), pid, q>} : \phi, \Delta \tag{4}$$

where `e'` is as explained above. Ideally, $\texttt{f}(Y)$ is a library function which has been already specified and verified; in this case goal (3) is (an instance of) a lemma and can be eliminated. If not, we use the method described in the next section to achieve this. It is also possible, that we have no access to the implementation of the function; in this case we have to leave the goal open and the final proof will be relative to the correctness of the specification. We focus our attention on goal (4). Since we intend to show that the function call results in a value, we obviously hope to be able to prove that the pre-condition $\psi$ is a consequence of the assumptions ,. So, dealing with `prepost` should result in the two proof goals replacing goal (4):

$$, \vdash \psi, \Delta \tag{5}$$

$$, , X : \texttt{eval}(\theta) \vdash \texttt{proc<e'}(X)\texttt{, pid, q>} : \phi, \Delta \tag{6}$$

where `eval` is like `prepost` but is not relativised on a precondition. `eval` states, that either (i) $X$ is a value variable $V$ satisfying $\theta$, or otherwise (ii) $X$ reduces via a silent (i.e. side-effect-free) step to another expression $X'$ satisfying $\texttt{eval}(\theta)$. In other words, dealing with `eval` should result in two proof goals replacing goal (6):

$$, , V : \theta \vdash \texttt{proc<e'}(V)\texttt{, pid, q>} : \phi, \Delta \tag{7}$$

$$, , X \xrightarrow{\tau} X', X' : \texttt{eval}(\theta) \vdash \texttt{proc<e'}(X)\texttt{, pid, q>} : \phi, \Delta \tag{8}$$

where goal (7) is the desired goal (2). It remains to eliminate goal (8). We assumed that variable $X$ stands for a side-effect-free Erlang expression occurring at redex point in `e'`. As a consequence, the only actions of $\texttt{e'}(X)$ are the silent actions of $X$. We also assumed that property $\phi$ is insensitive to the number of internal actions, e.g. of the shape $\mu Z.(\phi' \vee \langle\tau\rangle\, Z)$. These considerations imply, that dealing with $\phi$ should result in a goal replacing goal (8):

$$, , X' : \texttt{eval}(\theta) \vdash \texttt{proc<e'}(X')\texttt{, pid, q>} : \phi, \Delta \tag{9}$$

This goal is an instance of goal (6). It was obtained through unfolding a least-fixed-point formula (namely `eval`) on the left-hand side of the turnstyle symbol, and can hence be eliminated by using the discharge mechanism mentioned in the previous section.

For given shapes of formula $\phi$ the reduction outlined above can even be performed algorithmically. An important such case is when $\phi$ is itself a `prepost` formula; in this case the transition from goal (8) to goal (9) becomes straightforward and easy to mechanize.

# 4 Specification and Verification of Side-effect-free Expressions

In general, specification of the interaction behaviour of Erlang programs is a difficult task for which no systematic method has been developed so far. In the previous section we showed how to factor out the reasoning about side-effect-free Erlang expressions from the general reasoning about Erlang systems. Once we are in the realm of side-effect-free Erlang expressions we can apply well-known verification techniques.

**Natural Semantics.** Our verification method is based on a small-step operational semantics for Erlang. The proof rules contain, among other types of essertions, labelled-transition assertions of the shape $P \xrightarrow{\alpha} Q$. When reasoning about side-efect-free expressions, however, one usually prefers to work directly with the reflexive and transitive closure $e \longrightarrow^* v$ relating the expression $e$ with

the value $v$ resulting from the (terminating) evaluation (i.e. sequence of side-effect-free computations) of $e$. This kind of assertion is usually denoted $e \Downarrow v$, and the semantics based on a set of rules for reasoning in this style is usually called *natural (operational) semantics* [9]. It is the style of reasoning that we would like to employ, and this is easy to achieve, since the $\Downarrow$ predicate is definable as a least fixed-point formula through the transition relation. However, there is one significant complication: such a semantics is sufficient only if we are dealing with closed expressions $e$. But our specification and verification method is parametric in its nature, and in this case we have to somehow relate in a single construct the resulting values to the (free) parameters in $e$. This idea brings us to another well-known concept, namely the one of (weakest) pre-conditions and (strongest) post-conditions.

**Pre-conditions and Post-conditions.** A classical method for verification of sequential programs is the axiomatic method of Hoare [10]. It is based on assertions of the shape $\{\psi\}e\{\theta\}$, the intuitive semantics of which, in our context, is: given the parameters of $e$ satisfy the pre-condition $\psi$, then execution of $e$, provided it terminates, results in a value satisfying the post-condition $\theta$. We follow the same idea, but require termination; a correctness notion known as *total correctness*.

The typical sequent we have to consider is of the following shape:

$$, \ \vdash \ \mathtt{f}(Y_1, ..., Y_n) : \mathtt{prepost}(\psi, \theta), \Delta$$

where $\mathtt{f}(Y_1, ..., Y_n)$ is a function call, the body of the definition of which is side-effect free, and where `prepost` relates the values resulting from evaluating the function call to the values of the input parameters $Y_1, ..., Y_n$ of `f`. Intuitively, $\mathtt{prepost}(\psi, \theta)$ states that given $\psi$ holds, evaluation of $\mathtt{f}(Y_1, ..., Y_n)$ terminates with a value satisfying $\theta$. In our property specification language `prepost` can be defined as follows:

$$\mathtt{prepost}(\psi, \theta) \ = \ (\psi \Rightarrow \mathtt{eval}\ \theta)$$

$$\begin{array}{l} \mathtt{eval}\ \theta \Leftarrow \\ \quad \lambda E : \mathtt{ErlangExpression}. \\ \qquad \exists V : \mathtt{ErlangValue}. \ (V = E \ \wedge \ \theta\, V) \\ \quad \vee\ E : \langle \tau \rangle\, \mathtt{true} \wedge [\tau]\, \mathtt{eval}\ \theta \end{array}$$

where $\tau$ refers to side-effect-free (also called *silent*) computation steps.

The crucial question is how to handle function calls within the body of $\mathtt{f}(Y_1, ..., Y_n)$. A natural approach is to use the same technique as outlined in the previous section. When $\phi$ is a `prepost` formula, the transition from goal (8) to goal (9) becomes straightforward. We obtain the following admissable proof rule, given that $e_2$ occurs at redex point of $e_1$ (and given $\phi$ is a `prepost`

formula):

$$\text{(ValCut)}\ \frac{\Gamma\ \vdash\ e_2 : \texttt{prepost}(\psi,\theta), \Delta \qquad \Gamma, V : \theta\ \vdash\ e_1(V) : \phi, \Delta \qquad \Gamma\ \vdash\ \psi, \Delta}{\Gamma\ \vdash\ e_1(e_2) : \phi, \Delta}$$

which plays an important rôle in our proofs.

# 5 Example: The Quicksort Algorithm

In this section we illustrate our approach on the well-known quicksort algorithm. Figure 1 gives an implementation of the algorithm as an Erlang module.

**Specification.** The algorithm is to be specified as a satisfaction pair of the shape $\texttt{sort}(L) : \phi_{sort} L$ where $\phi_{sort} L$ is a formula of type $\texttt{prepost}(\psi_{sort} L, \theta_{sort} L)$. The pre-condition $\psi_{sort}\ L$ can be given as a satisfaction pair $L : \texttt{list}$ where `list` is a list specification playing the rôle of a list type. For the purposes of the present paper it is sufficient to consider a list as either being the emptylist or being decomposable into a head element and a tail list:

$$\begin{array}{l}
\texttt{list} \Leftarrow \\
\quad \lambda L : \texttt{ErlangValue}. \\
\qquad L = \texttt{[]} \\
\quad \vee\, \exists P, R : \texttt{ErlangValue}. \\
\qquad\quad L = \texttt{[}P\texttt{|}R\texttt{]} \\
\qquad \wedge\, R : \texttt{list}
\end{array}$$

The post-condition $\theta_{sort}\ L$ should relate the resulting list $L'$ to the argument list $L$. The usual way to relate the two lists is to require $L'$ to be a sorted permutation of $L$:

$$\begin{array}{l}
\theta_{sort}\ L \triangleq \\
\quad \lambda L' : \texttt{ErlangValue}. \\
\qquad \texttt{isSorted}\ L' \\
\qquad \wedge\, \texttt{isPermutation}\ L\ L'
\end{array}$$

where the predicates `isSorted` and `isPermutation` are as expected (definitions omitted).

We have also to specify the `split` function, which is called within the body of `sort` (we omit the specification of `append`). As with `sort`, the specification $\phi_{split}\ (P, R)$ of `split` is given as a formula $\texttt{prepost}(\psi_{split}\ (P, R), \theta_{split}\ (P, R))$. The pre-condition $\psi_{split}\ (P, R)$ can be taken to be $R : \texttt{list}$. The post-condition $\theta_{split}\ (P, R)$ specifies the resulting value as a pair $\{S, B\}$ of Erlang values (lists), such that the concatenation $S \cdot B$ is a permutation of $R$, $P$ is smaller than any

```
-module(sort).
-export([sort/1]).

sort ([]) -> [];
sort ([Pivot|Rest]) ->
  case split(Pivot, Rest) of
    {Smaller, Bigger} ->
      append(sort(Smaller), [Pivot|sort(Bigger)])
  end.

split(Pivot, L) ->
  split(Pivot, L, [], [])

split(Pivot, [], Smaller, Bigger) ->
  {Smaller, Bigger};
split(Pivot, [H|T], Smaller, Bigger) when H < Pivot ->
  split(Pivot, T, [H|Smaller], Bigger);
split(Pivot, [H|T], Smaller, Bigger) when H >= Pivot ->
  split(Pivot, T, Smaller, [H|Bigger]).
```

Figure 1: The Quicksort Algorithm as an Erlang Module.

element of $B$, and $P$ is bigger than any element of $S$:

$$
\begin{array}{l}
\theta_{split}\ (P, R) \triangleq \\
\quad \lambda V : \texttt{ErlangValue}. \\
\quad\quad \exists S, B : \texttt{ErlangValue}. \\
\quad\quad\quad V = \{S, B\} \\
\quad\quad \wedge\ \texttt{isPermutation}\ (S \cdot B)\ R \\
\quad\quad \wedge\ \texttt{isSmaller}\ P\ B \\
\quad\quad \wedge\ \texttt{isBigger}\ P\ S
\end{array}
$$

**Verification.** We now proceed with sketching a correctness proof. The initial proof goal is as follows:

$$\vdash \texttt{sort}(L) : \phi_{sort}\ L \tag{10}$$

Unfolding $\phi_{sort}$ leads to the new goal:

$$L : \texttt{list} \vdash \texttt{sort}(L) : \texttt{eval}(\phi_{sort}\ L) \tag{11}$$

Unfolding the definition of `list` yields the two new goals:

$$\vdash \texttt{sort([])} : \texttt{eval}(\phi_{sort}\ \texttt{[]}) \tag{12}$$

$$R : \texttt{list} \vdash \texttt{sort}([P|R]) : \texttt{eval}(\phi_{sort}\ [P|R]) \tag{13}$$

Goal (12) is easily eliminated by chosing the empty list for the existentially quantified Erlang value in the body of the `eval` property, since the empty list is (trivially) both sorted and a permutation of itself. Proceeding with goal (13), we unfold the definition of `sort`, obtaining the following goal:

$$R : \texttt{list} \vdash \texttt{case split}(P, R)\ \texttt{of}\ \ldots : \texttt{eval}(\phi_{sort}\ [P|R]) \tag{14}$$

We have reached a point where the redex of the expression is a function call. This is where we can apply rule (ValCut) introduced in the previous section, yielding three new goals:

$$R : \texttt{list} \vdash R : \texttt{list} \tag{15}$$

$$R : \texttt{list} \vdash \texttt{split}(P, R) : \phi_{split}\ (P, R) \tag{16}$$

$$R : \texttt{list}, V : \theta_{split}\ (P, R) \vdash \texttt{case}\ V\ \texttt{of}\ \ldots : \texttt{eval}(\phi_{sort}\ [P|R]) \tag{17}$$

Goal (15) is trivial. Goal (16) is just another instance of the typical sequent adressed in the previous section. By unfolding $\theta_{split}$ and the `case` statement, goal (17) is reduced to:

$$R : \texttt{list}, , \ \vdash \texttt{append}(\texttt{sort}(S), [P|\texttt{sort}(B)]) : \texttt{eval}(\phi_{sort}\ [P|R]) \tag{18}$$

where , is the form ula list:

$$\texttt{isPermutation}\ (S \cdot B)\ R, \texttt{isSmaller}\ P\ B, \texttt{isBigger}\ P\ S$$

Here again we can apply rule (ValCut), as well as some properties of `list`, to obtain:

$$S : \texttt{list}, , \ \vdash\ \texttt{sort}(S) : \texttt{eval}(\theta_{sort}\ S) \tag{19}$$

$$B : \texttt{list}, , \ \vdash\ \texttt{sort}(B) : \texttt{eval}(\theta_{sort}\ B) \tag{20}$$

$$, \ \vdash\ \texttt{append}(L_S, [P|L_B]) : \phi_{append}\ (L_S, [P|L_B]) \tag{21}$$

$$, , L_S : \theta_{sort}\ S, L_B : \theta_{sort}\ B, L : \theta_{append}\ (L_S, [P|L_B])\ \vdash\ L : \theta_{sort}\ [P|R] \tag{22}$$

Goals (19) and (20) are instances of goal (11), and can therefore be eliminated (note that the least fixed-point formula `list` was unfolded along the way, allowing application of the rule of discharge). Goal (21) is once again an instance of the typical sequent adressed in the previous section. What is left is goal (22), in which no Erlang expressions occur anymore, just variables. The treatment of such sequents is entirely standard and outside the scope of this paper.

The details of the verification of the quicksort algorithm can be found in a separate paper [4]. The approach to verification of side-effect free code is highly compositional which greatly facilitated the reuse of proof results. Around 30 lemmata were proved and used to produce a well-structured and economic proof.

# 6 Conclusion

We outlined an idea for an approach for factoring out the reasoning about the behaviour of side-effect-free Erlang expressions from the reasoning about general Erlang systems, and the subtask of verifying side-effect-free Erlang expressions, which was illustrated on the concrete example of the well-known quicksort algorithm. Future effort is needed to support the proposed approach by theoretical results, such as a proof of admissability of the ValCut rule.

Because of its simplicity, it is appealing to attempt to adapt the same reasoning scheme to Erlang expressions which do exhibit side-effects. This will in general be connected with significant complications, especially when the side-effects include process spawning, and remains a topic of future research.

**Acknowledgement.** The authors would like to thank Mads Dam and Lars-åke Fredlund at the Swedish Institute of Computer Science for carefully reading and commenting on the manuscript. Special thanks are also due to the anonymous referree who rejected the paper but provided a series of excellent suggestions for improving the scientific value of the paper.

## References


[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang (Second Edition).* Prentice-Hall International (UK) Ltd., 1996.

[2] T. Arts and M. Dam. Verifying a distributed database lookup manager written in Erlang. In *Proc. Formal Methods Europe'99,* Lecture Notes in Computer Science, 1708:682–700, 1999.

[3] T. Arts, M. Dam, L.-å. Fredlund, and D. Gurov. System description: Verification of distributed Erlang programs. In *Proc. CADE'98,* Lecture Notes in Artificial Intelligence, 1421:38–41, 1998.

[4] G. Chugunov and L.-å. Fredlund. Verifying sequential Erlang programs. Technical Report T2000:02, Swedish Institute of Computer Science, 2000.

[5] M. Dam. Proving properties of dynamic process networks. *Information and Computation*, 140:95–114, 1998.

[6] M. Dam, L.-å. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In *Compositionality: the Significant Difference,* H. Langmaack, A. Pnueli and W.-P. de Roever (eds.), Springer, 1536:150–185, 1998.

[7] L.-å. Fredlund. Towards a semantics for Erlang. Unpublished manuscript*, Swedish Institute of Computer Science*, 1999.

[8] L.-å. Fredlund and D. Gurov. A framework for formal reasoning about open distributed systems. In *Proc. ASIAN'99,* Lecture Notes in Computer Science, 1742:87–100, 1999.

[9] C. A. Gunter. *Semantics of Programming Languages.* The MIT Press, 1992. (See chapter 4.1).

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[11] D. Kozen. Results on the propositional $\mu$-calculus. *Theoretical Computer Science*, **27**:333–354, 1983.

[12] D. Park. Finiteness is mu-Ineffable. *Theoretical Computer Science*, **3**:173–181, 1976.

[13] G. D. Plotkin. A structural approach to operational semantics. Aarhus University report DAIMI FN-19, 1981.

# Specification-based Testing of Synchronous Software*

L. du Bousquet F. Ouabdesselam I. Parissis J.-L. Richier N. Zuanon
LSR-IMAG, BP 72, 38402 Saint Martin d'Hères, France
E-mail: {ldubousq, ouabdess, parissis, richier, zuanon}@imag.fr

## Abstract

*Test data generation and test execution are both time-consuming activities when done manually. Automated testing methods promise to save a great deal of human effort. This especially applies to reactive programs which have complex behaviors over time and which require long test sequences.*

*In this article, we present Lutess, a testing environment for synchronous reactive software. Lutess produces automatically and dynamically test data with respect to some environment constraints of the program under test. Moreover, it allows to trace the test execution and spot the situations where the program violates its properties.*

*Lutess offers several specification-based testing methods. They aim at simulating more realistic environment behaviors, producing relevant data to test thoroughly a given property or driving the program under test into interesting situations. To produce the test data, the methods use different types of guides: statistical distribution of the input generation, properties, or behavioral patterns.*

*Lutess proved to be powerful and easy to use in industrial case studies. Lutess won the Best Tool Award of the First Feature Interaction Detection Contest. The tool is described hereafter from both practical and formal points of view.*

**Keywords** Automated testing, synchronous reactive software, telecommunications systems, Lustre.

## 1 Introduction

A reactive software must continually respond to signals from its environment, and must satisfy temporal constraints so that it can capture all the external events of concern.

Synchronous programs are a sub-class of reactive software. They are deterministic, they are never blocked, and they satisfy the synchrony hypothesis [2] which states that every reaction of the software application to external events is theoretically instantaneous (actually, fast enough to ensure that the environment remains invariant during the computation of the reaction).

Reactive or synchronous systems are often safety-critical and must be thoroughly validated to ensure that they meet their requirements mostly by means of formal verification, or intensive simulation or testing.

In this paper, we are concerned with functional (black box) testing of synchronous pieces of software. The purpose of functional testing is to reveal errors in order to help the tester get confidence in the software correctness [14]. No testing hypotheses [3] is made on the software behavior (no regularity for example), nor on the software input space (no uniformity for example). Therefore, the validation of reactive software requires that it does maintain its relation with its environment over long sequences of exchanges and the number of input-output relations (test cases) to be managed is really large. These relations can't be easily computed by hand, since the reactive system input and output usually depend on the system history (and not

*This work has been partially supported by a contract between CNET-France Telecom and University Joseph Fourier, #957B043.

*This paper is a combination of three other articles [8, 10, 21].*

only on its current input). Thus, testing should be automated in order to make it easier, improve its quality and lower its cost.

Lutess is a testing environment that supports highly automated testing of synchronous reactive systems [19]. Lutess is mainly suited to the test of the control part of reactive software since it deals with programs involving only boolean input and output signals. High level specifications of telecommunication services and control-command software are typical examples of Lutess application fields. Lutess offers different testing methods in order to fit the tester needs as well as possible. For instance, test data can be produced so that the most used operations would receive the most testing (as in [18]), others can be randomly generated or based upon an input partition (as in [15]).

The aim of this paper is to provide an overview of the tool and its foundations. The usefulness of each testing method is illustrated with an example concerning the validation of a telecommunication feature specification, namely the Call Forwarding No Reply.

The paper is organized as follows. Section 2 gives a brief description of the principles of Lutess. Section 3 provides an example of the application of the synchronous approach to the modeling of a telephony system. Section 4 presents the test data generation methods provided by Lutess from a practical point of view, section 5 details their formal foundations and section 6 outlines the test data selection algorithms. Section 7 is devoted to the implementation of the tool. Section 8 explores the advantages and the shortcomings of the tool. Section 9 introduces related work.

## 2 Lutess

Our approach to functional testing of synchronous software consists in examining whether a program satisfies some stated properties. These properties are requirements imposed on the program behaviors, such as "a user's phone goes back to its idle state every time the user goes on the hook". An important point of this kind of validation is that it is done under assumptions about the possible behaviors of the environment interacting with the software. When one is not concerned with the software robustness, it makes no sense to take into account impossible environment behaviors. For example, it is physically impossible for the user of a telephone to go on the hook twice without going off the hook in between. When considering a telephony system, only sequences among which "go off" and "go on" actions alternate are meaningful with respect to testing.

### 2.1 Architectural overview

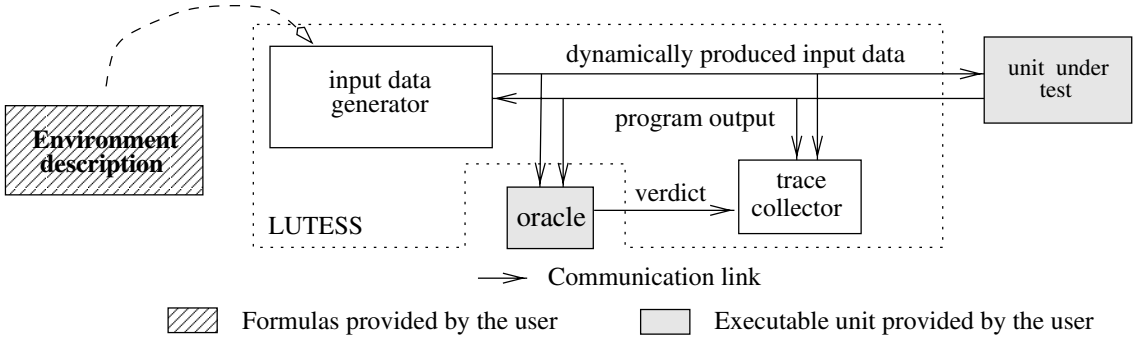



**Figure 1. Lutess**

Lutess requires three elements: the *unit under test* (that is, the software or software component under test), its *environment description* and an *oracle* (as shown in figure 1). The oracle is an implementation of the software requirements. Lutess constructs automatically the test harness which builds a test data generator, links the generator, the unit under test and the oracle, coordinates their executions and records the sequences of input-output values and the associated oracle verdicts (test sequences).

The test is operated on a single action-reaction cycle, driven by the generator. The generator randomly selects an input vector for the unit under test and sends it to the latter. The unit under test reacts with an output vector and feeds back the generator with it. The generator proceeds by producing a new input vector and the cycle is repeated. The oracle observes the program inputs and outputs, and determines whether the software requirements are violated.

The test data generator is automatically built by Lutess from an environment description written in Lustre[1] [4]. This description is provided as a single syntactical unit, called a *testnode* [20]. Examples of environment description and oracle properties are given in section 3.

The unit under test and the oracle are both executable programs with boolean inputs and outputs. They must have a synchronous behavior but they have not to be necessarily supplied as Lustre programs.

To begin the test data generation, one has to feed Lutess with a probability seed, which is used to initialize a classical random number generator. Keeping in mind that the behavior of a synchronous program is deterministic, i.e. in a given state, its response to a given input value is always the same, one can note that the use of such a generator allows to reproduce any experiment, by using the same seed. For a given program and a given environment description, Lutess requires different seeds in order to produce different test sequences.

Moreover, the user has to specify the number ($n$) and the length ($l$) of the test sequences that Lutess has to produce. The process which produces the $n$ test sequences of $l$ values is called a *test run*. During a test run, the program is reset in its initial state at the beginning of each new test sequence.

Finally, Lutess includes a "trace collector" which provides 3 functions:

- Storing the input, output and oracle data (boolean values) into specific files.
- Displaying the traces in a textual mode, defined by the user (an example is given in table 1). This makes the manual trace analysis more comfortable.
- Replaying a test sequence (for example using different oracles).

### 2.2 Lutess testing methods

During a test run, at each cycle (or step), the Lutess generator randomly selects an input vector for the system under test. Basically, the input is selected using the environment description (black-box testing), and assuming that the data distribution is uniform. But the user can also define:

- an input statistical (partial) distribution; the generator will produce inputs according to the given distribution;
- some (safety) properties; the generator will select preferably inputs which potentially drive the system under test toward those properties violation;
- some scenarios (behavioral patterns); the generator will select preferably inputs which follow the scenario.

These methods are described in sections 4 and 5.

## 3 Example

As an illustration of Lutess application, we consider a telephony system offering the Call Forwarding No Reply feature (CFNR)[2]. This feature allows a subscriber to have his incoming calls redirected when he does not answer within a given delay. The feature is dynamically activated and deactivated. The number to which calls are redirected is also dynamically set.

The telephony system is modeled from the users' viewpoint. Its environment includes the physical telephones which are linked to the system (figure 2). The system we consider is composed of 4 users (called A, B, C, D).

System inputs (issued by the environment) are events describing the actions performed on the phones: $\text{On}_i$, $\text{Off}_i$, $\text{Dial}_i(j)$, $\text{CFon}_i(j)$, $\text{CFoff}_i$, with $i$ and $j \in \{\text{A}, \text{B}, \text{C}, \text{D}\}$). The event $\text{CFon}_i(j)$ indicates that the

[1] Lustre is both a synchronous programming language and a temporal logic.

[2] This example is taken from a case study aiming at modeling feature specifications from their ETSI descriptions [7].

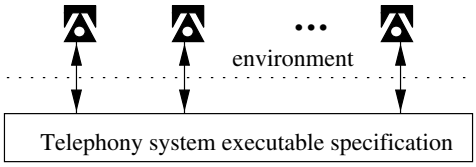



**Figure 2. Telephony System Model**

user $i$ requires the activation of his CFNR feature to forward his calls towards $j$; the event CFoff$_i$ means that user $i$ demands his CFNR feature to be deactivated.

Outputs are signals which produce specific tones at the terminal (such as Busy-Tone, Ringing-Tone, ... ). Each output signal identifies the state of the phone. In this example, we use the traditionnal Basic Call Model [16] which depicts the call processing as state machines. A phone has 7 states, which are *idle* (I), *dialing* (D) *waiting* (W), *alerting* (A), *talking* (T), *ringing* (R), and *exception* (E)[3].

We suppose that the telephony system has a synchronous behaviour: at a given instant, it reads its inputs and computes instantaneously its outputs, and so on forever. Moreover, for sake of modelling simplicity, it is assumed that at most one of the system inputs can be true at the same time.

To perform the validation of this system, the human tester has to exhibit the environment description and the system requirements (oracle properties). In this example, the two sets of formulas are provided in Lustre.

Lustre [4] is a programming language for synchronous programs, which is declarative and data-flow oriented. It corresponds to a linear past temporal logic which offers usual arithmetic, boolean and conditional operators and two specific temporal operators: **pre**, the "previous" operator, and $->$ the "followed-by" operator. In Lustre any variable or expression is intended to be a function of a discrete time (time is assimilated to the set of natural numbers, $0$ denoting the initial instant). In other words, a Lustre expression denotes the sequence of values it takes over the different instants of time. Let *E* and *F* be two expressions denoting the sequences $(e_0, e_1, \ldots, e_n \ldots)$ and $(f_0, f_1, \ldots, f_n \ldots)$; **pre**$(E)$ denotes the sequence $(nil, e_0, e_1, \ldots, e_{n-1} \ldots)$ where $nil$ is an undefined value. $E -> F$ denotes the sequence $(e_0, f_1, \ldots, f_n \ldots)$.

Lustre allows the specifier to define its own logical or temporal operators to express invariants. For example, in this paper, we use the temporal operator **once_from_to**(A, B, C) to specify that property A must hold at least once between the instants where B and C occur. The exact Lustre definition is:

```
node once_from_to(A, B, C: bool) returns(X: bool);
let                    -- Note: implies(A, B) computes the value of A ⇒ B
    X = implies (after(B) and C, once_since(A, B));
tel;
node once_since(C, A: bool) returns(X: bool);
let
    X = if A then C
            else if after(A) then C or pre(X)
            else true;
tel;
node after(A: bool) returns(X: bool);
let
    X = false -> pre(X or A);
tel;
```

[3]A phone is waiting when a number has been dialed and the connection has not been established yet. It is alerting when the connection is established but the party has not gone off the hook yet. When an error occurs, the phone enters the exception state until its user goes on hook.

### 3.1 Environment description

1. As stated before, at most one event can occur at each instant of time. Considering the events to be *On$_i$, Off$_i$, Dial$_i$(j), CFon$_i$(j), CFoff$_i$*, with $i$ and $j \in \{$A, B, C, D$\}$, this constraint is written in Lustre as below:
   (E1) #(*On$_A$*, *Off$_A$*, *Dial$_A$*, ..., *CFoff$_D$*)
   where # is a Lustre operator stating that "at most one element of the parameter list is true".

2. A user can't go off (resp. on) the hook twice without going on (resp. off) the hook in between:
   (E2) $\mathbf{once\_from\_to}(On_i, \mathbf{pre}\ Off_i, Off_i)$ **and**
   $\mathbf{once\_from\_to}(Off_i, \mathbf{pre}\ On_i, On_i)$.

3. A user should dial only if his telephone emits the *DialingTone*:
   (E3) $Dial_i \Rightarrow DialingTone_i$

4. A user can (try to) activate and deactivate the CFNR service only when his telephone emits the *DialingTone*:
   (E4) $\forall j, (CFon_i(j)\ \mathbf{or}\ CFoff_i) \Rightarrow DialingTone_i$

The environment constraints E1, E2, E3, E4 have to be inserted in a testnode (see below). As it can be noted, the testnode inputs (resp. outputs) are the system's outputs (resp. inputs). This should be understood as "the generator receives the program outputs as inputs, and generates (i.e. returns) input data for the program. The selected input data satisfies the properties contained in the *environment* argument list".

**testnode** *Environment (o1, o2, ..., $o_m$ : program_outputs)*
**returns** *(i1, i2, ..., $i_n$ : program_inputs);*
  **var** *l1, l2, ..., $l_k$ : local_variables;*
  **let**
    *environment(E1, E2, E3, E4);*
  **tel**;

Obviously, achieving a complete specification of the environment is not realistic. Thus, any detected violation of the software requirements has to be analyzed by the tester, since it can result from many causes: an error in the tested unit, an environment behavior which should have been discarded, or an incorrect requirement statement.

### 3.2 Oracle properties (system requirements)

As a preliminary definition, we say that the CFNR feature is *invoked* for a user, if the latter is a CFNR subscriber which has activated this service, and if he/she does not answer a call within the time delay.

1. A call will be forwarded if (1) the callee feature is invoked and (2) the maximum number of forwards is not reached. This bound is a service provider option which was set to 3 for our example.

2. A call can be forwarded only if the service has been previously activated by the callee, and if the latter did not deactivate the service in the meantime.

3. A forwarded call will be redirected to the last user which has been designated by the subscriber.

It is easy to write in Lustre an oracle program from these properties. One has to express each property in Lustre by defining intermediary variables and by using Lustre classical operators. Consider, for instance, the predicate *LastUser(x)* that takes into account the last activation of the feature by user $x$:

$$LastUser(x) = \mathbf{if}\ CFon(x,y)\ \mathbf{then}\ y\ \mathbf{else}\ \mathbf{pre}\ LastUser(x)$$

With this predicate, we can express the last of the above properties as follows:

$$p_3 = (CallForward(x,y) => LastUser(x) = y)$$

*CallForward(x, y)* is true whenever a call for $x$ is forwarded to $y$ ($x, y \in \{A, B, C, D\}$).

Then, from the Lustre expression of these properties, say $p_1, p_2$ and $p_3$, we build a Lustre program the inputs of which are the inputs and outputs of the program under test. Its unique output is the conjunction of the oracle properties expressed in Lustre:

**node** *Oracle (program_inputs; program_outputs)*
**returns** *(res : boolean);*
  **var** *l1, l2, . . . , $l_k$ : local_variables;*
  **let**
    *res* = $p_1$ and $p_2$ and $p_3$ *;*
  **tel***;*

# 4 Principles and usages of the testing methods

## 4.1 Basic random testing

According to this basic technique test data are generated only with respect to the environment specification without any additionnal consideration (black-box testing). This is the weakest test data selection criterion one can define for synchronous software. The test data generation is performed in such a manner that the data distribution is uniform. Table 1 gives an example of a trace that Lutess has produced with this method, according to an output format defined by the tester.

```
 1: -      -   -      -   -      -   -      -   I I I I True
 2: OffA   -   -      -   -      -   -      -   D I I I True
 3: CFonA  (D) -      -   -      -   -      -   E I I I True
 4: -      -   -      -   -      -   -      -   E I I I True
 5: -      -   -      -   -      -   OffD   -   E I I D True
 6: -      -   -      -   -      -   DialD  (D) E I I W True
 7: -      -   -      -   OffC   -   -      -   E I D W True
 8: -      -   -      -   CFonC  (B) -      -   E I E E True
 9: -      -   OffB   -   -      -   -      -   E D E E True
10: -      -   DialB  (A) -      -   -      -   E W E E True
11: -      -   -      -   OnC    -   -      -   E W I E True
12: -      -   -      -   -      -   OnD    -   E E I I True
13: OnA    -   -      -   -      -   -      -   I E I I True
14: -      -   -      -   -      -   -      -   I E I I True
15: OffA   -   -      -   -      -   -      -   D E I I True
16: CFonA  (C) -      -   -      -   -      -   E E I I True
17: -      -   OnB    -   -      -   -      -   E I I I True
18: -      -   OffB   -   -      -   -      -   E D I I True
19: -      -   CFonB  (D) -      -   -      -   E E I I True
20: -      -   -      -   OffC   -   -      -   E E D I True
21: -      -   -      -   OnC    -   -      -   E E I I True
<a><-------------------- b ----------------><-- c --><d >
   < User A >< User B >< User C >< User D >
```

(a) Step number;
(b) $\text{User}_x$ action and its parameter ($\text{Off}_x$, $\text{On}_x$, $\text{Dial}_x$(y), $\text{CFon}_x$(y), $\text{CFoff}_x$; $x, y \in \{A, B, C, D\}$);
(c) $\text{Phone}_x$ state (Idle, Dialing, Waiting, Alerting, Talking, Ringing, Exception);
(d) Oracle verdict (issued by the oracle defined in section 3.2).

**Table 1. A trace generated by Lutess**

### Empirical observations

Very often, a uniform distribution is far from the expected real software use. Indeed, test data in table 1 show that some users' phones stay off the hook for long periods of time in Exception state (i.e. after receiving a Busy Line indication), e.g. user A between states 4 and 13. In reality, a user would have quickly gone on the hook in such a situation. Similarly, many generated behaviors consist in alternating going off and on the hook, performing no action in between (user C, step 20 and 21), which is not a common behavior. We also noticed that, on the whole, every user tries to call himself/herself as often as any other user (user

D, step 6) or to activate the CFNR feature several times in a row (user A, steps 3 and 16). In the real world, such behaviors rarely occur, and are most of the time the result of wrong actions.

In order to test or analyze more realistic simulations, one may want to specify its own statistical environment distribution. With Lutess, this is possible thanks to probabilities that one can associate with program inputs.

### 4.2 More realistic random testing

Lutess offers facilities to define in the testnode a multiple probability distribution [24] in terms of conditional probabilities associated with the unit under test input variables [6]. The variables which have no associated probabilities are assumed to be uniformly distributed. A conditional probability assignment defines, for an input variable, its probability to be set to true when a given condition is met (when no condition is provided the probability is unconditional). The conditions are Lustre expressions. An algorithm is implemented in Lutess to automatically translate a set of conditional probabilities into an operational profile (and vice versa).

An operational profile describes how users employ a system (a system usage). Using an operational profile to guide testing insures that the operations involved in the system usage of concern will receive the most testing [18].

Let us try this method on our example. The conditional probabilities are chosen in order to overcome the problems exhibited by the previous empirical observations. For instance, to decrease the time spent by one user's phone in the Exception state, we specify that the probability to go on the hook is high while the phone is in the Exception state.

$$\langle OnA, 0.9, \mathbf{pre}\, ExceptionA \rangle$$

Let $c_1, c_2, \ldots, c_s$ be a list of conditional probabilities. Similarly to the environment constraints, the conditional probabilities are declared in the testnode, in the following way:

**testnode** *Environment (o1, o2, ... , $o_m$ : program_outputs)*
**returns** *(i1, i2, ... , $i_n$ : program_inputs);*
  **var** *l1, l2, ... , $l_k$ : local_variables;*
  **let**
    *environment(E1, E2, E3, E4);*
    *proba($c_1, c_2, \ldots, c_s$);*
  **tel**;

#### Empirical observations

Regarding the last unrealistic aspect mentioned in the previous subsection, we defined about 15 conditional probabilities for each user. There are 5 possible actions for each user, and approximately 3 conditional probabilities per action which may have different values depending on the phone states. For instance, the probability to go on the hook is usually different in the states Exception, Dialing and Talking.

A realistic environment simulation may not produce data which test rare but important and interesting features of the program. To overcome this problem, Lutess has two different methods which consist in testing in a more relevant manner some given properties or to drive the program into interesting situations. These methods produce data according to two types of guides: (invariant) properties and behavioral patterns.

### 4.3 Property-oriented testing

Property-oriented testing is aimed at selecting test data which facilitate the detection of property violations. At each cycle, this method automatically generates values which are relevant to test the considered properties.

We say that an input data is relevant to test a property, when the program reaction is liable to cause an instantaneous failure with respect to this property. For instance, let's consider the simple property $\mathcal{P} : i \Rightarrow o$, where $i$ (resp. $o$) is an input (resp. output) of the unit under test. When $i$ is false, the unit under test cannot

falsify $\mathcal{P}$. When $i$ is true, the unit under test will falsify $\mathcal{P}$ if it returns the value false for $o$. Hence, $i$ =true is relevant to test $\mathcal{P}$.

Input values which are relevant to the considered properties are favored over the other input values. But the random selection process is fair enough to let those latter values be exercised. In Lutess, the properties chosen to guide the generator ($s_1, s_2, \ldots, s_z$) have to be defined with the environment description, in the testnode, by means of the *safety* operator. Conditional probabilities can also be used in combination with this method.

**testnode** *Environment (o1, o2, ... , $o_m$ : program_outputs)*
**returns** *(i1, i2, ... , $i_n$ : program_inputs);*
  **var** *l1, l2, ... , $l_k$ : local_variables;*
  **let**
    *environment(E1, E2, E3, E4);*
    *proba($c_1, c_2, \ldots, c_s$);*
    *safety($s_1, s_2, \ldots, s_z$);*
  **tel**;

### Empirical observations

One property of the telephony system is that the user's phone goes back to its idle state every time its user goes on the hook. Driving the generation with such a property led to favor the considered action, thus improving the tester's confidence in the system's reaction to this input. However, this resulted in every user tending to go on the hook as soon as possible; thus, many more realistic behaviors are never tested.

## 4.4 Behavioral pattern-based testing

As complexity grows, reasonable behaviors for the environment may reduce to a small part of all possible ones with respect to the constraints. Some interesting features of a system may not be tested efficiently since their observation may require sequences of actions which are too long and complex to be randomly frequent.

The behavioral pattern-based method aims at guiding further the input generation so that the most interesting sequences are produced. A behavioral pattern characterizes those sequences by listing the actions to be produced, as well as the conditions that should hold on the intervals between two successive actions (figure 3). Regarding input data generation, all sequences matching the pattern are favored and get higher chance to occur. To that, desirable actions appearing in the pattern are preferred, while inputs that do not satisfy interval conditions get lower chance to be chosen. The generation method is usually invoked with environment constrained test data. Behavioral patterns are stated using a trace-like notation which is automatically translated in Lustre expressions.

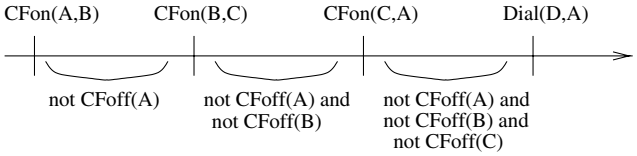
### Empirical observations

To avoid loops in the forwarding, specifying the CFNR feature requires that no more than 3 redirections are ever performed on a single call in a row. When checking what could happen in the case of more then 3 redirections, we noticed that this situation had little chance to occur. The use of a pattern has resulted in an increase of the number of 3 redirections occurrences in shorter test sequences. Figure 3 shows the graphical representation of such a pattern.

# 5 Formal framework

This section provides a formal framework for the testing methods, in order to show explicitly their applicability and some of their limits.

In the following, for any set $X$ of boolean variables, $V_X$ denotes the set of values of the variables in $X$. $x \in V_X$ is an assignment of values to all variables in $X$.




Upper conditions describe the sequence of actions to be produced.
Lower conditions are interval conditions.

**Figure 3. Example of a behavioral pattern**

### 5.1 Formal definition of an environment simulator

The abstraction of an environment simulator is derived from an I/O machine. This environment simulator is non deterministic, i.e. it uses an non-deterministic method to generate values respecting the environment constraints.

**Definition 1** *An environment simulator (or a generating machine) is defined as* $M_{env} = (Q, q_{init}, O, I, t, env, out_{env})$ *where*

- $O$ *(resp.* $I$*) is the set of the UUT output (resp. input) variables.*
- $Q$ *is a finite set of states,*
- $q_{init} \in Q$ *is the initial state (also denoted* $q_0$*),*
- $env \subseteq Q \times V_I$ *represents the environment specification.*
- $t : Q \times V_O \times V_I \rightarrow Q$ *is the (total) transition function.*
- $out_{env}$ *is a method which, given* $q \in Q \backslash \{q \mid \nexists i (q, i) \in env\}$*, chooses one element from* $S_{env}(q) = \{i \mid (q, i) \in env\}$*, the set of all valid UUT inputs.*

In every state, a new UUT input is issued. Next, the UUT computes an output which enables a transition. This behavior can be expressed in terms of UUT inputs ($i_k$) and outputs ($o_k$).

$$
\begin{array}{|l}
For\ k = 0, \ldots \\
\quad i_k \leftarrow out_{env}(q_k) \\
\quad read(o_k) \\
\quad q_{k+1} \leftarrow t(q_k, o_k, i_k)
\end{array}
$$

(B)

**Remark 1**: Consider a testnode handling two software inputs $(i, j)$, whose environment constraint is *env = pre i and j*. When the associated I/O machine is in a state where *pre i = false*, there is no input value for which the constraint holds. However, if $i$ is always set to true, the machine would never be blocked. Thus, an environment simulator has no guarantee to be reactive. For a given $q$, the set $S_{env}(q)$ may be empty. The theoretical means to determine whether the generator is reactive is to compute the set of reachable states, or its complement (i.e., the set of states leading inevitably to the violation of $env$). These computations are based on a least fixed point calculation which can be impracticable [13, 22]. This is why, in terms of implementation, we don't try to compute $S_{env}(q)$ a priori. We rather try to detect blocking situations during the generation.

**Remark 2**: By default, $out_{env}$ is implemented as a method that consists in selecting inputs from $S_{env}(q)$ according to an equally probable distribution.

### 5.2 Formal definition of a property-guided machine

A property-guided machine is a generating machine which generates test sequences (cf. section 6.2) which are more liable to invalidate a predicate (cf. section 4.3).

**Definition 2** *Let* $M_{env} = (Q, q_{init}, O, I, t, env, out_{env})$ *be a generating machine and* $f_P \subseteq Q \times V_O \times V_I$ *be a predicate representing a property* $P$.
*A UUT input value* $i \in V_I$ *(adequately) tests* $P$ *on state* $q \in Q$ *(adequate*$_P$*(q, i)) iff* $\exists o \in V_O,\ \neg f_P(q, o, i)$.

**Definition 3** *A property-guided machine is a generating machine* $M_P = (Q, q_{init}, O, I, t, env, out_P)$ *where*

- $P$ *is a conjunction of properties,*
- *Let* $S_{env \cap adequate_P} = \{i \in V_I \mid (q, i) \in env \cap adequate_P\}$. *The method* $out_P$ *chooses a value from* $S_{env \cap adequate_P}$ *if this set is not empty; otherwise it selects a value from* $S_{env}$.

Whenever it is possible to produce an input value which adequately tests the properties, all input values which do not test adequately the properties are ignored.

**Remark 3**: Note that the adequate test data is searched for in the current state. Thus the technique is limited to an instantaneous guiding. When considering a safety property like $pre\ i \Rightarrow o$, the generator does not discover that setting $i$ to true will test the property at the following step. Moreover, a property such as $(not\ i\ or\ o)$ **and** $(pre\ i\ or\ o)$ would be adequately tested only with values capable of falsifying $(not\ i\ or\ o)$, that is $i$ being set to $true$. This prevents it from selecting an adequate data for the rest of the property $(pre\ i\ or\ o)$, since $pre\ i$ will always be $true$.

## 5.3 Formal definition of an operational profile-guided machine

An operational profile-guided machine is a generating machine that generates test sequences (cf. section 6.3) that conform to a given operational profile (cf. section 4.2).

**Definition 4** *An operational profile-guided machine is a generating machine* $G_{prof} = (Q, q_{init}, O, I, t, env, out_{CPL})$ *where*

- $CPL = (cp_0, cp_1, \ldots, cp_k)$ *is a list of conditional probabilities. Each* $cp$ *is a 3-tuple* $(i, v, f_{cp})$ *where* $i$ *is an input variable (*$i \in I$*),* $v$ *is a probability value (*$v \in [0..1]$*), and* $f_{cp}$ *is a condition (*$f_{cp} \subseteq Q \times V_O \times V_I$*).* $v$ *denotes the probability that the variable* $i$ *takes on the value true when the condition* $f_{cp}$ *holds.*
- $out_{CPL}$ *is such that the selection method is no longer equally probable and depends on the conditional probability list.*

When the conditional probability list is empty, the machine is equivalent to the basic one. The conditional probability list (partially) overrides the by-default equally probable distribution of the basic generating machine.

## 5.4 Formal definition of a pattern-guided machine

A pattern-guided machine is a generating machine that generates test sequences (cf. section 6.4) that follow a given behavioral pattern (cf. section 4.4).

A behavioral pattern (BP) is made out of alternating and ordered instant conditions and interval conditions. The instant conditions must be satisfied one after the other as time progresses. Each interval condition shall be continually satisfied between the two successive instant conditions which border it. A behavioral pattern characterizes the class of input sequences that match the sequence of conditions.

A behavioral pattern (BP) is built with the following syntax rule, where a simple predicate (SP) is a Lustre boolean expression which does not include the current outputs:

$$BP ::= [SP]\ SP \ \mid \ [SP]\ SP\ BP$$

The non-bracketed predicates represent the instant conditions, while the bracketed predicates correspond to interval conditions. $[true]\ CFon(A, B)\ [not\ \ CFoff(A)]\ CFon(B, C)$ is an example of a BP. BPs provide a means to partially describe a sequence: the inputs between two instant conditions may take any value provided that the interval condition holds.

With a behavioral pattern is associated a *progress* variable which indicates what prefix of the BP has been satisfied so far. To any value this variable can take corresponds a pair of predicates {*inter*, *cond*} which describes the next-to-appear predicate and the predicate that should continually hold in the meantime.

**Definition 5** *A pattern-guided machine is defined as* $G_{pat} = (Q, q_{init}, O, I, t, env, out_{BP}, progress)$ *where*

- $(Q, q_{init}, O, I, t, env, out_{BP})$ *is a generating machine*
- $BP = [true]cond_0[inter_1]cond_1 \ldots cond_{n-1}[inter_n]cond_n$
- *progress is an integer variable taking its value over* $V_{progress} = [-1, 0..n+1]$. *It is the progress index on* $BP$.
- *Let* $S_{\mathcal{H}}, S_{\mathcal{L}}, S_{\mathcal{N}} : Q \times V_{progress} \rightarrow V_I^*$ *be sets of input variables defined as,* $\forall q \in Q, \forall j \in V_{progress}$*:*
  - $S_{\mathcal{H}}(q, j) = \{i \in V_I \mid (q, i) \in cond_j \cap env\}$
  - $S_{\mathcal{L}}(q, j) = \{i \in V_I \mid (q, i) \in \neg inter_j \cap \neg cond_j \cap env\}$
  - $S_{\mathcal{N}}(q, j) = \{i \in V_I \mid (q, i) \in inter_j \cap \neg cond_j \cap env\}$
- *Given* $q$ *and* $j$*, the current state and progress values, the method* $out_{BP}$ *first selects a non-empty set among the above, then performs the standard value selection within this set. As a side effect,* $out_{BP}$ *also computes the next value for progress:*
  - *if* $S_{\mathcal{H}}(q, progress)$ *is chosen,* $progress$ *is incremented,*
  - *if* $S_{\mathcal{L}}(q, progress)$ *is chosen,* $progress$ *is set to* $-1$*,*
  - *if* $progress = -1$ *or* $n+1$*,* $progress = 0$.

Intuitively, the partition is motivated by the status of the transitions regarding the progression of the guiding process: $S_{\mathcal{H}}$ includes all input that make the process go forward, $S_{\mathcal{L}}$ groups those that lead to the process stopping, while $S_{\mathcal{N}}$ gathers all transitions that do not affect the process.

**Remark 4**: Definition 5 does not ensure that the guiding process will lead to the completion of the pattern, i.e. to generate sequences that match it. Indeed, there may exist a reachable state for which a $progress$ value makes both $S_{\mathcal{L}}$ and $S_{\mathcal{H}}$ empty. If the guiding process makes the machine reach this state, the process can't progress nor regress anymore and becomes quiescent for the remaining of the test. Many other similar situations may occur, that prevent from completing the pattern. However, all of them are due to an incorrect description of the pattern. This description should be cautiously performed.

## 6 Test data selection

The automaton obtained by compiling the environment constraints is coded using a symbolic notation in which the states are represented by a set of boolean variables, and the transitions by boolean functions.

The environment constraints (i.e. the $out_{env}$ method) are implemented as a Binary Decision Diagram (BDD) [1] (for sake of presentation, in the figure, the BDDs is represented by a Shannon tree (ST)).

For example, figure 4 shows the ST associated with a BDD built by Lutess for the following constraint : "at most one entry among B1, B2 and B3 is available at each time". In the ST, 0 and 1 stand for respectively false and true. Each node of the diagram carries a variable and each of its outgoing branches is labelled with the value taken by that variable. The left(resp. right) sub-ST corresponds to the assignment of a false (resp. true) value to the root variable. A path from the ST root to a leaf represents an input state. If the input state is valid with respect to the environment constraints, the terminating leaf carries a true value. The input space contains 8 input states, among which only 4 are valid.

All the generation techniques rely on the same principle. The test data generator uses the environment BDD to randomly select one input state which satisfy the constraints, so that the associated boolean function takes a true value.

### 6.1 Random testing by environment simulation

The basic random generation algorithm produces equally probable input values. To guarantee to all the valid input vectors an equal probability, the value of $e$ is set in function of the following probabilities:

$$p(e = \texttt{true}) = \frac{v_1}{v_0 + v_1} \quad \texttt{and} \quad p(e = \texttt{false}) = \frac{v_0}{v_0 + v_1}$$

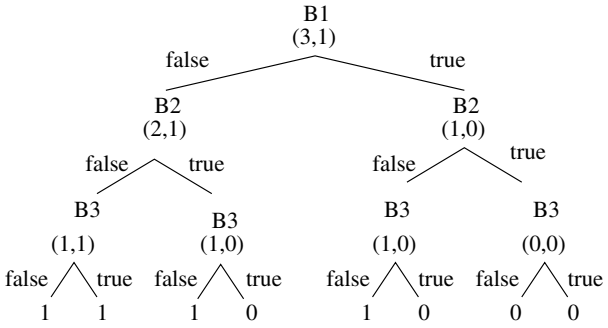



**Figure 4. Labelled BDD for equally probable generation.**

where $v_0$ (resp. $v_1$) is the number of distinct paths leading to a leaf carrying a $true$ (resp. a $false$) value in the sub-ST the root of which is the node associated with $e$. $v_0$ and $v_1$ are computed at the begining of the simulation process for all the input nodes of the BDD.

At each cycle, the generator performs four operations:

- locate, in the diagram describing the environment constraints, the sub-diagram corresponding to the current values of the state,
- generate a random value for the software inputs satisfying the boolean function associated with that diagram,

In other words, the generator searches in the diagram associated with the constraints a path leading to a true leaf.

### 6.2 Property-oriented testing

This technique is implemented by building a new BDD from the output method and the properties to be tested. The resulting BDD allows to check whether a given state and a given value of the inputs both satisfy the environment constraints and are liable to exhibit an error with respect with the properties. The basic algorithm is modified as follows:

- locate, in this late diagram, the sub-diagram corresponding to the current value of the state,
- check whether there exists at least one value for the inputs which can lead to a true leaf in this diagram,
- if positive, randomly select one of these values; otherwise, perform the basic algorithm.

### 6.3 Operational profile-based testing

The generation algorithm uses both the previous labelled BDD and the conditional probability list.

Let $CP(e)$ = $((p_1, ce_1), (p_2, ce_2), \ldots, (p_r, ce_r))$ be a list of conditional probabilities associated with the input variable $e$. In $CP(e)$, $p_j$ denotes the probability that the variable $e$ takes on the value true when the condition $ce_j$ is true. The selection function assigns a value to $e$ according to the following probabilities:

$$\begin{cases} p(e = \texttt{true}) = & \texttt{if } ce_1 \texttt{ then } p_1 \\ & \texttt{else if } ce_2 \texttt{ then } p_2 \\ & \texttt{else if } \ldots \\ & \texttt{else if } ce_r \texttt{ then } p_r \texttt{ else } \frac{v_1}{v_0+v_1} \\ p(e = \texttt{false}) = & (1 - p(e = \texttt{true})) \\ & \textit{with } v_0 \textit{ and } v_1 \textit{ referring to the basic labelling} \end{cases}$$

### 6.4 Behavioral pattern-based testing

Given the pattern to be matched, the method drives the generator to consider at every cycle the pair of predicates {*inter*, *cond*} corresponding to the current value of the *progress variable*. At each step, first,

the input space is computed to get all the possible inputs meeting the environment specification. It is then divided into three categories: $S_{\mathcal{H}}$, $S_{\mathcal{L}}$ and $S_{\mathcal{N}}$ as stated in definition 5.

A probability is assigned to each category so that an input in the first one would be favored over an input in the third category, which, itself, would be preferred to an input from the second category. These probabilities are determined with respect to the cardinality of each partition and to given weights associated with them: $w_{\mathcal{H}}$, $w_{\mathcal{L}}$ and $w_{\mathcal{N}}$. A partition is said to be of higher priority than an other if its weight is greater.

The input selection is a two-step process. First, a category is selected according to the determined probabilities. Each category $c$ in $\mathcal{C}$=$\{S_{\mathcal{H}}, S_{\mathcal{L}}, S_{\mathcal{N}}\}$ has a probability $p_c$ of being selected:

$$p_c = \frac{w_c * card(c)}{\sum_{j \in \mathcal{C}} w_j * card(j)}$$

Then, an input is chosen in an equally probable manner from the selected category. As a result, the probability for any input *i* in *c* to be chosen is $p_{i,c}$:

$$p_{i,c} = \frac{1}{card(c)} * p_c = \frac{w_c}{\sum_{j \in \mathcal{C}} w_j * card(j)}$$

The implementation of the algorithm is also based on the environment BDD. Each predicate in the pattern is represented by a BDD. The predicate BDDs and the environment BDD are combined to identify the input sets $S_{\mathcal{H}}$, $S_{\mathcal{L}}$ and $S_{\mathcal{N}}$. These BDD are labelled in the very same manner than for the basic generation.

Every generation step involves therefore the traversal of the three diagrams corresponding to the current value of *progress*. The traversal leads to the subdiagrams corresponding to the current environment state, where the cardinality for $S_{\mathcal{H}}$, $S_{\mathcal{L}}$ and $S_{\mathcal{N}}$ can be retrieved, thanks to the labelling. The selection is then performed with respect to the given weights and the calculated cardinalities.

## 7 Tool implementation and validation

The tool code represents 26000 lines of C++. Lutess has been used intensively during several case studies, among which the "Feature Interaction Detection Contest" held in association with the 5th Feature Interaction Workshop [9, 11]. The goal was to detect possible and undesired interactions between twelve telecommunication services. For this case study, the test process for each of the 78 configurations involved 10 to 20 sequences of 1000 to 10000 steps each. On the whole, each configuration has been tested for around 1 million test cases. The Lutess tool was run over 1500 times.

For this case study, we also considered applying a model-checker Lesar [12] to evaluate the ability of verification method to detect feature interactions [5]. Preliminary results show that the model-checker cannot deliver a result in most of the 78 configurations, because of lack of time or memory amount. On the contrary, Lutess always returns a verdict.

Building the BDD structure corresponding to a given environment is the most expensive part of the testing process. In our experiments, environments included between 32 and 45 constraints, plus up to 8-step patterns or 40 conditional probabilities. It has always been possible to perform this computation and to run the test on a Sparc Ultra-1 station with 128 MB of memory. Maximum of required virtual memory amounts to 100 MB. Though, as the number of constraints describing the environment increases, the BDD complexity rises and its generation lasts longer. For the less-constrained environments that we produced, 6 seconds on CPU were necessary, while the most-constrained environments required about 30 minutes for the corresponding BDD to be generated. As a comparison, a 1000 test run lasts about 2 minutes once the BDD has been generated[4]. So, the more the environment is constrained, the more relevant is the test (since the whole test case is more realistic), but the longer is the BDD generation.

Several $\chi^2$ tests were performed in order to check that the statistical methods produce data according to the different assumptions (i.e. that the basic statistical method produces data in an equally-probable way and that the method guided by conditional probabilities produces data with respect to the defined probabilities). Those tests have shown that these assumptions are valid.

[4]This second phase of the testing process is proportional to the length of the test sequence.

# 8 Advantages and limitations

## 8.1 Advantages in using Lutess

Lutess offers a unified framework for synchronous program testing. Basically, a generator produces test data which satisfy an environment description. Lutess proposes different types of guidelines the user can use to describe a more realistic environment or make the test more relevant. Unlike the environment description, these additional guidelines are not to be strictly enforced. As a result, all valid behaviors are still possible, while the more reasonable ones are more frequent. The model of the environment is thus more "realistic". The environment description and the guidelines have to be described in the same language (Lustre) and in the same framework (the testnode).

The use of conditional probabilities or patterns proved to be highly profitable when prototyping the application: these techniques allow to have a quick feedback on the correction of the implementation. Then, when it comes to validate the implementation (test its conformance to the specification), these techniques drive the environment to follow a realistic evolution. Meanwhile, thanks to the probabilistic aspect introduced in both methods, the behaviors of the environment may vary and involve rare and unforeseen scenarios. Such cases, close to the expected behavior –yet unexpected– are realistic and thus worth to be tested.

Lutess has a user-friendly interface (figure 5). It offers the user an integrated environment:

- to define the environment description, the oracle and the unit to be tested (in the fields *Program under test, Oracle* and *Environment*),
- to command the construction of the test harness, and to build constrained random generators (with *Begin*, *Kill* and *Continue* buttons,
- to set the random seed, the number and the length of the data sequences,
- to compile Lustre programs, to format the sequences of inputs, outputs and verdicts and to replay a given sequence with a different oracle (with *Tools* menu and *Redo* button),
- to visualize the progression of the testing process. Usually, Lutess does not stop the test generation process at the first oracle violation. This is especially useful for checking when some specific event occurs.

The three components required by Lutess (the unit under test, the environment description and the oracle) are just connected together and not compiled into a single executable code. This allows the tester to easily change a component, for example to replay a test sequence with a new oracle, or to fix the environment specifications.

## 8.2 Limitations

Lutess can only generate data for boolean input and output synchronous programs. We have always been able to by-pass this potential drawback yet, by using boolean vectors for enumerated data types.

For the moment, it is possible to use property-oriented testing in combination with conditional probabilities. But it isn't possible to use behavioral patterns with conditional probabilities. We are currently working on this point.

Specifying the software environment by means of invariant properties is a rather delicate task. Indeed, one should adequately choose a set of properties which do not "overspecify" the environment. Overspecifying may prevent some realistic environment behaviors from being generated.

The theory underlying Lutess does not provide a means to evaluate when the test should be stopped. In fact, it is quite hard to define a meaningful coverage criterion. For instance, classical coverage criteria (coverage of code instructions or branches of control flow graph) are very loosely related to the set of the possible program behaviors. Different experiments have been conducted to examine how code coverage could be realated with fault detecting. While basic criteria such as instruction coverage or branch coverage have been easily met, long sequences of test cases have been generated without resulting in any increase in the multiple conditions coverage criterion, beyond some level.


**Figure 5. Lutess interface**

## 9 Related work

Jagadeesan et al. have presented a technique and a toolset that represent the most similar work to Lutess [17]. Compared to Lutess, this approach appears to be limited in several respects. There is no guiding using operational profiles or scenario-like methods. Environment constraints are only taken into account to restrict the size of the input space. Inputs are selected with uniform weights. The whole process is based on the compilation of the oracle, the application and the test harness into one single executable code; recompiling is necessary after each modification, which caused the biggest dissatisfaction, according to what the authors said.

As we said before, Lutess can only generate data for synchronous programs with boolean inputs and outputs. In [23], Halbwachs et al. describe another synchronous testing tool, Lurette, which was built to take into account numerical data. Lurette requires also three elements, and like Lutess, needs a Lustre environment description. Lurette has no elaborated strategies for boolean data generation, but has a strategy for integer and real data generation.

## 10 Conclusion and future work

In this article, we presented Lutess, a highly automated testing environment for synchronous software and illustrated its use on an example. This automation allows to transfer the human efforts from the classical tester's chores (selecting the data, determining the result validity) to more defect prevention tasks (e.g., developing specifications).

Lutess offers several specification-based testing methods in order to fit the tester needs as well as possible. These methods aim at simulating more realistic environment behaviors, producing relevant data with respect to some properties or interesting situations. These methods produce test data using different type of guides, which are conditional probabilities, properties, and behavioral patterns.

We mainly conducted two experiments: a first case study of feature specification validation based on the ETSI recommendations [7], and a second one in the framework of the FIW contest [9]. Experience has confirmed that this approach is highly cost-effective. Both case studies showed that the guiding techniques were excellent at finding problems involving rare scenarios. This positive experience was reinforced by the valuable application of Lutess in the software specification stage, which helped get confidence in these specifications. All this has certainly contributed to make Lutess the "best tool" of the FIW contest [11].

Trace analysis is an important task, even if the verdict is automatic, since it can reveal unsuspected problems. Besides, writing relevant specifications in the appropriate format for test data generation should be facilitated. An environment to support these tasks is under consideration. It should integrate proving techniques to decide on formulae equivalence. Future directions also include criteria to determine when to stop testing and a notion of error coverage associated with the existing testing techniques.

## References


[1] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27:509–516, 1978.

[2] A. Benveniste and al. Synchronous Technology for Real-Time systems. In *The 1994 Real-Time Conferences*, pages 104–122, Teknea, 1994.

[3] G. Bernot, M-C. Gaudel, and B. Marre. Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6:387–405, 1991.

[4] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages (POPL 87), Munich*, pages 178–188. ACM Press, 1987.

[5] L. du Bousquet. Feature Interaction Detection using Testing and Model-checking, Experience report. In *World Congress on Formal Methods*, Toulouse, France, September 1999.

[6] L. du Bousquet, F. Ouabdesselam, and J.-L. Richier. Expressing and implementing operational profiles for reactive software validation. In *9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, 1998.

[7] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Incremental Feature Validation : a Synchronous Point of View. In *Feature Interactions in Telecommunications Systems V*, pages 262–275. IOS Press, 1998.

[8] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: a Specification-driven Testing Environment for Synchronous Software. In *21st International Conference on Software Engineering*, pages 267–276. ACM Press, May 1999.

[9] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Feature Interaction Detection using Synchronous approach and Testing. *Computer Networks and ISDN Systems*, to be published, 2000.

[10] L. du Bousquet and N. Zuanon. An Overview of Lutess, A Specification-based Tool for Testing Synchronous Software. In *14th IEEE International Conference on Automated Software Engineering*. IEEE, October 1999.

[11] N.D. Griffeth, R. Blumenthal, J.-C. Gregoire, and T. Otha. Feature interaction detection contest. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications Systems V*, pages 327–359. IOS Press, 1998.

[12] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Programming Language LUSTRE. *IEEE Transactions on Software Engineering*, pages 785–793, september 1992.

[13] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous Observers and the Verification of Reactive Systems. In *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Twente*. Workshops in Computing, Springer Verlag, 1993.

[14] D. Hamlet. Software Quality, Software Process and Software Testing. *Advances in Computers*, 1995.

[15] D. Hamlet and R. Taylor. Partition Analysis Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, pages 1402–1411, december 1990.

[16] ITU-T. Principles of intelligent network architecture. Recommandation Q.1201, 1993.

[17] L.J. Jagadeesan, A. Porter, C. Puchol, J.C. Ramming, and L. Votta. Specification-based Testing of Reactive Software: Tools and Experiments. In *19th International Conference on Software Engineering*, 1997.

[18] J. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, pages 14–32, march 1993.

[19] F. Ouabdesselam and I. Parissis. Testing Synchronous Critical Software. In *5th International Symposium on Software Reliability Engineering*, Monterey, USA, 1994.

[20] I. Parissis. *Test de logiciels synchrones spécifiés en Lustre*. PhD thesis, Université Joseph Fourier, Grenoble, France, september 1996.

[21] I. Parissis and F. Ouabdesselam. Specification-based Testing of Synchronous Software. In *4th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, San Francisco, USA, 1996.

[22] P. Ramadge and W. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM J. Control and Optimization*, 25(1):206–230, january 1987.

[23] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*. IEEE, 1998.

[24] J. Whittaker. *Markov chain techniques for software testing and reliability analysis*. PhD thesis, University of Tenessee, 1992.

# Formalization and Testing of Reference Point Facets

Ina Schieferdecker, Mang Li, Axel Rennoch

GMD FOKUS
Kaiserin-Augusta-Allee 31, D-10589 Berlin, Germany
phone: +49 30 3463-7000, fax: +49 30 3463-8000

{schieferdecker, m.li, rennoch}@fokus.gmd.de
www.fokus.gmd.de/tip

***Abstract***

*The paper introduces a new concept to express the architecture and behavior of distributed systems in a formal, detailed and extensible manner in terms of reference point facets (RP-facets). RP-facets are based on the well-established concept of reference points as used in ODP and TINA. Facets describe statical and dynamic aspects of reference points as well as pre- and post-conditions for their use. The paper gives a mathematical characterization of RP-facets, defines a specification template for the definition of RP-facets and derives a conformance test method for their validation. An example taken from the TINA retailer reference point shows the application and practical use of RP-facets.*

## 1 Introduction

TINA (Telecommunications Information Networking Architecture [12]) is an open system architecture for telecommunication systems in a multi-vendor environment. TINA combines modern methodologies and techniques, such RM-ODP [6] and CORBA [9] to support the development of large-scale systems. Core concepts and specifications of TINA have been established. TINA is in its maturity phase, where conformance evaluation plays an important role. In TINA, telecommunication stakeholders are characterized by their business roles, e.g. consumer, retailer or third-party service provider. Since every stakeholder represents an autonomous administrative domain, the implemented sub-systems used by stakeholders operate in a heterogeneous, unpredictable, and uncontrollable environment. Inter-domain reference points are introduced to ensure the interoperability of the various sub-systems [13].

The TINA architecture addresses a wide range of issues and provides a complex set of concepts and principles. It has been partitioned into several models, subsystems, components, etc. in order to handle the complexity. An essential partitioning concept is that of reference points (RPs). Reference points consist of a set of interfaces together with potential interactions at these interfaces. Reference point specifications define conformance requirements for a relationship between or within administrative domains of distributed systems. The TINA reference point concept follows the RM-ODP conformance assessment principles [6]. An example for an inter-domain reference point is the Retailer Reference Point [14].

Key issues for multi-vendor systems are interoperability and interworking. Reference points as a collection of conformance requirements are the basis to increase the likelihood for interworking and interoperability. Conformance testing is an effective and efficient means to validate the overall

functionality of a multi-vendor system. It is a well-established alternative to the otherwise needed many-to-many test setups for individual components and/or sub-systems to ensure their interoperability and interworking.

The Conformance Testing Methodology and Framework (CTMF) [5] is a well accepted technology in the area of protocol testing. CTMF defines test architectures and the test notation TTCN (Tree and Tabular Combined Notation) for the evaluation of capability and behavioral conformance of protocol implementations. CTMF allows the modelling and specification of both centralized and distributed test systems. It has been used for ISDN, ATM, Internet protocols and many others. A recent work shows also the usability of CTMF for object-oriented systems [3].

Reference points provide a simple straight-forward means to express the TINA architecture in terms of objective requirements for conformance. However, current defined TINA reference points tend to be too large. They are inadequately structured and do not allow incremental specification, implementation, and testing.

Therefore, the design for testability of reference points is a requirement to enable and further facilitate the testing process for distributed systems. In order to support conformance testing more effectively and efficiently, we propose a new partitioning concept for reference points: the concept of *reference point facets* (RP-facet)[1] [2] [16].

Reference points can be composed from RP-facets and/or segmented into RP-facets. Each of these RP-facets (or subtopics) has it's own concepts, partitioning, information model, and other details. Conformance can be tested separately for each of these RP-facets. Thus, a system may be tested at multiple levels with respect to various RP-facets representing different aspects of a reference point. This kind of testing is consistent with the current usage of TINA specifications, and allows a vendor to implement limited roles in the business of service provisioning.

In this paper, we present the concept of and specification techniques for RP-facets in Section 2 and 3, resp. Section 4 discusses RP-facet based test specification and a conformance test method based on RP-facets. An example showing the overall approach is presented in Section 5. Conclusions finish the paper.

## 2 TINA Reference Points

TINA inter-domain reference points[3] are located at the border of administrative domains in a telecommunication system and are used to define conformance and interoperability requirements for the business relationships between the telecommunication stakeholders, which are in certain business roles. The TINA business model (see Figure 1) defines five business roles: consumer, retailer, third-party service provider, broker and connectivity provider. It defines the following inter-domain reference points:

1. The term facet is used in the OMG CORBA Component Model (CCM). In order to avoid misunderstandings, RP-facet is used instead.
2. Please note that although we consider reference point facets under the realm of TINA the results of this work is in general applicable to distributed system, which use a notion of reference points as interface/set of conformance requirements to the outside.
3. Intra-domain reference points are not considered in this paper.

- Retailer inter-domain reference point (Ret)
- Broker inter-domain reference point (Bkr)
- Third-party inter-domain reference point (3Pty)
- Retailer-to-retailer inter-domain reference point (RtR)
- Connectivity service inter-domain reference point (ConS)
- Terminal connection inter-domain reference point (TCon)

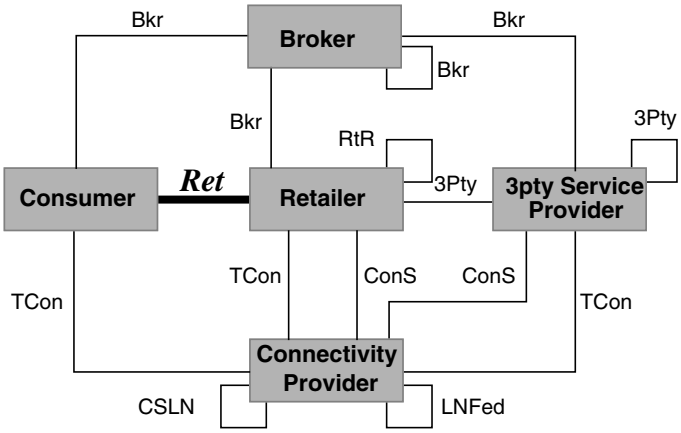



**Figure 1** TINA business model

- Layer network federation inter-domain reference point (LNFed)
- Client-server layer network inter-domain reference point (CSLN)

The current TINA RP interfaces are operational. That is, the interactions over interfaces occur in form of operation invocations. Via an operation, a client requests the execution of some functionality by the server object that provides the operational interface. Typically, an operation invocation returns the results (after successful termination or exceptions) to the client. “Oneway” operations are special case of operations that do not require a response to the client.

In general, the computational viewpoint of RPs are characterized by object interfaces using computational languages. TINA RPs are specified using the ODL (Object Definition Language) [15]. An ODL specification defines objects with their interfaces and object groups, which constitute e.g. an RP. ODL provides syntax for the structural description of systems only. A formalization of behavioral specification is not prescribed.

We consider in this paper the Ret-RP between consumer and retailer as an example. In terms of telecommunication services, the retailer serves as the service provider and the consumer as the service user. The Ret-RP offers generic access to telecommunication services, operations for the discovery and start of operational, management, and administrative service offerings, operations for the control and management of service sessions such as announcement, termination, suspension, invitation, notification for the service users participating in a service session.

Ret-RP is separated into an access part and a usage part. The access part contains interfaces that are required to establish a contractual relationship between consumer and retailer, which is referred

to as an access session. A service session can be built only upon an access session. The usage part of Ret-RP captures service session related interfaces. Ret-RP features are indicated either as mandatory or optional. This differentiation is significant for conformance testing.

## 3 The RP-Facet Concept

*RP-facets* define refinements of TINA reference points. An RP-facet is to enable interaction among components with separable concerns. It is a meaningful and self-standing portion of functionality. An RP-facet is a minimal set of conformance criteria, a TINA testing can be associated with. RP-facets are the basis for determining test purposes and generating test cases for TINA reference points.

Each reference point should be composed of one or more RP-facets. Typically, there will be a "core" facet that provides some minimum set of functionality. Additional interfaces and interactions can be specified to provide additional functionality. An RP-facet depends on the presence of the “core” facet and may depend on the presence of other RP-facets.

The RP-facet concept facilitates conformance testing, which is in particular based on the observation of system behavior. Thus, a purpose-oriented functional description in terms of use scenarios is proposed. The functionality of an RP-facet is specified by the signature and behavior of operations[1]. Operations provide services to object’s environment, whereas interfaces represent access points for services.

Typically, an inter-domain TINA reference point separates two business roles with distinguished functionality. An RP-facet is associated with one of the architectural parts separated by the reference point, referred to as *RP-facet role*.

The RP-facet role is to denote the functionality of interest in relation to the corresponding reference point. The dynamic aspect of operations is described by *use scenarios*. The purpose-oriented use scenarios describe potential interactions between the RP-facet role and it’s environment.

Before we define the notion of RP-facet, we need to define miscellaneous notions such as dependent operations and self-containment. A reference point is defined by a set of interfaces, each of which offers functionality to the outside via operations.

*Definition 1:* *An interface[2] I has a set of operations $SO_I$. $SO_I$ is divided into the set of mandatory and optional operations $SO_I^{mand}$ and $SO_I^{opt}$, resp., referring to the set of operations, which need or resp. can be offered by this interface. It holds that $SO_I^{mand} \cap SO_I^{opt} = \emptyset$ and $SO_I^{mand} \cup SO_I^{opt} = SO_I$.*

*Definition 2:* *A reference point R has a set of interfaces $SI_R$. $SI_R$ is divided into the set of mandatory and optional interfaces $SI_R^{mand}$ and $SI_R^{opt}$, resp., referring to the set of interfaces, which need or resp. can be offered by this reference point:*

1. Operational interfaces are considered currently. The results are directly applicable to event interfaces. Stream interfaces will be considered in a further work.
2. An interface denotes here an interface instance of an interface type, i.e. potentially there are a number of interfaces of the same interface type at a reference point.

- $I \in SI_R^{mand}$ *iff* $SO_I^{mand} \neq \emptyset$
- $I \in SI_R^{opt}$ *iff* $SO_I^{mand} = \emptyset$

*It holds that* $SI_R^{mand} \cap SI_R^{opt} = \emptyset$ *and* $SI_R^{mand} \cup SI_R^{opt} = SI_R$.

*Assumption 1:* *Subsequently we assume that the set of all operations $SO_R$ of reference point R, i.e* $SO_R = \bigcup_{I \in SI_R} SO_I$, *is not empty.*

*Definition 3:* *Let o be an operation at an interface I of reference point R, i.e. $o \in SO_I$. Let $o_1..o_n$ be further operations at R, i.e. $o_i \in SO_R$, i=1..n.*
*o is dependent on $o_1..o_n$ if the invocation of o requires previous invocations of $o_1..o_n$.*[1]
*o is independent if for all n there is no sequence of operations $o_1..o_n$, on which o is dependent.*

The dependent operations are either specified explicitly or derived from the use scenarios of the reference point.

*Definition 4:* *The dependence relation $dep_{I,R} \subseteq SO_I \times \wp(SO_R)$ of operations at interface I, where $\wp(SO_R)$ denotes the powerset of all operations of reference point R is defined such that*
$\forall o \in SO_I \; \forall so = \{o_1..o_n\} \in \wp(SO_R)$: $(o, so) \in dep_{I,R}$ *iff*

- *o is dependent on $o_1..o_n$ and*
- $\forall o_k \in SO_R$: *if o is dependent on $o_k$ then $o_k \in so$.*

*Lemma 1:*

- $\forall o \in SO_I : \exists!(o, \{o_1..o_n\}) \in dep_{I,R}$, *i.e. $(o, \{o_1..o_n\})$ in $dep_{I,R}$ is unique.*
- *o is an independent operation iff* $(o, \emptyset) \in dep_{I,R}$.

An RP-facet is self-contained in terms of functionality. Self-containment is defined with respect to the dependence relation. It is the core property of an RP-facet. Please note that the set of operations of an interface may be used only partially in an RP-facet:

*Definition 5:* *An RP-facet $F_R$ is a set of operations of a reference point with the following properties:*

- *it is a non-empty set and*
- $\forall I \in SI_R \; \forall o \in SO_I \; \forall (o, \{o_1..o_n\}) \in dep_{I,R}$ : *if* $o \in F_R$ *then* $o_i \in F_R$, i=1..n.
  *(the self-containment property)*

*The set of all RP-facets is denoted by $SF_R$.*

*Assumption 2:* *Subsequently, we assume that $SO_R$ is self-contained.*

Within the same reference point, dependent operations are captured by the same RP-facet:

*Lemma 2:*

- $\forall I,J \in SI_R \; \forall o1 \in SO_I \; \forall o2 \in SO_J$: *if* $o1 \in F_R$ *and o1 is dependent on o2, than* $o2 \in F_R$.

1. The dependence relation can be further refined to cover further aspects of dependencies. For example, if operation `o2` is only executable when operation `o1` returns `x`, then `o2` can be defined to be result-dependent on `o1`. Or, if interface `iB` is only reachable through an operation `o1` of interface `iA`, then `o2` can be defined to be reachable-dependent on `o1`.

- *For each RP, there exist a partitioning into RP-facets $F_i \in SF_R$, i=1..n, such that*
  *$SO_R = \cup_{i=1..n} F_i$ and $F_i \cap F_j = \emptyset$ for $i \neq j$.*
- *$SO_R$ is an RP-facet.*

RP-facets can be ordered. This order will be used to identify necessary steps in conformance testing:

*Definition 6:* *The order relation $\leq$ on RP-facets uses the subset relation:*
*$\forall F1, F2 \in SF_R$: $F1 \leq F2$ iff $F1 \subseteq F2$.*

*Lemma 3:*
- *$SO_R$ is the maximal element of $\leq$, i.e. $\forall F \in SF_R$: $F \leq SO_R$.*

The core is used to denote the mandatory, self-contained subset of a reference point. It is the set of all operations that need to be offered at an reference point in order to have it self-contained with respect to the dependence relations and complete with respect to the mandatory operations. If the core is empty then the complete reference point is an optional one.

*Definition 7:* *The core $C_R$ of a reference point R is the set of all mandatory operations of all mandatory interfaces of R with all their dependent operations, i.e.*

- *$\forall I \in SI_R^{mand}$ $\forall o \in SO_I^{mand}$: $o \in C_R$ and*
- *$\forall o \in C_R$, $\forall so \in SO_R^n$: $(o, \text{so}) \in dep_{I,R}$ : $so \subseteq C_R$.*

*Assumption 3:* *Subsequently, we assume that $C_R$ is non-empty.*

*Lemma 4:*
- *$C_R$ is unique.*
- *$C_R$ is an RP-facet.*

To support incremental specification, RP-facets are built cohesively, with the core as the origin. This leads to the definition of core-based RP-facets.

*Definition 8:* *A core-based RP-facet $F_{R,C}$ is an RP-facet that contains all operations of the core, i.e. $C_R \subseteq F_{R,C}$. The set of all core-based RP-facets is denoted by $SF_{R,C}$.*

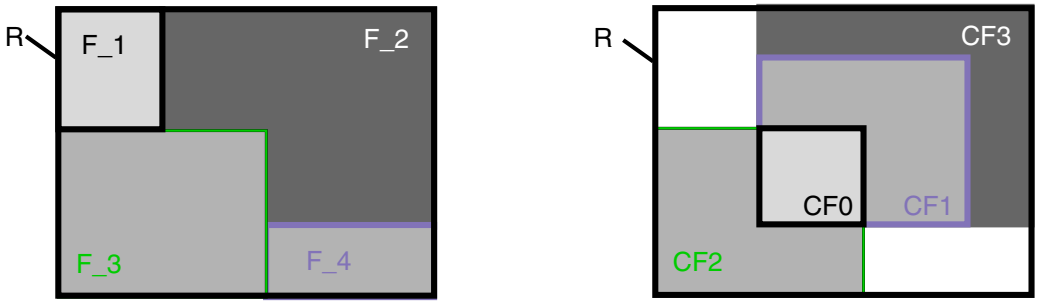
A core-based RP-facet covers at least the core and possibly additional optional operations.

*Lemma 5:*
- *$C_R$ is a core-based RP-facet.*
- *$SO_R$ is a core-based RP-facet.*

A reference point has a core and may have zero or more additional cohesive core-based RP-facets.

*Lemma 6:*
- *$C_R$ is the minimal element of the order relation $\leq$ on $SF_{R,C}$,*
  *i.e. $\forall F \in SF_{R,C}$: $C_R \leq F$.*
- *$SO_R$ is the maximal element of the order relation $\leq$ on $SF_{R,C}$,*
  *i.e. $\forall F \in SF_{R,C}$: $F \leq SO_R$.*
- *If $C_R = SO_R$ then is $SF_{R,C}$ a singleton.*

The relation of RP-facets and core-based RP-facets are depicted in Figure 2.




**Partitioning** of R into RP-Facets F_1 .. F_4 with $SO_R$= F_1 ∪ F_2 ∪ F_3 ∪ F_4 and F_1 ∩ F_2 =∅, etc.

**Hierarchies** of core-based RP-facets CF0 .. CF3 of R with CF0 ≤ CF1 ≤ CF3 ≤ $SO_R$ and CF0 ≤ CF2 ≤ $SO_R$.

**Figure 2** Reference Points and its RP-Factes

The conformance test method (see Section 5) will be based on the concept of core-based RP-facets and their hierarchies[1], as they naturally reflect the mandatory and optional requirements for a reference point and their relation.

## 4 RP-Facet Specification

Making the RP-facet concept practical is essential for real, industrial relevant systems. This is possible by providing a development method for RP-facets in combination with appropriate specification techniques. Even more, the unambiguous specification of an RP-facet including its static and dynamic models is crucial for testability. As any formalization reduces misinterpretation of the system under test, a formal specification supports in particular automated test generation and the possibility to validate tests for their soundness against the specification.

The reuse of specification parts of the reference point under test and therefore the reuse of specification techniques for distributed system is desired as it makes test development more efficient and allows a better integration of system development with test development.

Our approach for specifying RP-facets is based on the Object Definition Language (ODL) [7] for signatures of RP-facets in combination with Message Sequence Charts (MSC) [8][2]. Additions are needed to cover specific aspects of RP-facets according to the concepts introduced in the previous section. The specification template for RP-facets compresses:

- indication to the related reference point and the RP-facet role,
- statical specification of the RP-facet in ODL, and

1. Please note that for every core-based facet F there is at least the following hierarchy $C_R \leq F \leq SO_R$.
2. We concentrate currently on the on the functional aspect in the behavioral specification of reference points. Extensions to support description of operational aspects, e.g. QoS, usage, will be elaborated in future work.

- behavioral specification of the RP-facet in terms of use scenarios, including representations of dependence relations in MSC.

Further, we use the standard test notation TTCN (Tree and Tabular Combined Notation) to formulate test cases for RP-facets.

The RP-facet specification and test case generation cycle is presented in Figure 3. ASN.1 is the commonly used data representation form by MSC and TTCN. Thus, mappings for data types and constants from ODL to ASN.1 need to be defined.

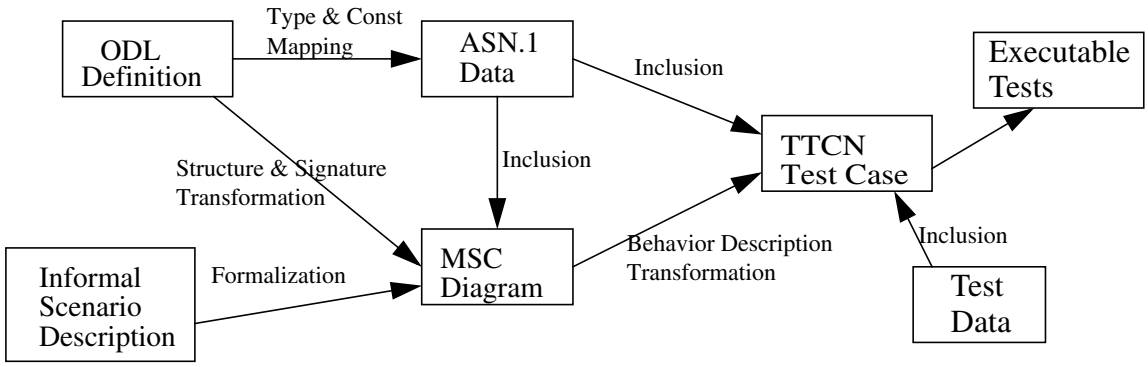



**Figure 3** RP-facet specification and test generation

### 4.1 Structural Specification Template

The template for the RP-facet structural specification is an extension of the TINA reference point specification template [13], which uses TINA-ODL [15]. ITU-T ODL [7] adopts most of the concepts and definitions of TINA-ODL. Thus, the following discussion on ODL refers to ITU-T ODL.

ODL is a superset of the OMG IDL (abbr. as IDL). In fact, most of the current TINA reference points are specified using IDL only. An example of the TINA Retailer Reference Point (Ret-RP) [14] specification is shown below.

```
#include "TINACommonTypes.idl"

module TINAProviderInitial {
 interface i_ProviderInitial {
   void requestNamedAccess (
     in TINACommonTypes::t_UserId userId,
     in TINACommonTypes::t_UserProperties userProperties,
     out Object namedAccessIR,
     out TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
     out TINAAccessCommonTypes::t_AccessSessionId asId
     ) raises ();
 };
};

module TINARetRetailerInitial {
 interface i_RetailerInitial: TINAProviderInitial::i_ProviderInitial {
 };
};
```

It specifies the interface *i_RetailerInitial* of the retailer domain, that inherits definitions of the interface *i_ProviderInitial*. Domains where interfaces reside, are indicated in the naming of modules and interfaces, e.g. *TINAProvider*, *TINARetRetailer*. *TINAProvider* is a generalized business role and can be specialized to a consumer, retailer, third-party service provider, broker and connectivity provider. In our example, *TINAProvider* is specialized to *TINARetRetailer*. The counter part of a provider is a user, which is at the Ret-RP a consumer.

Interactions at the Ret-RP between a retailer and a consumer involve not only interfaces provided by the retailer, but possibly also those interfaces supported by the consumer. This reflects the object model, in which an object supports interfaces, where services can be used by clients, and may require interfaces that are provided by other objects in its environment. Thus, there is a need for the identification of *supported* and *required* interfaces. The ODL's object template provides a notion for this purpose. Further, the template is also used to indicate the reference point and the role related to an RP-facet.

As shown in the following example, the module *TINARet* corresponds to the considered reference point. The facet role is represented by the object template identifier *Retailer* prefixed by the keyword *CO* (originated from Computational Object). Related interfaces are declared respectively behind the keywords *requires* and *supports*.

```
module TINARet {
 CO Retailer {
   requires
     Consumer::i_ConsumerInitial;
   supports
     i_RetailerInitial;
     i_RetailerAccess;
 };
};
```

Dependence relations of operations and interfaces (see Section ) allow reuse of ODL definitions by inclusion. A systematic document structuring eases system evolution.

### 4.2 ODL to ASN.1 Data Mapping

The ODL to ASN.1 mappings for data types and constants are in-line with the rules defined in [3][1]. Rules for basic type translation are shown in Table 1. Structure types are mapped according to Table 2.

| ODL Types | ASN.1 Type |
|---|---|
| long, unsigned long, long long, unsigned long long, short, unsigned short | INTEGER |
| char, wchar, string, wstring | GraphicString |
| octet | OCTET STRING(SIZE(1)) |
| boolean | BOOLEAN |
| void | NULL |
| float, double, long double | Real |

**Table 1** Mapping rules for basic types

1. This work is based on CORBA 2.2 specification. Mappings for IDL types included in the most recent and up-coming CORBA specifications, e.g. the *value* type, will be considered in future work.

| ODL Type | ASN.1 Type |
|---|---|
| struct | SEQUENCE |
| sequence | SEQUENCE OF |
| enum | ENUMERATED |
| array | SEQUENCE SIZE(n) OF |
| any | CHOICE |
| union | SEQUENCE |

**Table 2** Mapping rules for structured types

ODL *exception* declarations are *struct*-like. Hence, they are mapped to ASN.1 *SEQUENCE* types.

The mapping for the *Object* type is aligned to the OMG *interoperable object reference* (IOR) concept. An IOR is the global representation of the corresponding object and is composed of ASCII characters. For systems that are compliant with this concept, ASN.1 *IA5String* type is used.

### 4.3 Behavioral Specification Template

Message Sequence Charts (MSC) is a graphical and formal trace language defined by ITU-T [8]. MSC describes interactions between message-passing instances. MSC-2000 [8] is a new version of the standard that has been approved only recently. It has improved structural, data and time concepts. Method calls are introduced to support the description of control flows.

To use MSC for use scenarios of RP-facets, some structure and signature transformations are required.

*Rule 1* *MSC diagrams for an RP-facet are organized by an MSC document. The identifier of the MSC document is equivalent to the name of the RP-facet.*

An MSC document defines an instance kind for an RP-facet. It contains instances, messages, timer and MSC diagram declarations. In addition, a data language to be used in the MSCs can be declared.

*Rule 2* *The RP-facet role, the environment of the RP-facet role, every supported/required interfaces are mapped to separate instances.*

Instances for the RP-facet role and supported interfaces form the scope of the RP-facet, while other instances represent the scope of the environment. Interface instances play the role of service supplier. Instances of the RP-facet role and its environment are of the service consuming role.

*Rule 3* *The order relation between RP-facets, e.g. $F1 \leq F2$, is represented by inheriting the MSC document for F1 into the MSC document for F2.*

Inheriting a MSC document into another results in inheriting all declarations and MSCs from the inherited into the inheriting MSC document. This reflects the idea that for $F1 \leq F2$, *F2* covers *F1* completely as it is.

*Rule 4* *The dependence relation is represented by MSC expressions or high-level MSC (HMSC), where the MSC sequential operator is used to order the individual operation invocations as a sequence of simple MSCs reflecting separate operation invocations and the MSC*

*alternative operator for the subsequent behaviour in accordance to the potential outcomes of operation invocations.*

*Rule 5* *MSC specifications for RP-facets consists of two diagram types:*

- *High-level MSCs (HMSC) give an overview on the main structure and dependencies at the RP-facet. Here, references to further MSCs (usually simple MSC, see below), which are typically executed sequentially and combined with guards, are used. Enhanced use scenarios of RP-facets contain also parallel interactions at different interfaces. The operands of parallel expressions are represented by separate MSC instances.*
- *Simple MSCs contain a detailed definition of allowed message exchange and timer events between involved MSC instances. Further, they allow the usage of constructors for behavior control (e.g. alternatives, loops etc.) and guarded executions.*

MSC expressions, which can be graphically represented by HMSC, are the basic concept to represent the dependence relation between operations. If *o1* needs to be invoked before *o2*, it will be represented by *M1 seq M2* with *M1* reflecting the invocation of *o1* and *M2* the invocation of *o2*. In the case that several outcomes of *o1* and/or *o2* are possible within *M1* and/or *M2*, the alternative operator *alt* in combination with conditions will be used in addition. Please note that more complex behavior definitions for RP-facets will use also parallel, loop and optional expressions.

*Rule 6* *An ODL operation declaration is transformed to MSC message declarations. Mandatory is a message corresponding to a request on the operation. If the operation is not a "oneway" operation, a message in accordance with reply on the operation is also defined. If appropriate, each potential exceptional outcome of the operation is translated into a separate message.*

MSC asynchronous messages are used instead of method calls to make the representation of alternative operation invocation outcomes, in particular under exceptional conditions, more readable. In addition, the mapping to TTCN in the case of asynchronous messages is straightforward (see also Section 5).

The rule for attribute transformation is defined analogously:

*Rule 7* *An ODL attribute declaration is transformed to MSC message declarations. Mandatory is a message corresponding to the "get" operation on the attribute. If the attribute is not "readonly", a message in accordance with the "set" operation on the attribute is also defined.*

The data concepts of MSC-2000 allows the flexible use of a data language of the user's choice. No MSC specific data language is defined. MSC-2000 provides syntactical and semantical functions as interfaces to the use of external data languages within MSC. The definition of these functions for using ASN.1 in MSC at these interfaces is currently under work.

## 5 RP-Facet Based Testing

The RP-facet concept, in particular the self-containment property of an RP-facet, supports system evolution by incremental specification and implementation. In addition, RP-facets provide also testable specifications:

- The identification of the *RP-facet role* and its communication parties leads to the definition of the scope of the System Under Test (SUT) as well as the environment of the SUT, which will be emulated by components of the Test System (TS).
- The *structural specification* of the RP-facet to be tested, in form of ODL and ASN.1 definitions, can be shared by the TS.
- The formalization of *behavioral description* of RP-facet use scenarios in MSC supports an automated generation of tests.
- The *self-containment* property of RP-facets supports the identification of well-defined states of the SUT to achieve reproducible test results.
- The operation dependencies of an RP-facet define requirements on the sequence of test execution.

In order to support an efficient test development, we propose to use abstract test specifications. Our approach is based on the standard test notation TTCN [5].

### 5.1 Test Specification

TTCN (Tree and Tabular Combined Notation) was designed for conformance testing of OSI protocol implementations [5]. The test architecture is based on an asynchronous communication between SUT and TS. PCO (Point of Control and Observation) is an abstract location, where stimuli are sent to the SUT and reactions of the SUT are observed, either in form of Protocol Data Units (PDUs) or Abstract Service Primitives (ASPs). In decentralized test architectures, where typically several Parallel Test Components (PTCs) in addition to the Main Test Component (MTC) communicate with the SUT, more than one PCOs can be assigned to a PTC.

The analogy to the asynchronous message passing mechanism of MSC facilitates the transformation of MSC constructs to TTCN constructs. At first, it leads to the representation of MSC messages as TTCN abstract service primitives (ASPs)[1]:

*Rule A* *MSC messages representing ODL operations or attributes are translated into TTCN ASPs.*

According to Rule 6 and Rule 7, the ASPs are denoted by *request-ASP*, *reply-ASP* and *exception-ASP*.

Further, PCOs, test components and test configurations need to be identified for the test system. Due to the distinction of *supported* and *required* interfaces, two classes of PCOs can be derived from an MSC instance:

*Rule B* *A MSC instance representing a provided interface of the RP-facet role is interpreted by a client-PCO over which request-ASPs are sent to the SUT and reply-ASPs or exception-ASPs from the SUT are observed.*

*Rule C* *A MSC instance representing a required interface of the RP-facet role is interpreted by a server-PCO over which request-ASPs from the SUT are received and reply-ASPs or exception-ASPs to the SUT are sent.*

The assignment of one PCO to one PTC is not stringent, but recommended. The semantics of a

1. The selection of ASPs instead of protocol data units (PDUs) is based on the analogies between the object model and the OSI reference model. Please refer to [3] for details.

PTC is constrained by the class of PCOs it has. A PTC is in a client role when it communicates via a client-PCO with the SUT, and vice versa. Hence:

*Rule D* *Only PCOs of the same class, i.e. either client-PCOs or server-PCOs, can be assigned to a PTC. The assignment of more than one PCOs to a PTC is allowed, as long as the processing of test events, e.g. parallel sending of ASPs, is not restricted.*

The MSC inline expressions allow behavioral composition of event structures within a MSC. The operators refer to alternative (*alt*), parallel composition (*par*), iteration (*loop*), exception (*exc*) and optional (*opt*) parts. The *alt* operator, used in the example presented in the paper (Figure 5), defines alternative executions of MSC sections. In TTCN, the distinction between sequentialized and alternative behavior is identified by the indentation level of TTCN statements (subsequent TTCN events have a higher indentation as preceding events). Therefore:

*Rule E* *MSC inline expressions are expressed in TTCN by a combination of appropriate indentation levels, TTCN conditions and GOTO-statements.*

The TTCN timer concept addressing start, time-out and cancellation of timers is sufficient to cover MSC timer events.

The derivation of TTCN test descriptions from HMSCs is as follows:

*Rule F* *MSC references are mapped in TTCN to test step calls. MSC conditions are directly interpreted by TTCN qualifiers.*

TTCN test steps are a macro-like kind of subroutines. They are also used in case of RP-facet specifications representing extensions of previously specified smaller RP-facets. For example, it is typical that the test specification derived from a small RP-facet specification (e.g. from the minimal core-based RP-facet) will become the preamble (i.e. the very first test behavior at the beginning of a test description) of another "bigger" RP-facet test specification.

### 5.2 Test Campaign Derivation

In general, software testing is time and cost intensive, i.e. critical for large systems. Therefore, CTMF [4] gives advise for practical test purpose identification and for the grouping of test cases. We define a test suite structure according to core-based RP-facet hierarchies of the reference points under test. The sequence of test execution for the reference points under test is derived from the dependence relation between its operations.

The basic idea is to start with testing the core of a reference point and then to test incrementally by a repeating selection and testing of small extensions of the set of already tested operations. Each extension should comprise a complete core-based RP-facet.

At first, we define the ordered sequence of RP-facets to be tested:

- the minimal core-based RP-facets is tested first
- subsequently, other core-based RP-facets are tested according to their hierarchy.

Secondly, we define the sequence of testing operations within an RP-facet:

- Independent operations are those that can be tested without any preconditions (i.e. without preambles in the test case body).
- Dependent operations can be tested only of the operations they are depending on have been tested already successfully.

An algorithm for the test method is as follows. For simplicity, we assume that the system under test S realizes reference point R by means of core-based RP-facets $CF_1..CF_n$ with $C_R=CF_1 \leq ... \leq CF_n=SO_R$.

Let *T* be the set of already tested operations at R. *T* is divided into $T_P$ and $T_F$. $T_P$ refers to the set of operations that passed all tests. $T_F$ comprises those operations for which at least one test failed. Further, let *I* be the set of operations that are not testable as they depend on operations, which failed their tests or belong also to *I*. Let *N* be the core-based RP-facet under test in the current testing iteration.

**Start**: T= ∅, I= ∅, i=1, N= $CF_i$.

**Iteration i**:

*Step I:* *Select* $o \in N$ *with* $(o, \emptyset) \in dep_{I,R}$ *:*

/* independent operations */

*Execute the tests for o .*

*If o passes all tests, then* $T_P = T_P \cup \{o\}$ *else* $T_F = T_F \cup \{o\}$.

*In any case,* $N=N\backslash\{o\}$

*Repeat until no further operations o with* $(o, \emptyset) \in dep_{I,R}$ *exist .*

*Proceed with Step II.*

*Step II:* *Select* $o \in N$ *with* $(o, \{o_1..o_m\}) \in dep_{I,R}$ *and* $\forall j,j=1..m: o_j \in T$

/* dependent operations whose preconditional operations have been tested successfully*/

*If* $\exists j=1..m: o_j \in T_F$ *, then* $I = I \cup \{o\}$.

*Else, execute the tests for o .*

*If o passes all tests, then* $T_P = T_P \cup \{o\}$ *else* $T_F = T_F \cup \{o\}$.

*In any case,* $N=N\backslash\{o\}$ .

*Repeat until no further operations o with* $((o, \{o_1..o_m\}) \in dep_{I,R}$ *and* $\forall j,j=1..m: o_j \in T)$ *exist .*

*Proceed with Step III.*

*Step III:* *Select* $o \in N$ *with* $(o, \{o_1..o_m\}) \in dep_{I,R}$ *and* $\exists j=1..m: o_j \in I$

/*dependent operations for which not all preconditional operations are tested successfully*/

*Then* $I = I \cup \{o\}$ *and* $N=N\backslash\{o\}$ .

*Repeat until no further operations o with* $((o, \{o_1..o_m\}) \in dep_{I,R}$ *and* $\exists j=1..m: o_j \in I)$ *exist .*

*Proceed with Step IV.*

*Step IV:* *Select* $o \in N$ *with* $(o, \{o_1..o_m\}) \in dep_{I,R}$ *and* $\exists j=1..m: o_j \in N$

/*dependent operations with cyclic dependencies*/

*Execute the tests for* $o_j$.

*If* $o_j$ *passes all tests, then* $T_P = T_P \cup \{o_j\}$ *else* $T_F = T_F \cup \{o_j\}$.

*In any case,* $N=N\backslash\{o_j\}$ .

*Proceed with Step III.*

*Repeat until no further operations o with* $(o, \{o_1..o_m\}) \in dep_{I,R}$ *and* $\exists j=1..m: o_j \in N$ *exist . Proceed with Step V.*

*Step V: If N empty and not yet termination, take* $i=i+1$, $N=CF_i \setminus (T \cup I)$ *and proceed with Step II.*

**Termination**: *If* $T \cup I = SO_R$ *terminate.*

Interface operation tests will comprise static operation header tests as well as dynamic testing of operations semantics. First, the static header tests result from combinations of valid/invalid parameters and test values according to the interface signature and constraints [10]. The other test groups, which focus on testing of valid/invalid sequences of operations at RP-facets, can be derived using traditional test derivation algorithms well known from e.g. LTS or EFSM based test generation methods implemented in several academic and commercial test derivation tools.

## 6 An Example

This section presents an example on how the proposed concepts and specification techniques are applied to the TINA Retailer reference point (Ret-RP). The retailer is the focus of the consideration.

The following ODL definition is a simplified representation of [14] (see also Section 4.1). It indicates the reference point *Ret* and the RP-facet role *Retailer*. It defines further: *Retailer* provides a *i_Initial* interface and a *i_Access* interface; *Retailer* uses the *i_Initial* interface of the *Consumer*; the operation *namedAccess* of the *Retailer*'s *i_Initial* interface requires an input parameter for passing some user information, and provides an output parameter for returning a reference of requested object, and in case that the passed user information is invalid an *PropertyError* exception is raised. The consumer domain services are defined in a separate document *Ret_Consumer.odl*.

```
#include "Ret_Consumer.odl"
module Ret {
 CO Retailer {
   requires
     Consumer::i_Initial;
   supports
     Retailer::i_Initial;
     Retailer::i_Access;

   interface i_Initial {
     void namedAccess (
       in UserProperty userInfo,
       out Object i_na;
     ) raises (PropertyError);
     ...};
   interface i_Access {...};
 };};
```

From the textual description of Ret-RP business scenarios, two RP-facets can be derived:

- The core facet *Ret_Retailer_core* involves login and logout of a consumer at the retailer domain. The retailer's interface *i_Initial* and the operation *namedAccess* are used by login.
- An additional facet *Ret_Retailer_add1* is based on the core facet. It is to start a service after a successful login, and to terminate the service before the logout.

*Ret_Retailer_core* is organized by the MSC document and High-level MSC (HMSC) presented in Figure 4. According to Rule 2, five instances (prefixed by **inst**) are defined: *Retailer*, *Retailer_i_Initial*, *Retailer_i_Access*, *Consumer* and *Consumer_i_Initial*. The operation *namedAccess* is mapped to three messages (indicated by **msg**), respectively for the request, reply and exception related to the operation (see Rule 6). The inclusion of data types and constants translated to ASN.1 is enabled by the **language** and **data** constructs. The HMSC *Ret_Retailer_core_msc* uses two utility MSCs *Login* and *Logout*, and conditions *idle*, *LoginFailed* and *LoginSuccessful.* It describes the dependency of the logout activity on a successful login.

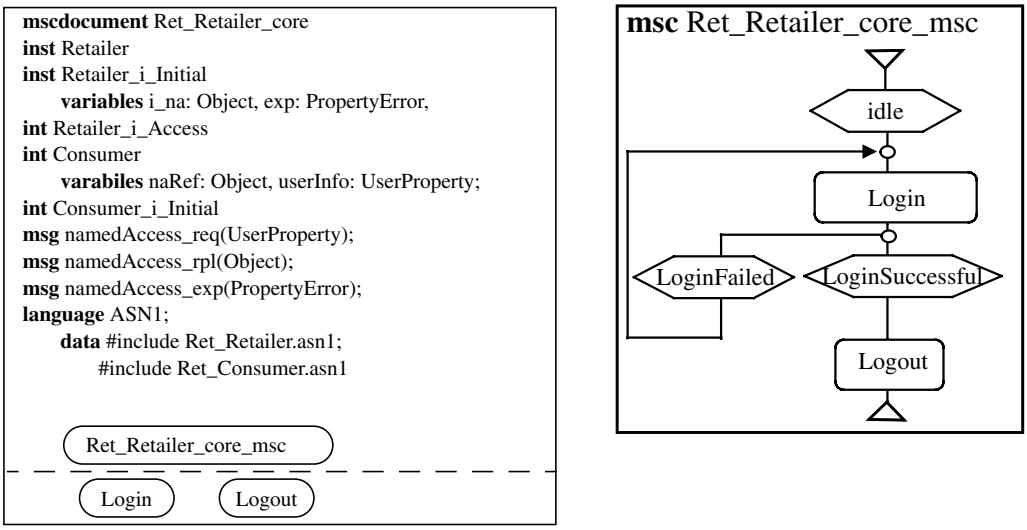



**Figure 4** MSC example of Ret_Retailer_core

Details of purpose-oriented use scenarios of RP-facets are described by MSC event traces. Figure 5 shows the message exchanges between a *Consumer* instance and a *Retailer_i_Initial* instance in relation to the login activity. The two alternative outcomes of a request on the operation *namedAccess* are represented by use of the MSC inline expression *alt*. Data used in message parameters and/or conditions are defined in the data part of the MSC document.

The cohesive relation of *Ret_Retailer_add1* to *Ret_Retailer_core* is in particular reflected by the reuse of declarations and utility MSCs, as presented in Figure 6. To support start service related operation, declarations of the messages *startService_req*, *startService_rpl* and *startService_exp* are added to the core facet MSC document. Furthermore, two new utility MSCs are introduced: *StartService* and *EndService*. *Ret_Retailer_add1_msc* extends the core facet's HMSC by a description of the logical relation between *StartService* and *EndService* MSCs.

Table 3 shows the dynamic part of the TTCN specification of a test case in accordance with the Login MSC (Figure 5). The tabular form is simplified to ease the understanding.

In this example, the SUT is an implementation of the retailer domain core RP-facet. This test case is to evaluate the login activity at the retailer's *i_Initial* interface. The TS emulates the behavior of a client of *i_Initial*. It uses a client-PCO named *PCO1_Retailer_i_Initial* defined using Rule B. The purpose of this test case is to verify: after a request on the operation *namedAccess* with valid

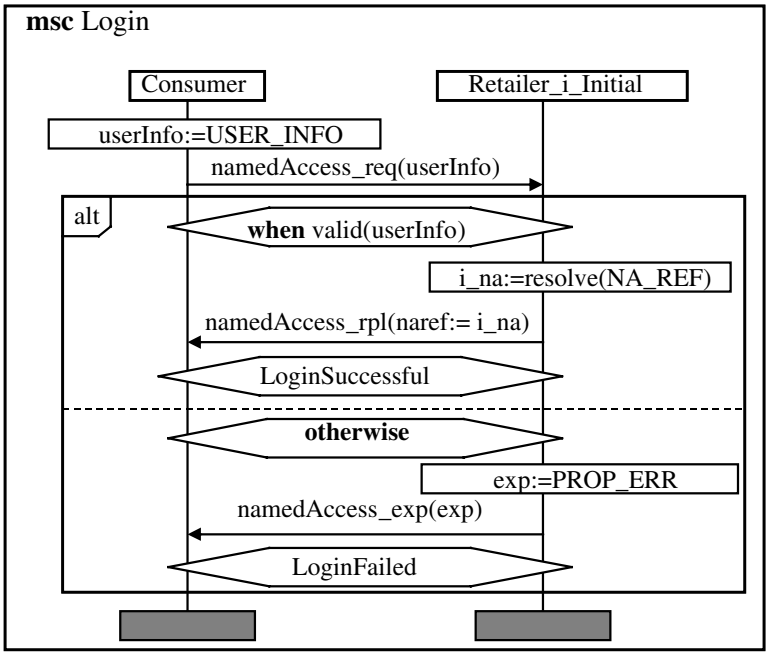



**Figure 5** Login MSC diagram

user information is sent to the SUT, a reply of *namedAccess* is received by TS (see line 2 and 4). To indicate the ASP kind, the request-ASP is prefixed by *pCALL*, and the reply-ASP is prefixed by *pREPLY.*

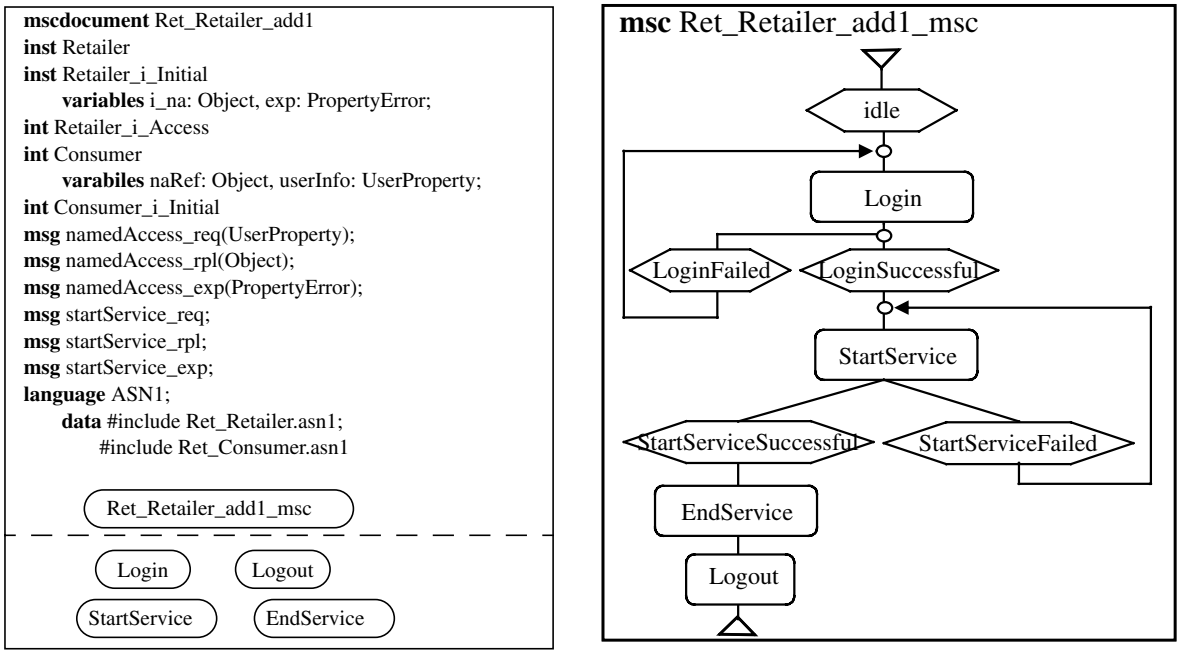



**Figure 6** MSC example of Ret_Retailer_add1

The test step *GetInitialRef* (line 1) used as preamble is mainly intended to allow the resolution of object references that will be used in the test case. It is not derived directly from the MSC. It is a general purpose test step. In addition, a timer *Timer1* is used to ensure that a test event (including time-out events) will occur after a given time even in case that the SUT does not answer.

The postamble *Logout* (line 5) after the receive event recalls the MSC reference *Logout* in the HMSC of the core RP-facet.

| **Test Case Dynamic Behavior** | | | | |
|---|---|---|---|---|
| **No** | **Label** | **Behavior Description** | **Constraints Ref** | **Verdict** |
| 1 | | +GetInitialRef | | |
| 2 | | PCO1_Retailer_i_Initial ! **pCALL**_Retailer_i_Initial__**namedAccess** | pCALL_namedAccess_s1 | |
| 3 | | START Timer1 | | |
| 4 | | PCO1_Retailer_i_Initial ? **pREPLY**_Retailer_i_Initial__**namedAccess**<br><br>CANCEL Timer1 | pCALL_namedAccess_r1 | (P) |
| 5 | | +Logout | | |
| 6 | | ?TIMEOUT Timer1 | | I |

**Table 3** TTCN test case example

The implementation and execution of TTCN-based test case in the CORBA environment is discussed in [3].

## 7 Conclusions

The goal of the work presented in this paper is to provide concepts and means for testable specifications that facilitate conformance and interoperability testing for distributed systems. The approach is based on the RM-ODP and TINA reference point concept. It refines reference points into self-contained and extensible facets, referred to as RP-facets, in order to allow incremental specification, implementation and testing of distributed systems at their reference points.

Formalization is key to the concept. Besides a mathematical characterization of RP-facets, a template for structural and behavioral specifications of RP-facets as well as a conformance test method are elaborated. The specification template consists of ODL, ASN.1, and MSC to provide adequate information detail for the derivation of abstract test cases in TTCN. The combination of all these specification techniques is shown by an example.

In addition to a thorough usability study of the approach, future work will be on the support of further testing aspects, e.g. operational conformance under load situation, what requires extensions of the RP-facet specification template. Additional issues of test campaigns, which are partly discussed in the paper, such as efficient test strategy, will be also considered.

## 8 References


[1] R. V. Binder: Testing Object-Oriented Systems, Models, Patterns and Tools, Addison-Wesley, 1999.

[2] S. Ghosh, A.P. Mathur: Issues in Testing Distributed Component-Based Systems.- In Proc. of the First Intern. ICSE Workshop on Testing Distributed Component-Based Systems, Los Angeles, U.S.A, May 1999.

[3] M. Li, I. Schieferdecker, A. Rennoch: Testing the TINA Retailer Reference Point, Proceedings of ISADS'99, Tokyo, Japan, March 1999.

[4] ISO/IEC 9646-2: Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework - Part 2: Abstract test suite specification, 1991.

[5] ISO/IEC 9646-3: Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework - Part 3: The Tree and Tabular Combined Notation (TTCN), edition 2, Dec. 1997.

[6] ITU-T Rec. X.901 | ISO/IEC 10746-1: 1995, Open Distributed Processing - Reference Model Part 1, Geneva, Swiss.

[7] ITU-T Z.130: Object Definition Language (ITU-ODL), March 1999.

[8] ITU-T Z.120: Message Sequence Charts (MSC'2000), Nov. 1999.

[9] OMG: Common Object Request Broker Architecture (CORBA), version 2.3, 1999.

[10]A. Rennoch, J. de Meer, I. Schieferdecker: Test Data Filtering, 9. GI/ITG-Fachgespräch "Formale Beschreibungstechniken für verteilte Systeme", München (D), June 1999.

[11]Steedman, D.: Abstract Syntax Notation One (ASN.1), Technology Appraisals Ltd., 1990.

[12]TINA-C: Overall Concepts and Principles of TINA, version: 1.0, Feb. 1995.

[13]TINA-C: TINA Reference Points, version 3.1, Jun. 1996.

[14]TINA-C: Ret Retailer Reference Point Specification, version 1.1, 1999.

[15]TINA-C: Object Definition Language (TINA-ODL), version 2.3, Jul. 1997.

[16]TINA-C: TINA-CAT WorkGroup Request for Proposals, TINA Conformance Testing Framework, version 1.0, Jul. 1999.

# Towards a Mechanised Software Development Method

Bing Wu[1], Luming Lai[2], and D.R.W. Holton[3]

[1] B.Wu@scm.brad.ac.uk,
Home page: http://www.personal.comp.brad.ac.uk/~bwu/
[2] L.M.Lai@scm.brad.ac.uk,
Home page: http://www.personal.comp.brad.ac.uk/~lmlai
[3] D.R.W.Holton@scm.brad.ac.uk,
Home page: http://www.personal.comp.brad.ac.uk/~drwholton
University of Bradford, Bradford, BD7 1DP, West Yorkshire, UK

**Abstract.** Formal methods (FM) consist of a set of techniques and tools that are based on mathematical modeling and formal logic and which are employed to specify and verify requirements and designs for computer systems - both hardware and software. Moreover, the growing criticality and complexity of software has led to increased interest in applying FM to software specification and design as well.
The paper will develop a more practical software development method by integrating the Refinement Calculus with Z, both developed at Oxford University, and develop a software development environment in which software can be formally specified and refined into program code with all the refinement steps proved correct by the tool. We present a case study of using a refinement tool prototype, ZRefiner to refine a Z speification to code. The three-tier structure of ZRfiner deals with different applications in different tiers and makes ZRefiner more flexible and efficient. Finally, potential theoretical and practical problems of implementing such a mechanised tool are discussed.

**Keywords:** Formal methods, Z, Refinement Calculus, Mechanised software development

## 1 Introduction

Many of the most serious problems in software design and implementation result from imprecise, ambiguous, incomplete, misunderstood, or just plain incorrect statements in requirements specifications. Describing large, complex systems with thousands or even millions of properties is extremely difficult using natural language.

Formal methods (FM) consist of a set of techniques and tools that are based on mathematical modeling and formal logic and which are employed to specify and verify requirements and designs for computer systems - both hardware and software. Early successes in the use of FM were more frequently obtained in the

design of computer hardware, and this domain remains an important application area for FM. However, the growing criticality and complexity of software has led to increased interest in applying FM to software specification and design as well.

Z [8, 16] is a well established specification language that has a distinguishing mechanism of modularization: the schema calculus. It allows us to formalise individual requirements separately and join them together by schema operators. Its success is evident: many case studies [8] have already been developed, some of which involve industrial applications [6]. A wide range of tools [24] that support several aspects of its use in specification have been implemented. However, none of them support refinement using Z. The main objective of our research is to develop a fully integrated tool for Z which supports refinement and good graphical user interfaces.

Although Z is a well established specification language and the refinement calculus was developed many years ago by Back, Morgan and Morris, [1, 11, 10], the integration of both is not as simple as originally thought. There are some fundamental problems to be solved for a smooth integration. For example, the refinement calculus is based on the weakest precondition semantics but Z on set theory; the specification statements in the refinement calculus use pre- and postconditions but Z uses the schema calculus which puts abstraction above everything else.

Several proposals for solving these problems can be found in the literature. The first proposal for integrating Z with the Refinement Calculus is presented in [9], where the differences between Z and the notation of the Refinement Calculus are analyzed and, in the light of these considerations, rules that translate schemas and some schema expressions to programs are suggested. [19] has proposed a different form of integration, where schemas themselves have been regarded as commands of the language of the refinement calculus. Refinement of schemas is accomplished either by laws that are similar to the translation rules in [9] that apply to schema expressions, or by verification instead of calculation, when none of these laws apply. There is no equivalent to the rule of [9] that translates any schema.

Another approach is suggested in [18], where generalizations of the Z conjunction and disjunction schema operators are introduced into the language of the refinement calculus so that specification statements can be combined and the Z incremental style of building specifications can be used. The aim is to achieve a refinement calculus that can cope with large specifications. However, the other schema operators, which also contribute to the success of the Z style, are not considered and it is not clear how they can be added to the refinement calculus. Our approach is similar to Ward's. But we propose to use Z throughout the whole development process, which are more readable than the specificaiton statements.

[15] defined a notation for documenting the development of ADA programs from Z specifications. Despite the fact that a language similar to that of the refinement calculus was used, the proposal consisted of designing the programs directly in ADA and then giving an account of their correctness by using the no-

tation and literate programming was suggested as part of Cleanroom, a method of software development that recommended the use of formal specifications and refinement, but that did not indicate any particular technique.

In [3–5], ZRC - a refinement calculus for Z based on Morgan's calculus [10] was developed. The method allows specification using Z schemas and provides some conversion laws to convert Z schemas into the specification statements in the refinement calculus. Then the refinement is done in the refinement calculus. A weakest precondition semantics for Z was developed so that the soundness and completeness of the conversion laws are proved. The drawback of such an approach is that the abstraction of Z is lost as soon as the refinement starts.

The motivations of producing efficient programs that has a mathematically sound basis and allows the use of calculational techniques are certainly best served by the approaches advocated in [9, 20, 19, 22]. Also motivated by Cavalcanti's work, we develop a prototype refinement tool: ZRefiner, with the support for spcification and refinement of Z. We introduce the weakest precondition semantics for the modified Z, which is used to replace the specification statements during refinement.

This paper is organised as follows. Section 2 investigates Z and the refinement calculus. Section 3 introduces the modified Z specification. Section 4 presents a weakest precondition semantics for the modified Z. Section 5 introduces a integrated refinement calculus for the modified Z, and presents some refinement laws we use in the paper. Section 6 presents the prototype structure of ZRefiner, which is based on the three-tier architecture. Section 7 introduces the birthday book case study using ZRefiner. Section 8 discusses the advantages and distantages of ZRefiner and the refinement calculus for the modified Z, and also presents our future work.

## 2 Z and The Refinement Calculus Overview

### 2.1 Z Overview

Z is a specification language which uses schemas and schema operators to construct complex specifications from component specifications. Z specifications have good hierarchical structures. There are two basic kinds of schemas: one specifies the states of a system and the other specifies the operations which can be performed on these states. In the following example, we describe this by specifying a class manager. The same example can be found in [9].

**Example 2.1** We can use the following state schema,

*Class*

$$yes, no : \mathbb{P}\, Student$$

---

$$yes \cap no = \{\}$$
$$\#(yes \cup no) \leq max$$

to describe the state of a system to record which students in a class have (*yes*) or have not (*no*) completed a set of exercises.

If we want to enlarge the state of this system, for example adding students information such as their registration numbers into the system, we can add another state schema by the schema calculus.

*RegistrationNumbers*

$$regnumber : Student \rightarrow Registration$$

---

$$\forall x, y : Student \mid x \neq y \bullet regnumber(x) \neq regnumber(y)$$

Then the overall state of the enlarged system is

$$Class \wedge RegistrationNumbers.$$

An operation to enrol a new student into the class could be described by

*EnrolOk*

$$\Delta Class$$
$$s? : Student$$

---

$$s? \notin yes \cup no$$
$$\#(yes \cup no) < max$$
$$yes' = yes$$
$$no' = no \cup \{s?\}$$

The new student will not have done the excises, so he will be put into set *no*. Now the declaration introduces the state before and after the operation(in $\Delta Class$) and the input variable $s?$. The predicate shows the relation between the variables of the state before(*yes* and *no*) and the state after($yes'$ and $no'$) and $s?$.
□

The Z schemas also have an "open-world" view about variables which do not occur in its signatures [18]. It places no restrictions on those variables. For example, in the above schema *Class* does not mention variable *regnumber* but schema *RegistrationNumber* has *regnumber* in its signature. So $Class \wedge RegistrationNumber$ can change *regnumber* according to *RegistrationNumber*.

### 2.2 The Refinement Calculus

*Refinement calculus*, developed independently by Back, Morgan and Morris, [1, 11, 10], provides a uniform method for deriving programs from specifications. The calculus extends a programming language with an abstract specification construct. The calculus defines formally an ordering between specifications that allows one specification to be substituted for another. The semantics of both the programming language and the specification construct are defined by Dijkstra's weakest precondition semantics [7].

The Refinement Calculus introduces a specification statement into the guarded command language. Its novelty is the banishment of the differences between specifications and program code. Therefore, software development can start with an abstract specification statement and then refine it gradually into program code through mixed terms which may contain both specification statements and program code.

For example, the notation $w := E$ is an *assignment command*, and is also the program code. It changes the state so that the variable $w$ is mapped to the value $E$, and all other variables are unchanged. Assignments form the basis of imperative programming language since they are easy to execute. Below we give a law of refinement for assignment.

**Law 22 (assignment)** *If* $pre \Rightarrow post[w \backslash E]$, *then*

$$w, x : [pre \mid post] \sqsubseteq w := E.$$

To demonstrate a refinement example, we also give the following laws of refinement:

**Law 23 (strengthen postcondition)** *If* $post' \Rrightarrow post$, *then*

$$w : [pre, post] \sqsubseteq w : [pre, post'].$$

**Law 24 (weaken precondition)** *If* $pre \Rightarrow pre'$, *then*

$$w : [pre, post] \sqsubseteq w : [pre', post].$$

The following example shows a refinement using the refinement calculus.

**Example 2.5** The following specification sets $x$ to either 1 or 2 when it is non-negative:

$$\begin{array}{ll} & x : [x \geq 0, x = 1 \vee x = 2] \\ \sqsubseteq & \{Law\ 24\ weaken\ precondition\} \\ & x : [true, x = 1 \vee x = 2] \\ \sqsubseteq & \{Law\ 23\ strengthen\ postcondition\} \\ & x : [true, x = 1] \\ \sqsubseteq & \{Law\ 22\ assignment\} \\ & x := 1 \end{array}$$

□

The informal meaning of a specification statement $w : [pre \mid post]$ is as follows:

> If the initial state satisfies the precondition *pre*, then change *only* the variables listed in the frame $w$ so that the resulting final state satisfies the postcondition *post*.

According to this explanation, the specification statement has a "closed world" view about the variables which do not occur in the frame $w$. That is, those variables cannot be changed by the specification statement. This does not cause any problem for the Refinement Calculus because it does not allow the construction of specifications from individual ones using the schema operators. The specification statement always specifies operations on the overall state. This simply wipes out all the advantages of the schema calculus, thus Z.

### 2.3 Conclusion

There are two ways to integrate Z with the Refinement Calculus. One is suggested in [18] where two of the schema operators, conjunction and disjunction, are introduced into the Refinement Calculus such that the composition of specification statements are possible. However, these two operators turn out to be non-monotonic with respect to the refinement relation.

Another ways for the integration is to introduce the Refinement Calculus into Z. Once again, we face the problem of non-monotonicity of some of the schema operators, including the conjunction and disjunction operators.

The goal of a development in a refinement calculus is to start with a specification at a high level of abstraction that captures only the essential properties of the system and transform this to program code. A key factor in a refinement calculus's ability to achieve this goal is the use of a single wide-spectrum language that covers both specification and program code [2, 13]. A well designed wide-spectrum language allows specifications which are concise and which give maximum freedom to the developer when choosing an implementation. Development in a refinement calculus proceeds via transformations which replace (possibly non-executable) specification constructs with (executable) programming language constructs. Thus, at any stage of the development the object being reasoned about is usually a mix of specification and programming constructs written in the wide-spectrum language.

Based on this reason, we propose a modified Z, which separates pre- from postcondition, as the language of the refinement calculus. Then we define the schema calculus for the modified Z which makes the conjunction operator monotonic with respect to the refinement relation.

## 3 The Modified Z Specification Language

Z, as a "pure" specification language, tries to distance itself as far away from implementation as possible. It uses abstract data types, allows the introduction of new signatures at anywhere, and it does not care about the pre- and postconditions of operations. If we compare Z with VDM, we can see one important difference between them. That is, VDM is a refinement development method, whereas Z is just a specification language. The result of this is that the specification formulae in VDM uses keywords like a programming language and is

syntactically more complicated than Z schemas. Of course, with refinement development in mind, VDM also separates precondition from postcondition. If we want to use Z during the development of software, Z needs to be modified.

The closed-world view means that any variable in the signature of a specification which is not also in its frame can not be changed. As a result, the combination of two specification can not in general produce a new specification which can change all of the variables in both frames.

To solve this problem, we remove the frame of the specification statement, and introduce the Z declaration part into the specification statement. The final specification is similar to Z schema except that the predicates of the Z schema are divided into two parts: the *precondition part* and the *postcondition part*. We call this schema as *modified Z schema*. The *declaration part* introduces the state variables of a schema. Thus, the variables in the signature of a specification are also in its declaration part. As the same as the specification statement in [17], the postcondition of an operation in a modified Z specification can be used to specify those variables that do not change.

*Schema*

| |
|---|
| *Declaration Part* |
| *Precondition Part* |
| *Postcondition Part* |

$w : [pre, post]$

We can also use an abbreviation:

$Schema \mathrel{\widehat{=}} [decl \mid pre \mid post]$.

We refer to the pre- and postconditions as *pre Schema* and *post Schema*, and the declaration as *decl Schema*.

**Example 3.1** The operation schema *EnrolOk* can then be specified by

*EnrolOk*

| |
|---|
| $\Delta Class$ |
| $s? : Student$ |
| $s? \notin yes \cup no$ |
| $\#(yes \cup no) < max$ |
| $yes' = yes$ |
| $no' = no \cup \{s?\}$ |

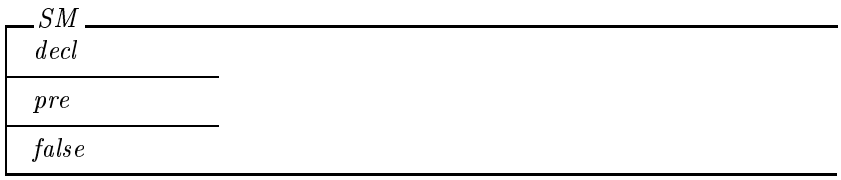
□

The reasons for doing this are as follows. First, the Z schema operators are not monotonic with respect to the refinement relation. This means that we cannot retain the schema operators during the refinement development and will lose

the hierarchical structures of Z schema expressions at the very beginning of the refinement process. Another reason for separating precondition from postcondition is that it is not easy to see the refinement steps in schema which rely on separate pre- and postconditions. For example, it is hard to see the refinement of a schema description of an operation directly into an iteration because we can hardly see the loop invariant and the bound function in the schema description.

The advantages of doing this is that it makes the modified Z more expressive. For example, we can have miraculous specifications, which can be useful during refinement.

**Example 3.2** We define the miraculous specifications:




The schema is called **miracle** because it implements anything. Miracles arise "accidentally" in program development when we make an incorrect design step. It is discussed in more detail in [12].
□

While composing a modified specification with the other modified specification, we introduce two operations into the modified Z notations: the schema conjunction and the schema disjunction, in order to keep the feature of the open-world view. We also extend schema operations to be monotonic with respect to the refinement relation except schema disjunction. Since the cases joined by schema disjunctions are mutually exclusive, we prove that schema disjunctions can also be defined by alternations. By keeping the mutual conditions of the schema disjunction, we can refine the disjoined schemas individually. Thus, we can leave the refinement of schema conjunctions and disjunctions to the later stage and keep the Z incremental style of developing specifications. The above details are presented in [21].

## 4 The Weakest Precondition Semantics

### 4.1 The Weakest Preconditions

*Weakest preconditions* were first introduced in [7], where they are used to define the semantics of Dijkstra's language of guarded commands:

> The condition that characterizes the set of all initial states such that activation will certainly result in a properly terminating happening leaving the system in a final state satisfying a given postcondition is called "the weakest precondition corresponding to that postcondition."

If the system is denoted by $S$ and the desired postcondition by $\psi$, then we denote the corresponding weakest precondition by

$$wp.S.\psi \ .$$

If the initial state satisfies $wp.S.\psi$, it is guaranteed to establish eventually the truth of $\psi$. Because $wp.S.\psi$ is the weakest precondition, we also know that if the initial state does not satisfy $wp.S.\psi$, this guarantee cannot be given, since the system may end in a final state not satisfying $\psi$ or it may even fail to reach a final state at all.

The meaning of $wp.S.\psi$, the weakest precondition for the initial state such that activation will certainly result in a properly terminating happening, leaving the system $S$ in a final state satisfying the postcondition $\psi$, allows us to give the *wp* semantics for specification statements [10] and Z specifications [4]. This paper also extends the *wp* semantics to the modified Z specifications.

**Definition 41 (Weakest precondition)** *The weakest precondition of a modified Z schema* $[d \mid pre \mid post]$ *can be defined by*

$$wp.[d \mid pre \mid post].\psi \mathrel{\widehat{=}} pre \land (\forall\, ds', do! \bullet post \Rightarrow \psi)$$

*where* $d$ *declares the set of all the schema variables:* $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$. $ds$ *declares the set of state variables,* $ds'$*, the corresponding dashed variables,* $di?$*, the input variables,* $do!$*, the output variables.*
□

In the definition above, termination is captured by *pre*, and correctness is captured by $(\forall\, ds', do! \bullet post \Rightarrow \psi)$.

### 4.2 The Guarded Commands

A *Guarded Command*, is a statement 'pre-fixed' by a boolean expression termed a guard. A *guard* is a formula which selects those states to which its associated command applies. A *command* is the associated program to be executed. The guarded command itself is written

$$G \to P,$$

where $G$ is a guard and $P$ a program.

To execute a guarded command we first evaluate the guard, then execute the statement if the guard is true, otherwise do nothing. The following example shows the execution of a guarded command.

**Example 4.2**

$$x \geq 0 \to x := x - 1$$

The command $x := x - 1$ can be executed only if its guard $x \geq 0$ is true.
□

Semantics for ordinary guarded commands are introduced in [7, 10]. Here we give the weakest predicate transformers for all the basic guarded commands. The semantics includes assignment, sequential composition, alternation and iteration. The semantics is defined in terms of the weakest precondition with respect to a postcondition $\psi$.

$$
\begin{aligned}
wp.\ \mathbf{skip}\ .\psi &\mathrel{\widehat{=}} \psi \\
wp.\ \mathbf{abort}\ .\psi &\mathrel{\widehat{=}} \mathbf{false} \\
wp.(x := E).\psi &\mathrel{\widehat{=}} \psi[x \backslash E] \\
wp.(P;\ Q).\psi &\mathrel{\widehat{=}} wp.P.(wp.Q.\psi) \\
wp.\{pre\}.\psi &\mathrel{\widehat{=}} pre \wedge \psi \\
wp.[post].\psi &\mathrel{\widehat{=}} post[' \backslash] \Rightarrow \psi \\
wp.\ |\,[\ \mathbf{var}\ dvl \bullet P]\,|\ .\psi &\mathrel{\widehat{=}} \forall\, dx' \bullet wp.P[vl, vl' \backslash x, x'].\psi \\
wp.\ |\,[\ \mathbf{con}\ dcl \bullet P]\,|\ .\psi &\mathrel{\widehat{=}} \exists\, dx \bullet wp.P[cl \backslash x].\psi \\
wp.(\mathbf{if}\ \ \Box i \bullet G_i \rightarrow P_i\ \mathbf{fi}).\psi &\mathrel{\widehat{=}} (\vee i \bullet G_i) \wedge (\wedge i \bullet G_i \Rightarrow wp.P_i.\psi) \\
wp.(\mathbf{do}\ \ \Box i \bullet G_i \rightarrow P_i\ \mathbf{od}).\psi &\mathrel{\widehat{=}} \exists\, k \geq 0 \bullet H_k(\psi)
\end{aligned}
$$

**Fig.1.** Predicate transformers for guarded commands

In the semantics for the iteration, the conditions $H_k(\psi)$ is given by

$$
H_k(\psi) \mathrel{\widehat{=}} \begin{cases} \psi \wedge \neg\ (\vee\ i \bullet G_i) & k = 0, \\ wp.(\mathbf{if}\ \ \Box i \bullet G_i \rightarrow P_i\ \mathbf{fi}).H_{k-1}(\psi) \vee H_0(\psi) & k > 0. \end{cases}
$$

## 5 A Integrated Refinement Calculus

In this section, we briefly describe a refinement calculus for the modified Z, which is based on Morgan's refinement calculus but uses Z schemas(modified) instead of the specification statement (for more details, see [26]).

The refinement calculus of Back, Morgan and Morris [1, 11, 10], are all based on the work of Dijkstra [7]. Each of the calculus creates a wide-spectrum language by extending a simple imperative programming language with specification constructs. The semantics of both the programming constructs and the specification constructs of this language are given in terms of Dijkstra's weakest preconditions.

The correctness of program transformations is also characterised in terms of weakest preconditions. A large number of refinement rules which capture both traditional design intuitions and guarantee correctness have been proven using this characterisation. These rules range from simple laws which introduce local variables to complex laws which show that recursive procedures are correct.

In this section, we present the refinement calculus for the modified Z: its conversion and refinement laws. Most of the conversion laws are based on those

of [9, 3]. The refinement laws are, on the whole, based on those of Morgan's calculus [10].

Our main objective is to formalise the refinement calculus for the modified Z. Fortunately, a lot of work have been done based on specification statements [9, 3].

During the refinement steps, refinement results produced by the modified Z is more readable and concise than that using the specification statements [4]. Our refinement calculus uses the modified Z as the development language. It has a schemas calculus like Z and also splits the precondition and the postcondition for the refinement. So it obtains the advantages of Z and refinement calculus as well.

Additionally, in [21] we introduce the monotonic schema operations into the modified Z schema and make it possible for the incremental style of program development. This is a distinctive attribute of our work.

In [9], the method for refining Z specifications is divided into two steps. First change notation to the refinement calculus notation, then apply algorithm refinement using the refinement calculus. The first step is also called notation change. The first change is to convert from the undashed/dashed convention to the use of a subscript 0, and to shorten the names of th names of variables, if necessary, also removing ? and ! suffices from the names of input and output variables. Then use a basic law, which is based on Implicit Precondition abbreviation, to translate from Z schemas to specification statements:

$$w : [(\exists\, w : T \mid inv \bullet pred)[w/w_0], pred].$$

The precondition of the specification statement is calculated by existentially quantifying the output variables and dashed variables in the schema's predicate, as described in [25].

As in [3], we use the dashed variables in the modified Z rather than 0-subscript variables in the specification statements, for the sake of simplicity, and keeping the dashing convention of Z to maintain the compliance with this notation. Similarly, we can get the basic conversion law for the translation from the Z schemas to the modified Z schemas.

**Law 51 (Basic Conversion)**

$$\begin{aligned} &[\Delta S;\ di?;\ do! \mid p] \\ &= [d \mid inv \wedge \exists\, ds';\ do! \bullet inv' \wedge p \mid inv' \wedge p] \end{aligned}$$

*where* $S \mathrel{\widehat{=}} [ds \mid inv]$ *and* $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$.

By way of illustration, we consider the specification of the class manager presented in Example 2.1.

**Example 5.2** We convert the Z operation *EnrolOK* to the modified Z schema by applying *Basic Conversion Law.*

$$EnrolOk =$$

*EnrolOk*

$\Delta Class$
$s? : Student$

---

$s? \notin yes \cup no$
$\#(yes \cup no) < max$

---

$yes' = yes$
$no' = no \cup \{s?\}$

□

Z Schemas which specify operations that do not change the state, have the form

$$[\Xi S;\ di?;\ do! \mid p].$$

They can be written as:

$$[\Delta S;\ di?;\ do! \mid p \wedge s1' = s1 \wedge \cdots \wedge sn' = sn]$$

where $\alpha ds = s1 \cup \cdots \cup sn$ and $s1, \cdots, sn$ are state components of schema $S$. Since schemas of this form occur very frequently in Z specifications, we present an additional conversion law as below.

**Law 53 (Basic Conversion)**

$$\begin{aligned} & [\Xi S;\ di?;\ do! \mid p] \\ = & [d \mid inv \wedge \exists\, do! \bullet p[\alpha ds' \backslash \alpha ds] \mid p] \end{aligned}$$

*where* $S \mathrel{\widehat{=}} [ds \mid inv]$ *and* $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$.

Based on the refinement laws in [10, 9, 3], we present the refinement laws for the modified Z. As with Morgan's refinement calculus, the refinement calculus for the modified Z supports procedures, recursion and data refinement. It is completely formalised in terms of the weakest preconditions. The target language of the refinement calculus is an extension of Dijkstra's guarded command language [7]. The laws are listed by the alphabetic order in the appendix. The soundness of all the laws are proved in [25].

In our refinement calculus, the refinement of a Z specification typically begins with the application of a conversion law that transforms its operation schemas into the modified Z schemas, in which the precondition and the postcondition are separated, and then proceeds with the application of proper refinement laws. The case study presented in the paper uses some refinement laws of the refinement calculus of the modified Z, which are also included in the appendix.

## 6 The ZRefiner Structure

We intend to develop a prototype refinement tool, ZRefiner, which supports the development of specification and the refinement of Z. This section presents the structure of the prototype ZRefiner. In general, the structure of ZRefiner is divided into three tiers as in Fig.2.

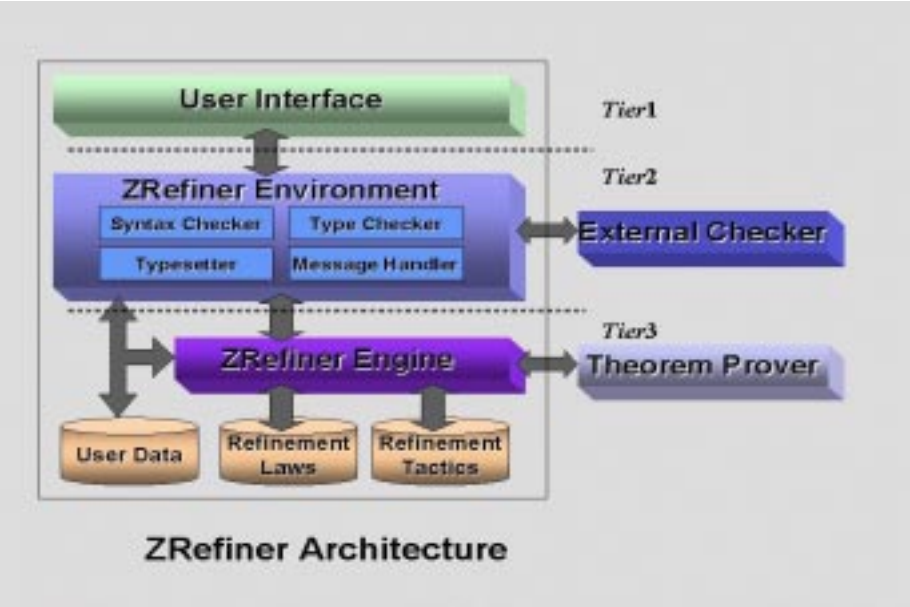



**Fig.2.** ZRefiner Architecture

The first tier is the User Interface which supports all the major features expected of a standard editor including Cut, Copy, Paste, etc. In the User Interface, a user can input a Z specification in the editor, or use the File menu to load a Z specification from a disk file. Fig.3 is a sample of ZRefiner user interface.

The second tier is the ZRefiner Environment, which provides a system environment to handle the messages between the User Interface and the ZRefiner engine. The ZRefiner environment has four components: the syntax checker, the type checker, the typesetter and the message handler. A user may ask the system to analyse the current Z specification. The message handler is in charge of the message handling. First, it will pass the control to the syntax checker to check the syntax of the current Z specification and get the feedback of the check result which is sent back to the user interface. Secondly, it will ask the type checker to do the Z type checking for the current Z specification. The message handler can also connect to an external checker to do some other checks, e.g. domain

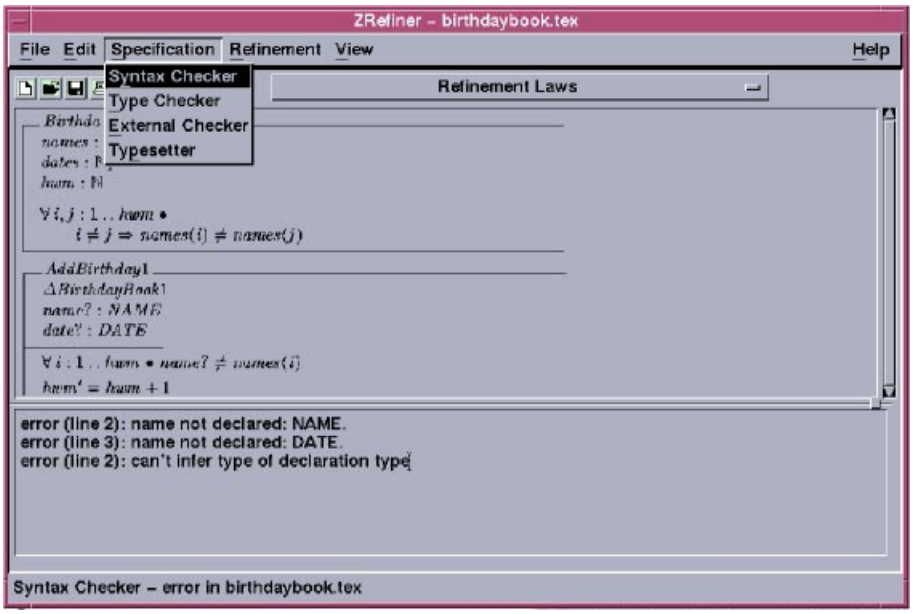



**Fig.3.** ZRefiner User Interface

checking. All message communication between the ZRefiner environment and the external checkers are controlled by the message handler.

The typesetter is used to display the output. The user of ZRefiner can choose between two display modes: text mode and graphical mode. For instance, a Z specification can be displayed in LaTeX format. By using a typesetter, a graphical display with the standard Z schema can be obtained in the user interface so that it is more straightforward and friendly than using the text mode. In ZRefiner, the display of the modified Z schema is also supported by the typesetter.

The third tier of ZRefiner is the ZRefiner engine. The ZRefiner engine is the main part of the refinement tool. It interacts with the message handler and the external theorem prover. When the user chooses to refine a Z specification, the message handler will pass the user's command and this Z specification to the ZRefiner engine. The user's input includes the information about the refinement law selected so that the ZRefiner engine can locate the law and the tactics from the databases. Then it will process the refinement according to the refinement law. During the calculation, the external theorem prover is automatically called by the ZRefiner engine for the proof of certain obligations and the calculation of the precondition. After that, the user can get the refinement results from the user interface.

The advantage of the three-tier structure of ZRfiner is that it is more flexible. As we can see from Fig.1, the tier-1 is in charge of user interaction. The tier-2 is mainly in charge of the specification processing while the tier-3 is managing

the refinement process. Additionally, it provides greater application scalability, lower maintenance and increases reuse of components of all tiers.

# 7 A Case Study on Z Specification Development

## 7.1 Overview and Motivation

To demonstrate our refinement tool, we use the classic case study of the birthday book specified by Spivey [16]. This case study was also used in [3, 4].

## 7.2 Z Specification

The birthday book is a simple system which records people's birthdays and is able to query people's birthdays in its database. For the sake of simplicity, we start the development from a concrete version of the birthday book specification and use only two operations.

The Z specification of a birthday includes basic types: $NAME$ and $DATE$, which $NAME$ is the given set of people's names and $DATE$ is the given set of dates.

$$[NAME, DATE]$$

We describe the state space of the system as a schema named $Birthdaybook1$. The birthday book is represented by two arrays which modeled by functions from the set $\mathbb{N}_1$ of strictly positive integers to $NAME$ or $DATE$. The birthday for $name[i]$ is the corresponding element $dates[i]$ of the array $dates$. The variable $hwm$ (for 'high water mark') shows how much of the array is in use.

$$\begin{array}{|l}
\textit{BirthdayBook}1 \\
\hline
names : \mathbb{N}_1 \rightarrow NAME \\
dates : \mathbb{N}_1 \rightarrow DATE \\
hwm : \mathbb{N} \\
\hline
\forall\, i, j : 1 \,..\, hwm \bullet \\
\quad i \neq j \Rightarrow names[i] \neq names[j] \\
\hline
\end{array}$$

This state schema introduces three state variables. The state invariant states that all the names are different. The first operation $AddBirthday1$ adds a new person into the database. (We ignore the exception cases.) We increase $hwm$ by one and add the record of the new name and date into the arrays.

*AddBirthday*1

$\Delta BirthdayBook1$
$name? : NAME$
$date? : DATE$

---

$\forall i : 1 \mathinner{..} hwm \bullet name? \neq names[i]$
$hwm' = hwm + 1$
$names' = names \oplus \{hwm' \mapsto name?\}$
$dates' = dates \oplus \{hwm' \mapsto date?\}$

The operation *FindBirthday*1 queries the person's birthday at the database. We output the birthday of the person *name*? who is recorded in the array: *names*.

*FindBirthday*1

$\Xi BirthdayBook1$
$name? : NAME$
$date! : DATE$

---

$\exists i : 1 \mathinner{..} hwm \bullet$
$\quad name? = names[i] \land date! = dates[i]$

### 7.3 Refinement

In the following part of the section, we develop the final code for the birthday book by using ZRefiner.

ZRefiner uses the modified Z schemas rather than the specification statements to develop the final program. For the sake of conciseness, ZRefiner takes pred*Schema* as the predicates of *Schema*. The first step of the refinement in ZRefiner is transforming the schema calculus into a modified Z schema of the form: $[d \mid pre \mid post]$. In the Refinement menu of ZRefiner, we select and apply *Basic Conversion Law* to *AddBirthday*1 and obtain the specification statement shown below:

$$AddBirthday1 = \{Basic\ Conversion\}$$

*AddBirthday*1

$names, names' : \mathbb{N}_1 \rightarrow NAME$
$dates, dates' : \mathbb{N}_1 \rightarrow DATE$
$hwm, hwm' : \mathbb{N}$
$name? : NAME$
$date! : DATE$

---

pred *BirthdayBook*1
$\forall i : 1 \,..\, hwm \bullet name? \neq names[i]$

---

pred *BirthdayBook*1′
$\forall i : 1 \,..\, hwm \bullet name? \neq names[i]$
$hwm' = hwm + 1$
$names' = names \oplus \{hwm' \mapsto name?\}$
$dates' = dates \oplus \{hwm' \mapsto date?\}$

We apply *Sequential Composition Introduction Law* in ZRefiner to obtain a sequential composition of two schemas:

$AddBirthday1 \sqsubseteq \{Sequential\ Composition\ Introduction\}$

$|\,[\ \mathbf{con}\ X : \mathbb{N}_1 \ \bullet$

*AddBirthday*1_1

$names, names' : \mathbb{N}_1 \rightarrow NAME$
$dates, dates' : \mathbb{N}_1 \rightarrow DATE$
$hwm, hwm' : \mathbb{N}$
$name? : NAME$
$date! : DATE$

---

pred *BirthdayBook*1
$\forall i : 1 \,..\, hwm \bullet name? \neq names[i]$ ;

---

pred *BirthdayBook*1′
$\forall i : 1 \,..\, hwm \bullet name? \neq names[i]$
$hwm' = hwm + 1$

*AddBirthday*1_2

$names, names' : \mathbb{N}_1 \rightarrow NAME$
$dates, dates' : \mathbb{N}_1 \rightarrow DATE$
$hwm, hwm' : \mathbb{N}$
$name? : NAME$
$date! : DATE$

---

pred *BirthdayBook*1
$\forall i : 1 \,..\, hwm \bullet name? \neq names[i]$
$hwm = X + 1$

---

pred *BirthdayBook*1′
$\forall i : 1 \,..\, hwm \bullet name? \neq names[i]$
$hwm' = X + 1$
$names' = names \oplus \{hwm' \mapsto name?\}$
$dates' = dates \oplus \{hwm' \mapsto date?\}$

$]\,|\,|$

It is easy to see that an assignment can refine the above specification by applying *Assignment Introduction Law* in ZRefiner and get the following code for *AddBirthday*1_1 and *AddBirthday*1_2:

$AddBirthday1\_1 \sqsubseteq \{Assignment\ Introduction\}$
$\quad hwm := hwm + 1$
$AddBirthday1\_2 \sqsubseteq \{Assignment\ Introduction\}$
$\quad names[hwm], dates[hwm] := name?, date?$

Now we pick up the other operation: *FindBirthday*1. Again, we use *Basic Conversion Law* to make the precondition and the postcondition displayed in the different of the specification in ZRefiner.

$FindBirthday1 = \{Basic\ Conversion\}$

*FindBirthday*1

$names : \mathbb{N}_1 \rightarrow NAME$
$dates : \mathbb{N}_1 \rightarrow DATE$
$hwm : \mathbb{N}$
$name? : NAME$
$date! : DATE$

---

pred *BirthdayBook*1
$\exists\, i : 1 \,..\, hwm \bullet name? = names[i]$

---

$\exists\, i : 1 \,..\, hwm \bullet name? = names[i] \land date! = dates[i]$

We may refine this specification using the refinement law for iteration, which should find a proper birthday for a certain person in the database. To proceed, we first introduce an auxiliary variable as the loop index, by applying *Variable Introduction Law* in ZRefiner.

$\sqsubseteq \{Variable\ Introduction\}$
$\quad |\,[\,\mathbf{var}\ k : \mathbb{N}_1 \bullet$

*FindBirthday*1

$names : \mathbb{N}_1 \rightarrow NAME$
$dates : \mathbb{N}_1 \rightarrow DATE$
$hwm : \mathbb{N}$
$name? : NAME$
$date! : DATE$
$k, k' : \mathbb{N}_1$

---

pred *BirthdayBook*1
$\exists\, i : 1 \,..\, hwm \bullet name? = names[i]$

---

$\exists\, i : 1 \,..\, hwm \bullet name? = names[i] \land date! = dates[i]$

$\quad ]\,|$

Next, variable $k$ is put into the specification which then is refined to a sequential composition with a sub-specification *FindBirthday*1_1 and an assignment of the assigning $dates[k]$ to $date!$.

$\sqsubseteq \{Following\ Assignment\ Introduction\}$

$$
\begin{array}{|l}
\hline
\textit{FindBirthday}1\_1 \\
names : \mathbb{N}_1 \rightarrow NAME \\
dates : \mathbb{N}_1 \rightarrow DATE \\
hwm : \mathbb{N} \\
name? : NAME \\
date! : DATE \\
k, k' : \mathbb{N}_1 \\
\hline
\text{pred } BirthdayBook1 \\
\exists\, i : 1 \,..\, hwm \bullet name? = names[i] \\
\hline
\exists\, i : 1 \,..\, hwm \bullet name? = names[i] \land \\
\quad dates[k'] = dates[i] \\
\hline
\end{array}
\quad ; \quad date! = dates[k]
$$

We still need to refine the above schema. By choosing the invariant

$$\exists\, i : 1 \,..\, hwm \bullet name? = names[i] \land \forall\, i : 1 \,..\, k - 1 \bullet name? \neq names[i]$$

we apply *Sequential Composition Introduction Law* in ZRefiner to obtain a sequential composition of two schemas: *FindBirthday*1_1_1 and *FindBirthday*1_1_2, which is split by the loop invariant.

*FindBirthday*1_1 $\sqsubseteq \{Sequential\ Composition\ Introduction\}$

$$
\begin{array}{|l}
\hline
\textit{FindBirthday}1\_1\_1 \\
names : \mathbb{N}_1 \rightarrow NAME \\
dates : \mathbb{N}_1 \rightarrow DATE \\
hwm : \mathbb{N} \\
name? : NAME \\
date! : DATE \\
k, k' : \mathbb{N}_1 \\
\hline
\text{pred } BirthdayBook1 \\
\exists\, i : 1 \,..\, hwm \bullet name? = names[i] \\
\hline
\exists\, i : 1 \,..\, hwm \bullet name? = names[i] \\
\forall\, i : 1 \,..\, k' - 1 \bullet name? \neq names[i] \\
\hline
\end{array}
\quad ;
$$

*FindBirthday*1_1_2

$$names : \mathbb{N}_1 \rightarrow NAME$$
$$dates : \mathbb{N}_1 \rightarrow DATE$$
$$hwm : \mathbb{N}$$
$$name? : NAME$$
$$date! : DATE$$
$$k, k' : \mathbb{N}_1$$

---

$$\exists\, i : 1 \,..\, hwm \bullet name? = names[i]$$
$$\forall\, i : 1 \,..\, k-1 \bullet name? \neq names[i]$$

---

$$\exists\, i : 1 \,..\, hwm \bullet name? = names[i] \land dates[k'] = dates[i]$$

It is clear to see that *FindBirthday*1_1_1 is to establish the initial invariant by the following assignment:

*FindBirthday*1_1_1 $\sqsubseteq$ {*Assignment Introduction*}
  $k := 1$

*FindBirthday*1_1_2 should be refined to an iteration. By strengthening post-condition, we obtain the standard iteration specification, which includes the loop invariant and the boolean guard.

*FindBirthday*1_1_2 $\sqsubseteq$ {*Strengthen Postcondition*}

*FindBirthday*1_1_2

$$names : \mathbb{N}_1 \rightarrow NAME$$
$$dates : \mathbb{N}_1 \rightarrow DATE$$
$$hwm : \mathbb{N}$$
$$name? : NAME$$
$$date! : DATE$$
$$k, k' : \mathbb{N}_1$$

---

$$\exists\, i : 1 \,..\, hwm \bullet name? = names[i]$$
$$\forall\, i : 1 \,..\, k-1 \bullet name? \neq names[i]$$

---

$$\exists\, i : 1 \,..\, hwm \bullet name? = names[i]$$
$$\forall\, i : 1 \,..\, k'-1 \bullet name? \neq names[i] \land name? = names[k']$$

By selecting *Iteration Introduction Law* in ZRefiner, we obtain the following iteration easily:

*FindBirthday*1_1_2 $\sqsubseteq$ {*Iteration Introduction*}
  **do** $name? \neq names[k] \rightarrow$

$$
\begin{array}{l}
|[\ \textbf{var}\ k : \mathbb{N}_1 \bullet \\
\quad k := 1; \\
\quad \textbf{do}\ name? \neq names[k] \rightarrow \\
\quad\quad k := k + 1 \\
\quad \textbf{od}; \\
\quad date! := dates[k] \\
]|
\end{array}
$$

**Fig.3.** The final code of *FindBirthday*1

*FindBirthday*1_1_2_1

$$
\begin{array}{l}
names : \mathbb{N}_1 \rightarrow NAME \\
dates : \mathbb{N}_1 \rightarrow DATE \\
hwm : \mathbb{N} \\
name? : NAME \\
date! : DATE \\
k, k' : \mathbb{N}_1 \\
\hline
\exists\, i : 1 \,..\, hwm \bullet name? = names[i] \\
\forall\, i : 1 \,..\, k - 1 \bullet name? \neq names[i] \land name? \neq names[k] \\
\hline
\exists\, i : 1 \,..\, hwm \bullet name? = names[i] \\
\forall\, i : 1 \,..\, k' - 1 \bullet name? \neq names[i] \land 0 \leq hwm - k' < hwm - k
\end{array}
$$

**od**

As we can see, the part of the modified Z schema is embedded in the **do**-loop. It is so clear to see that the loop body must increase the variant $hwm - k$. So we obtain the following refinement result of the loop body:

$$
\begin{array}{l}
FindBirthday1\_1\_2\_1 \sqsubseteq \{Assignment\ Introduction\} \\
\quad k := k + 1
\end{array}
$$

So comes the final code in ZRefiner, also shown in Fig.3.

In the next section, we compare the differences between the refinement method which ZRefiner used and other methods, and distinguish their advantages and disadvantages.

## 8 Conclusion

The case study illustrates that a Z specification can be developed to code under ZRefiner. In ZRefiner, the first step of development begins with selecting and applying a conversion law that transforms a Z operational schema into a modified

Z schema, in which the precondition and the postcondition are separated, then proceeds with the application of proper refinement laws supported by ZRefiner.

All the proof obligations required by every refinement step are automatically handled by ZRefiner. The reliability of the refinement is assured. The ZRefiner users need not consider any proof procedure but concentrates on the refinement development. The efficiency of the development method is also greatly improved.

Another advantage of using ZRefiner is that the refinement steps produced by ZRefiner is more readable and concise than that using the specification statements [4]. ZRefiner uses the modified Z schema as the development language. It has a schemas calculus like Z and also splits the precondition and the postcondition for the refinement. So it obtains the advantages of Z and refinement calculus as well.

Case studies using the specification statements and pure Z schemas can be found in [27].

### 8.1 Future Work

Since the system we produced here is only a prototype system, a lot of work has to be done to make it practical. Our future work of ZRefiner includes more case studies and practical work on the tool development.

We can see from the previous section, that Z schemas are much more readable than the specification statements in the refinement calculus which do not support any abstraction, since some additional information is hidden by Z notations: $\Delta$ and $\Xi$. We can also see that two notations are used in ZRefiner: the Z notation and the modified Z notation. Is it possible to use only one notation because it is more difficult for a developer to use one notation for specification and the other for development? Since Z notation is widely accepted, it is a good idea to use them in the refinement calculus. The formalisation of the refinement calculus using Z is an interesting motivation for the future investigation.

In the refinement calculus, the final code is guarded commands, which is difficult to execute by the machine. To make the final code be a programming language, such as C, is also an interesting topic. In [23], we investigate a case study of refining Z to C code.

Much work is to be done on strategies for refining Z specifications. Simplicity and efficiency should be considered during refine steps. Based on this, the new refinement laws may be introduced to some steps.

A wide range of tools [24] that support several aspects of its use in specification have been implemented. However, none of them support refinement using Z. The main objective of our research is to develop a fully integrated tool for Z which supports refinement and good graphical user interfaces.

With the three-tier structure of ZRefiner, it is easier to develop and refine its components separately. We can develop the components of each module first, and then integrate them into the three-tier architecture. For the user interface, we wish to develop a perfect window-based system. Some tools can be used to develop the user interface, such as Java Workshop, Motif, Sun Workshop Visual and Tcl/Tk, etc.

Since the technologies for syntax checking, type checking and typesetting are widely available, the development of these components should not be a problem. The major work should be the message handler in tier-2. There will also be a lot of work to be carried out to build a good refinement engine for ZRefiner. Again, we can make use of existing theorem prover. Z/EVES, for example, [14] is a general tool which supports the analysis of Z specifications in several ways: syntax and type checking, schema expansion, precondition calculation, domain checking, and general theorem proving. Since the latest version of Z/EVES supports using the API, it would be interesting to integrate Z/EVES with ZRefiner. Of course, there are other good theorem provers we can investigate [24].

## Acknowledgments

Special thanks to Dr. Ana Cavalcanti for her generosity in sending her thesis and papers to us and also valuable discussions. Thanks also to Mark Saaltink for his help with Z/EVES.

## References

1. R. J. R. Back. On the Correctness of Refinement Steps in Program Development. PhD thesis, University of Helsinki, 1978.
2. F. L. Bauer, H. Ehler, A. Horsch, B. Moller, H. Partsch, O. Puakner and P. Pepper. The Munich Project CIP: Volume II: The Program Transformation System CIP-S, Lecture Notes in Computer Science p.292, Springer-Verlag, 1987.
3. A. L. C. Cavalcanti. A Refinement Calculus for Z. PhD thesis, Oxford University, Computing Laboratory, Programming Research Group, Technical Monograph PRG-123,1997.
4. A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC - A Refinement Calculus for Z. Formal Aspects of Computing, 10(3): 267-289, 1998
5. A. L. C. Cavalcanti and J. C. P. Woodcock. A Weakest Precondition Semantics for Z. The Computer Journal, 41(1):1-15,1998.
6. B. P. Collins, J.E. Nicholls and I. H. Sorensen. Introducing Formal Methods: the CICS Experience with Z. IBM Hursley and PRG Oxford University, 1987.
7. E. W. Dijkstra. A Discipline of Programming. Prentice-Hall, 1976.
8. Ian Hayes. Specification Case Studies. Prentice Hall, second edition, 1993.
9. S. King. Z and the Refinement Calculus. VDM'90 VDM and Z - Formal Methods in Software Development, volume 428 of Lecture Notes in Computer Science, pages 164-188, Kiel-FRG, Spriner-Verlag, April 1990.
10. C. C. Morgan. Programming from Specifications. Prentice Hall, 2nd edition, 1994.
11. J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. Science of Computer Programming 9, pp. 287-306, 1987.
12. C. C. Morgan, P. H. B. Gardiner, and K. A. Robinson. On the Refinement Calculus. Springer-Verlag, 1993.
13. H. A. Partsch. Specification and Transformation of Programs : a Formal Approach to Software Development. Springer-Verlag, 1990.
14. Mark Saaltink. The Z/EVES System. ORA Canada, In ZUM'97: The Z Formal Specification Notation (10th International Conference of Z Users, Reading, UK, April 1997, Proceedings)

15. C.T. Sennet. Demonstrating the compliance of Ada programs with Z specifications. In C.B. Jones, R.C. Shaw, and T. Denvir (eds), *5th Refinement Workshop*, Workshops in Computing, London, pp. 70-87, Springer Verlag, 1992.
16. J. M. Spivey. The Z Notation - A Reference Manual, 2nd edition. Prentice Hall, 1992.
17. Mark Utting. An Object-Oriented Refinement Calculus with Modular Reasoning. PhD thesis, University of New South Wales, Australia, 1994.
18. N. Ward. Adding Specification Constructors to the Refinement Calculus. In J. C. P. Woodcock and P. G. Larsen, editors, FME'93: Industrial-Strength Formal Methods, volume 670 of Lecture Notes in Computing Science, pages 652-670. Springer-Verlag, 1993.
19. J. B. Wordsworth. Software Development with Z. Addison-Wesley Publishers, 1992.
20. J. C. P. Woodcock. Implementing Prompted Operations in Z. 5th Refinement Workshop, Workshops in Computing, London, 1992. Prentice Hall.
21. B. Wu. Towards a Mechanised Software Development Method. MPhil to PhD Transfer Report. Dept of Computing, University of Bradford, Dec 1999.
22. Jim Woodcock and Jim Davis. Using Z : specification, refinement, and proof. Prentice Hall, 1996.
23. B. Wu, D.R.W. Holton and L. Lai. Case Study: A Sales Database. Technical Report. Dept of Computing, University of Bradford, Jan 2000.
24. B. Wu and L. Lai. A Tool Survey on Z Specification and Refinement. Technical Report. Dept of Computing, University of Bradford, July 1999.
25. B. Wu and L. Lai. Calculating the Pre- and Postcondition. Technical Report. Dept of Computing, University of Bradford, Aug 1999.
26. B. Wu and L. Lai. A Weakest Precondition Semantics for the Modified Z. Technical Report. Dept of Computing, University of Bradford, Aug 1999.
27. B. Wu and L. Lai. Combining the Refinement Calculus with Z – Case Studies. Technical Report. Dept of Computing, University of Bradford, Aug 1999.

## A Refinement Laws Used in the Paper

**Law A1 (Assignment Introduction)** *If* $pre \Rightarrow post[vl'\backslash el][\alpha ds'\backslash \alpha ds]$, *where dvl declares the variables of vl, vl contains no duplicate variables, vl and el have the same length and have no free dashed variables, and the corresponding variables of vl and expression of el have the same type, then*

$$[d;\ dvl;\ dvl' \mid pre \mid post] \sqsubseteq vl := el$$

*where* $d \mathrel{\widehat=} ds \cup ds' \cup di? \cup do!$.

**Law A2 (Basic Conversion)**

$$\begin{aligned} & [\Delta S;\ di?;\ do! \mid p] \\ = {} & [d \mid inv \wedge \exists\, ds';\ do! \bullet inv' \wedge p \mid inv' \wedge p] \end{aligned}$$

*where* $S \mathrel{\widehat=} [ds \mid inv]$ *and* $d \mathrel{\widehat=} ds \cup ds' \cup di? \cup do!$.

**Law A3 (Basic Conversion)**

$$\begin{array}{l} \quad [\Xi S;\ di?;\ do! \mid p] \\ = [d \mid inv \wedge \exists\, do! \bullet p[\alpha ds' \backslash \alpha ds] \mid p] \end{array}$$

*where* $S \mathrel{\widehat{=}} [ds \mid inv]$ *and* $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$.

**Law A4 (Following Assignment Introduction)** *If* $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$ *and dvl declares the variables of vl, vl contains no duplicate variables, vl and el have the same length and have no free dashed variables, and the corresponding variables of vl and expression of el have the same type, then*

$$\begin{array}{l} \quad [d;\ dvl;\ dvl' \mid pre \mid post] \\ \sqsubseteq [d;\ dvl;\ dvl' \mid pre \mid post[vl' \backslash el[\alpha ds, vl \backslash ds', vl']];\ vl := el \end{array}$$

**Law A5 (Iteration Introduction)** *If vrt is an integer, each gi and vrt have no free dashed variables, then*

$$\begin{array}{l} \quad [d \mid inv \mid inv[\alpha ds \backslash \alpha ds'] \wedge \neg\ (\vee\ i \bullet gi[\alpha ds \backslash \alpha ds'])] \\ \sqsubseteq \textbf{do}\ \Box i \bullet gi \rightarrow [d \mid inv \wedge gi \mid [\alpha ds \backslash \alpha ds'] \wedge 0 \leq vrt[\alpha ds \backslash \alpha ds'] < vrt]\ \textbf{od} \end{array}$$

*where* $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$.

**Law A6 (Sequential Composition Introduction)** *If mid is a predicate and it has no free dahsed variables and the variables of post are not free in ds, then*

$$\begin{array}{l} \quad [d;\ dx;\ dx' \mid pre \mid post] \\ \sqsubseteq [d \mid pre \mid mid[\alpha ds \backslash \alpha ds']];\ [d;\ dx;\ dx' \mid mid \mid post] \end{array}$$

*where* $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$ *and dx declares the variables of x, dx′, the corresponding dashed variables.*

**Law A7 (Sequential Composition Introduction)** *If mid is a predicate and cl is the constants, x is the variables, cl and x have the same length and the constants of cl have the same type as the corresponding variables of x, the variables of cl and cl′ are not free in mid and* $[d;\ dx;\ dx' \mid pre \mid post]$*, then*

$$\begin{array}{l} \quad [d;\ dx;\ dx' \mid pre \mid post] \\ \sqsubseteq | [\ \textbf{con}\ dcl \bullet \\ \qquad\quad [dx;\ dx';\ do1! \mid pre \mid mid];\ [d;\ dx;\ dx' \mid mid[x \backslash cl]['\backslash] \mid post[x \backslash cl]] \\ \quad ]| \end{array}$$

*where dcl declares the constants of cl, dx declares the variables of x, dx′, the corresponding dashed variables,* $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$ *and* $do1! \subseteq do!$.

**Law A8 (Strengthen Postcondition)** *If* $pre \wedge npost \Rightarrow post$*, then*

$$[d \mid pre \mid post] \sqsubseteq [d \mid pre \mid npost]$$

**Law A9 (Variable Introduction)** *If the variables of vl and vl′ are not free in* $[d \mid pre \mid post]$ *and are not dashed, then*

$$[d \mid pre \mid post] =| [\ \textbf{var}\ dvl \bullet [d;\ dvl;\ dvl' \mid pre \mid post]] |$$

*where dvl declares the variables of vl.*

# Integrating formal methods into the development cycle of a safety-critical embedded software system

P.G. Bertoli, A. Cimatti, P. Traverso
Istituto per la Ricerca Scientifica e Tecnologica, Povo, Italy
E-mail: {bertoli,cimatti,leaf}@irst.itc.it

**Abstract**

This paper describes a technology transfer project where formal specification and verification techniques have been applied in the development of a safety-critical embedded software system. IRST was directly involved in the development of the system, jointly working with the design engineers of a leading company in the design of embedded systems. The project was subject to two major requirements. First, a tight integration of the formal methodologies into the existing development cycle was to be achieved in order to enhance the quality of the design. Second, it was necessary to limit the impact of a new, potentially costly methodology. During the project, a structured specification methodology was defined, tailored to the structure of the system under analysis. This methodology combines the use of the commercial tool OBJECTGEODE with a custom support tool, developed during the project, for the automatic generation of executable models, starting from the formal specification of subcomponents.

**Keywords: Safety-critical systems, formal verification, formal specification methodologies, early debugging, model checking.**

## 1 Introduction

Formal methods have a great potential of application in the development of complex industrial systems [1]. They can be expressive and unambiguous specification methods, and formal verification tools provide for powerful debugging in early stages of design. For these reasons, in certain application

fields, e.g. railways, formal methods are even becoming part of standards [2, 5]. However, the application of formal methods does not come for free. Formal methods require a training effort; they can increase costs, slow down the process of development, and involve changes on the development cycle.

This paper describes a project developed by a major company in collaboration with IRST, where formal methods have been integrated into the development process of a safety-critical industrial system. The system under design is a complex, safety-critical, embedded control system, realized by several distributed, communicating software subsystems. The system features several modes of operation, and performs complex interactions with the environment. The details of the system (simply called SYSTEM in the following) and the name of the company cannot be disclosed at this stage of the project. However, this paper is rather independent of the specific features of the SYSTEM. We will focus on the methodological aspects of the project, and particularly on the solutions adopted during the development to limit the costs of formal methods, though retaining their benefits. Given the complexity and the safety-critical nature of the SYSTEM, a major project requirement was to take advantage of formal methods to develop a high-quality design. This was to be achieved within strict project deadlines, and involved the training on-the-job of design engineers on the formal methodology.

In order to meet such requirements without introducing a major bottleneck, the introduction of formal methods was carefully evaluated. The development process, based on a spiral model, was structured and selectively integrated with the application of formal systems to the design of the most important components and functions. This was achieved by structuring the informal specifications in a modular way, and integrating them with formal descriptions of the subcomponents and of the system requirements. Model checking was used to validate the actual formal descriptions used as part of the specification. In order to make the validation task feasible, a tool was developed to produce formal models of the SYSTEM and its environment starting from the independent formal specifications of the subcomponents. This allowed to easily adopt a variety of abstractions depending on the property under analysis. The formal description and extensive simulation have been useful to validate a core of the system specifications, pinpointing some flaws in the starting informal specifications and feeding the implementors with detailed and non-ambiguous descriptions of the functions.

The paper is structured as follows. Section 2 provides an overview of the SYSTEM. Section 3 describes in detail the requirements and constraints

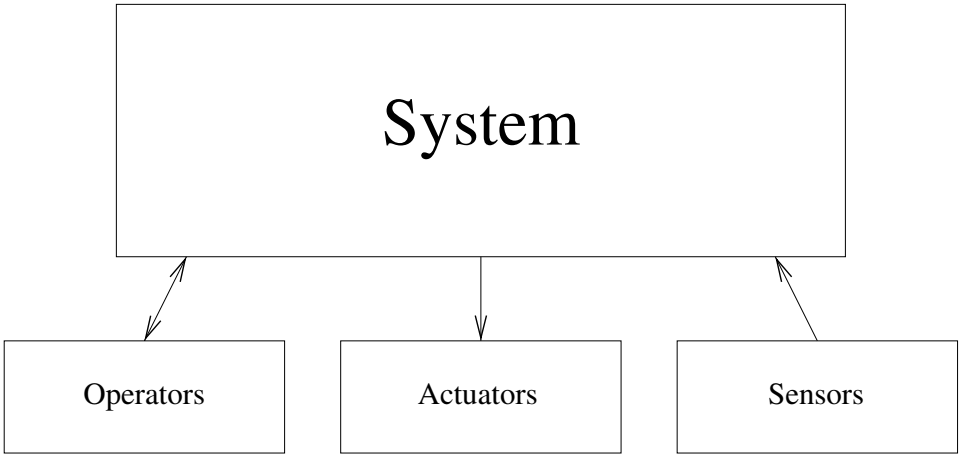



Figure 1: The SYSTEM and its environment

of the project. Section 4 discusses the specific difficulties in the project and the adopted solutions, focusing on the design of a custom specification and validation methodology, and of a tailored support tool. Section 5 discusses some results. Section 6 draws some conclusions and sketches possible future work.

## 2 Informal description of the SYSTEM

### 2.1 Environment

The SYSTEM operates within of a complex environment, interacting with a number of different actors, using several communication protocols (as shown in Fig. 1):

- a number of sensors are connected to SYSTEM. They convey heterogeneous data concerning the physical status of the environment and time-varying constraints which must be obeyed by the SYSTEM. For instance, informations indicating the faulty status of a controlled device can suggest that the SYSTEM must enter a different (e.g. degraded) mode of operation.

- several actuators of different kind allow the SYSTEM to control the op-

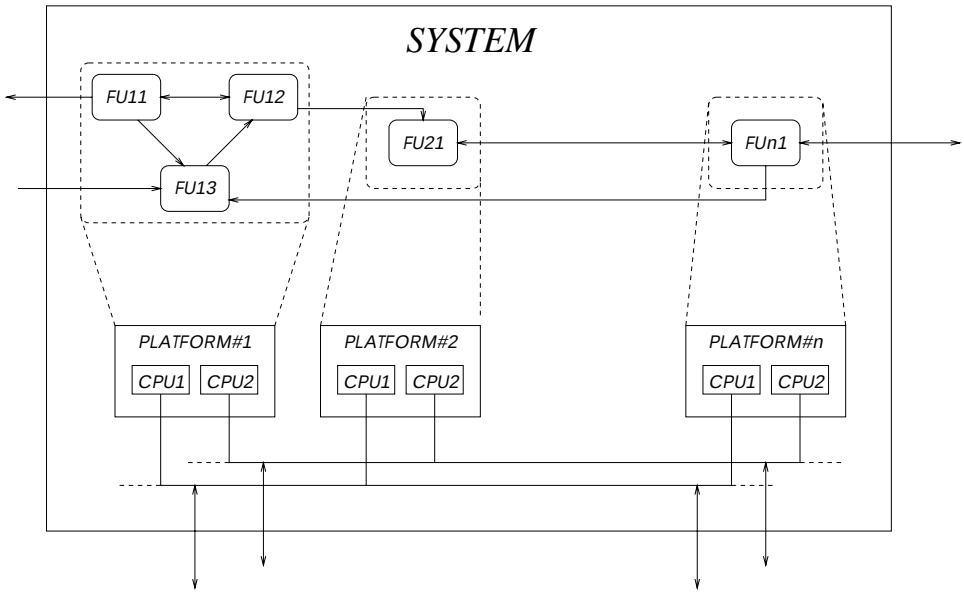



Figure 2: The System architecture

erations according to the specified rules and the status of the external environment.

- human operators may interact with the System, sending commands, selecting operation modes, and providing additional information to respond to data requests.

## 2.2 Functionalities

The main functionality of the System is to determine a safe behaviour which meets the constraints specified from the environment. In order to do so, the System is required to analyze the information acquired from different sources, integrate it, and respond suitably to the resulting conditions. This functionalities can be performed according to different operation modes, which can be selected by a human operator or entered depending on the informations conveyed by sensors. The System must guarantee - as much as possible - the safety of the operations in spite of possible misbehaviours of some of the actors interacting with it. For instance, in case of communication

failure with the external environment, the SYSTEM might aim at a safe, "inactive" state. Under certain particular conditions, however, it must be possible for the human operator to override the behaviour rules determined by the SYSTEM. Finally, the system must be fail-safe, i.e. it must be able to tolerate faults without producing unsafe behaviours, possibly maintaining some of its most important functionalities.

### 2.3 Architecture

In order to provide a better guarantee against hardware failures, the SYSTEM adopts an architecture based on redundancy, shown in figure 2. At the hardware level, the SYSTEM is built on several independent hardware platforms. Each platform runs a pair of CPUs, with a 2-out-of-2 exclusion logic, a run-time checking mechanism ensuring that the application program is executed consistently. The platforms communicate using a field bus systems based on redundancy.

At the software level, the SYSTEM is composed of several distributed programs, running on the different platforms. Each program performs one or more blocks of functionalities, called *"functional units"*. Functional units communicate according to a point-to-point paradigm, implemented by a complex communication protocol. In normal functioning mode, one of the platforms is in charge of performing the SYSTEM's high level functions and commanding the other platforms. The other platforms take care of interacting with the external environment, performing logging activities, commanding the actuators, and so on. The SYSTEM also implements a form of functional redundancy, by doubling the platforms able to perform the functions of the master platform. Two such units are run in parallel, one actually performing the task while the other is in a "hot standby" status. A master/slave switch protocol is used to guarantee that a correctly functioning platform is in charge. Finally, the SYSTEM is a multi-master system: in case both master units fail, one of the slave units is designed to guarantee that a core of functionalities are provided.

## 3 The project

The aim of the project was to integrate formal modeling and verification techniques within the development cycle of the SYSTEM in order to enhance the confidence on the correctness of the SYSTEM itself.

In order to pursue this objective, it was agreed to proceed to the formalization of a core of critical functionalities, and validate them against some of the designed tests. The selection of such paradigmatic functionalities and tests was carried out based on the analysis of the informal functional system specifications, provided as an input by the project's committant.

The project featured some relevant additional requirements and constraints. In particular:

1. the design team was required to use the ObjectGeode set of tools as the means for formal modeling and validation. ObjectGeode adopts SDL [4] as its basic modeling language (although the StateChart formalism [7] is also supported). Various formats are allowed for describing properties that should be obeyed by the system: propositional stop conditions, message sequence charts ([6]), GOAL observers [9]. The key tool of the ObjectGeode set is a SDL simulator which allows various forms of explicit state model checking, e.g. exhaustive, interactive.

2. The software architecture was to be taken into account in the analysis of the integration of the formal methods into the development process. The details of the software architecture are explained in section 4.

Finally, the project was subject to rather tight time constraints. The activity was carried out by a design team involving 6 people over eight months of elapsed time. This included a cross-fertilization phase where the Irst team and the design engineers exchanged know-how the use of formal methods and on the application.

# 4 The approach

## 4.1 Structured formal specification methodology

Integrating formal methods into the development of the System presented a variety of issues, related to its nature and size. It is widely known that applying formal methods to formally validate large software systems can be very hard, and often unfeasible, mostly because of the heavy computational costs of validation processes. A typical problem when using model checking techniques such as those implemented by the ObjectGeode tools consists in the state explosion of the model. This is due to the enormous number of combinations of the state variables which might occur during the simulation

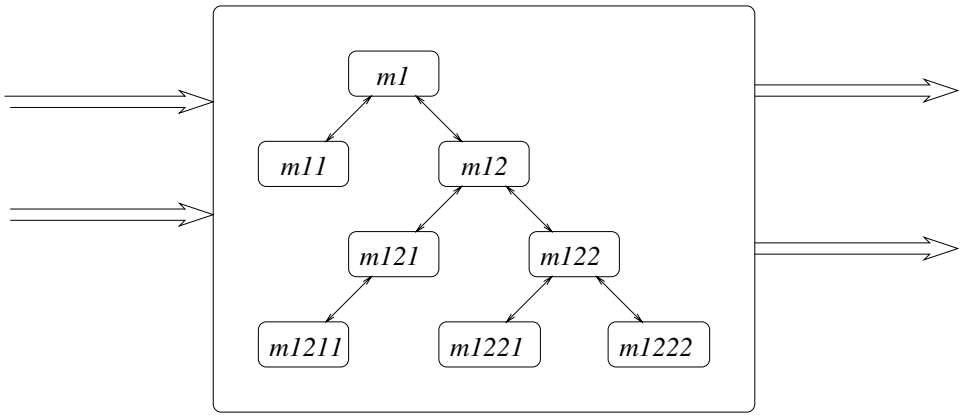



Figure 3: The software architecture of a functional unit

of the model. Several techniques can be used to reduce - often by orders of magnitude - the size of the search space, e.g. driving the search, splitting the search space, abstraction [3, 8]. The task of selecting and fruitfully applying a combination of these techniques is very hard by itself. For instance, the correct degree of abstraction of a concrete system depends on the properties which we intend to prove on it, and on finding a suitable representation for those parts of the system which are most critical in terms of computational effort during the simulation.

A different issue stems from the requirement of integrating formal methods into a dynamic development process, where specifications may evolve over time. This implies that a tight connection must be established between the system specifications, the test specifications and their respective formal modeling, and that it must be possible to maintain conveniently such a connection over time. This can only be achieved by structuring the specifications and the models in a tightly coupled and modular way. Thus we had to design a modeling methodology featuring (a) modularity and (b) amenability to (efficient) formal validation. We took advantage of the structure of the SYSTEM software architecture, preliminarily analyzed by the software team of the industrial partner, to design a tailored specification and validation methodology for the SYSTEM. Each platform realizes one or more functional units by executing them in a reactive loop. Each functional unit must terminate before the next is run; a time-out alarm mechanism is imple-

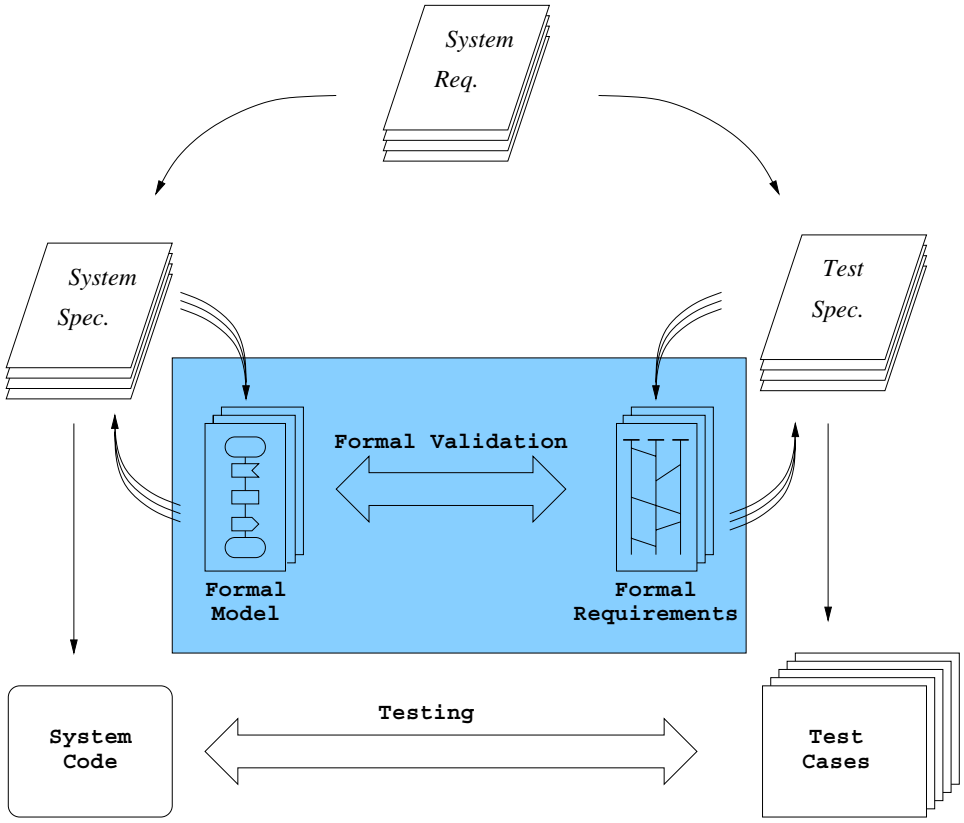



Figure 4: The development cycle of the SYSTEM

mented. In the reactive loop, each functional unit first performs the input, then executes, and finally delivers the outputs to the other units. Figure 3 depicts the software architecture of one of functional units: a functional unit is described as a tree hierarchy of finite state machines (*"machines"* from now on). Machines transfer control synchronously by activation/return signals which may deliver a variable number of values.

Based on this architecture, we chose to model each machine independently, and to describe formally the interfaces presented to the other machines. Such a model would be integrated as a part of the machine's specification. Thus we impacted the previously existing specification methodology by providing a clear rational for modular partitioning of the specifications,

and by integrating formal and informal aspects of the specifications into a unique repository. In this way, modifications to machines are easily reflected into the specifications and the formal modeling alike. Figure 4 clarifies our integration model of formal methods into the standard development cycle of the SYSTEM. The shaded box represents the formal phase of the development. In the standard development cycle, informal system requirements are the starting point to obtain informal system and test specifications. These are translated by the software team into system code and test suites respectively; these, in turn, are used in the testing phase. By formally defining the system and test specifications, a formal validation phase becomes possible independently from the existence of the system code. Indeed, it is possible to intertwine formal modeling/validation and coding phases in order to start from a selected kernel of the system, and to enrich it by adding details at subsequent phases. This allows the V&V teams and the coding teams to work in a pipeline chain, reducing the time impact of the integration. The discovery of inconsistencies in the formal validation phase has an impact over both the formal and informal descriptions of the system and/or of the test specifications. Moreover, by modeling subcomponents independently, it becomes easy to design a variety of abstractions of a machine, keeping its external interface, thus allowing for test-driven modelings of the system.

### 4.2 Support tools

The OBJECTGEODE tools do not provide any specific means to model and validate nets of hierarchies of synchronous finite state machines defined independently. This is due, in particular, to the nature of the SDL language they adopt. Basically, SDL allows the specification of finite state machines (SDL processes, and sub-processes called services) which communicate asynchronously via signals sent over point-to-point channels; a signal may convey a fixed number of values. Each process features an independent variable namespace, shared by its sub-processes. Standard function and procedure constructs are also provided. Within this frame, we identified two main choices to describe the combination of functional units starting from independent descriptions of machines:

- Describe each functional unit as an SDL process, and each machine of the unit as an SDL service, representing procedural invocations by input/output handshaking protocols. However, this solution adds to the complexity of the modeling, since single synchronous invocations expand into sequences of SDL constructs. Furthermore, this solution

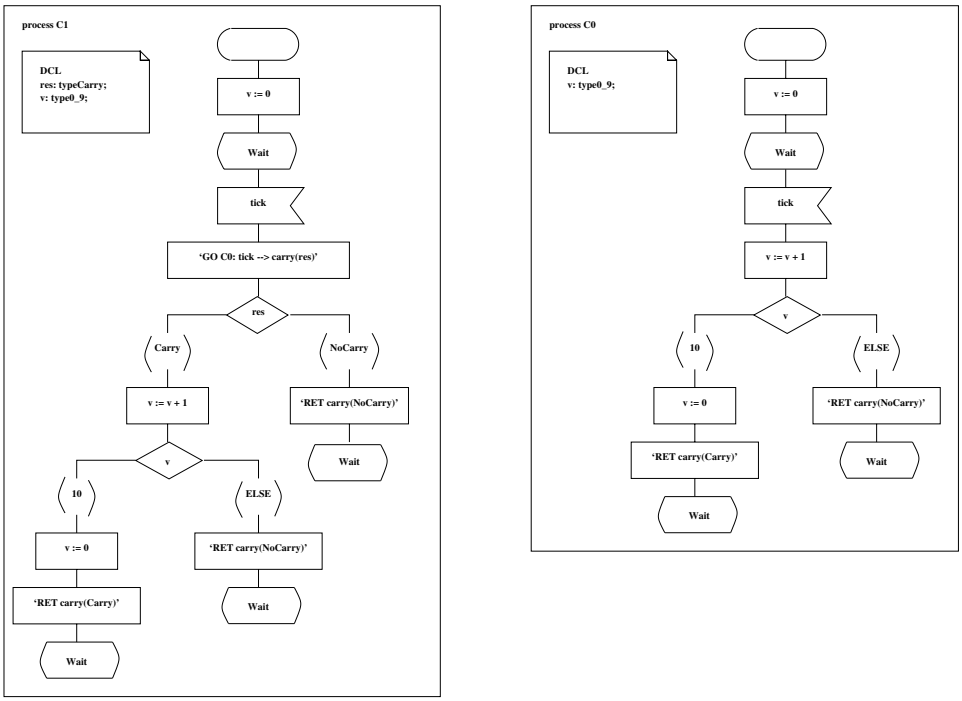



Figure 5: An example of synchronous communication

involves modeling intermediate states to represent the points of control transfer. We experimentally observed that such additional states cause a combinatorial state explosion, making it often unfeasible to proceed to the validation task. On the other this modeling style allows for the observation of intermediate control states, which might be useful for debugging purposes when analyzing simulation traces.

- Describe each functional unit as an SDL process, and each machine of the unit as an SDL procedure. This solution avoids the intermediate states problem caused by the services, and procedural invocations would be represented via the SDL procedure call mechanism. However, several representational issues arise. First, the SDL only admits fixed-arity interfaces for procedures, whereas machines may be invoked in several different modes by their parents using different sets of parameters. This would force the formal modeler to take care of defining

and using complex "union" interfaces. Moreover, persistent states would be represented by global variables; this raises the issue of name clashing between variables referring to different machines - a big issue when dealing with units containing dozens of machines. On the other hand, removing the intermediate states enhances the model's simulation efficiency but decreases its traceability: when a misbehaviour of a functional unit is found, it becomes hard to detect which machine caused it, since no intermediate control state is recorded within the trace.

Our solution to these problems involved the following steps:

1. design a tailored extension to SDL in order to express in a compact and meaningful way synchronous control transfer between machines (as well as asynchronous communication between functional units). We called this extension SDL$^+$. The extension is minimal: the standard SDL send/receive constructs are adopted for modeling asynchronous communication; a pair of constructs (*"GO"* and *"RET"*) are added to model synchronous control transfer. Figure 5 shows a simple example of usage of the synchronous control transfer constructs. The machines C1 and C0 realize a simple two-digit decimal counter. C1 keeps track of the decades values and plays the master to C0, which keeps track of units. The counter is activated by sending a "Tick" signal to C1. C1 activates C0 in turn. C0 updates the units value and informs C1 whether a carry occurred. If needed, C1 updates the decades value, and returns a carry result. Some details, e.g. global type declarations, are omitted for simplicity.

2. design a simple language to describe the topology of the system, and of each functional unit composing it. In particular, the language describes non-ambiguously the hierarchical structures, and allows tagging each machine with an identifier that indicates whether its executable realization should follow the service or procedure modeling paradigm. Moreover, it is possible to specify that certain machines are "fake", i.e. empty placeholders in the hierarchy. This allows to conveniently select relevant parts of a model, e.g. depending on the test that we intend to execute on it.

3. build a custom software to compose hierarchies of independent machines (specified in SDL$^+$) into a unique finite state machine (represented in standard SDL). Such a tool models and combines the

machines following the service or procedure paradigm, according to annotations provided together with the topology of the system. This is useful when debugging complex models: the traces produced by verifying procedure-based models can provide a detailed explanation of the model's behaviour, by using them to drive the execution of the corresponding service-based models (which in most cases would not be amenable to exhaustive verification). We called the tool SdlSdl.

We stress here that the tool is tailored to the architecture of the System, in order to obey the tight time constraints of the project. For instance, no attempt has been made to consider hierarchies other than trees (e.g. DAGs). Also, we developed the tool to feature a very simple error handling, and no attempt is made for error recovery. Figure 6 describes the behaviour of the SdlSdl tool. The tool receives the following input:

- A declaration of the functional units composing the System;
- For each functional unit, the description of the hierarchy of machines it contains, using the simple language described previously;
- For each machine, its description in the SDL$^+$ format. The tool cross-checks informations concerning the topology of the hierarchies in the functional units using the SDL$^+$ descriptions. Moreover, it is in many cases capable of completing an incomplete topological specification.

As an output, the executable SDL model of the system is produced, where procedures or services are used to model machines according to the user's specification. This makes it very easy to produce both the service-based and procedure-based executable models of the system, and use both of them, taking advantage of their different features. For instance, we used procedure-based models to produce bug traces, and service-based models to track bugs down by driving the search with such traces. Moreover, the SdlSdl tool produces a report file which describes the complete topology of the system, and the set of the signatures of the machines. This can be useful to complete/cross-check the functional specifications.

# 5 Results

We applied the specification and validation methodology to the selected functionalities of the System. In particular, we focused on some critical

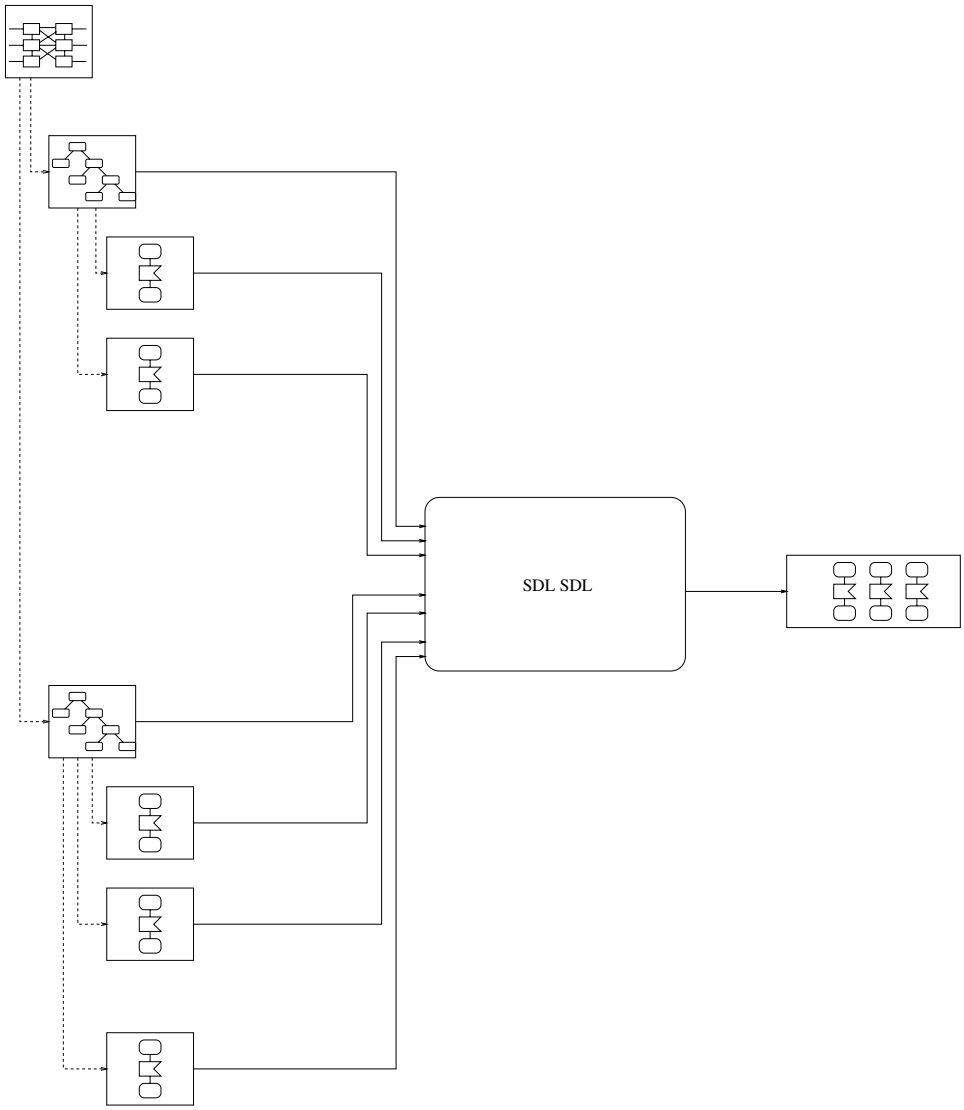



Figure 6: The SdlSdl conversion

functionalities of the master functional unit. We described the startup protocol for the SYSTEM, and some functionalities related to the handling of vital commands coming from the environment.

The resulting models have been suitable to exhaustive validation under selected environmental hypothesis; a typical test would involve visiting in the order of 20000 states, each state occupying approximately 1600 bytes of memory. An exhaustive simulation run would take 30 to 70 seconds on a Sun Sparc 10 workstation running Solaris 2.5 and equipped with 128 Megabytes of RAM. As a result of our exhaustive simulations, we pinpointed some minor underspecifications and misalignements between the specifications of the various machines. More importantly, when testing against the formal modeling of two of the system requirements derived from the test specifications, we discovered two problems which might had potentially led the SYSTEM to unsafe behaviours. The specifications were revised accordingly. At a methodological level, a major result consisted in designing the structured specification methodology, which is currently in use for the ongoing project.

## 6 Conclusions

Our experience in integrating formal methods into the development cycle of a safety-critical software systems has highlighted the advantages and risks of the formal approach. In particular, in order to reduce the cost of the introduction of an additional formal specification/validation, we found it vital to design a custom, application-dependent specification methodology, which takes advantage of the specific system structure. Also relevant to the success of the project has been the design of a specific custom tool to allow the exploitation of existing, powerful general commercial verification tools on our custom methodology. We think that these conclusions have a character of generality, i.e. that the success of applying formal methodologies is strictly related to the possibility of tayloring them to the specific object under exam. This may allow scaling up in the dimension of the tractable problems, which is the real issue in handling the formal modeling and validation of real-life systems.

## References

[1] J. Bowen. Formal Methods in Safety-Critical Standards. Oxford University Computing Laboratory Technical Report, 1995.

[2] J. Bowen. The Industrial Take-Up of Formal Methods. Oxford University Computing Laboratory Technical Report, 1995.

[3] E. Clarke, O. Grumberg, and D. Long. Model Checking. In *Proceedings of the International Summer School on Deductive Program Design*, Marktoberdorf, Germany, 1994.

[4] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL - Formal Object-oriented Language for Communicating Systems.* Prentice Hall Europe, 1997.

[5] European Commitee for Electrotechnical Standardization. European Standard - Railway Applications: Software for Railways Control and Protection Systems. EN 50128, 1995.

[6] J. Grabowski, P. Graubmann, and E. Rudolph. The standardization of Message Sequence Charts. In *Proceedings of the IEEE Software Engineering Standards Symposium*, pages 48–63, Brighton, September 1993. IEEE Computer Society Press.

[7] D. Harel and E. Gery. Executable Object Modeling with Statecharts. In *Proceedings of the 18th international conference on Software engineering*, pages 246–257. ACM, March 1996.

[8] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[9] Verilog. *ObjectGeode SDL Simulator User Manual - The GOAL language.* Verilog, 1999.

# Conditions for synthesis of communicating automata from HMSCs

Loïc Hélouët, Claude Jard

*IRISA/CNRS*
*Campus de Beaulieu*
*35042 Rennes cedex FRANCE*
helouet@irisa.fr,jard@irisa.fr
http://www.irisa.fr/pampa

**Abstract**

Formal methods can now be used at early stages of the development process. This increases the need for consistency between two levels of formalism: a declarative level of scenario type, and an operational level of automata type. It is particularly true in the telecommunications world with the joint use of standardized languages like High-level Message Sequence Charts (HMSCs) and SDL. So it is natural to consider the transition from one level to another by automated synthesis mechanisms. Algorithms for synthesis of SDL communicating systems from HMSCs have been proposed these last years. However, the theoretical power of HMSCs is such that only a subset of HMSCs can be reasonably treated. Identification of the limits of synthesis is still in its early stages. In this article we show for the first time a necessary condition so that synthesis preserves behaviours. This condition relies on a property of generalized local choice and reconstructibility of the local sequencing. It is decidable and we present algorithms that could be implemented in practical synthesis tools.

*Key words:* Message Sequence Charts, synthesis, distributed systems.

## 1 Introduction

Specification of distributed systems starts at a high level of abstraction. An intuitive formalism must be used to capture the behaviours of communicating entities, leaving implementation details for later refinement steps. Two main approaches can be distinguished. The sequential approach (mostly based on communicating automata) emphasizes sequences of events within processes of the system. The specification is given as behaviours of single processes, that contain communication events. Languages like SDL or Estelle (5) can be used for this purpose. They are equipped with tools that allow many formal manipulations, ranging from simulation to code generation. The second approach

is communication-based. It emphasizes on communications between processes. The specification is given through patterns of actions and communications performed on different processes. Use cases and scenarios are examples of such kind of specification formalisms.

The main drawback of the sequential approach is the difficulty to reason about global properties (even the simple fact that a given sending event matches a receiving event). On the other hand, sequential communicating processes can be directly implemented, as they just represent local sequences of communication primitives and internal actions.

Scenarios have the advantage to give a global view of the system activity. Causalities and concurrency are explicitly represented. Nevertheless, composing scenarios is not an easy task, and the overall meaning of a specification can be less intuitive than expected. For example, new concurrency may be introduced during sequential composition. Furthermore, scenarios are not directly implementable (in a distributed way). This may explain why they are often restrained to documentation purposes in methodologies like UML (8), and in the SDL-based tools.

Transforming a scenario model into a set of communicating finite state machines is a first step towards their implementation. This goal needs to provide answers to questions such as:

- Is a scenario implementable?
- If the answer is no, what should be changed to make it implementable?
- Is there a class of scenarios which is known to be easily implementable?

Algorithms for synthesis of SDL communicating systems from HMSCs have been proposed these last years. But the theoretical power of HMSCs is such that only a subset of HMSCs can be reasonably treated. Identification of the limits of synthesis is still in its early stages. In this article we show for the first time a necessary and sufficient condition such that synthesis preserves behaviours. This condition relies on a property of generalized local choice and reconstructibility of the local sequencing. It is decidable and we present algorithms, which could be implemented in practical tools for synthesis.

This article is organized as follows: first section describes HMSCs and defines the notion of language behaviours and of reconstructibility. Section 3 provides a state of the art of synthesis methods from HMSCs. Section 4 presents our target model of communicating finite state machines (CFSM) and formalizes the synthesis approach used by (1; 7) to synthesize SDL from HMSCs. Section 5 outlines the problems met by this approach, and proposes a condition on HMSCs that would make it valid with respect to languages equivalence.

## 2 Message Sequence Charts

This section introduces Message Sequence Charts, a scenario formalism standardized by the ITU (11). MSCs are defined by two levels of specification: basic Message Sequence Charts, which define simple communication scenarios, and High-level Message Sequence Charts, a kind of scenario automaton, that composes basic charts.

### *2.1 Basic Message Sequence Charts*

Basic Message Sequence Charts (bMSCs for short) model a communication pattern between processes (called *instances*). Each instance defines a sequence of events, and is represented by a vertical axis. An event can be a message emission or reception, a timer operation, or an atomic action. As no precise meaning is associated to timer events within this article, they will be considered as atomic actions. Notice the very important fact that communications are closed: sending and receiving of a message are localized in the pattern. This explains the decidability results presented in the paper. A bMSC defines precedence relations on events: a message emission must precede the corresponding reception, and events are totally ordered along instance axis. Therefore, a bMSC can be formalized as a finite, non-autoconcurrent labeled partial order.

A bMSC is a tuple $M = (E, \leq, I, \phi)$ where:

- $E$ is a finite set of events,
- $\leq$ is a partial order relation (antisymmetric, reflexive and transitive) called *causal order* on events,
- $I$ is a set of names of instances that perform at least one action in $M$, and is called the set of *active instances* of $M$.
- $\phi : E \longrightarrow I$ is a labeling of events. It is required that this labeling is not auto-concurrent, which means that events belonging to the same instance are totally ordered (form chains):

$$\forall (e_1, e_2) \in E^2,\ \phi(e_1) = \phi(e_2) \implies (e_1 \leq e_2) \vee (e_2 \leq e_1)$$

Slightly abusing the notation, we will note $\phi(E) = \{i | \exists e \in E \wedge \phi(e) = i\}$ the set of instances appearing in any set of events $E$. For any MSC $M$, we will note $min(M) = \{e \in E | \nexists e' \neq e \wedge e' \leq e\}$ the set of minimal events of $M$, and $min_i(M) = e \in E | \phi(e) = i \wedge \forall e', \phi(e') = i, e \leq e'$ the minimal event on instance $i \in I$. For any event $e \in E$, $pred(e) = \{e' | e' \leq e\}$ will denote the set of predecessors of $e$. We will also denote by $em(e)$ the sending event corresponding to the receiving event $e$. For any labeled order

$M = (E_M, \leq_M, I_M, \phi_M)$, for any set $E' \subseteq E_M$, we will denote by $M_{/E'}$ the restriction of $M$ to events of $E'$.

The events of $E_M$ can be of three types: `send(m) to j` for the emission of the message $m$, `rec(m) from j` for the reception of the message $m$, and `action` for an internal event (atomic action, or operation on a timer). We will often note $!m$ the sending event, and $?m$ the receiving event for a message $m$.

Consider, for example bMSC $M$ in Figure 1. $M$ represents a communication pattern between two processes $A$ and $B$, and can be formalized by a labeled partial order $M = (E_M, \leq_M, I_M, \phi_M)$ where :

- $E_M = \{ \quad e_1 = \texttt{send(m1) to B}, e_2 = \texttt{send(m2) to B},$
  $\qquad\qquad e_3 = \texttt{rec(m1) from A}, e_4 = \texttt{rec(m2) from A}\}$
- $\leq_M = \{(e_1, e_2), (e_1, e_3), (e_2, e_4), (e_3, e_4)\}$
- $I_M = \{A, B\}$
- $\phi_M = \{(e_1, A), (e_2, A), (e_3, B), (e_4, B)\}$

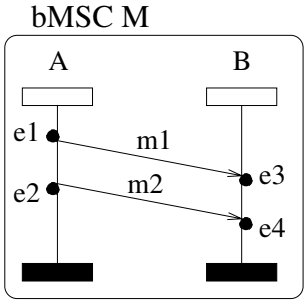



Fig. 1. An example bMSC.

Standard notation of bMSCs also allows for the definition of a zone on an instance axis called co-region, in which events are not ordered. According to the assumed semantics of co-regions, it can mean that events are concurrent, or that the order is not yet defined, and should be specified in further refinements of the specification. As we do not consider co-regions as a central point for our approach, we leave them for further extensions of our work. Therefore, events will be considered as totally ordered on an instance axis.

### *2.2 bMSC automata (HMSC)*

bMSCs only allow for the specification of simple scenarios. A higher level notation called High-level Message Sequence Charts (HMSCs for short) is used to define more elaborated behaviours. Longer patterns can be constructed by sequentially composing bMSCs. HMSC $H_1$ in Figure 2 is a sequential composition of bMSCs $M_1$ and $M_2$. The semantics of the sequence of bMSCs

defined in the standard is a weak sequential composition[1]. The result is an instance-by-instance concatenation, where, for each instance, the maximum event of the first bMSC is linked to the minimum event of the second bMSC. This gives to MSCs an interesting expressive power since communication messages can be accumulated between instances by concatenating basic patterns. For algorithmic reasons, some authors restrict the composition to a strong sequencing, forcing a synchronization barrier between each pattern. We think this dramatically decreases the modeling power of MSCs. Let us consider again the example of Figure 2. Imposing a strong sequencing between $M_1$ and $M_2$ forces instance $A$ to wait for a synchronization with instance $B$ before sending message $m_2$. Therefore, an implementation of $H_1$ assuming strong sequencing between $M_1$ and $M_2$ would have to introduce new communications, and the only trace defined by HMSC $H_1$ would be $!m_1.?m_1.!m_2.?m_2$. However, we think there is no reason to delay a process in a distributed system when no message reception is expected. Moreover, the bMSC decomposition of a specification is arbitrary, and is more the result of a need to reduce the size of charts than the expression of a synchronization. So, HMSC $H_1$ in Figure 2 should be considered equivalent to HMSC $H_2$ in Figure 3, and defines two possible traces: $!m_1.?m_1.!m_2.?m_2$, and $!m_1.!m_2.?m_1.?m_2$.

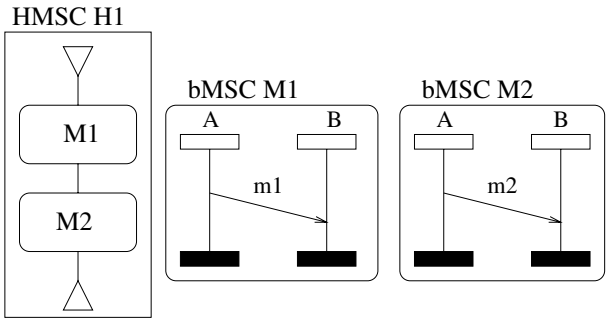



Fig. 2. HMSC $H_1$: sequence of bMSCs $M_1$ and $M_2$.

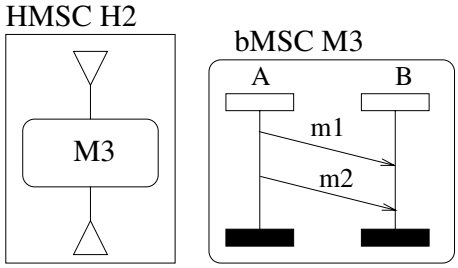



Fig. 3. HMSC $H_2$ equivalent to HMSC $H_1$ in Figure 2.

Let us define the sequencing operator $\circ$ on two MSCs $M_1 = (E_1, \leq_1, I_1, \phi_1)$ and $M_2 = (E_2, \leq_2, I_2, \phi_2)$: $M_1 \circ M_2 =< E, \leq_{M_1 \circ M_2}, I_1 \cup I_2, \phi >$, where:

[1] Weak sequential composition is close to the Pratt's local sequencing (9), where $\phi$ defines locality.

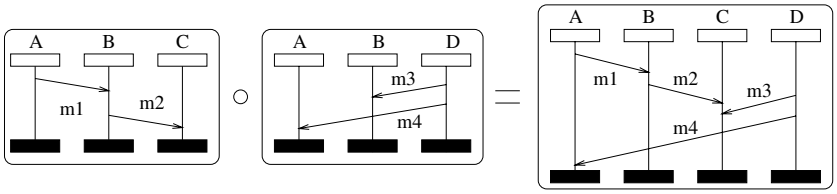



Fig. 4. Chain by chain concatenation of basic message sequence charts.

- $E$ is the disjoint union of $E_1$ and $E_2$ : $E = \varphi_1(E_1) \cup \varphi_2(E_2)$ with $\varphi_1(E_1) \cap \varphi_2(E_2) = \emptyset$ and $\varphi_1, \varphi_2$ are two isomorphisms.
- $\forall e, e' \in E$, $e \leq_{M_1 \circ M_2} e'$ iff $\varphi_1^{-1}(e) \leq_1 \varphi_1^{-1}(e')$ or $\varphi_2^{-1}(e) \leq_2 \varphi_2^{-1}(e')$ or $\exists (e_1, e_2) \in \varphi_1(E_1) \times \varphi_2(E_2)$ : $\phi_1(\varphi_1^{-1}(e_1)) = \phi_2(\varphi_2^{-1}(e_2)) \wedge \varphi_1^{-1}(e) \leq_1 e_1 \wedge e_2 \leq_2 \varphi_2^{-1}(e')$
- $\forall e \in E$, $\phi(e) = \phi_1(\varphi_1^{-1}(e))$ if $e \in \varphi_1(E_1)$ or $\phi(e) = \phi_2(\varphi_2^{-1}(e))$ if $e \in \varphi_2(E_2)$

More intuitively, sequential composition consists in ordering events $e_1$ in bMSC $M_1$ and $e_2$ in bMSC $M_2$ if they are situated of the same instance, and then calculating the transitive closure of the partial order obtained. An example of sequential composition is provided Figure 4.

The standard HMSC notation also contains a parallel composition operator, that will not be considered in this article. In most of the cases, HMSCs comprising parallel composition can be translated into HMSCs comprising only sequence and choice operators. Now, consider HMSC $H_3$ in Figure 5: HMSCs $H_4$ and $H_5$ are composed within a parallel frame (parallel composition is denoted by $\|$).

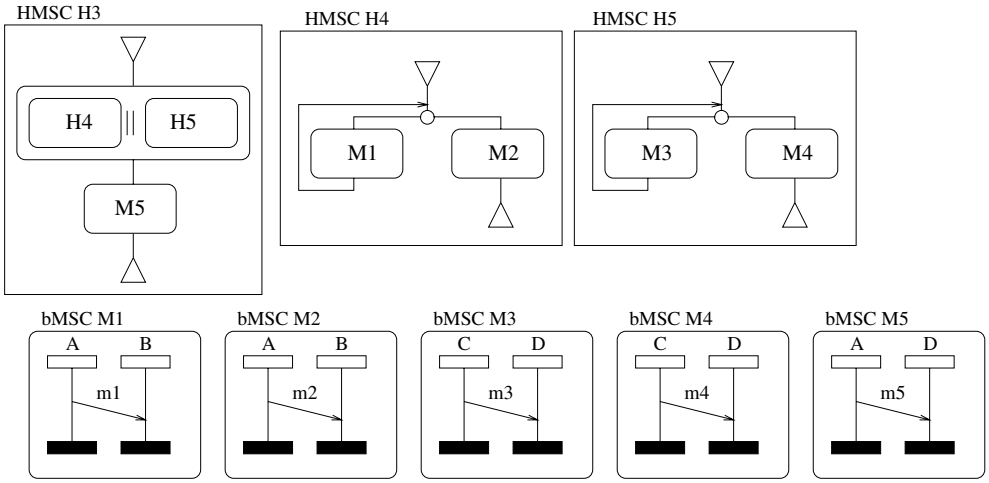



Fig. 5. HMSC $H_3$ comprising two parallel loops.

This example comprises two potentially infinite loops that behave in parallel. When one of the operands stops looping, the other specification also have to stop. Therefore, this specification contains an implicit synchronization, that can be compared to the non-local choice described in section 5.2. As we want to avoid that kind of specification, we limit our approach to HMSCs comprising

choices, sequences, and loops. Those HMSCs can be defined using a finite automaton on bMSCs.

**Definition 1** *A* HMSC *is a graph* $H = (N, \longrightarrow, \mathcal{M}, l, n_0)$*, where:*

- $N$ *is a finite set of nodes,*
- $\longrightarrow$ *is the transition relation* ($\subseteq N^2$)*,*
- $\mathcal{M}$ *is a set of bMSCs, on disjoint sets of events. Each bMSC* $M \in \mathcal{M}$ *is a tuple* $M =< E_M, \leq_M, I_M, \phi_M >$*,*
- $l$ *is a labeling function on transitions* ( $l : N^2 \longrightarrow \mathcal{M}$ )*,*
- $n_o$ *is the starting node of the graph.*

**Definition 2** *A finite* path *of a HMSC* $H$ *is a word* $p = n_1..n_k \in N^*$ *such that* $\forall i \in 1..k-1, (n_i, n_{i+1}) \in \longrightarrow$*. Each path* $p = n_1...n_k$ *defines a unique order* $O_p = M_1 \circ .. \circ M_{k-1}$ *where* $\forall i \in 1..k-1, M_i = l(n_i, n_{i+1})$*. An* initial path *is a path starting from* $n_0$*.*

**Definition 3** *A HMSC* $H$ *defines a partial order family* $\mathcal{O}(H)$*, which is the set of orders* $\{O_{p_1}, O_{p_2}, ..\}$ *associated to the set of initial paths* $\{p_1, p_2, ...\}$ *of the automaton. By considering the total orderings that are compatible with at least one order of* $\mathcal{O}(H)$*, we can define the* language *accepted by a HMSC.*

*Let* $M = (E, \leq, I, \phi)$ *be a bMSC, and let* $w = e_1..e_n$ *be a word of* $E^*$*.* $w$ *is a* linearization *of* $M$ *if and only if:* $\forall i \in 1..n-1, \forall j > i, (e_j, e_i) \notin \leq$*. Let us call* $\mathcal{L}(H)$ *the language described by a HMSC* $H$*.* $\mathcal{L}(H)$ *is the prefix closed set of linearizations defined by the elements of* $\mathcal{O}(H)$*. Let us note* $E_{\mathcal{M}} = \bigcup_{M \in \mathcal{M}} E_M$ *the set of events of* $H$*. A word* $w \in E^*_{\mathcal{M}}$ *is a word of* $\mathcal{L}(H)$ *if and only if:*

$$\exists v = w.u \wedge \exists p = n_0..n_k \text{ initial path of } H \wedge v \text{ is a linearization of } O_p$$

**Definition 4** *A* choice *in a HMSC* $H$ *is a node with more than one successor. A choice* $c$ *defines an alternative between scenarios. Any loop-free path starting from* $c$ *will be called a* branch *of the choice* $c$*.*

### *2.3 Reconstructibility*

A difficult question raised by synthesis is how to impose locally on the events of each instance, the order globally defined by the HMSC. Knowing that the message receptions are undergone and not controlled, a key point will be the possibility of rebuilding the desired order from the order of the received messages. It is what we call reconstructibility.

**Definition 5** *Let* $R$ *be a partial order relation on a set of events* $E$*, and* $\phi$ *be a labeling of* $E$*. A* non-local transitive reduction *of* $R$ *is the set of pairs*

$(e, e') \in R$ *such that* $\phi(e) \neq \phi(e')$ *and* $\nexists e''|(e, e'') \in R \wedge (e'', e') \in R$. *It is denoted by* $e \rightarrowtail e'$.

In a bMSC, the non-local transitive reductions of the causal order are the pair of events associated to communications (when $e \rightarrowtail e'$, then $e$ is a message emission and $e'$ is the corresponding reception).

**Definition 6** *The* message-transitive closure *(or mt-closure, for short) of a partial order relation* $R$ *is written* $R^{*mt}$, *and is a relation* $R'$ *such that* $(e, e') \in R'$ *if and only if:*

- *i)* $(e, e') \in R$, *or*
- *ii)* $\exists e'' \in E$ *such that* $eR'e'' \wedge e''R'e'$, *or*
- *iii)* $\exists e_1, e_2 \in E^2$ *such that* $\phi(e1) = \phi(e_2) \wedge e_1 R' e_2 \wedge e_1 \rightarrowtail e \wedge e_2 \rightarrowtail e' \wedge \phi(e) = \phi(e') \wedge (e', e) \notin R$.

Obviously, as $^{*mt}$ is a closure operation, any element of $R$ must be in $R^{*mt}$ (condition *i*)). *ii*) expresses transitivity on the causality relation between events. *iii*) $e_1$ and $e_2$ are message emissions, and $e$ and $e'$ are the corresponding receptions. As no ordering between $e$ and $e'$ exist, and as any pair of event of the same instance must be ordered, the order between message receptions is the same as the order between the corresponding emissions.

**Definition 7** *A pair* $(e, e')$ *is said to be* reconstructible *by message-transitive closure in a relation* $R$ *if and only if* $(e, e') \in R^{*mt}$

The pair $(e, e')$ in the first situation of Figure 6 is reconstructible. Dotted arrows symbolize the reconstructed edges. On the other hand, the pair $(e, e')$ in the second situation is not reconstructible, as $e_1 \not\rightarrowtail e$. The pair $(e', e)$ of the third situation is not reconstructible either, as the pair $(e, e')$ already exists.

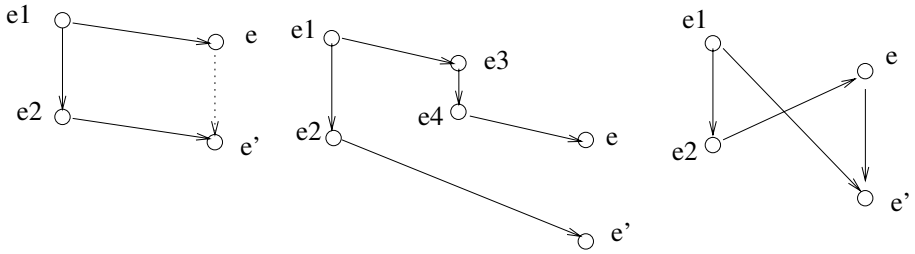



Fig. 6. Order reconstruction allowed by FIFO communications.

A synthesis method from HMSCs to communicating automata will be considered as correct if any ordering information is preserved by the transformation, or can be reconstructed using the mt-closure.

## 3 State of the art

Synthesizing protocols from HMSCs is not a new problem. Several approaches have been proposed. In (10), MSCs are seen as descriptions of finite state machines communicating synchronously, composed by means of regular expressions. The proposed synthesis algorithm is to project the regular expressions on each instance, to produce a protocol (i.e. a set of synchronously communicating finite state machines). Then the traces of the synthesized system are compared with the traces allowed by the HMSC. If the set of traces are equivalent, then the protocol is considered correct. This method suffers two drawbacks. First, an erroneous protocol can be constructed from a specification containing a non-local choice (defined in section 5.2), and potentially leading to deadlock. Second, concurrency between MSCs is interpreted as interleaving. When concurrency appears within an iteration, some correct protocols may be considered as incorrect during the trace comparison.

(4) addresses the question whether a given bMSC can be the behaviour of an implementation model. This approach distinguishes five different buffering methods (ranging from one FIFO channel per message to synchronous communications). Some implementation models are equivalent (they allow for the implementation of the same bMSCs). Then, a definition of implementability for a class of architecture is provided. A bMSC $M$ is said to be weakly implementable for a class architecture if at least one trace of $M$ can be implemented. A bMSC $M$ is said to be strongly implementable if all traces of $M$ can be implemented. This article only addresses bMSCs, and implementability of two bMSCs $M_1$ and $M_2$ on a given architecture does not ensures that a composition of $M_1$ and $M_2$ is implementable using the same communication model.

(6) proposes to synthesize ROOM models from MSCs. ROOM charts are a kind of asynchronous state-charts. MSC sequential composition is considered as strong sequencing of orders, which reduces expressiveness (the language of a HMSC is the concatenation of linearizations of the scenarios, without any possibility of shuffling these words). The synthesis algorithm can be used on a subset of MSCs that do not contain non-local choices, message overtaking, or internal actions. Furthermore, MSCs are supposed to be normalized (i.e. bMSCs do not have common prefix at choice nodes), and each instance is supposed to execute at least one emission or one reception. Consequently, the non-local choice decision required by this article can be computed by considering only the immediate successors of choice nodes. We will see in section 5.2 that the standard case of weak sequential composition demands to search all reachable bMSCs to detect non-locality.

In (7), weak sequencing of HMSCs is considered. The HMSC is projected

on its instances, which gives a set of skeletons of finite state machines, that are translated into SDL processes. The synthesis method assumes an SDL-like communication channel between each pair of communicating processes. However, the SDL system allows more traces than those defined by the HMSC specification. This is due to the impossibility of preserving an order between message receptions from different senders. This approach is implemented in MOST (Moscow Synthesizer Tool).

In (1) SDL processes are synthesized according to a given communication architecture. The most permissive architecture associates a SDL channel to each pair of communicating instances (the approach is then very similar to (7)). Some more restrictive architectures may prevent even simple HMSCs from being implemented. Again, the synthesized protocol may produce traces that are not specified by the HMSC. This approach is implemented in the MSC2SDL tool.

## 4 Automatic derivation of communicating finite state machines

Communicating finite state machines (CFSMs for short) are a commonly used representation of distributed systems (each process is described by a finite state machine that can send or receive messages). Communication channels are FIFO: reception of a message $m$ can be performed if and only if $m$ is the first message that can be consumed from the channel (see (3) for example). This does not allow message overtaking, which can be nevertheless specified in HMSC. Instead of considering a complex buffering mechanism for each process in that case, we prefer to slightly modify the communication semantics. We consider a message reception as possible if the message is present in the queue. This communication semantics is very similar to SDL communication without implicit message consumption.

### *4.1 Synthesis method*

This section formalizes the approach defined in (1; 7). The target model for synthesis is finite automata communicating through queues, one queue being associated to each pair of automata.

Let $H = (N, \longrightarrow, \mathcal{M}, l, n_0)$ be a HMSC describing the behaviour of a set of communicating instances $I$, then:

- $E_{\mathcal{M}} = \bigcup_{M \in \mathcal{M}} E_M$ is the union of events of bMSCs in $\mathcal{M}$.

- $E_{|i} = \{e | \phi(e) = i\}$ is the restriction of $E_{\mathcal{M}}$ to the set of events performed by instance $i$.

A set of CFSM $\mathcal{A}_H = \{\mathcal{A}_i\}_{i \in I}$ can be computed from $H$. Each CFSM $\mathcal{A}_i$ is a tuple $\mathcal{A}_i = (S_i, E_i, \delta_i, s_{0_i})$ such that:

- $S_i \subseteq \mathcal{P}(E_i)$ is a set of states,
- $E_i = E_{|i}$ is a set of events,
- $\delta_i \subseteq S_i \times E_i \times S_i$ is a set of transitions. $(s, e, s') \in \delta_i$ if and only if:
  - $e \in s$, and
  - $s' = \{e' \in E_i | \quad \exists p$ path of $H$ such that $O_p = M_1 \circ .. \circ M_k$ , $e \in M_1$ and $e' = min_i(O_{p/E_{O_p} \setminus pred(e)})\}$,
- $s_{0_i} = \{e \in E_i | \exists p$ initial path of $H$, and $e = min_i(O_p)\}$ is the initial sate.

More intuitively, a state is a subset of fireable events. An event can be fired if it is contained in a state. The resulting state is composed of events located on the same path as $e$ that are allowed after the execution of $e$.

### *4.2 Language of CFSMs*

A set of CFSMs defines a possibly infinite transition system.

A **state** of a CFSM system composed of $K$ communicating machines is a pair $S = (\{s_i\}_{i \in 1..K}, \{w_{ij}\}_{i,j \in 1..K})$, where:

- $\{s_i\}_{i \in 1..K}$ is a set of local states,
- $\{w_{ij}\}_{i,j \in 1..K}$ is a word representing the messages transiting from $i$ to $j$,

The initial state for a CFSM system is $S_0 = (\{s_{0_i}\}_{i \in 1..K}, \{\epsilon, ..., \epsilon\})$. Considering any subword $w$ of a buffer $w_{ij}$, we will note $|w|_m$ the number of occurrences of the message $m$ in the word $w$.

A **transition** from a state of the CFSM system to another state is possible if one of the components of the system can perform it.

$$\text{- } \exists j \in 1..K \ : \ e \in s_j \wedge e = \texttt{rec(m) from i} \ \wedge \ w_{ij} = v.m.v' \wedge (s_j, e, s'_j) \in \delta_j$$
$$\wedge \forall e' = \texttt{rec(m') from i}: \ (s_j, e', s''_j) \in \delta_j, |v|_{m'} = 0$$

---

$$S \xrightarrow{e} S', \text{ where } S' = S_{\{s_j := s'_j; w_{ij} := v.v'\}}$$

A message of type $m$ sent by a process $i$ can be received by an automaton $\mathcal{A}_j$ in a state $s_j$ if the reception of $m$ is an event that can be fired from state $s_j$,

and if $m$ is the first message in the buffer from $i$ to $j$ that can be read in sate $s_j$.

$$\text{-} \quad \exists j \in 1..K \ : \ e \in s_j \wedge e = \texttt{send(m) to i} \ \wedge \ w_{ji} = v \wedge (s_j, e, s'_j) \in \delta_j$$

$$S \xrightarrow{e} S', \text{ where } S' = S_{\{s_j := s'_j; w_{ji} := v.m\}}$$

$$\text{-} \quad \exists j \in 1..K \ : \ e \in s_j \wedge e = \texttt{action} \ \wedge \ (s_j, e, s'_j) \in \delta_j$$

$$S \xrightarrow{e} S', \text{ where } S' = S_{\{s_j := s'_j\}}$$

A message emission or an atomic action can be performed by an automaton $\mathcal{A}_j$ in a state $s_j$ if and only if this event is allowed in $s_j$.

This definition of communications allows us to generate automata for specification containing message crossing, which would not be possible with a strict FIFO ordering. Consider the simple bMSC of Figure 7. This specification contains a message crossing, which can however be implemented by the communicating finite state machines of Figure 8.

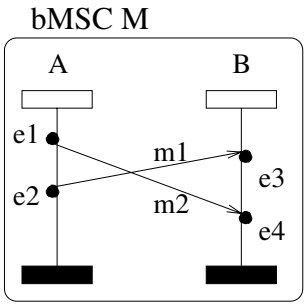



Fig. 7. A simple bMSC.

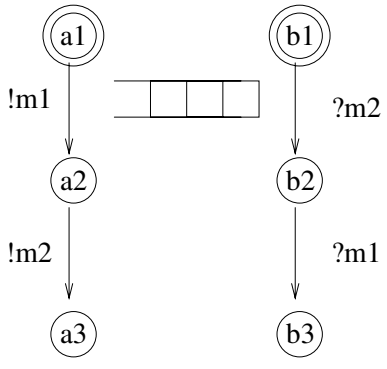



Fig. 8. A simple CFSM implementing bMSC $M$ of Figure 7.

Let us note $\mathcal{L}(\mathcal{A}_H)$ the language described by a set of communicating automata $\mathcal{A}_H$. A word $w = e_1.e_2..e_n \in E^*_{\mathcal{M}}$ is a word of $\mathcal{L}(\mathcal{A}_H)$ if and only if $S_0 \xrightarrow{e_1}$

$S_1 \xrightarrow{e_2} ... \xrightarrow{e_n} S_n$. We will write $S \xrightarrow{w} S'$ when a state $S'$ can be reached from state $S$ using $w$. We will denote by $s(w)_i$ the state reached by the automaton $\mathcal{A}_i$ after execution of $w$.

## 5 Conditions for language equality

A protocol $\mathcal{A}_H$ synthesized from a HMSC $H$ is considered as correct if $\mathcal{A}_H$ and $H$ define the same language.

### *5.1 Soundness*

The first step is to show that the synthesized CFSM is able to produce all the behaviours defined by the original HMSC.

**Theorem 8** *For any HMSC $H$, $\mathcal{L}(H) \subseteq \mathcal{L}(\mathcal{A}_H)$*

**proof:**

Proving $\mathcal{L}(H) \subseteq \mathcal{L}(\mathcal{A}_H)$ is equivalent to showing:

$$\forall w \in E^*_{\mathcal{M}}, w \in \mathcal{L}(H) \Longrightarrow w \in \mathcal{L}(\mathcal{A}_H)$$

This is also equivalent to proving the property:

$$P : \forall n \in I\!N, \forall w \in E^n_{\mathcal{M}}, w \in \mathcal{L}(H) \Longrightarrow w \in \mathcal{L}(\mathcal{A}_H)$$

Obviously, $P$ is true for $n = 0$. Let us show that $P$ true for $n$ implies $P$ true for $n+1$. Let us suppose $P(n)$ true, and $P(n+1)$ false. Then, it means that there is a word $w \in E^n_{\mathcal{M}}$, and an event $e \in E_{\mathcal{M}}$ such that: $w.e \in \mathcal{L}(H) \wedge w.e \notin \mathcal{L}(\mathcal{A}_H)$. Therefore $\exists p = M_1 \circ ... \circ M_k$, path of $H$ such that $w.e$ is a prefix of a linearization of $p$, and $\phi(e) = i \wedge (e \notin s(w)_i \vee (e = \text{rec(m) from } j \wedge w_{ij} \neq v.m.v')$

- $e \notin s(w)_i$ if and only if there is a predecessor $e'$ of event $e$ on instance $i$ and $e'$ has not been executed in $w$. Therefore, $w.e \notin \mathcal{L}(H)$, contradiction.
- $e = \text{rec(m) from } j \wedge w_{ij} \neq v.m.v'$ also leads to a contradiction, as $w.e \in \mathcal{L}(H)$ implies that any predecessor (and consequently an emission of $m$) appears in $w$.

Therefore, we have proved $\mathcal{L}(H) \subseteq \mathcal{L}(\mathcal{A}_H)$. □

Unfortunately, the synthesis method defined previously does not ensures that $\mathcal{L}(H) = \mathcal{L}(\mathcal{A}_H)$. Let us consider HMSC $H_6$ of Figure 9: the traces defined by $H_6$ are $!m_1.?m_1$ and $!m_2.?m_2$. The communicating finite state machines synthesized from $H_6$ (see Figure 10) also describe the traces $!m_1.!m_2$ and $!m_2!m_1$. HMSC $H_6$ defines an implicit synchronization between instances $A$ and $B$, that must agree on the scenario to perform. Clearly, this specification can not be implemented without an additionnal synchronisation mechanism. Such a situation is called non-local choice.

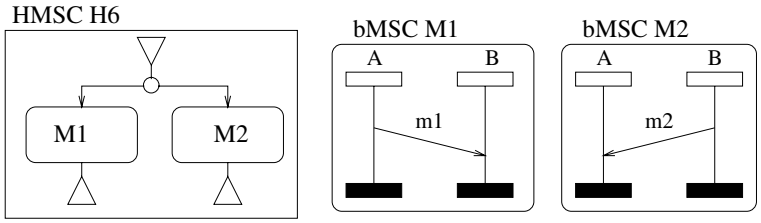



Fig. 9. A non-local HMSC.

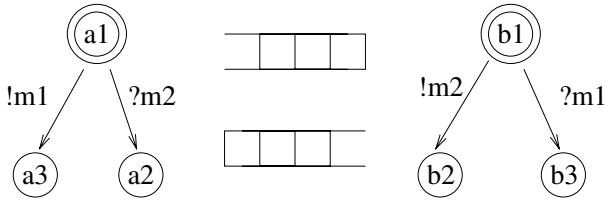



Fig. 10. CFSM synthesized from HMSC $H_6$ in Figure 9.

### *5.2 Non-local choice*

The generally admitted meaning of non-local choice (2) is when more than one instance can decide to perform a scenario or another at a choice node. The intended behaviour is that the first instance able to perform the choice chooses a behaviour. The next instances reaching the same iteration of this choice have to conform to the chosen scenario. This results in a behaviour in which an instance "knows" what to do at a choice node without any communication. However, non-local choices can not be implemented without adding communications to the system. In some particular cases, however, non-local choices may express concurrency between scenarios, as in example of Figure 14.

A definition of non-local choice was previously given in (2). This definition assumes that any instance should communicate with other instances on each branch of a choice. This assumption limits the search for non-local choice to the set of outgoing edges. However, when considering weak sequential composition of bMSCs with disjoint set of instances, non-local choice is not a local property. Therefore, a global definition of non-local choice must be provided. Consider HMSC $H_7$ in Figure 11: choices seem to be local, but the decision to perform a scenario can be taken by $A$ or $C$. The definition has thus to be extended.

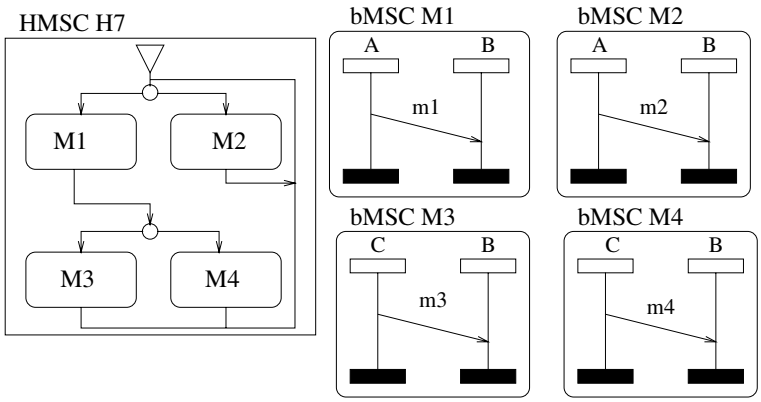



Fig. 11. Non-local choice located on more than one choice node.

**Definition 9** *Let $c$ be a choice node. $c$ is* local *if and only if:*

$\forall p_1 = c.n_1...n_k$, *maximal loop-free path of* $H$ $(c \notin n_1...n_k)$ *from* $c$, $\forall p_2 = c.n'_1...n'_k$ *maximal loop-free path of* $H$ *from* $c$, $\forall e_1 \in min(O_{p_1})$, $\forall e_2 \in min(O_{p_2}), \phi(e_1) = \phi(e_2)$.

For a local choice $c$, the instance deciding the behaviour at this point of the specification will be called the *deciding instance* of choice $c$. From the definition of non-local choice, the algorithm is straightforward. Note that locality can be checked on loop-free paths (adding a pattern that already appears in a path does not add minimal events).

**Algorithm:**

**for all** $c$,`choice node in` $H$ **do**
  $P = \{(c.n, I, J) | c \xrightarrow{M} n \wedge I = \phi(min(M)) \wedge J = \phi(M)\}$
  $MAP = \emptyset$ `/* Maximal acyclic paths */`
  **while** $P \neq \emptyset$ **do**

$$MAP = MAP \cup \{(w.n, I') | w = n_1...n_k \wedge n_k \xrightarrow{M} n \wedge n \in w \wedge (w, I, J) \in P \wedge I' = I \cup (\phi(min(M)) - J)\}$$

$$P = \{(w.n, I', J') | (w, I, J) \in P, w = n_1...n_k \wedge n_k \xrightarrow{M} n \wedge n \notin w \wedge J' = J \cup \phi(M)\ I' = I \cup (\phi(min(M)) - J)\}$$

  **end while**
  $DI = \bigcup_{(w,I) \in MAP} I$ `/* deciding instances */`
  **if** $|DI| > 1$ **then**
    $H$ `contains a non-local choice`
  **end if**
**end for**

As we only consider loop-free paths of HMSC $H$ this algorithm terminates. From now on, we will write $Loc(H)$ when a HMSC $H$ does not contain non-local choices.

**Theorem 10** $Loc(H) \not\Longrightarrow \mathcal{L}(\mathcal{A}_H) \subseteq \mathcal{L}(H)$

*Ensuring that a HMSC $H$ is local does not ensures that the synthesis produces a CFSM that is language equivalent to $H$*

The automata produced for HMSC $H_8$ of Figure 12 are described in Figure 13. One can easily check that
$w = !m1.?m1.!m2.!m3.?m3.!m4.?m4....$ in a trace of $\mathcal{A}_H$, but not of $H$ (in $H$, $m2$ must be received before $m4$ in that trace).

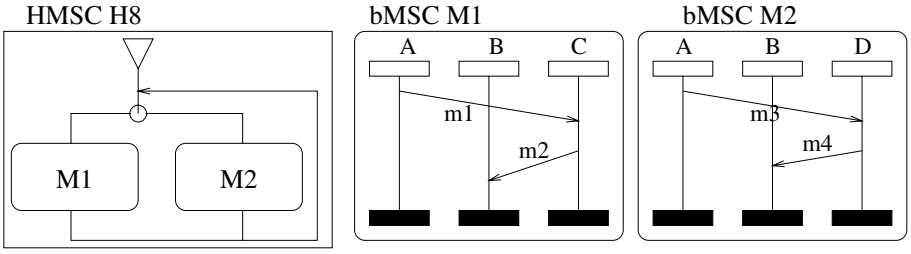



Fig. 12. HMSC $H_8$

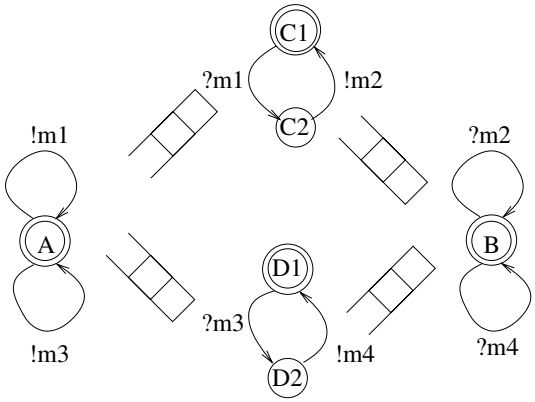



Fig. 13. CFSM synthesized from HMSC $H_8$ Figure 12

When synthesizing a CFSM system from a HMSC, the local ordering between events may be lost at choice nodes. Consequently, a specification may reach a state in which two messages $m$ and $m'$ can be received from different senders by the same automaton $\mathcal{A}_i$. In the HMSC specification, $m$ and $m'$ are messages from different bMSCs, and therefore their receptions are ordered (either `rec(m)` $\leq$ `rec(m')` or `rec(m')` $\leq$ `rec(m)`). If $\mathcal{A}_i$ do not have enough information to reconstruct the ordering between `rec(m)` and `rec(m')`, a wrong trace is allowed. This is what happens in the specification Figure 12: if bMSC $M_1$ is choosen before bMSC $M_2$, then $m_1$ is sent before $m_3$, and $m_2$ must be received before $m_4$. Unfortunately, $\mathcal{A}_B$ do not have enough information to prevent the reception of $m_4$.

**Theorem 11** $\neg Loc(H) \not\Longrightarrow \mathcal{L}(\mathcal{A}_H) \not\subseteq \mathcal{L}(H)$

*Even if $H$ is a non-local Message Sequence Charts, the CFSM implementation*

*synthesized from $H$ can be correct (consider HMSC $H_9$ in Figure 14).*

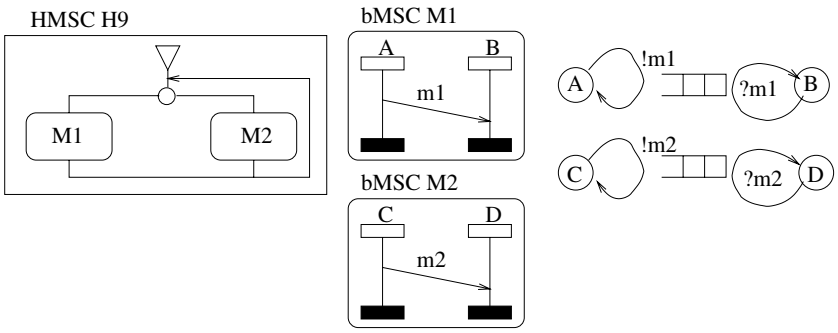



Fig. 14. Non-local HMSC with language equality.

### *5.3 Local sequencing reconstructibility*

The locality condition is not strong enough to ensure that the CFSM generation of Section 4 produces a set of automata that is trace equivalent to the HMSC specification.

**Definition 12** *A choice $c$ of a HMSC $H$ is said to be reconstructible if and only if:*

- *$c$ is a local choice, and*
- *$\forall p$ path of $H$ from $c$ such that $O_p = B_1 \circ B_2 \circ ... \circ B_n$ with $B_i, i \in 1..n$ branch of $c$ , $\forall x$ non-deciding instance, $\forall (e, e') \in p : e = min_x(B_i) \wedge e' = min_x(B_j) \wedge 1 \leq i < j \leq n$, $(e, e')$ is a reconstructible pair of $\leq_{O_p}$ from $\leq_{O_p} \setminus (E_{i_{|x}} \times E_{j_{|x}})$ by mt-closure.*

More intuitively, a choice is reconstructible if removing the local ordering due to bMSC sequencing on any non-deciding instance does not affect the mt-closed ordering.

**Definition 13** *A HMSC $H$ is said to be reconstructible if and only if any choice in $H$ is reconstructible. This property will be written $Rec(H)$.*

Imposing HMSC reconstructibility is weaker than requiring any message to be acknowledged. The example of Figure 15 shows a HMSC in which two messages are emitted by different instances and received on a single instance. However, the order between the receptions is preserved by the translation in Figure 16.

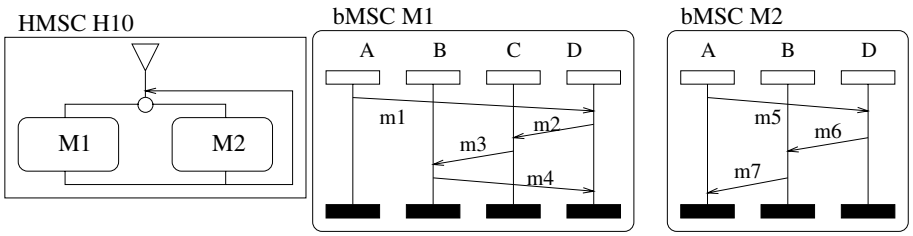



Fig. 15. Reconstructible HMSC $H_1 0$.

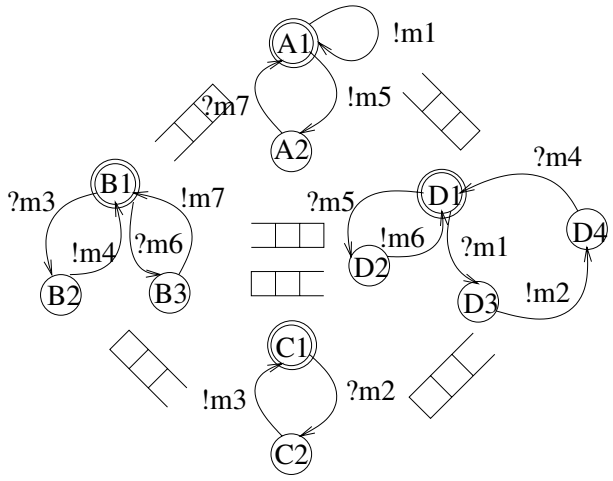



Fig. 16. CFSM synthesized from HMSC $H_1 0$ in Figure 15.

**Theorem 14** $Loc(H) \wedge \mathcal{L}(\mathcal{A}_H) \subseteq \mathcal{L}(H) \Longrightarrow Rec(H)$

**proof:**
Let us prove that $\mathcal{L}(\mathcal{A}_H) \subseteq \mathcal{L}(H) \wedge \neg Rec(H)$ leads to a contradiction.

If $Rec(H)$ does not hold, then there is a choice node $c$ in $H$ such that:
$B_1 =< E_1, \leq_1, A_1, I_1, \alpha_1 >$ and $B_2 =< E_2, \leq_2, A_2, I_2, \alpha_2 >$ are two branches of $c$, $\exists p$ path of $H$ from $c$ with $O_p = B_1 \circ B_2$. There exists an instance $x$, $\exists r_1 \in E_1$, $r_2 \in E_2$, receiving events such that: $r_1 = min_{\phi(r_1)}$, $r_2 = min_{\phi(r_2)}$, $r_1 =$rec$(m_1)$ from $i$, $r_2 =$ rec$(m_2)$ from $j$, $i \neq j$.

Let us call $O$ the order $O = (\leq_{O_p} -(E_{1_x} \times E_{2_x}))^{*mt}$. From the definition of $Rec(H)$, we have $(r_1, r_2) \notin O$. As $(r_1, r_2)$ can not be reconstructed, for any predecessor $e'$ of $r_2$ in $B_2$, and for any successor $e$ of $r_1$ in $B_1$, we have $(e, e') \notin O$.

According to the locality of $H$ there are two deciding events $d_1 \in E_1$, $d_2 \in E_2$ such that $d_1 < r_1$, $d_2 < r_2$ and $\phi(d_1) = \phi(d_2)$. From the structure of the HMSC $H$ and from the derivation method, we know that there is an automaton $\mathcal{A}_x \in \mathcal{A}_H$ containing a state $s_x$ such that $r_1$ and $r_2$ can be fired from $s_x$. So, from any global state $S$ containing $s_x$, $r_1$ and $r_2$ can be fired if $m_1$ and $m_2$ have been sent.

We know that there is an initial path $p_v = n_0....c$ in $H$ leading to $c$. So, any linearization $v$ of $O_{p_v}$ is a word of $\mathcal{L}(H)$. As $\mathcal{L}(H) \subseteq \mathcal{L}(\mathcal{A}_H)$, we also have $v \in \mathcal{L}(\mathcal{A}_H)$. Consequently, $\exists S$ such that $S_0 \xrightarrow{v} S$. As we know that $r_1$ and $r_2$ are minimal events, we have $s_x \in S$.

So, we can find a word $w = v.d_1.u.em(r_1).d_2.u'.em(r_2)$ where $u$ contains any predecessor of $d_2$ in $O_p$ in $\mathcal{L}(H)$. We also know that $w \in \mathcal{L}(H)$, therefore $S_0 \xrightarrow{w} S'$ and as $r_1$ have not been fired, $s_x \in S'$. As $m_1$ and $m_2$ have been sent, and as $\phi(em(r_1)) \neq \phi(em(r_2))$, $w.r_2$ is a word of $\mathcal{L}(\mathcal{A}_H)$. $w.r_2$ is not a word of $\mathcal{L}(H)$, as $d_1 < d_2$ implies $r_1 < r_2$ in any word of $\mathcal{L}(H)$. This contradicts $\mathcal{L}(\mathcal{A}_H) \subseteq \mathcal{L}(H)$. □

**Theorem 15** $Rec(H) \Longrightarrow \mathcal{L}(\mathcal{A}_H) \subseteq \mathcal{L}(H)$

**proof:**
Let us suppose that $Rec(H) \wedge \mathcal{L}(\mathcal{A}_H) \not\subseteq \mathcal{L}(H)$, and let us show that it leads to a contradiction.

If $\mathcal{L}(\mathcal{A}_H) \not\subseteq \mathcal{L}(H)$, then there exists a word $w$ such that $w \in \mathcal{L}(\mathcal{A}_H) \wedge w \notin \mathcal{L}(H)$. From the construction method, we know that $w \neq \epsilon$, so $w$ is of the form $w = v.e$, and such that $v \in \mathcal{L}(\mathcal{A}_H) \wedge v \in \mathcal{L}(H)$. As $v \in \mathcal{L}(H)$, then there exists a path $p \in H$ such that $v$ is a prefix of a linearization of $O_p$.

$v.e \notin \mathcal{L}(H)$ may hold for two reasons:

- i) $\forall p' = p.n_{k+1}....n_{k+n}$, $e \notin E_{O_{p'}}$. Therefore, $v.e \notin \mathcal{L}(\mathcal{A}_H)$. Contradiction.
- ii) $\forall p' = p.n_{k+1}....n_{k+n}$, there exists a sequence of events $e_1.e_2...e_k$ such that $\forall i \in 1..k, e_i \leq_{O_{p'}} e$, and $v.e_1...e_k.e \in \mathcal{L}(H)$.

As $v.e \in \mathcal{L}(\mathcal{A}_H)$, there exists a local state $s_x$ such that $S_0 \xrightarrow{v} S \wedge s_x \in S \wedge (s_x, e, s'_x) \in \delta_x$. As $\mathcal{L}(H) \subseteq \mathcal{L}(\mathcal{A}_H)$, $v.e_1...e_k.e$ is also a word of $\mathcal{L}(\mathcal{A}_H)$. Three cases may appear: $e$ is a sending event, an internal action, or a receiving event.

- Suppose $e$ is a sending event, or an internal action. Then there exists an event $e_j, j \in 1..k$ such that $\phi(e_j) = \phi(e) \wedge \forall i < j, \phi(e_i) \neq \phi(e)$. Therefore, $\exists (s_x, e_j, s''_x) \in \delta_x$. $e$ and $e_j$ are minimal event on instance $x = \phi(e)$ for branches of a choice $c$. According to the locality property, $e_j$ and $e$ are deciding events for their respective branch. So, $O_{p'}$ is of the form $O_{p'} = M_1 \circ ... \circ M_l \circ B_1 \circ B_2$, where $B_1$ contains $e_j$ and $B_2$ contains $e$. Therefore, $\exists p''$ path of $H$, such that $O_{p''} = M_1 \circ ... \circ M_l \circ B_2$. So $v.e \in \mathcal{L}(H)$. Contradiction.
- Suppose $e$ is a receiving event. As $v.e \in \mathcal{L}(\mathcal{A}_H)$, any emission needed for executing $e$ is performed in $v$. Again, $v.e \notin \mathcal{L}(H)$ implies that there is an event $e_j, j \in 1..k$ such that $\phi(e_j) = \phi(e) \wedge \forall i < j, \phi(e_i) \neq \phi(e)$, and $\exists (s_x, e_j, s''_x) \in \delta_x$. Still considering the locality of $H$, $e_j$ is also a receiving event, and there is a choice between two branches $B_1$ and $B_2$,

$e_j = min_x(B_1)$, and $e = min(B_2)$. If $\phi(em(e)) = \phi(em(e_j))$, then according to the semantics of reception, $e$ cannot be executed before $e_j$ by $\mathcal{A}_H$. So, $\phi(em(e)) = \phi(em(e_j))$. Of course, $\phi(e)$ is not a deciding instance. So there is a choice $c$ between $B_1$ and $B_2$, and a path $B_1 \circ B_2$, such that $(e_j, e)$ can not be reconstructed. Contradiction with $Rec(H)$.

□

From theorem 14 and theorem 15, the following property holds true:

$$Loc(H) \Longrightarrow \Big(Rec(H) \Longleftrightarrow \mathcal{L}(\mathcal{A}_H) \subseteq \mathcal{L}(H)\Big)$$

Using theorem 8, this property becomes:

$$Loc(H) \Longrightarrow \Big(Rec(H) \Longleftrightarrow \mathcal{L}(\mathcal{A}_H) = \mathcal{L}(H)\Big)$$

Note that the proposition $Rec(H) \Longleftrightarrow \mathcal{L}(\mathcal{A}_H) = \mathcal{L}(H)$ is false (due to theorem 11). Example of Figure 14 exhibits a non-local specification where $\mathcal{L}(\mathcal{A}_H) = \mathcal{L}(H)$. Such a kind of specification contains hidden parallelism within choices (instances $A, B$ and $C, D$ never synchronize).

### *5.4 Decision of reconstructibility*

This section provides an algorithm for deciding the reconstructibility of a HMSC $H$. The reconstructibility property can be decided on prefix of paths originating from a choice.

**Proposition 16** *$\forall c$, choice of $H$, $c$ is reconstructible if for any pair of branch $B_i$, $B_j$ such that $\exists p$, path from $c$ and $O_p = B_i \circ B_j$, for any non-deciding instance $x$, $(min_x(B_i), min_x(B_j))$ is reconstructible from $\leq_{B_i \circ B_j} -(E_{i_{|x}} \times E_{j_{|x}})$ by mt-closure.*

**Algorithm:**

**for all** $c$, `choice node of` $H$ **do**
  $P = \{(c.n, M) | c \xrightarrow{M} n\}$
  $C = \emptyset$ `/* Cycles */`
  $MAP = \emptyset$ `/* Maximal loop-free paths */`
  **while** $P \neq \emptyset$ **do**
    $C = C \cup \{M \circ M' | (w = c.n_1..n_k, M) \in P \wedge n_k \xrightarrow{M'} c\}$
    $MAP = MAP \cup \{M \circ M' | (w = c.n_1..n_k, M) \in P \wedge n_k \xrightarrow{M'} n \wedge n \in w\}$
    $P = \{(w.n, M \circ M') | (w = c.n_1..n_k, M) \in P \wedge n_k \xrightarrow{M'} n \wedge n \notin w\}$
  **end while**
  **for all** $(B_i, B_j) \in C \times (MAP \cup C)$ **do**
    **if** $\exists x \in I | (min_x(B_i), min_x(B_j)) \notin (B_i \circ B_j - (B_{i|x} \times B_{j|x}))^{*mt}$ **then**
      Order can not be reconstructed
    **end if**
  **end for**
**end for**

## 6 Conclusion

This paper proposed a formal definition of non-local choice for HMSCs with weak sequential composition. Non-local choices can be detected on loop-free (and therefore finite) paths of the HMSC. Then a procedure for synthesizing CFSMs was proposed. The absence of non-local choices in a HMSC is not a sufficient condition for ensuring the synthesis produces a correct protocol: order reconstructibility is also required. Reconstructibility of an HMSC is also decidable on loop-free paths, and could be implemented as a front-end for protocol synthesis tools.

When a specification is non-reconstructible, some acknowledgment messages can be added to transform it into a reconstructible one. A possible extension of this work would be to consider an automation of these transformations. Adding stamps to messages when necessary could also avoid executing non-specified traces. Considering stamps as well as introducing co-regions (a weaker ordering on instances than total ordering) would probably need to modify the CFSM target model.

## 7 Acknowledgments

The authors are very grateful to professor Ferhat Khendek at Concordia University, for helpful discussions on formal aspects of synthesis from MSCs, and

on his tool MSC2SDL. We are also very grateful to Benoît Caillaud for his comments on this work and suggestions for future extensions.

## References

[1] M.Abdalla, F.Khendek,G.Butler, New Results on Deriving SDL Specifications from MSCs, in Proceedings of 9th SDL forum, R.Dssouli, G.Bochman & Y.Lahav editors, pp 51-66.

[2] H. Ben-Abdallah and S. Leue, Syntactic Detection of Process Divergence and non-Local Choice in Message Sequence Charts, in: E. Brinksma (ed.), Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems TACAS'97, Enschede, The Netherlands, April 1997, Lecture Notes in Computer Science, Volume 1217, p. 259 - 274 Springer-Verlag, 1997

[3] D.Brand, P.Zafiropulo, On communicating Finite-State Machines. Journal of the ACM, vol 30, No 2, April 1983, pp 323-342.

[4] A. Engels, S. Mauw, M.A. Reniers: A Hierarchy of Communication Models for Message Sequence Charts. In: T. Mizuno, N. Shiratori, T. Higashino and A. Togashi (ed.): Formal Description Tecniques and Protocol Specification, Testing and Verification, Proceedings of FORTE X and PSTV XVII '97, pages 75-90, Osaka, Japan, 18-21 November 1997. Chapman & Hall.

[5] International Standard ISO 9074, Information Processing Systems - Open Systems Interconnection - Estelle: A Formal Description Technique Based on an Extended State Transition Model, Geneve, 1989.

[6] S.Leue, L.Mehrmann, M.Rezai, Synthesizing ROOM Models from Message Sequence Chart Specifications, 13th IEEE Conference on Automated Software Engineering, Honolulu, Hawaii, October 1998.

[7] N.Mansurov, D.Zhukov, Automatic Synthesis of SDL models in use case Methodology, in Proceedings of 9th SDL forum, R.Dssouli, G.Bochman & Y.Lahav editors, pp 225-240.

[8] OMG, Unified Modelling Language 1.1, September 1997.

[9] V.Pratt, Modeling Concurrency with Partial Orders, International Journal of Parallel Programming, Vol 15, No 1, 1986, pp 33-71

[10] K.Yamanaka, S.Komura, J.Kato, H.Ichikawa, Deriving Protocols from Message Sequence Charts in a Communicating Processes Model.

[11] TU-TS Recomendation Z.120: Message Sequence Chart 1996 (MSC96) Technical Report, ITU-TS, Geneva, 1996.

# A Practical Method to Integrate Abstractions into SDL and MSC based Tools [1]

Maria-del-Mar Gallardo and Pedro Merino

*Dpto. de Lenguajes y Ciencias de la Computacion, University of Malaga, 29071 Malaga, Spain*

**Abstract**

Many industrial oriented tools that employ SDL as the basis to develop complex systems support very detailed specifications in order to perform tasks such as simulation, code generation or testing. But the size of the SDL model constructed for these purposes makes the verification of *MSC*s more resource consuming, due to the well-known problem of the state space explosion. This paper presents a proposal in the direction of optimising the verification of *MSC*s without limiting the use of SDL in the other tasks. The main contribution of the paper is the definition of a practical automatable method to obtain intermediate SDL specifications suitable for the efficient verification of MSCs. The correctness of the transformation is supported by the definition of a semantics framework that allows us to compare different SDL models of the same system, each one with a particular abstraction level.

*Key words:* automatic verification; abstract interpretation; SDL

## 1 Introduction

Automatic verification based on formal methods is becoming the most widespread technique to increase confidence in the correctness of critical systems [18] [16] [4]. However, in many projects, formal specifications are employed as very detailed descriptions for simulation, automatic code generation or simple documentation, and these uses are not compatible with high quality verification. The reason is that automatic verification is only fruitful when the model is an abstract representation of the real system, with exactly the details necessary to ensure that satisfaction of interesting properties in the model implies satisfaction in the real system [19]. Nevertheless, many companies consider that

[1] Supported by CICYT TIC99-1083-C02-01, Spain

the cost of creating verification-oriented models may not be compensated by potential improvement in the quality of their products and the verification is usually poorer than desired. This problem is also a challenge in current industrial oriented CASE tools like SDT[26] or Object Geode [27], which employ the international standard formal methods SDL [20] and MSC [21] for specification and analysis of correctness in the earlier phases of system development.

The problem that we address in this paper is how to improve the quality of the verification of SDL models against properties described with MSC without interfering with the use of these languages for the other development phases. We describe an automatable approach for transforming SDL code in order to obtain more abstract models to be verified using less time and memory. This transformation method can be integrated into the tools that support verification and the other phases of development.

*Related Work*

Our method is based on the Abstract Interpretation technique (AI) [2], which has been mainly employed to improve the verification of systems against properties described with temporal logic ([3] [6] [23]). AI is based on the idea of approximation: every program data is approximated by a higher level description by means of an abstraction function. When combined with verification, a given abstraction function is employed to transform a model into a new (abstract) one over which properties are analyzed. This approach was employed in [3] and [6] for the verification of temporal properties expressed with CTL and $\mu$-calculus, respectively. Recently, Graf and Saïdi used the theorem prover PVS for the automatic construction of the abstract model [10]. This work has been extended to obtain a textual specification, which can be again abstracted [1], [25] [24].

Dwyer and his colleagues use an Abstraction Library with already designed abstraction functions in order to allow the translation of high level programming languages (such as ADA or JAVA) to the input language to model checker [7] [8]. This idea is also presented in [13].

*Contributions*

Compared with these related works, our proposal has the following main characteristics:

(1) The method is applied to verify properties represented as *Messages Sequence Charts* (*MSC*s), instead of temporal logic properties. The use of this formalism makes easier the integration of abstraction in some kinds of industrial applications, such as telecommunications, because the requirements are usually represented in this manner.

(2) The abstraction produces a new textual (and graphical) SDL model, so it can be analysed with the same tools than the original one. Furthermore, new transformations can be applied to the abstract version.
(3) We consider two kinds of users with different knowledge about the abstraction techniques. The non-expert user takes the abstraction functions from the Abstraction Library to automatically transform the model. The expert designer defines these abstraction functions for different application domains.

The paper is devoted to presenting the overall methodology to use abstraction into SDT and the theoretical background to ensure property preservation [2]. The correctness is analyzed by defining the so-called generalized semantics of SDL, which puts stress on the operational aspects of the language which are not affected when abstracting data or operations (this idea was used in [22], [15] and [5]). The rest of SDL characteristics are parameters of the semantics in such a way that if we change them (by means of an appropriate abstraction) the high level behaviour of the system remains unchanged. This approach allows us to relate different SDL models of a given system, each one with a particular level of detail, and its has been proved useful to reason about correctness conditions for transformation [13]. Other existing semantics of SDL are more well sutiable for describing the language and for implementing tools, but they are too details to be used for reasoning in our context.

The organisation of the paper is as follows. In Section 2 we describe the verification capabilities of SDT and the subsets of SDL and MSC employed in the paper are described. Sections 3 to 5 describe the practical aspects of the paper. Section 3 presents the methodology to extend SDT with abstraction capabilities. Section 4 contains the kinds of abstractions that we consider, and Section 5 shows an example of how to employ the Abstraction Library to reduce the state space of a system. Sections 6 to 8 describe the formal framework to ensure the correctness of the transformations. Section 6 describes a generalized semantics of SDL to reason about property preserving transformations of SDL. Section 7 and Section 8 give the conditions to preserve correction and the correctness results, respectively. In Section 9, we present conclusions and future work.

## 2 Preliminaries

Although some knowledge of SDL and MSC is assumed, we now describe the subsets of SDL and MSC considered in the paper and give an example that is employed in the following sections. This section also explains the verification

[2] Note that the same method can be integrated in similar tools like Object Geode

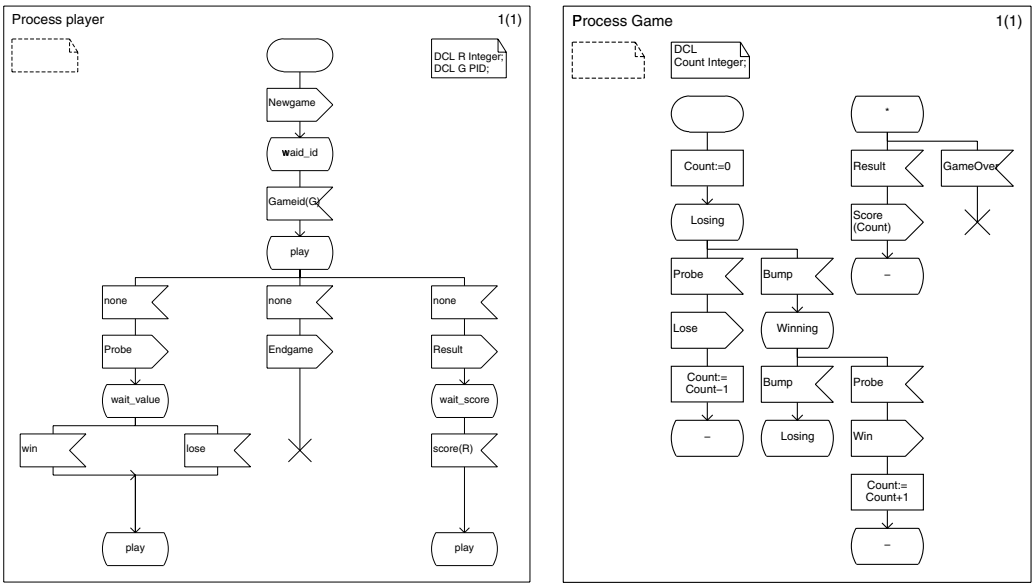



Fig. 1. Processes Player and Game

capabilities of SDT.

### *2.1 Subsets of SDL and MSC*

The formal languages SDL and MSC are in continuous evolution with new standards from the International Telecommunication Union (ITU) periodically. The last versions incorporate structural language constructs, essentially composition and object oriented concepts. Although the results in the paper can be extended to cover full SDL-92 as well as High Level MSC, we will concentrate in what is called Basic SDL in [20] and Plain MSC in SDT documentation. Both languages have textual and graphical forms. As these representations are equivalent, it is usual to present the specifications in the graphical form and reserve the textual form for automatic processing.

The subset of SDL includes SDL processes, states, start and stop symbols, input and output of signals with data, tasks, decisions, save symbols, and timer mechanisms. Nondeterministic behaviour modeled by spontaneous inputs and nondeterministic decisions is also considered.

**Example 1** *This subset of SDL is enough to model examples such as the usual* `DeamonGame` *system, which is a de facto standard for SDL tutorials and papers (partially given in Fig. 1). The process* `Game` *has an internal status, which can have either state* `Winning` *or* `Losing`*. The status is changed every now and then by a process* `Deamon`*. A Player has to guess when the status of the system is* `Winning`*. The system status is probed issuing the signal* `Probe`*. If the status during probing is* `Winning`*, you gain one point, otherwise one point is lost.*

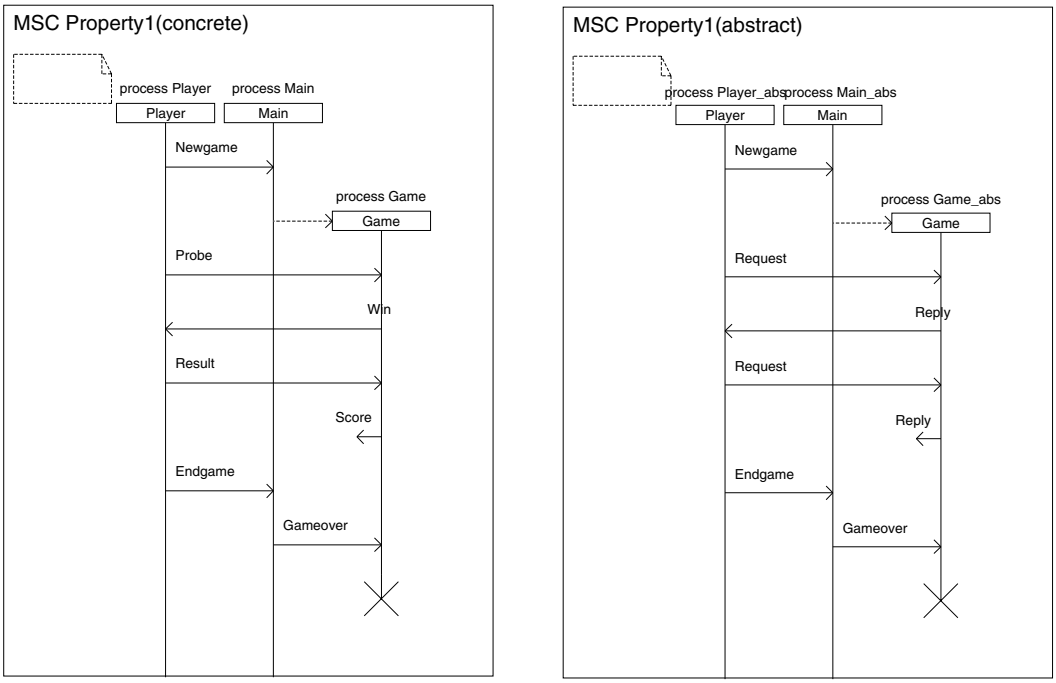



Fig. 2. a) *MSC* for DeamonGame b) *MSC* for abstract DeamonGame

*The current score can be asked for using the* `Result` *signal. Our version of the system contains a non-deterministic process* `Player` *that makes any number of guesses before sending the* `Endgame` *signal.*

MSC is usually employed to represent requirements as the message interchange between communicating entities and their environment. When employed with SDL, those messages would coincide with the signals, which are sent (consumed) from (by) any part of the specification (an SDL system, a block or a process). An *MSC* is a particular diagram (or text specification) which represents a particular scenario related to an SDL system. Fig. 2 a) shows an *MSC* with a message interchange in the system described above.

### *2.2 Verification in SDT*

The role of SDL in SDT is to be the kernel language to design the software, while the MSC language is employed to represent sequences of *Observable events*, which are only a small subset of the actions in SDL (for example send signal and consume signal). Thus the sequence $MSC = ev_1 \rightarrow \ldots \rightarrow ev_k$ represents a user requirement. The verification of MSC against SDL is the most important kind of analysis in this tool. Fig. 3 represents the role of both languages in the analysis of the systems. As in other similar tools, the simulation can produce *MSC*s that can be employed as requirements for verification. The user can also construct/modify the *MSC*s by using the graphical editor. The

verification can be performed with two different objectives:

(1) Check that no execution path in the SDL system produces a sequence of events *matching* the $MSC$ (the SDL model does not verify the $MSC$). Otherwise, the validator produces a trace with the erroneous behaviour.
(2) Check that the SDL model can exhibit at least one execution sequence producing the scenario represented by the $MSC$ (the SDL model verifies the $MSC$). Again, the validator shows the traces that match the $MSC$.

The *matching* between a path in SDL and the scenario in the $MSC$ holds iff all observable events produced by the entities in the $MSC$ occur in the SDL path in the same order. The events related to other entities (blocks, processes) are ignored. The starting point of the execution sequence in the SDL system can be an arbitrary state, but it must be the same than in the $MSC$ diagram. The meaning of each kind of verification depends on the user purposes. The first one is especially powerful for locating and removing undesired scenarios in the behaviour of the system, while the second one is employed in order to know if the system can respond in a particular way.

Apart from the verification of $MSC$s, the validator can be also employed to check other important properties such as absence of deadlock, invariants, and errors due to undesirable behaviours of the SDL constructors (invalid receiver for a signal, range errors, operators errors, etc.) In order to deal with huge state spaces, the validator can use bit state and random walk (see [18]).

## 3 Extending SDT with Abstractions

Our approach to extend SDT with abstractions consists in transforming the SDL specification into intermediate versions, which are only employed for verification purposes, as it is shown in Fig. 4 with dark ground. Given a detailed SDL model $M$, we try to construct and verify abstract models $M_i^*$ until $M$ is shown to be correct or until specific errors are found. The verification consists in checking non-satisfaction of $MSC$, but also absence of deadlock can be checked in the abstract model.

The construction of an abstract model $M_i^*$ is automatic, and the user only has to choose the abstraction function $\alpha$ suitable for the kind of system and for the specific $MSC$.

Given the initial model $M$ and the $MSC$ representing the undesirable scenario, the first step is to obtain the abstract model $M_i^*$ by using $\alpha$. The abstract version of the $MSC$ ($MSC_i^*$) is obtained by a similar mapping pro-

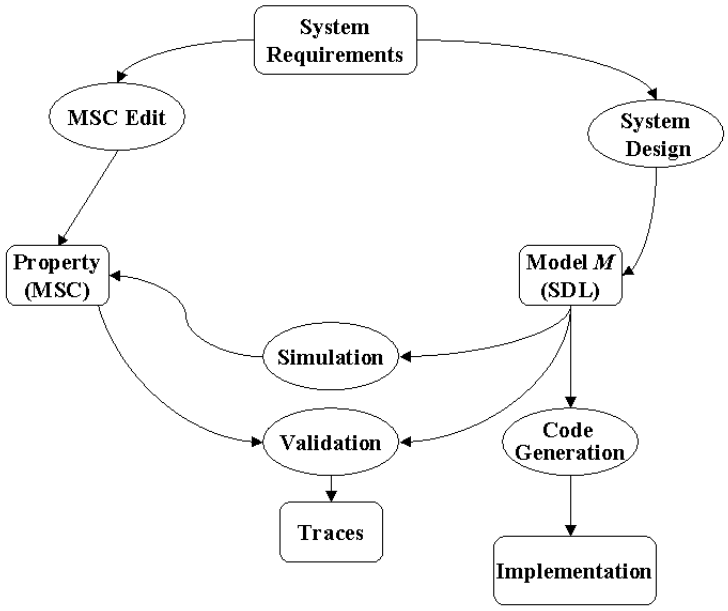



Fig. 3. The role of SDL and MSC in SDT

cess, but many times can be easily obtained manually. Then, we employ the standard SDT validator and simulator in the following way:

(1) If $M_i^*$ does not verify $MSC_i^*$ then, by using the property preserving results discussed in Sect. 5, $M$ does not verify $MSC$. Therefore $M$ can be employed to follow on with the development cycle in SDT (e.g., code generation).
(2) If errors are found in an abstract model, we employ the abstract trace produced by the validator to construct a concrete counterexample for $M$, and use the simulator to check if this counterexample is possible in $M$. If $M$ shows the error, then we must modify $M$ and start again with the verification. Otherwise, we refine the abstraction function and produce a new abstract model (and if necessary, a new abstract $MSC$) preserving more information.

The essential point is to choose a function $\alpha$ that preserves enough relevant information to decide whether the verification results for $M_i^*$ can be extended to $M$. As in other related works, it is necessary that the user posses a sufficient knowledge of the system model in order to choose the proper abstraction function. The provision of an Abstraction Library to automatically construct correct abstract models can solve most of the difficulties, thus giving an important added value to the whole development tool. This library contains transformation rules to implement different abstractions, and it is constructed

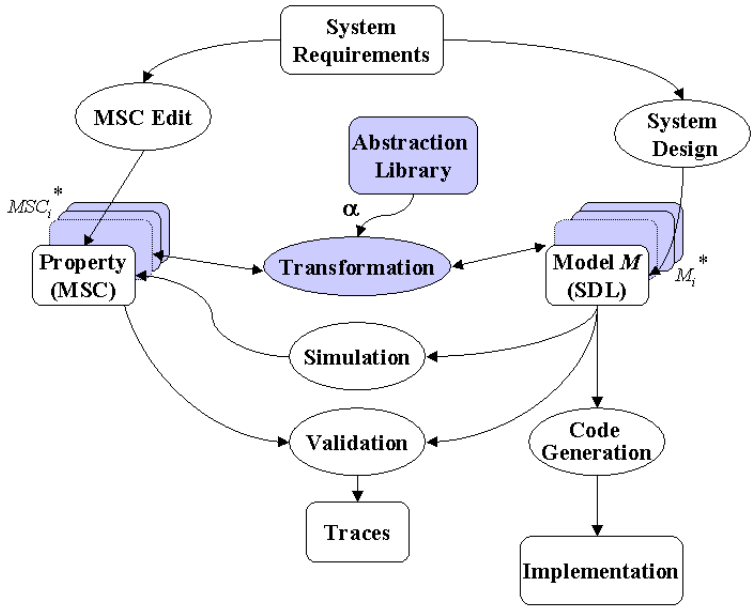



Fig. 4. Overview of extended SDT

depending on the application domain. The kind of information and it use is discussed in the following sections.

**Example 2** *As an example to justify this approach, let us consider the following property over the DeamonGame system: "In all paths, it is impossible for the* `Player` *to leave the system without consuming all the response signals from the process* `Game`*." This property must be encoded with several MSCs in order to consider all the possible kinds of requests from the* `Player` *and the responses from the* `Game`*. Even if we only consider one successful request-reply followed by one non-completed request-reply, as in Fig. 2 a), the number of MSCs to be considered is large. The verification of each one of these MSCs makes the validator run out of memory due to the presence of the counters* `Count` *and* `R` *in the model. So we need some kind of abstraction to solve the problem.*

## 4 Abstracting data and signals

In this section we explain the approach to obtain abstract SDL models. The process starts by abstracting some data and signals. Then, as an effect of this abstraction, all SDL instructions that work with this signals and variables

have to be modified. The modification consists in replacing every instruction with its abstract version. More formally, the method is described as follows.

Let $M = P_1||\ldots||P_n$ be an SDL system involving the concurrent execution of $n$ processes.

- Let us suppose that $P_j$ contains $s_j$ local variables $v_1, \ldots, v_{s_j}$, each one ranging over a (non-empty) set of values $D_{j_i}$. Let us define $D_j$ and $D$ as $D_j = D_{j_1} \times \ldots \times D_{j_{s_j}}$ and $D = D_1 \times \ldots \times D_n$.
- Let $SG$ be the set of signals defined in M.
- Let $Q_j$ be the domain of all possible values that $q_j$ may store. Let $Q$ be $Q_1 \times \ldots \times Q_n$. If $SG^i = \underbrace{SG \times \ldots \times SG}_{i}$ then $Q_j \subseteq \bigcup_{i \geq 0}(SG^i)$.
- Let $Inst$ be $Inst_1 \times \ldots \times Inst_n$, $Inst_j$ being the set of instructions of $P_j$.

Using these definitions, the set of system states is $SState = D \times Q \times Inst$. Let $\alpha = \alpha_d \times \alpha_q : D \times Q \rightarrow D^* \times Q^*$ be an abstraction function which transforms each concrete state (in which temporally instructions are not being taken into account) into an abstracted one. We assume that $(D^*, \leq_d^*)$ and $(Q^*, \leq_q^*)$ are posets where, as is classic in AI, partial orders represent the relative precision of the approximation of every abstract data. Sometimes, for clarity in the exposition, we will use $\alpha_d$ and $\alpha_q$ over simple variables, signals and queues instead of tuples.

Given $\alpha_d$ and $\alpha_q$, we define an approximation of the instructions $\alpha_{inst}$ as the function which transforms every concrete instruction into an abstract one. $\alpha_{inst}$ renames the original instruction and changes data and signals for the corresponding abstract ones using $\alpha$. $\alpha_{inst}$ is the key to construct the abstract model. At this point $\alpha_{inst}$ is only a renaming function with new data. But, we are interested in obtaining $M^*$ as an SDL model, so we need an SDL implementation of every $\alpha_{inst}$. In the following, $\alpha_{inst}$ will denote this implementation, and it is aasumend to be executed as an atomic instruction in order to preserve the necesarry correction conditions.

Let $Inst^*$ be $Inst_1^* \times \ldots \times Inst_n^*$, each $Inst_j^*$ being the set of abstract instructions of $P_j$ and let $SState^*$ be $D^* \times Q^* \times Inst^*$, the set of system states.

Finally, let us define $M^*$ as the abstract system obtained by substituting each instruction $i$ of $P_j$ by $\alpha_{inst}(i)$.

**Example 3** *We can attack the problem of the state-space explosion in example 2 by abstracting some signals between* `Player` *and* `Game` *and the local variables in these processes as follows:*

$\alpha_q$ *(*`Probe`*)* = `Request` $\alpha_q$*(*`Result`*)* = `Request`
$\alpha_q$ *(*`Win`*)* = `Reply` $\alpha_q$ *(*`Lose`*)* = `Reply` $\alpha_q$ *(*`Score`*)* = `Reply`

$\alpha_d$ (`Player.R`) = `0` $\alpha_d$ (`Game.Count`) = `0`

*This abstraction of data must be followed by the transformation of the instructions that manipulate these data. The next section shows how to use the Abstraction Library to obtain $M^*$.*

## 5 Using the Abstraction Library

To be rigorous, abstraction is carried out from an abstraction function $\alpha$ which transforms values and instructions over actual data and signals into abstract ones, as defined in the previous section. However in practice, the function $\alpha$ is not directly chosen by the user but it is the result of selecting a set of abstraction functions from the Library. Each single abstraction function defines the SDL code to implement specific abstractions of variables or signals. For example, the abstraction of signal (`S1`) with (`S2`) is defined in the Library with the function `AbstractSignal` as follows:

```
AbstractSignal_Output((S1(*),S2) ) = output  S2
AbstractSignal_Input(S1(*),S2) ) = input  S2
AbstractSignal_Save(S1,S2) ) = save  S2
```

where the string appended to the name of the function is the original SDL instruction (employed for automatic processing) and the returned value is the abstract version of the instruction. The `*` symbol represents the parameters in `S1`, which are ignored by this abstraction. Following the same notation, some of the operations for the abstraction VariableToConstant that abstracts an integer variable (`V`) by a constant (`C`) can be defined as

```
VariableToConstant_Add(V,C,E)) = TaskV := C
VariableToConstant_Sub((V,C,E)) = TaskV := C
```

The definition of both abstract instructions ignore the expression to be added or subtracted (`E`). Note that all possible kinds of sentences/operations over the variable should be define in the Library in order to be usable for transforming any SDL model. It is particularly interesting to define abstract versions of the boolean expressions that preserves all the branches of the initial SDL model (for example to use decision or provided). See [7] for samples of other abstract functions with this features.

To use the Library, the user must select the abstractions (the global $\alpha$)

to be applied to the SDL model by using

```
apply(
      AbstractSignal(probe,request)
      AbstractSignal(result,request)
      AbstractSignal(win,reply)
      AbstractSignal(lose,reply)
      AbstractSignal(score,reply)
      VariableToConstant(Player.R,0)
      VariableToConstant(Game.Count,0)       )
```

Then the MSC and the SDL code are automatically obtained. The transformation of a Plain MSC can be easily performed by renaming the Signals in the Input and Output events, as shown in Fig. 2b. The variable abstraction in the SDL model produces $DeamonGame^*$, and the abstraction of signals in $DeamonGame^*$ produces $DeamonGame^{**}$. The code for $DeamonGame^{**}$ (see Fig. 5) is obtained replacing the current instructions with the abstract versions as shown in Table 1.

Both abstract models are already suitable to perform a more efficient verification.Absence of deadlock is proved by inspecting 706 states in $DeamonGame^*$ and 479 states in $DeamonGame^{**}$, respectively. The state-space of the initial model (with Symbol-Sequence as the transition mode) has more than 900,000 states (see Table 1), and the verification of $DeamonGame$ leaves the tool out of memory after the 900,000 states.

The non-verification of the $MSC$ described in Fig. 2a is proved by inspecting 1139 states in $DeamonGame^*$. But, if we have problems with memory then we can use the abstract version $MSC^*$ (Fig. 2b) to prove $M \not\models MSC$ by inspecting only 583 states in $DeamonGame^{**}$. It is important to note that there are several $MSC$s that produce the same abstract diagram $MSC^*$. For this particular property, the verification over the original model must consider a set of concrete diagrams, and not only the one in Fig. 2. So, we have also saved the user from having to construct such diagrams. Table 2 summarises the verification results obtained with the three SDL models when using a maximum depths of 100 and 1000 to truncate the execution paths. The number of $MSCviolations$ represent SDL paths where observable events occurs in steps not matching with the $MSC$. As it was expected this number decreases when the model is more abstract.

These results confirm the advantages of the abstraction method, but we still need to study whether the verification of the abstract SDL model produces useful information about the initial model.This topic is discussed in

Table 1
Replacement of instructions

| Current SDL instruction | Abstract instruction |
|---|---|
| output probe | output request |
| output result | output request |
| input probe | input request |
| input result | output request |
| output win | output result |
| output lose | output result |
| input win | input result |
| input lose | output result |
| Task Count := Count + 1 | Task Count := 0 |
| Task Count := Count - 1 | Task Count := 0 |
| Task R := R + 1 | Task R := 0 |
| Task R := R - 1 | Task R := 0 |

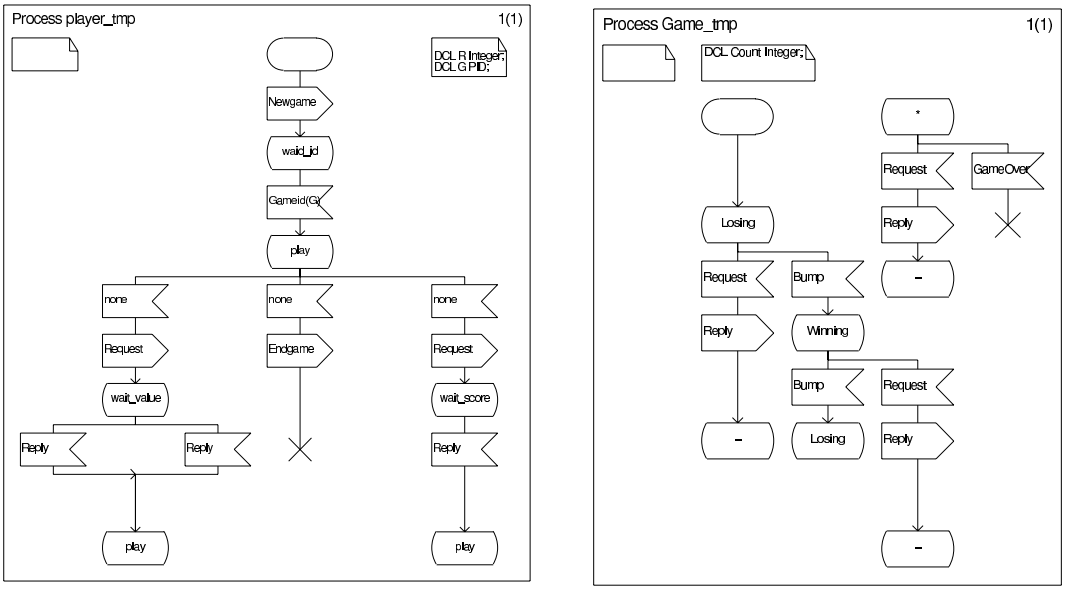



Fig. 5. Abstract version of processes Player and Game

the following sections.

## 6 A Generalized Semantics of SDL

As explained in the Introduction, the generalized semantics describes the operational behavior of a program, making explicit the domain-dependent model

Table 2
Verification results

| Model | States to discard deadlock | States to discard $MSC$ | $MSC$ Violations |
|---|---|---|---|
| DeamonGame (depth 100) | 52998 (truncated) | 53135 (truncated) | 48 |
| DeamonGame* (depth 100) | 709 | 1139 | 19 |
| DeamonGame** (depth 100) | 479 | 583 | 14 |
| DeamonGame (depth 1000) | > 900.000 (truncated) | > 900.000 (truncated) | — |
| DeamonGame* (depth 1000) | 479 | 583 | 14 |
| DeamonGame** (depth 1000) | 709 | 1139 | 19 |

characteristics influenced by the abstraction, such as data and instructions. In this section, we define such a generalized semantics for the subset of the SDL language considered, which will be used to reason about property preserving abstractions.

Every system $M \in SDL$ is a sequence of process instances $M = P_1||\ldots||P_n$ which run in parallel. Let $Inst$ be the set of basic instructions from which the processes are constructed. $Inst$ includes states labels, the assignment instruction, the Boolean and arithmetic operators, the `decision` and `nextstate` instructions, the instructions for sending and for receiving signals (denoted by $Input$ and $Output$, respectively), and the constructors `save` and `provided` for declaring saving signals and guards. Let us define $State$ the set of process states and $Decl$ as the declarative part of the system (types and variables). $State$ includes the special states `start` and `stop`. Then, every process is described as:

$$P = Decl; \{State : Tran\} \qquad Tran = \{(Input|null); Inst\}$$

$null$ representing the continuous signals.

In short, the behaviour of the process is as follows. Every state represents an internal process state defined by the programmer. In each one of these states, the process will carry out a transition, which will usually begin reading a signal from the input queue, and it will follow on with a sequence of arbitrary instructions. The transition will end with a `nextstate` instruction which will provoke the process into jumping to another state or with the `stop` state which will end the process [3] . As is usual, we make no assumptions about the speed of the processes or about the time spent in a transition or in a state. In addition, the delay due to the transportation of signals through channels is not considered in our framework, but we assume that signals are instantaneously transmitted.

We now present the generalized semantics of SDL, introducing the

[3] We are not considering the structural concepts in SDL as *blocks*, *procedures* or *services* since we are interested in the internal behaviour of the processes

definition gradually in order to be clear.

(1) Let $State_j$ and $Inst_j$ be the sets of the internal states (labels) and instructions of the process $P_j$, respectively. We assume that $State_j \subseteq Inst_j$.
(2) Let $SState$ be the set of tuples $(l_1, \ldots, l_n, q_1, \ldots, q_n, i_1, \ldots, i_n)$ representing the internal state of the system. For each process $P_j$, $l_j$, which is also a tuple, holds the actual value of every local variable at a point during the execution, $q_j$ is the content of the input queue, and $i_j$ the instruction just executed by $P_j$. $q_j = \emptyset$ and $q_{j_k}$ will denote that the queue $q_j$ is empty and the $k^{th}$ signal of $q_j$, respectively.
(3) Let $Sequence$ be the set of finite or infinite sequences of system states.
(4) Let $Initial : SDL \rightarrow SState$ be the function which returns the initial state of every system, i.e. variables have been initialized, $q_j = \emptyset$ and each $i_j$ is an special instruction which precedes the first one in every process $(start)$.
(5) Let $just\_exe_j : SState \rightarrow Inst_j$ be the just-executed function which, given a process and a system state, returns the last instruction of the process executed, i.e., $just\_exe_j((l_1, \ldots, l_n, q_1, \ldots, q_n, i_1, \ldots, i_j, \ldots, i_n)) = i_j$.
(6) The function $eval : BoolExp \times SState \rightarrow \{false, true\}$, $eval(exp, s)$ returns the evaluation of the Boolean expression $exp$ in the state $s$.
(7) $input_{i_1}, \ldots, input_{i_k}$ denote the $k$ instructions of $P_j$ in the state $i \in State_j$ that can enable a transition. $\forall 1 \leq n \leq k \;\; input_{i_n}$ may be a standard input instruction as *input signal*, a continuous signal or a spontaneous transition.
(8) Let $next\_inst_j : Inst_j \rightarrow \wp(Inst_j) \cup \{end\}$ be the function which returns the set of instructions that textually can follow $i \in Inst_j$.
- If $i \in State_j$ then $next\_inst_j(i) = \{input_{i_1}, \ldots, input_{i_k}\}$.
- $next\_inst_j($ `nextstate` ns $) = \{$ `state` ns $\}$.
- $next\_inst_j($`stop`$) = end$, and otherwise
- $next\_inst_j(i) = \{ni\}$, $ni$ being the instruction which follows $i$ in $P_j$.

(9) Let $exec_j : Inst_j \times SState \rightarrow \{false, true\}$ be the executable function:
- $exec_j(i, s) = false$ if $i \in State_j$ and
  - $q_j = \emptyset$ and no transition $input_{j_s}$ is continuous or spontaneous.
  - $q_j \neq \emptyset, q_{j_1} = signal$, and some $input_{i_k} = input\ signal\ provided\ exp$, and $eval(exp, s) = false$.
- $exec_j(i, s) = true$, otherwise.

In short, $exec_j(i, s)$ returns $true$ if the instruction $i$ of process $P_j$ does not suspend in the state $s$. Note that we are not considering that a signal can be discarded.

(10) Let $next_j : Inst_j \times SState \rightarrow \wp(Inst_j) \cup \{delay, end\}$ be the function which given a process instruction returns the next instruction to be executed, i. e.,
- $next_j(i, s) = end$, if $next\_inst_j(i) = end$
- $next_j(i, s) = delay$, if $\forall\ ni \in next\_inst_j(i), exec_j(ni, s) = false$

- $next_j(i,s) = \{i_1, \ldots, i_k\}$, if $next\_inst_j(i) = \{ni\}$, $ni$ being a non-deterministic decision such as

  `decision` any ():
  i1;...; nextstate ns1;
  ik;...; nextstate nsk;
  enddecision.
- $next_j(i,s) = next\_inst_j(i)$, otherwise.

(11) Let $S : (Inst_1 \cup \ldots \cup Inst_n) \times SState \rightarrow SState$ be a semantics function which gives meaning to each SDL instruction.
$S(i, (l_1 \ldots, l_j, \ldots, l_n, q_1, \ldots, q_j, \ldots, q_n, i_1, \ldots, i_j, \ldots, i_n)) = s' =$
$(l_1 \ldots, l'_j, \ldots, l_n, q_1, \ldots, q'_j, \ldots, q_n, i_1, \ldots, i, \ldots, i_n))$
means that executing the instruction $i \in Inst_j$, when the system is in the state $s$, produces the evolution of the system towards the state $s'$. $S$ is an unspecified generic function ; we only substitute $i_j$ by $i$ in the state to indicate that $i$ is the last instruction executed in $P_j$. The high level behavior of the model is not dependent on this function and this is why we do not define it. In [20], the actual meaning of every SDL instruction can be found.

(12) Let $Trans : SState \rightarrow \wp(SState) \cup \{end, deadlock\}$ be the function which returns the states to which the system can evolve from a given state $s$:
- $Trans(s) = \cup_{j=1\ldots n}(\cup_{i \in (next_j(just\_exec_j(s),s) \cap Inst_j)}\{S(i,s)\})$,
  if $j \in \{1, \ldots, n\}$ exists such that $next_j(just\_exec_j(s), s) \notin \{delay, end\}$
- $Trans(s) = end$, if $\forall j = 1, \ldots, n$, $next_j(just\_exec_j(s), s) = end$, and
- $Trans(s) = deadlock$, otherwise.

The generalized semantics $Gen : SDL \rightarrow \wp(Sequence)$ is defined as:

$Gen(M) = \{s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k \rightarrow \ldots \in Sequence/Initial(M) = s_0,$
$\forall j > 0.(Trans(s_{j-1}) \notin \{deadlock, end\}, s_j \in Trans(s_{j-1}))\} \bigcup$
$\{s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k \not\rightarrow \in Sequences/Initial(M) = s_0,$
$Trans(s_k) \in \{deadlock, end\}, \forall 0 < j \leq k.(Trans(s_{j-1}) \notin \{deadlock, end\},$
$s_j \in Trans(s_{j-1}))\}$

*Gen* associates each *SDL* model $M$ with the set of all possible state sequences that $M$ can generate in different system executions. This semantics is useful for our purposes since:

a)each path corresponds to a possible system execution which must be analyzed when verifying the *MSC*s;

b) as the meaning of the operations $S$ and *eval* is not defined, it is possible to change it to construct the abstract models;

c) *Gen* allows us to easily reason about the relation concrete/abstract model. If we denote $Gen(M)$ with $Gen(M, eval, S)$ to emphasize the functions on which it depends, then $Gen(M^*, eval^*, S^*)$ is the generalized semantics of $M^*$

It is interesting to note that when verifying with SDT no interleaving between processes is carried out while a process is executing a transition except when an `output` or a `create` instruction is executed (this execution method is called Symbol-sequence in SDT). The semantics $Gen$ could have been defined considering a coarser size of atomic instructions, but this would have complicated the exposition unnecessarily, and would have reduced its applicability to other tools.

## 7 Property Preserving Abstractions

In this section, we first explain the conditions conditions to guarantee correctness. Then we discuss the preservation results.

### *7.1 Correctness Conditions*

As we explained before, $Gen(M, eval, S)$ and $Gen(M^*, eval^*, S^*)$ denote the generalised semantics of the models $M$ and $M^*$, respectively. The following conditions on the meaning of the abstract instructions (which is defined with $eval^*$ and $S^*$) will assure the correctness of the transformation $M \rightarrow M^*$:

(1) Let us assume that $eval^* : BoolExp^* \times SState^* \rightarrow \{false, true\}$ verifies the following correctness relation: $\forall s^* \in SState^*.(\forall s \in SState.\alpha_s(s) \leq_s^* s^* \Rightarrow eval(exp, s) \leq_b^* eval^*(\alpha_{inst}(exp), s^*)))$. The partial order used in the set $\{false, true\}$ is $false \leq_b^* true$ which, in our context, means that $eval^*$ returns $false$ only if the evaluation of $\alpha_{inst}(exp)$ in each concretization of the abstract state $s^*$ is $false$. Thus, $eval(exp, s) = false \not\Rightarrow eval^*(\alpha_{inst}(exp), \alpha_s(s)) = false$, however $eval^*(exp^*, s^*) = false \Rightarrow eval(exp, s) = false$, for all $exp \in BoolExp$ and $s \in SState$ such that $\alpha_{inst}(exp) = exp^*$ and $\alpha_s(s) \leq_s^* s^*$.

(2) Let $S^* : (Inst_1^* \cup \ldots \cup Inst_n^*) \times SState^* \rightarrow SState^*$ be an abstract function verifying the relation: $\forall s^* \in SState^*.(\forall s \in SState.(\alpha_s(s) \leq_s^* s^* \Rightarrow \alpha_s(S(i, s)) \leq_s^* S^*(\alpha_{inst}(i), s^*)))$. $S^*$ gives abstract meaning to the instructions of the abstract system $M^*$. As before, $S^*$ is not specified, we have only declared its correctness relation with $S$.

The first condition says that the abstract instructions cannot produce new suspensions with respect to the initial model $M$. The second one imposes that the effect of an abstract instruction must be an approximation of the effect of the respective concrete one (as is classic in abstract interpretation).

We can now give the preservation results of the abstraction that justify the practical use of the abstraction method. The main results are related to deadlock detection (Proposition 8), non-verification of $MSC$s (Proposition 11) and abstraction of abstract models (Corollary 12), but other properties can be analyzed in the semantics framework. The proofs of the results have been omitted, but they can be found in [12].

In the following, we assume that the abstraction $\alpha$ preserves the number of signals in the queues.

Given $seq = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k \rightarrow \ldots \in Sequence$ and $MSC = ev_1 \rightarrow \ldots \rightarrow ev_k$, we define $\alpha_{seq}(seq)$ as $\alpha_s(s_0) \rightarrow \alpha_s(s_1) \rightarrow \ldots \rightarrow \alpha_s(s_k) \rightarrow \ldots \in Sequence^*$ and $\alpha_m(MSC) = \alpha_{inst}(ev_1) \rightarrow \ldots \rightarrow \alpha_{inst}(ev_k)$.

**Definition 4** *Given $seq^*$, $seq^{*'} \in Sequence^*$, then $seq^* \leq^*_{seq} seq^{*'}$ iff for all $i \geq 0$, $s^*_i \leq^*_s s^{*'}_i$.*

**Theorem 5** *Let $Gen(M, eval, S)$ and $Gen(M^*, eval^*, S^*)$ be the semantics of the models $M$ and $M^*$ verifying the conditions presented in Sect. 7.1, then for each deadlock-free sequence $seq \in Gen(M, eval, S)$, an abstract deadlock-free sequence $seq^* \in Gen^*(M^*, eval^*, S^*)$ exists, such that $\alpha_{seq}(seq) \leq^*_{seq} seq^*$.*

We impose the condition that every state in each concrete execution path corresponds to an abstract state in an abstract execution path. This is a strong result as we need the whole concrete computation to be simulated step-by-step by the abstraction. However this constraint is necessary for analyzing $MSC$s since, otherwise, abstract model could lose observable events of the $MSC$.

We say that $M^*$ $\alpha$ -approximates $M$, and we denote it with $M \sqsubseteq_\alpha M^*$, when $M$ and $M^*$ verify Theorem 5.

**Proposition 6** *Let $M = P_1 || \ldots || P_n$ be an SDL system and $\alpha = \alpha_d \times \alpha_q : D \times Q \rightarrow D^* \times Q^*$ an abstraction function verifying the conditions of Sect. 7.1 and preserving the number of signals in the queues. Let $Inst^*$ be the set of abstract instructions derived from $\alpha$. Let $M^*$ be the system obtained by abstracting all the constants and instructions of $M$. Let us assume that for each instruction $i^* \in Inst^*$ an SDL implementation exists verifying the correctness conditions imposed in Sect. 7.1. Under these conditions, the model $MI^*$, obtained by atomically substituting each model instruction of $M^*$ by its implementation, verifies Th. 5.*

In the previous discussion, we have assumed that the concrete system

$M$ is deadlock-free. The next Proposition studies how to analyze deadlock in $M$. For this purpose, we impose the conditions presented in the following definition.

**Definition 7** *Given an abstraction function $\alpha$, we say that $\alpha$ verifies the executability conditions iff for each pair of states $s^* \in State^*$, $s \in State$, such that $\alpha_s(s) \leq_s^* s^*$, and for each $i \in Inst_j$, $exec_j(i,s) = exec_j^*(\alpha_{inst}(i), s^*)$.*

In short, executability conditions assure that the abstraction does not introduce additional suspension behaviors in the abstract model. Proving that an abstraction function $\alpha$ verifies such conditions consists in proving that the suspension behaviour of the original model instructions which can deadlock is not modified by $\alpha$.

In the following we will assume that $M \sqsubseteq_\alpha M^*$, $\alpha$ being an abstraction function verifying the executability conditions presented above.

**Proposition 8** *If $Gen(M^*, eval^*, S^*)$ has no execution sequence which deadlocks then $Gen(M, eval, S)$ has no deadlock either.*

Next proposition explains the relationship between the concrete and abstract systems when proving properties expressed using $MSC$s.

**Definition 9** *Given an SDL system $M$, an $MSC = ev_1 \rightarrow \ldots \rightarrow ev_k$ and $seq = s_0 \rightarrow \ldots \rightarrow \ldots \in Gen(M, eval, S)$, the subsequence $subseq = s_j \rightarrow \ldots \in Sequence$ $(j \geq 0)$ verifies $MSC$ ($subseq \models MSC$) iff a) $k = 0$, or b) if $s_i$ $(i \geq j)$ is the first observable event in $subseq$ then $s_i = ev_1$ and the subsequence $subseq = s_{i+1} \rightarrow \ldots$ verifies $MSC' = ev_2 \rightarrow \ldots \rightarrow ev_k$.*

**Definition 10** *Given an SDL system $M$ and an $MSC = ev_1 \rightarrow \ldots \rightarrow ev_k$, $M$ verifies $MSC$ ( $M \models MSC$ ) iff a sequence $seq \in Gen(M, S, eval)$ exists such as $seq \models MSC$. Otherwise, $M$ does not verify $MSC$ ( $M \not\models MSC$ ).*

**Proposition 11** *Given an SDL system $M$, an $MSC$ and its abstract version $MSC^* = \alpha(ev_1) \rightarrow \ldots \rightarrow \alpha(ev_k)$, if $M^* \not\models MSC^*$ then $M \not\models MSC$.*

**Corollary 12** *Let us assume that $M \sqsubseteq_{\alpha_1} M^*$, $M^* \sqsubseteq_{\alpha_2} M^{**}$, $MSC^* = \alpha_{1_m}(MSC)$ and $MSC^{**} = \alpha_{2_m}(MSC^*)$, then if $M^{**} \not\models MSC^{**}$ then $M \not\models MSC$.*

**Example 13** *Note that the abstractions employed in Section 5 verify the hypothesis of Proposition 6, because they satisfy the correctness conditions of Sect. 7.1, the original system has no* `provided` *instruction and they also preserve the number of signals in queues. So we can employ the abstract systems to prove absence of deadlock and no verification of $MSC$ in the original one.*

## 8 Conclusions and further work

We have presented an approach for optimising the verification of SDL systems, which is compatible with other tasks in current commercial tools such as SDT. Our method consists in the automatic transformation of the SDL model in order to obtain a simpler description, which can be analyzed for requirements such as absence of deadlock and non-satisfaction of $MSC$s. The method is based on the definition of a generalized semantics of SDL which is suitable to justify the correct transformation of SDL models into more abstract versions. In this work, like in other previous ones ([11] [13] [14]), we employ abstract interpretation as a technique to improve the automatic verification of systems.

Our method can be improved in different ways. The first important task is to complete the definition of a set of abstraction functions which can be easily selected by the user in order to perform the desired transformation, thus freeing the standard user from the need to think about correct simplifications of the SDL model. The second task is to obtain even more abstract transformed models by using information about the structure of the SDL specification, for example, the potential consumers or senders for given signals to be abstracted. We are currently working on the implementation of tools to employ the abstract interpretation technique within the SDT environment. Our aim is to obtain a user-friendly environment to improve the verification possibilities of this tool in order to stimulate users into performing more verification in the development cycle for software or other kinds of complex systems. As far as we know our approach to applying abstractions in the framework of SDL and MSC is an original contribution.

## References


[1] Bensalem S., Lakhnech Y., Owre S.: Computing Abstractions of Infinite State Systems Compositionally and Automatically. In Hu, A.J., Vardi, M.Y., (Eds.) Computer Aided Verification, LNCS-1427, Springer, 1998.

[2] Cousot P., Cousot R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Conf. Record of the 4th ACM Symp. on Principles of Programming Languages, 1977.

[3] Clarke E.M., Grumberg O., Long D.E.: Model Checking and Abstraction. ACM Transaction on Languages and Systems, 16(5), 1994.

[4] Clarke E.M., Wing J.M.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, 28(4), 1996.

[5] Cleaveland R., Riely J.: Testing-Based Abstractions for Value-Passing Systems. In Jonsson, B. and Parrow, J. (Eds) CONCUR 94, LNCS-836, Springer, 1994.

[6] Dams D., Gerth R., Grumberg O.: Abstract Interpretation of Reactive Systems. ACM Transactions on Programming Languages and Systems, 19(2), 1997.

[7] Dwyer M.B., Păsăreanuams C.S.: Filter-based Model Checking of Partial Systems, In Proc. ACM SIGSOFT Symp. on the Foundation of Software Engineering, 1998.

[8] Dwyer M.B., Hatcliff J., Smitdt D., Huth M., Stoughton A.: Bandera, http://www.cis.ksu.edu/santos/bandera/

[9] Ferdinand C., Martin F., Wilhelm R., Alt M.: Cache behavior prediction by abstract interpretation Science of Computer Programming, 35 (2), 1999.

[10] Graf S., Saïdi H.: Construction of Abstract State Graphs with PVS. In Grumberg, O., (Ed.) Computer Aided Verification, LNCS-1254, Springer, 1997.

[11] Gallardo M.M., Merino P., Troya J.M.: Relating Abstract Interpretation with Logic Program Verification. In Proc. of the 1st International Workshop on Verification, Model Checking and Abstract Interpretation, 1997.

[12] Gallardo M.M., Merino P.: A semantics and abstraction based Framework for the verification of SDL against MSC. Technical Report. Dpto. de Lenguajes y Ciencias de la Computacion. University of Malaga.

[13] Gallardo M.M., Merino P.: A Framework for Automatic Construction of Abstract Promela Models. In Dams D., Gert R., Leue S., Massink M. (Eds.) Theoretical and Practical Aspects of SPIN Model Checking, LNCS-1680, Springer, 1999.

[14] Gallardo M.M., Merino P.: Verifying Distributed Systems with Model Checking and Static Analysis To appear in Proc. of the 20th International Conference on Distributed Computing Systems - International Workshop on Distributed System Validation and Verification. April 2000.

[15] Giacobazzi R., Debray S.K., Levi G.: A Generalized Semantics for Constraint Logic Programs. In Proc. of the International Conference on Fifth Generation Computer Systems, 1992.

[16] Gunter C., Mitchell J.: Strategic Directions in Software Engineering and Programming Languages. ACM Computing Surveys, 28(4), 1996.

[17] Halbwachsa N.: About synchronous programming and abstract interpretation Science of Computer Programming, 31 (1), 1998).

[18] Holzmann G.J.: Design and Validation of Computer Protocols. Prentice-Hall, 1991.

[19] Holzmann G.J.: Designing Executable Abstractions. In Proc. of Formal Methods in Software Practice, 1998.

[20] ITU-T Recommendation Z.100, Specification and Description Language (SDL), 1993.

[21] ITU-T Recommendation Z.120, Message Sequence Chart (MSC), 1994.

[22] Jones, N.D., Søndergaard, H.: In Abramsky S., Hankin C., (Eds.) A Semantics-based framework for the abstract interpretation of Prolog. Abstract Interpretation of Declarative Languages, Ellis Horwood, 1987.

[23] Loiseaux C., Graf S., Sifakis J., Boujjani A., Bensalem S.,: Property Preserving Abstractions for the Verification of Concurrent Systems. Formal Methods in System Design, 6, 1995.

[24] Rusu V., Singerman E.: On proving Safety Properties by Integrating Static Analysis, Theorem Proving and Abstraction. In Cleaveland, W.R., (Ed.), Tools and Algorithms for the Construction and Analysis of Systems, LNCS-1579, Springer, 1999.

[25] Saïdi H., Shankar N.: Abstract and Model Check while you Probe. In Halbwachs, Peled, D., (Eds.) Computer Aided Verification, LNCS-1633, Springer, 1999.

[26] Telelogic: Telelogic TAU User Manuals, 1999. Available in http://www.telelogic.com.

[27] Verilog: ObjectGEODE Method Guidelines, 1999. Available in http://www.verilogusa.com.

# EXPERIENCES WITH TOOL DEVELOPMENT OF SDL IN COMBINATION WITH ASN.1 FOR COMMUNICATION PROTOCOL APPLICATIONS

**Ralf Schröder**
**Martin v. Löwis of Menar**

Humboldt Universität zu Berlin
Institut für Informatik
Lehrstuhl für Systemanalyse
Unter den Linden 6
10099 Berlin, Germany
tel: +49 30 2093 3120
fax: +49 30 2093 3111
e-mail: r.schroeder/loewis@informatik.hu-berlin.de

SDL in Combination with ASN.1, as a standardized variant of SDL, is a formal description technique. The language is used for the documentation, standardization, and verification of distributed communication systems. It is even possible to generate executable programs from such specifications. This paper introduces the tool environment SITE. The environment supports a compilation from SDL and ASN.1 to C++ or Java.
The ASN.1 support was extended for the application of SITE in industrial projects. The communication aspect of ASN.1 (encoding of values) is now available for use in specifications, and more ASN.1 constructs than the standardized combination covers, can be used (e.g. macro applications). The paper shows the application area of SITE, presents some details for the data implementation, and explains the current state of the ASN.1 extensions in SITE.

**Keywords**: SDL, ASN.1, Basic Encoding Rules, C++/Java code generation, SITE, code optimization, SDL environment communication

# Introduction

The widely-used language SDL [Z.100] is important for the specification of telecommunication systems, especially standardized systems. The highlights of the language are the intuitively understandable semantics of all base concepts and the graphical representation with excellent tools support. Also, SDL is alive. The ITU-T (International Telecommunication Union - Telecommunication standardization sector) recommends SDL as standardized specification technique and maintains the language actively.

A significant aspect for applications is the data concept of SDL. For today's protocol specifications, e.g. the small and broad band ISDN protocol family or the IN area, the complexity of data descriptions grows dramatically. However, the abstract data concept of SDL is neither accepted by the users nor completely implemented by tool vendors. The combination of SDL with ASN.1 is a good compromise to satisfy user requests for better data description techniques and tool vendors with their implementation problems.

ASN.1 [X.680] is a standardized data description technique defined in the OSI context [X.200]. The language allows the specification of data structures and values. It does not support the description of behavior, e.g. the manipulation of data values with operators. In addition to the language definition, there are standardized encoding rules for ASN.1 values. The key idea is to give an abstract description of data (ASN.1 - Abstract Syntax Notation One) together with basic encoding rules for values. An ASN.1 compiler can be used to generate data structures and corresponding encoding functions for the concrete target language (cf. figure 1). There are two standardized versions of ASN.1. The old one [X.208] is deprecated. The new one, last revised in 1997, includes multiple encoding schemes.

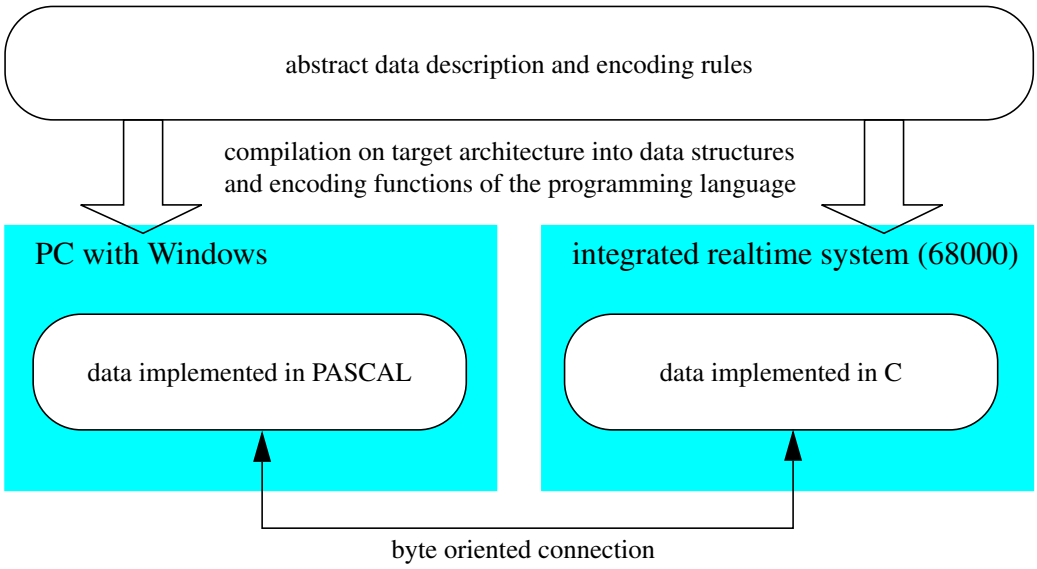



*Figure 1: Communication scenario*

The motivation of the combination of both languages SDL and ASN.1 comes from ASN.1 based protocol descriptions with informal behavior part. The replacement of the informal behavior part by a formal SDL description would require a transformation of the ASN.1 data to SDL data. If done manually, this is a rather redundant step, and it implies problems with the tool support for the translated SDL data structures. The combination of SDL with ASN.1 unifies the strong data description methods of ASN.1 with the expressive behavior part of SDL. Additionally, the possibility to encode ASN.1 data values can be used for an universal communication with an SDL system specification.

The SDL Integrated Tool Environment [SITE] of Humboldt University provides an SDL to C++ or Java compilation process, which is used for simulation, development of prototypes as well as real world applications. SITE supports the combination of SDL with ASN.1 together with the Basic Encoding Rules of ASN.1 [X.209]. This paper discusses two aspects:

1. ASN.1 for data descriptions in SDL, and
2. ASN.1 as base for universal communication with SDL systems.

The first aspect is standardized by ITU-T [Z.105], however based of the old ASN.1 version. SITE supports a more open approach with respect to the ASN.1 combination, i.e. it provides basic compatibility to the standard as well as extensions. Section 1 is concerned with the general use of SITE, especially the ASN.1 combination. The differences to the Z.105 approach are worked out. Details with respect to the ASN.1 code generation support are outlined in section 2. The second aspect goes beyond the standardized combination of both languages. This is discussed on different realizations with SITE in section 3. Finally, additional ASN.1 language concepts, which are not part of Z.105 are considered in section 4. These concepts were developed and implemented on user requests.

# 1. Development of applications with SITE components

SDL has two language representations: the graphical syntax SDL/GR, and the textual syntax SDL/PR. The the compilation process of SITE is based on SDL/PR. Normally, SDL specifications are developed graphically, so that a third-party editor is necessary. There are several SITE extensions to support different commercial editors directly, e.g. when diagnosing specification errors, implementing file include features, and language extensions. Currently, a direct connection is being considered between SITE and the commercial SDL tool Cinderella [Cin99] based on the Cinderella Application Program Interface.

This section explains how SITE can be used for building programs from SDL, where the ASN.1 part is considered separately. Finally, the application areas of SITE technology are presented.

## 1.1 SITE components in one view

SITE is an open collection of tools, which are able to handle SDL specifications. Tool components can be compiled for any platform with a standard C compiler. An overview of tools for the compilation process is given in figure 2.

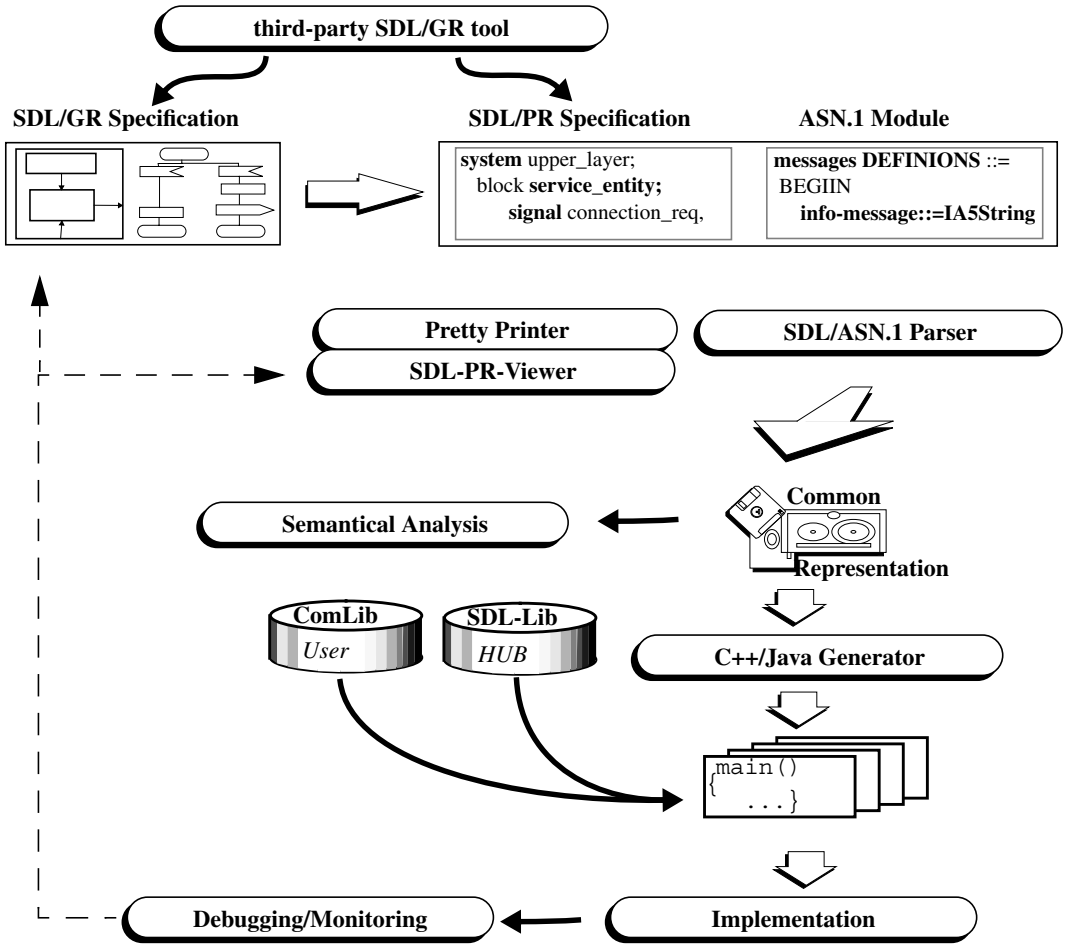



*Figure 2: SITE compilation process*

- A syntax analysis (Parser component) transforms an SDL/PR specification into an architecture-independent interchange format called *Common Representation* (CR). This format is the base for independent development of SITE components. In principal, a graphical editor could produce a CR instance directly; this option is currently being investigated as a strategy for integration with the Cinderella tool.
- The semantical analysis checks a specification (as CR instance) according to the static semantics of SDL. Other SITE components assume a semantically correct specification.
- The SDL code generator produces C++/Java code from the specification based on an abstract library interface. There are different libraries for C++: a simulation library, two libraries with underlying user-provided communication libraries and a CORBA

[OMG] based library for prototype programs. The generated code includes methods to produce a complete simulation interface, e.g. traces and state information. The library implementation decides about the use of these information, e.g. with a debug flag for the C++ compiler. The Java generation is under development. A development branch, which is used by Siemens AG, is an ASN.1 to Java compiler which supports Basic Encoding Rules.

- There are some components for back interpretation of test results. Most of them redirect results to the SDL/PR source. So for the simulation a small simulator interface with an SDL/PR viewer is available.
- Because from SDL/GR generated textual specifications are unreadable (in general the SDL/PR source code is annotated with layout information), a pretty printer can be used. Besides the reformatting process, the pretty printer is able to create PostScript, LaTeX and HTML output with syntax highlighting and for HTML optional cross reference links.

From each SDL system an executable program is produced, which can communicate with the operating system environment dependent on the used libraries. Packages can be compiled separately to a (static or dynamic) C++ library. The SDL developer normally uses an automatic production process provided by scripts, compilation technologies (*make*) or, so planned with Cinderella, interactively from the the graphical editor. Properties of all SITE components (e.g. case sensitive name handling, language extensions) can be controlled by command line options in a wide range.

## 1.2 Combining SDL and ASN.1

In 1995 ITU-T published the standardized combination of SDL with ASN.1 [Z.105]. This is a syntactic combination, which defines a new language version of SDL. ASN.1 type definitions and expressions can be used as SDL sort definitions and expressions. Such a direct combination is a compromise for both languages, e.g.:

- ASN.1 parts are not case sensitive,
- SDL keywords are not allowed in ASN.1 parts and vice versa,
- complex ASN.1 construct are not supported (ASN.1 macros),
- encoding of ASN.1 (tagging) is ignored.

The semantic base of the combination is a mapping from the additional ASN.1 constructs to SDL (in general data definitions).

The SITE solution for the combination of SDL with ASN.1, which was developed about 1993, influenced the standardized development. The functionality for basic data structures, i.e. the set of operations for ASN.1 data types, was derived from the SITE realization, so that on semantic level SITE is compatible to Z.105. However, the syntactic approach is different: ASN.1 specifications can be included „as is“ instead of SDL data definitions by using two keywords (cf. figure 4). Between these keywords, an analysis takes place according to the ASN.1 language definition. It is even possible to compile

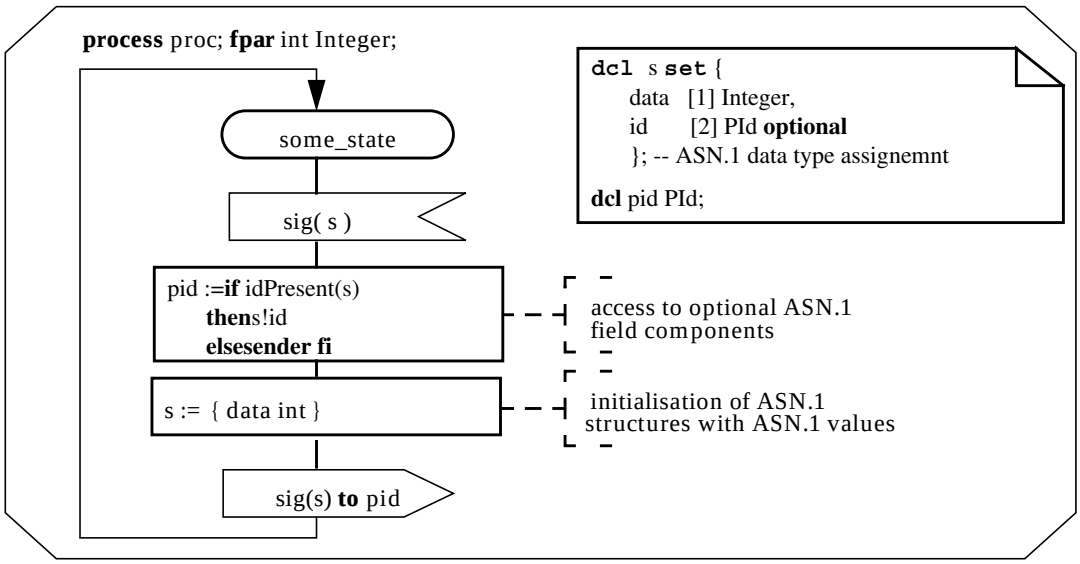



*Figure 3: Example SDL in Combination with ASN.1 according to Z.105*

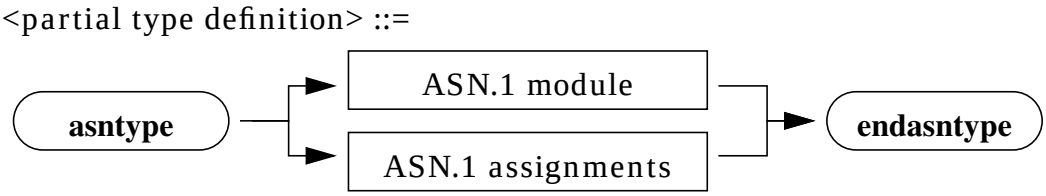



*Figure 4: ASN.1 integration with SITE*

ASN.1 modules stand alone. It is a question of tool components and additional combination rules, of course, to support more ASN.1 features than the standard X.105. Today, SITE tool components are implemented with the following features:

- application of Basic Encoding for ASN.1 values,
- selected ASN.1 macro applications and
- extensions of the current ASN.1 standard [X.680] with respect to the old [X.208] ASN.1 version.

The methodology behind that constructs is discussed in the next sections. A specification without these extensions can be transformed to a Z.105-like, semantically identical specification by introducing a ';' between ASN.1 assignments and dropping the keywords **asntype/endasntype**. In that sense, the SITE solution is a Z.105 restriction. Even ITU-T recognized that such a restriction is useful. The new revision of SDL in Combination with ASN.1 orients to the use of ASN.1 in complete modules; mixed notations are deprecated.

## 1.3 Application areas

SDL systems can be embedded in heterogeneous software systems. Possibly, the developers of such systems have their own application libraries for communication and scheduling as an abstraction layer for different software products. In such cases, the SDL code generation must follow the application design rules. The advantage of SITE is the high flexibility of all tool components, especially the code generation. Except for additional language features, only the interface between code generator and the user provided communication library has to be reimplemented (e.g. for the target language C++ adaptation of 10000 until 20000 lines of code). This abstraction between code generator and the user application library is called the SDL library. Within this library you have the choice between completeness of SDL semantics and performance of the program code. Moreover, the user can easy read the generated code because of the abstract library interface and the direct correspondence to the original SDL source. This is important for hand implementation of functionality, for example data base access.

The feasibility of the adaptation mentioned above is proved by two major projects:

1. **FTZ Darmstadt of Deutsche Telekom AG**: Demonstration of prototype solutions for a B-ISDN protocol stack[BGKS+];
2. **Siemens AG Berlin**: Generation of software for IN service control points.

Other research projects are the development a simulation library for SDL and the replacement of the application library for communication and scheduling by a CORBA implementation.

# 2. Code generation for data

SITE code generation is designed for full SDL-96 and ASN.1 support. Nevertheless, a concrete code generator version can be restricted to a useful SDL subset, because

- of performance reasons,
- the lack of support from the application library.

With respect to data, the user is able configure many properties of the code generation itself. The way, how this can be done with SITE is outlined first. Because of the rapidly growing data part in protocol specifications, the generation of efficient code becomes important. The SITE code generator is able to perform optimization on SDL level. The optimization strategies are discussed, too. Finally, the support for data encoding is considered.

## 2.1 Data design

The SITE compiler components know the ASN.1 type constructions and generate the corresponding C++/Java classes. The generated code is based on the runtime library library, which provides base classes for the main SDL constructs. Here the simple data types like INTEGER or OCTET STRING are implemented as classes. The decision to use classes for simple types is a direct consequence of the following requirements:

- Code generation should be supported for all SDL constructs, especially object oriented features such as context parameters.
- Generated code should have a similar structure as the SDL specification. As a consequence, inheritance and context parameters are implemented by object oriented concepts of the target language and not by flattening the structure. Because simple types cannot be inherited (in C++), a container class becomes necessary.

The set of predefined types and their supported operators as well as SDL generator based constructions like SET OF BOOLEAN depends on the used library and can be specified in a separate SDL specification which stands for the standardized package *Predefined*. It can be modified by the user, e.g. to add own data structures. There are special comments to control the style of code generation with respect to:

- the use of specific names in the generated code,
- the signature of operators,
- translation of name class literals for the target language.

For example, if the library supports an operator *assert*, the declaration in *predefined.sdl* could be:

```
newtype Boolean
    literals
        /*$codegen: SDLBool::SDLTrue() */ True,
        /*$codegen: SDLBool::SDLFalse() */ False;
    operators
         /* other operators of Boolean */
         /*$codegen: ASSERT #S*/
        assert: Boolean -> Boolean;
endnewtype Boolean;
```

The code generator will generate the text „SDLBool::SDLTrue`()`“ (in general the space terminated string after `$codegen:`) for a literal *True* in an SDL expression or the call of a static operator „`dummy=ASSERT(condition)`“ (other variants: #C - constructor call of the actual inheritance level, #n - a member function of the n-th operator argument) for an SDL assignment „`task dummy:=assert(condition)`“. Moreover, the semantic analysis recognizes that an assignment „`task dummy:=assert(0)`“ has a signature error (0 is not of sort *Boolean*). Even generic literal definitions can be controlled by using regular expressions substitution, e.g to delete leading zeros of integer values for C++ code generation:

```
literals
  /*$codegen: 0*([1-9][0-9]*) \1 */
  nameclass ('0':'9')* ('0':'9');
```

As a consequence, the user is able to add the static semantics and code generation support by modifying the package *Predefined* or adding own (external) packages to the SDL specification with corresponding non-SDL libraries.

## 2.2 Code optimization

Because the SITE tools was also used in industrial context, the optimization of the generated code was necessary. The main directive for optimization is not to corrupt the SDL semantics. The analysis of most used structures as well as run time analysis of generated programs were the base of improvements. A list of optimization features w.r.t. data is given in the list below. The sequence is from highest to lower importance.

***Assignments:*** Value assignments are expensive operations. In general, the class object must be copied recursively according to the SDL semantics. In some situations the copy operation can be replaced by reference assignments. The SITE code generation is able to recognize some of these situations:

- assignment of a temporary created literal value, e.g. var := (. 42, True .);
- simple variable modifications, e.g. int_var := int_var + 1, string := string // 'appendix';

Because assignment optimization also works for signal output, operator and procedure calls, this kind of analysis is the most important one.

***Procedure optimization:*** Normally, procedures can be specified with states. As a consequence, the control flow of a procedure has to be integrated in the scheduling scheme of the target library. However, most procedures are used „simply“ to manipulate ASN.1 data values[1]. This kind of procedures can be excluded from the scheduling scheme.

The ASN.1 combination implies a high number of simple procedures. Therefore this kind of optimization is an important step, too. However, with the further development of SDL, especially the data concept, the use of procedures will be replaced by normal SDL operators or methods.

***Decision optimization:*** SDL decisions allow rather complex evaluation rules for the determination of the action branch. However, more than 90% of all decision constructs are simple: boolean decisions or branching by enumerations. These cases can be mapped to corresponding target language constructs in efficient manner.

***Loop optimization:*** SITE code generation starts an extra function at the begin of a label. As a consequence each jump is a complex (virtual) function call. Simple loops, e.g. for the initialization of fields are recognized by the code generation and the function call is replaced by simple jumps of the target language.

Graphical editors also include artificial labels. If the code generation is connected with a certain graphical editor, these labels could be eliminated, too. This feature is not implemented with SITE.

---

1. Design problem of SDL in combination with ASN.1: ASN.1 data definitions cannot define operators; one work around is the use of procedures.

***In-line fragments:*** References to constants (SDL synonyms) can be replaced by the value. In combination with expression evaluation by the code generator the result is a small performance improvement for initialization and loops. The disadvantage is the enlargement of the generated code.

The result of such an optimization extremely depends on the specification style. For example the (for optimization adapted) specification

```
/* a simple procedure, no scheduling stragegies */
procedure fib; fpar i Integer; returns ret Integer;
  start;
    decision i<=2; /* a simple if construction */
      (True): return 1;
      else:
    enddecision;
    /*  no creation of Integer objects, direct variable
        manipulation, e.g. i.increase(-1)
    */
    task    i := i-1, ret:=call fib(i),
            i := i-1, ret:=ret + (call fib(i));
    return;
endprocedure;
```

for the recursive calculation of fibonacci numbers called with 32 could be improved by factor more than 1000. The performance for that example is as good as a „normal“ C++ code. A simple roundtrip communication of two distributed SDL processes (socket connection, integer data parameters, Basic Encoding Rules) had a approximately doubled the signal transfer rate after optimization.

## 2.3 Basic Encoding

If the SDL library supports the standardized Basic Encoding Rules [X.209] of ASN.1, the complex type definitions need a direct support for encoding functions from the code generator. The support of BER implies the correct use of ASN.1 tags, e.g. unique tags in a CHOICE type. This support is out of scope of the language definition in Z.105, however. The SITE environment pays attention to ASN.1 tags but can ignore them on request with command line options, e.g. if the generation of coding functions can be suppressed. Analysis as well as code generation can be used for stand alone ASN.1 specifications, too. The basic design of the SITE encoding support is taken from the public domain ASN.1 tool *Snacc* [SaNe].

The generated coding scheme can be used implicitly as described in the next section as well as explicitly with additional operators. If the last feature is supported from the used SDL library, each data definition (Sort) has two implicitly defined operators:

```
Decode   : Sort -> Octet_String;
Encode   : Octet_String -> Sort;
```

An useful application of encoding operators is demonstrated later with the ASN.1 type ANY. The implementation of other encoding schemes than BER is imaginable.

# 3. Communication with SDL signals

The use of signals to the SDL system environment is a suitable way to communicate with an SDL system. SDL signals carries optional data values, which can be encoded using BER. This scenario assumes that signal parameters are ASN.1 data only or that SDL data structures are mapped to ASN.1 definitions. The SITE tool set is able to generate suitable coding functions for ASN.1 data definitions as well as selected SDL structures. The use of SDL data without direct ASN.1 representation is not recommended, because of the tool specific mapping. Such data are Array and SDL structure constructs (*newtype struct*), or even the simple sorts *PId* and *Time*.

The standardized coding is not enough, however. The byte stream must be transmitted somehow and a suitable addressing scheme has to be provided, too. Normally, the generated software is embedded into a communication system, e.g. a proprietary TCP implementation, which is able to transmit a byte stream to a certain communication partner. Often such an interface is not developed for the communication with more than one communication instance per endpoint, so that the SDL specific sender and receiver information has to be coded into a PDU somehow. The ASN.1 description

```
SignalPDU ::= SEQUENCE {
  signal_identification ID,
  sender PId,
  receiver PId OPTIONAL,
  route PATH OPTIONAL,
  encoded_parameter OCTET STRING
}
```

could encapsulate the basic SDL signal instance properties, where the definition of the data types *ID*, *PId*, and *PATH* are environment-dependent. Of course, an application library specific PDU could be used, too. The concrete realization contains the SDL library as the link between code generation and application library. There is a wide range of implementation details. Some details of the two major SITE applications as well as of our research projects (cf. section 1.3) are discussed here.

One SDL library implementation was used for a demonstration platform for advanced multimedia teleservices, based on a sophisticated broadband network and signalling system [BGKS+]. Most of the B-ISDN protocol layers was specified using SDL systems, ATM hardware driver and user applications was programmed in C++. Here the signal identification is a static assignment to numbers, each SDL external channel is mapped to a connection oriented communication peer of the communication library and PId values are represented by locally unique numbers. The connection of two SDL systems is managed without user configuration assuming equal channel and signal names. Such an external channel supports signal buffering, if the communication partner is temporary not available (not started). Normally, such a connection remains open until the program dies. The consistent channel description of two SDL systems can be checked with the semantic analysis. A non SDL program can use a name service to establish the connection even across operating system boundaries, e.g. between VxWorks, Solaris and

WinNT. The figure 5 demonstrates the connection scenario of two SDL systems, the name service and a central time server. The registered system name can be used as binding to any SDL channel, where this interface is used for test purpose only.

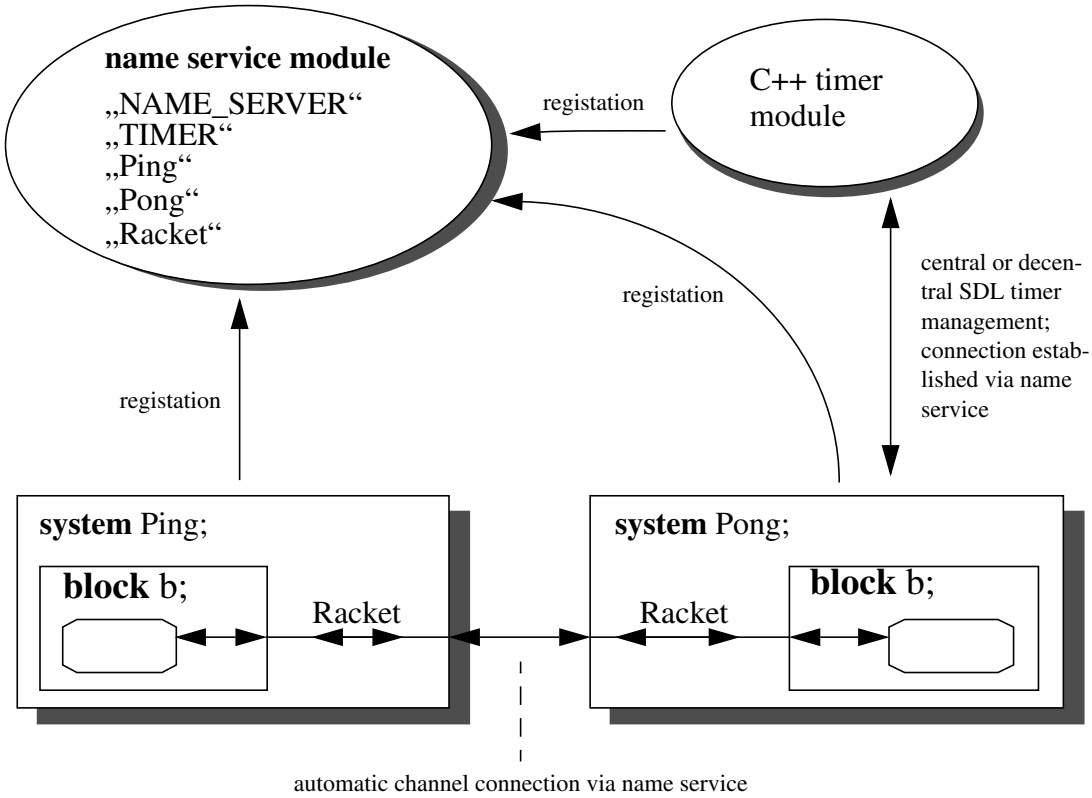



*Figure 5: SDL channel based communication*

The second SDL library implementation is part of a IN run-time-environment developed by Siemens AG for call control units. Here the communication infrastructure can be configured dynamically with a management information base (MIB)[CFSD]. The configuration includes the mechanism of the transport because of different interfaces to other programs, e.g. libraries for interacting with TCAP or SNMP. SDL packages with hand implemented data structures allow the access to MIB objects from the specification, e.g. to determine address information. There is s direct relation of a local SDL process address (PId value) to a globally unique, MIB based address information. In that scenario, connections can be established (and closed) on request dynamically. To avoid buffer overflows, SDL channels have no buffer, the specification must be able to manage discards signals.

A third solution for an SDL library with environment communication is the use of a CORBA implementation of ORBacus [OOC98] for the transport of PDU's. Here an IDL interface is provided for channel endpoints, PId's, signals and the system. The initial connection is established with an extra name service. Because of the two levels of encoding data parameters - BER for parameters and CDR for signals with the encoded

data parameter - a direct encoding of data with CDR is desirable. This technology is described in [Bie97]. This library was used for the prototype development of alternative transport protocols (B-ISDN [Q.2931]) of CORBA implementations [AIGN+].

# 4. Constructs beyond Z.105

Real world ASN.1 specifications consists of more language elements than the Z.105 supports. The following 3 extensions was developed in close teamwork with Siemens AG.

## 4.1 Communication with ASN.1 macros

At least the macro applications (or equivalent X.680-style information objects) OPERATION and ERROR should be supported by the combination of SDL with ASN.1. These constructs are used by protocols extensively. Useful is a mapping of macro values (object instances) to SDL signals, e.g. the specification

```
op OPERATION
     ARGUMENT INTEGER
     RESULT BOOLEAN
     ERRORS { timeout, unsupported }
   ::= localValue 42

timeout ERROR PARAMETER REAL ::= localValue 1001

unsupported ERROR ::= localValue 1002
```

introduces multiple SDL signals implicitly:

```
signal op(ROSE_ARG,Integer),
       op_RESULT(ROSE_RES,Boolean),
       timeout(ROSE_ERROR, Real),
       unsupported(ROSE_ERROR);
```

Contrary to the first intuition of SDL users, the ASN.1 OPERATION macro is *not* expressed as a remote procedure in SDL. The protocol stack used (TCAP, [Q.771]) defines different operation classes, e.g. an operation might “succeed” if there was no error in a certain period of time. Also, a service might need to invoke multiple operations simultaneously, which are then transmitted in a single PDU. Fine-tuning control flow based on remote procedures would not be possible.

The additional parameter is based on practical experience: a real world library for remote operations, e.g. an SS7 stack implementation, expects additional protocol information (e.g. invoke identifications). The exact contents of the parameter is specific to the application area. This functionality is supported for a Siemens AG specific SDL library to connect SDL specifications with an SS7 stack implementation of an other vendor.

Like ERROR and OPERATION, other information object definitions of the new ASN.1 standard could be directly supported by SDL in combination with ASN.1.

## 4.2 Rules of extensibility

These rules are an encoding enhancement for better version control of PDU‘s supported by the current ASN.1 version. The essence is that the encoder has to skip and store unknown encoded fragments and the decoder has to include the stored unknown arguments again. The unknown encoded strings have to be stored in placeholders of data structures. Possibly this placeholder is hidden or can be accessed by suitable operations. At least, it must be assignment-transparent:

```
UpperPDU ::= SEQUENCE{
  i Integer,
  ...
}
AdressedPDU:= SEQUENCE{
  my_addr PId,
  pdu UpperPDU
}

signal from_upper_level(UpperPDU),
       to_lower_level(AddressedPDU);

...

input from_upper_level(upperPDU);
  task newPDU := (. self, upperPDU .);
  output to_lower_level(newPDU);
  ...
```

This specification should be able to receive an encoded a value of type

```
UpperPDU-version2 ::= SEQUENCE{
  i Integer,
  ...,
  b Boolean
}
```

The additional field b is also part of the encoded value for signal *to_lower_level*. Choices and enumerations are straight forward, e.g.:

```
Version ::= enumerated { one(1), two(2),...}
dcl v Version;
...
decision v;
  (one) : ...
  (two): ...
  else : /* unknown versions */
enddecision;
```

## 4.3 ASN.1 ANY

The interpretation of the ASN.1 type ANY is given in Z.105:

```
newtype Any_type
endnewtype Any_type;
```

Obviously, this sort is not helpful, there are no values. A more convincing solution is already discussed in [Schr94]: the definition of *Any_type* depends on all defined data of the specification and provides operations to convert from and to a specific sort:

```
Seq ::= sequence {
  kind Integer,
  parameter any defined by kind
}

dcl seq Seq, b Boolean, i Integer;
...
decision seq!kind;
   (0): task b := Boolean(seq!parameter);
   (1): task i := Integer(seq!parameter);
   else : call ErrorMsg('some error...');
enddecision;
```

In context of ASN.1 there is a natural technical solution for the conversion: it is the application of the encoding rules. If conversion to the requested type fails, an SDL exception is raised in the SDL library implementation. Such an exception concept, which is already supported by SITE, will be part of the SDL language revision in 2000.

# 5. Conclusion

Even if the combination of SDL with ASN.1 is a remarkable progress for SDL applications, there are two major shortcomings:

1. the communication aspect is not standardized and
2. the generation of efficient target code remains rather tricky.

With the removing of ACT ONE from SDL and the introduction of object-oriented data, especially of a reference concept, in the new language revision SDL-2000 at least the second problem could disappear.

The embedding of SDL specifications in complex heterogeneous systems is opened with a question for encoding of SDL data values in the new ITU-T study period. The interface definition language IDL [OMG] could become more important than the ASN.1 integration. Currently, there is a standardized [Z.130], rather difficult mapping from ODL (an IDL adaption to telecommunication issues) to SDL in Combination with ASN.1. SDL-2000 allows much more direct mapping rules. SITE development already started in that direction!

# Literature


[AIGN+] AT&T, Iona, GMD Fokus, Nortel, Teltec DCU, Alcatel, Deutsche Telekom, Ericsson Telecommunications, Humboldt University of Berlin, Object Oriented Concepts, Open Environment Software, Telenor: Interworking Between CORBA And TC Systems, document telecom/98-10-13, OMG, 1998.

[BGKS+] Ulf Behnke, Michael Geipl, Gerd Kurzbach, Ralf Schröder, Nils Fischbeck, Renee Mundstock: Development of broadband ISDN telecommunication services using SDL'92, ASN.1 and automatic code generation. In Participation Proceedings of 8th international conference of Formal Description Techniques for Distributed Systems and Communication Protocols FORTE'92, Montreal,1995.

[Bie97] Frank Bielig: Implementierung einer SDL-Laufzeitbibliothek auf CORBA-Basis, Diplomarbeit, Humboldt-Universität, Institut f. Informatik, Berlin, 1997.

[CFSD] J. Case, M. Fedor, M. Schoffstall, J. Davin: A Simple Network Management Protocol (SNMP), RFC 1157, 1990.

[Cin99] Cinderella: Cinderella SDL. Technical Description, Danmark, 1999.

[KLS99] Gerd Kurzbach, Martin v. Löwis, Ralf Schröder: External Communication with SDL systems. In SDL'99 The Next Millennium, R.Dssouli, G.v.Bochmann, Y.Lahav (editors), proceeding of the Ninth SDL Forum Montreal, Canada, Elsevier, 1999.

[OMG] OMG: The Common Object Request Broker: Architecture and Specification. Revision 2.0, July 1996.

[OOC98] OOC: ORBacus for C++ and Java, Object-Oriented Concepts, Inc, 1998.

[Q.771] ITU: Functional Description of Transaction Capabilities, Recommendation Q.771, Genf, 1997.

[Q.1228] ITU: Interface Recommendation for intelligent network Capability Set 2, Geneva, 1997.

[Q.2931] ITU: Recommendation B-ISDN-DSS2-UNI-Layer 3 specification for basic call / connection control, Genf.

[SaNe] Michael Sample, Gerald Neufeld: Snacc 1.0: A High Performance ASN.1 to C/C++ Compiler. University of British Columbia, Vancouver, Canada, 1993.

[Schr94] Ralf Schröder: SDL'92 data handling in combination with ASN.1. master thesis, Department of Computer Science, Humboldt University Berlin, 1994.

[SITE] SDL Integrated Tool Environment of Humboldt University Berlin: http://www.informatik.hu-berlin.de/Themen/SITE.

[X.200] CCITT: Data Communication Networks Open System Interconnection {(OSI)} model and notation, service definition. Blue Book Recommendation X.200, Melbourne, November 1988.

[X.208] CCITT: Specification of Abstract Syntax Notation One (ASN.1), In Open Systems Interconnection [X.200] - Basis Reference Model. International Standard Recommendation X.208, conform with ISO8824, Melbourne, 1988.

[X.209] CCITT: Specification of Basic Encoding Rules of Abstract Syntax Notation One (ASN.1). In Open Systems Interconnection [X.200] - Basis Reference Model. International Standard Recommendation X.209, conform with ISO8825, Melbourne, 1988.

[X.219] CCITT: Data communication networks open system interconnection (OSI) Remote operations: model, notation and service definition. In Blue Book Recommendation X.200, Melbourne, November 1988.

[X.680] ITU-T: Data networks and open system communications, OSI networking and system aspects - Abstract Syntax Notation One (ASN.1): Specification of basic notation, ITU-T Recommendation X.680, 1997.

[Z.100] CCITT: SDL - Specification Description Language, International Standard Recommendation Z.100, Genevre,1992.

[Z.105] ITU: SDL in combination with ASN.1, Recommendation Z.105, Genf, 1997.

[Z.130] ITU: ITU Object Definition Language, Recommendation Z.130, Geneva,1999.

# Modeling and Verifying a Temperature Control System using Continuous Action Systems

Ralph-Johan Back Cristina Cerschi

Turku Centre for Computer Science (TUCS),
Lemminkäisenkatu 14 A, FIN-20520, Turku, Finland

**Abstract.** We describe and verify a real-time temperature control system for a nuclear reactor tank, using a generalization of action systems to hybrid systems as our formal framework. The analyzed control system is a linear hybrid system, combining discrete control with continuous dynamics. Our work can be seen as a case study on the applicability of the hybrid action system formalism to study the reachability problem, i.e., to prove that an unsafe state can not be reached by executing the system.

**Keywords** Action systems, Continuous action systems, Forward analysis, Invariance property, Reachability

## 1 Introduction

The safety-critical nature of most hybrid control systems has encouraged work on their formal modeling and verification. The goal is to provide a mathematical proof, which shows that a given model of a system satisfies its safety requirements. In this paper, we apply the generalized model of action systems for hybrid systems, developed by Back, Petre and Porres [3], to formally describe the real-time linear temperature control system inside a nuclear reactor tank. The model allows us to analyze the reachability properties of the system, in particular to prove that by executing the system we can not reach an unsafe state.

The action systems formalism, introduced by Back and Kurki-Suonio [4], is used to model and analyze parallel and distributed systems. The behavior of an action system is essentially that of Dijkstra's guarded iteration statement [6] on the state variables: the initialization statement is executed first, thereafter, as long as there are enabled actions, one action at a time is nondeterministically chosen and executed. The extension of this approach to continuous action systems [3] provides a unified framework for handling both discrete and continuous behavior. The continuous action system can be seen as a collection of time functions, defined in a piecewise manner. This approach lets us model both discrete and continuous functions in the same way, without separating discrete events from continuous laws. The system's formal representation is therefore homogenous, making its behavior easy to understand and reason about.

For general hybrid systems, the hybrid automata analysis methods [1, 2], can be applied with certain limitations. The automated analysis of hybrid systems [1] needs efficient algorithms to represent and approximate state sets. The verification methodology that is based on abstracted automata [14], which has simpler dynamics, faces the inconvenience that the abstractions that

are created depend on the property to be proved. Different specifications may require different abstractions, so proving different properties of the same hybrid system implies creating different abstractions.

This case study demonstrates how our model [3] can be applied to specify and prove safety properties for a temperature control system. This is a typical example of a hybrid control system. We show that we can use the same verification techniques for hybrid systems as for ordinary action systems.

The structure of this paper is as follows. In section 2 we briefly present the basic action system notion. Section 3 introduces the continuous action system approach. In section 4, we describe the temperature control system, using the continuous action system model. We also give the semantics of the system, by translating it into an equivalent (discrete) action system where time is explicitly advanced. Section 5 proves that the undesired state is not reachable, by proving an invariance property for the hybrid system, using the traditional forward analysis technique. Some aspects of the proofs in this section are presented in the Appendix. Section 6 presents an attempt to evaluate the formal model proposed in this paper and the verification technique used, following the discussion in [10]. Conclusions and related work are presented in section 7.

## 2 Action Systems

The *action systems formalism* [4] is a framework for specification and refinement of concurrent programs. It is based on an extended version of the *guarded command language* of Dijkstra [6].

An *action system* is in general a collection of *actions*, or guarded commands, which are executed one at a time. Parallel behavior is modeled by *interleaved* actions, i.e., by two or more actions that can be executed in any order.

An *action system* A is a statement of the form:

$$\mathrm{A} =_{\mathrm{def}} \ |[\ \mathrm{var}\ x : \mathrm{T} \bullet S_0; \mathrm{do}\ A_1 \ \square \ \ldots \ \square \ A_{\mathrm{m}}\ \mathrm{od}\ ]| : y \qquad (\mathbf{1})$$

on state variables (*attributes*) $x \cup y$. The *local* variables $x$ only exist during the execution of the action system. They are first initialized by $S_0$, after which the actions $A_1 \ldots A_{\mathrm{m}}$ are executed repeatedly, as long as one of them is enabled. The *global* variables $y$ exist before and after the execution of the action system. For specifying the state attributes, we define a finite set *Attr* of *attribute names* and assume that each attribute name in *Attr* is associated with a non-empty set of *values*. This set of values is the *type* of the attribute. If the attribute $x$ takes values from *Val*, we say that $x$ has the type *Val* and we write it as $x : Val$. The name and the type completely specify the attribute. We consider several predefined types, like "Real" for the set of real numbers, "$\mathrm{Real}_+$" for the set of non-negative real numbers and "Bool" for the boolean values $\{\mathrm{T}, \mathrm{F}\}$.

An *action A* is of the form

$$A :: = g \rightarrow S,$$

where $g$ is the *guard* of $A$ and $S$ is the statement (*body*) of $A$.

A statement $S$ is defined by the grammar:

$S :: =$ *abort* (abortion, nontermination)

| | | |
|---|---|---|
| | *skip* | (empty statement) |
| | $x := e$ | ((multiple) assignment) |
| | if $g_1 \rightarrow S_1 \square \ldots \square g_m \rightarrow S_m$ fi | (nondeterministic choice) |
| | $S_1; \ldots ; S_m$ | (sequential composition) |

where $S_1,\ldots, S_m$ are statements, $g_1 \ldots g_m$ are predicates (boolean conditions), $x$ a variable or a list of variables, and $e$ an expression or a list of expressions. The action "*abort*" always fails and is used to model disallowed behaviors. Actions can be much more general, but this simple syntax suffices for the purpose of this paper. An action $g \rightarrow S$ is *enabled* if its guard $g$ holds.

The execution of an action system is as follows. The initialization $S_0$ sets the variables to some specific values, using a sequence of assignments. Then enabled actions are repeatedly chosen and executed. The chosen actions will change the values of the variables in a way that is determined by the action body. Two or more actions can be enabled at the same time, in which case one of them is chosen for execution, in a demonically nondeterministic way. The computation terminates when no action is enabled. Termination of an action system means the termination of the control over the system, which means that the state will evolve no more, fixing the final values of the variables forever.

An action system is not usually regarded in isolation, but as a part of a more complex system. The rest of the system (the *environment*) communicates with the action system via shared (imported and exported) variables, referred to as the *global* variables.

A predicate $I$ is an *invariant of the action system* A, if it:

1. *holds* after the initialization, i.e., if

$$true \ \{| S_0 |\} \ I$$

2. *is preserved* by each action $g_i \rightarrow S_i$, i.e., if

$$I \wedge g_i \ \{| S_i |\} \ I, \ i = 1,\ldots,m.$$

Here $p \ \{| S |\} \ q$ denotes the standard partial correctness of statement $S$ with respect to precondition $p$ and postcondition $q$ [7].

## 3 Continuous Action Systems

A *continuous action system* [3] consists of a set of attributes that form the *state* of the system and a set of *actions* that act upon the attributes:

$$C \quad =_{def} \quad |(\text{var } x\text{: T} \bullet S_0 \text{ ; do } g_1 \rightarrow S_1 \square \ldots \square g_m \rightarrow S_m \text{ od })| : y \qquad (2)$$

Here $x = x_1, \ldots, x_n$ are the *controlled attributes* or *program variables* of the system, $S_0$ is a statement that initializes these attributes, while $g_i \rightarrow S_i$, $i = 1, \ldots, m$, are the *actions* of the system. The attributes $y = y_1, \ldots, y_k$ are defined in the environment of the continuous action system. An attribute $x$ is a function of time, where time is assumed to vary over the non-negative real numbers. The value of an attribute can be read and its value can be changed.

Changing the value means assigning to the attribute a new time function that may change the future behavior of the attribute but not its past.

An implicit variable *now* is used to denote the present time and can be referred to in expressions. By using the *now* variable in an expression, we can correlate the behavior of the model with the passage of time. Therefore, this formalism is well-suited for modeling real-time systems.

The initialization statement will set the attributes to some specific functions of time, using a sequence of assignments. The actions will change the values of the attributes in the prescribed way, provided they are enabled. This model allows two or more actions to be enabled at the same time, in which case one of the enabled actions is chosen for execution, in a (demonically) nondeterministic way.

The next time instance when an action is enabled may well be the same as the previous time instance when an action was enabled, i.e., time need not progress between two enabled actions. This allows us to model both discrete computation and continuous behavior in the same framework (a discrete computation does not take any time).

Functions are below described using $\lambda$ - abstraction, and we write *f.x* for the application of function *f* to argument *x*. We explain the meaning of *C* by translating it into an ordinary action system. The continuous action system's (*C*) semantical interpretation is given by the following (discrete) action system $\bar{C}$:

$$
\begin{aligned}
\bar{C} \;=\; |[\ \text{var}\ \ & now\colon \text{Real}_+,\ x\colon \text{Real}_+ \to \text{T} \bullet \\
& now := 0;\ \bar{S}_0;\ \text{N} \\
& \text{do} \\
& \quad \bar{g}_1 \to \bar{S}_1;\ \text{N} \ \square\ \ldots\ \square\ \bar{g}_m \to \bar{S}_m;\ \text{N} \\
& \text{od} \\
]|\ :\ & y,
\end{aligned}
$$

where

$$\text{N} =_{\text{def}}\ now := \text{next}.gg.now$$

and the operation "next" is defined by:

$$\text{next}.gg.t =_{\text{def}} \begin{cases} \min\{t' \geq t \mid gg.t'\} & \text{, if } \exists\ \text{t}' \geq \text{t such that } gg.t' \\ \infty & \text{, otherwise} \end{cases}$$

In $\bar{C}$, the attribute *now* is declared, initialized and updated explicitly. It models the moments of time that are of interest for the system, i.e., the starting time and the succeeding moments when some action is enabled. The value of *now* is updated by the statement "N". In the definition of "next", $gg = \bar{g}_1 \vee \ldots \vee \bar{g}_m$ is the disjunction of all guards of the actions. Thus, the function "next" models the moments of time when at least one action is enabled. Only at these moments can the future behavior of attributes be modified. If no action will be ever enabled, then the second branch of the definition will be followed. In this case the system terminates, i.e., the attributes will evolve forever according to the functions assigned to last. We assume in this paper that the minimum in the definition of "next" always exists, i.e., a *continuous action system* is well-defined when $\min\{t' \geq t \mid gg.t'\}$ is well-defined.

Let us introduce the notation:

$$f / t_0 / g =_{\text{def}} (\lambda t \bullet \text{if } t < t_0 \text{ then } f.t \text{ else } g.t \text{ fi}) \quad (3)$$

Thus, $f / t_0 / g$ behaves as $f$ before $t_0$, and as $g$ after $t_0$.

In $\overline{C}$, the condition $\overline{g_i}$ stands for the application of $g_i$ to *now*. For instance, $x = 0$ denotes $(x = 0).now \equiv (x.now = 0.now) \equiv (x.now) = 0$. An assignment $x_i := e$ in $S_i$ is again understood as denoting the following assignment in $\overline{S_i}$ :

$$x_i := x_i / now / e \quad (4)$$

The statement $\overline{S_i}$ is $S_i$ with these changes. The continuous action system is essentially just a way of defining a collection of time functions $x_1, \ldots, x_n$ over the non-negative reals, in a stepwise manner. The steps form a sequence of intervals $I_0, I_1, I_2, \ldots$, where each interval $I_k$ is either a left closed interval of the form $[t_i \ldots t_{i+1})$ or a closed interval of the form $[t_i, t_i]$, i.e., a point. The action system determines a family of functions $x_1, \ldots, x_n$, which are stepwise defined over this sequence of intervals and points. The extremes of these intervals correspond to the control points of the system where a digital discrete action is performed.

Another important observation regards the possibility of *Zeno behavior*. The definition of a continuous action system proposed in [3] does not guarantee that the sequence of generated intervals will cover all the non-negative reals. They might only cover an initial segment of these. In this case, there is a limit point of time that the action system reaches when the number of iterations reaches infinity. The simple explanation of the behavior of the hybrid system is then not sufficient. However, in that case, we assume that the system is restarted at the limit point, and repeat the process again. This is meaningful if all attribute values converge to a well defined value in the limit. This restart can be carried out as many times as needed. Thus, a continuous action system may have multiple limit points in its execution. The standard action system semantics does not allow multiple limit points, so this is a point where the semantics really has to be extended. For simplicity we will here assume that the system $\overline{C}$ does not exhibit *Zeno behavior*, i.e., that the value of *now* grows without bounds if the action system $\overline{C}$ does not terminate. Thus, a single limit point is sufficient. The absence of Zeno behavior means that the action system will define the values of the attributes for the whole domain of $\text{Real}_+$.

## 4 The Temperature Control System (TCS)

Our example system is taken from a study of hybrid systems using algorithmic techniques, by Alur et al. [1]. The system controls the coolant temperature in a reactor tank by moving two independent control rods. Controlling a nuclear reactor means controlling the multiplication of neutrons in the reactor core. When the control rods (which are made of materials that absorb neutrons) are pulled out of the core, more neutrons are available and the chain reaction speeds up, producing more heat. If they are inserted into the core, more neutrons are absorbed, and the chain reaction slows or stops, reducing the heat.

The goal is to maintain the coolant between the minimum temperature $\theta_m$ and the maximum temperature $\theta_M$. When the temperature reaches its maximum value $\theta_M$ the tank must be refrigerated with one of the rods. The temperature rises at a rate $v_r$ and decreases at rates $v_1$ and $v_2$, depending on which rod is being used. For safety reasons, a rod can be moved again only if

T time units have elapsed since the end of its previous movement. If the temperature of the coolant can not decrease because there is no available rod, a complete shutdown is required.

### 4.1 The Continuous Action System Model

This system can be described as a continuous action system , as follows. The system's variables are :

- $\theta$ that measures the temperature inside the reactor tank
- $x_1$ that measures the time elapsed since the last use of rod1
- $x_2$ that measures the time elapsed since the last use of rod2
- *state* that stores the state of the system.

In order to correlate the execution of an action with the passage of time, we introduce a *clock* variable, which measures the time elapsed since it was set to zero.

The operation

$$reset\ (c) =_{\text{def}}\ c := (\lambda t \cdot t - now) \qquad (\mathbf{5})$$

will reset the clock.

Note that the assignment $c := (\lambda t \cdot t - now)$ in the hybrid action system really stands for

$$c := (\lambda t \bullet \text{if } t < now \text{ then } c.t \text{ else } t - now \text{ fi}) \qquad (\mathbf{6})$$

in the translation of this system to an action system with explicit time.

The continuous action system for describing the temperature control system is as follows:

$TCS = |($ var $state$ : $\text{Real}_+$;
$x_1, x_2, c$ : $\text{Real}_+$;
$\theta$ : Real;
$state := 0$;
$reset(c)$;
$x_1 := (\lambda t \cdot T_1 + c.t)$;
$x_2 := (\lambda t \cdot T_2 + c.t)$;
$\theta := (\lambda t \cdot \theta_0 + v_r * t)$
do {cool with rod1}
$state = 0 \wedge \theta = \theta_M \wedge x_1 \geq T \rightarrow$
$reset\ (c)$;
$\theta := (\lambda t \cdot \theta_M - v_1 * c.t)$;
$state := 1$
□ {release rod1}
$state = 1 \wedge \theta = \theta_m \rightarrow$
$reset\ (c)$; $reset\ (x_1)$;
$\theta := (\lambda t \cdot \theta_m + v_r * c.t)$;
$state := 0$
□ {cool with rod2}

$state = 0 \wedge \theta = \theta_M \wedge x_2 \geq T \rightarrow$
$reset\ (c);$
$\theta := (\lambda t \cdot \theta_M - v_2 * c.t);$
$state := 2$
▯ {release rod2}
$state = 2 \wedge \theta = \theta_m \rightarrow$
$reset\ (c);\ reset\ (x_2);$
$\theta := (\lambda t \cdot \theta_m + v_r * c.t);$
$state := 0$
▯ {shutdown}
$state = 0 \wedge \theta = \theta_M \wedge x_1 < T \wedge x_2 < T \rightarrow$
$abort$
od
)| : $\theta_0, \theta_m, \theta_M, v_r, v_1, v_2, T_1, T_2, T$

Here, $T_1, T_2, \theta_0$ are constants, so that $0 \leq T_1 \leq T$, $0 \leq T_2 \leq T$, $0 \leq \theta_0 \leq \theta_M$.

The system is first initialized to *state*0, the clock is reset and at time point zero, inside *state*0, we have $x_1 := T_1$, $x_2 := T_2$, $\theta := \theta_0$. After this, the system starts evolving by increasing the time point *now* continuously. The first action is enabled when the system has reached the maximum temperature $\theta_M$ and the first rod is available ($x_1 \geq T$). The first action body is then executed: the clock is reset, and the tank is refrigerated with rod1 (the temperature $\theta$ starts decreasing linearly at rate $v_1$), and the system enters *state*1 by a discrete transition. The second action is enabled when the temperature reaches its minimum value $\theta_m$ and $state = 1$. The second action body is then executed: both *clock* variable $c$ and clock $x_1$ (that measures the passed time since the previous movement of rod1) are reset and the system returns to *state*0, where $\theta$ increases linearly at rate $v_r$. Similarly to the first action, action 3 is enabled if the system has reached the maximum temperature $\theta_M$ and the second rod is available ($x_2 \geq T$ : at least T time units have passed since it has been last used). The system then enters *state*2 where the temperature starts decreasing at rate $v_2$. Action 4, being symmetric to action 2, is enabled when the temperature reaches its minimum value $\theta_m$, and this time $state = 2$. The temperature then starts increasing at rate $v_r$. The last action has *abort* as its body, thus expressing that the shutdown state is not desired. It becomes enabled when the system is in *state*0, reaches its maximum temperature ($\theta = \theta_M$) and none of the rods is available: $x_1 < T \wedge x_2 < T$.

### 4.2 The Translated Action System Model

The translated action system $\overline{TCS}$, where time is explicitly advanced is as follows:

$\overline{TCS}$ = |[ var $state : \text{Real}_+ \rightarrow \{0,1,2,3\};$
$x_1, x_2, c : \text{Real}_+ \rightarrow \text{Real}_+;$
$\theta : \text{Real}_+ \rightarrow \text{Real};$
$start, now : \text{Real}_+ \bullet$
$now := 0;$
$state := (\lambda t \cdot 0);$
$c := (\lambda t \cdot t);$

$x_1 := (\lambda t \cdot T_1 + c.t)$;
$x_2 := (\lambda t \cdot T_2 + c.t)$;
$\theta := (\lambda t \cdot \theta_0 + v_r * t)$;
*start* := *now;*
*now* := min $\{t' \geq now \mid gg.t'\}$;
do {cool with rod1}
$state.now = 0 \wedge \theta.now = \theta_M \wedge x_1.now \geq T \rightarrow$
$c := c \,/\, now \,/\, (\lambda t \cdot t - now)$;
$\theta := \theta \,/\, now \,/\, (\lambda t \cdot \theta_M - v_1 * c.t)$;
$state := state \,/\, now \,/\, (\lambda t \cdot 1)$;
*start* := *now;*
*now* := min $\{t' \geq now \mid gg.t'\}$
□ {release rod1}
$state.now = 1 \wedge \theta.now = \theta_m \rightarrow$
$c := c \,/\, now \,/\, (\lambda t \cdot t - now)$;
$x_1 := x_1 \,/\, now \,/\, (\lambda t \cdot t - now)$;
$\theta := \theta \,/\, now \,/\, (\lambda t \cdot \theta_m + v_r * c.t)$;
$state := state \,/\, now \,/\, (\lambda t \cdot 0)$;
*start* := *now;*
*now* := min $\{t' \geq now \mid gg.t'\}$
□ {cool with rod2}
$state.now = 0 \wedge \theta.now = \theta_M \wedge x_2.now \geq T \rightarrow$
$c := c \,/\, now \,/\, (\lambda t \cdot t - now)$;
$\theta := \theta \,/\, now \,/\, (\lambda t \cdot \theta_M - v_2 * c.t)$;
$state := state \,/\, now \,/\, (\lambda t \cdot 2)$;
*start* := *now;*
*now* := min $\{t' \geq now \mid gg.t'\}$
□ {release rod2}
$state.now = 2 \wedge \theta.now = \theta_m \rightarrow$
$c := c \,/\, now \,/\, (\lambda t \cdot t - now)$;
$x_2 := x_2 / now \,/\, (\lambda t \cdot t - now)$;
$\theta := \theta \,/\, now \,/\, (\lambda t \cdot \theta_m + v_r * c.t)$;
$state := state \,/\, now \,/\, (\lambda t \cdot 0)$;
*start* := *now;*
*now* := min $\{t' \geq now \mid gg.t'\}$
□ {shutdown}
$state.now = 0 \wedge \theta.now = \theta_M \wedge x_1.now < T \wedge x_2.now < T \rightarrow$
*abort*
od
]| : $\theta_0$, $\theta_m$, $\theta_M$, $v_r$, $v_1$, $v_2$, $T_1$, $T_2$, $T$

Variable *now* has been explicitly introduced and the assignment *now* := min$\{t' \geq now \mid gg.t'\}$ gives the next time instance when the disjunction of the guards of the actions, $g_1 \vee g_2 \vee g_3 \vee g_4 \vee g_5$, holds (i.e., the next time when at least one guard is true, so some action is enabled).

We have introduced here the variable *start*, which stores the time moment when the system starts evolving in any state, after taking a discrete transition.

## 5 Reachability Verification

We know that if the temperature rises to its maximum and it cannot decrease because no rod is available, a complete shutdown is required. The question is whether the system will ever reach the shutdown state. A state σ' is reachable from the state σ if there is a run of the hybrid system H that starts in σ and ends in σ' [1]. Usually, we want to prove that some bad condition *g* (like the shutdown condition) is not reachable. This we can do by proving that some condition *I* is an invariant of the system, and that $I \Rightarrow \neg g$. As every reachable state satisfies *I*, this then shows that every reachable state satisfies $\neg g$, i.e. a state where *g* holds cannot be reached.

Let $\Delta\theta = \theta_M - \theta_m$. Clearly, the time the coolant needs to increase its temperature from $\theta_m$ to $\theta_M$ is $\tau_r = \Delta\theta / v_r$, and the refrigeration times using rod1 and rod2 are $\tau_1 = \Delta\theta / v_1$ and $\tau_2 = \Delta\theta / v_2$, respectively.

The sequence of heating and refrigeration is shown in Fig. 1:

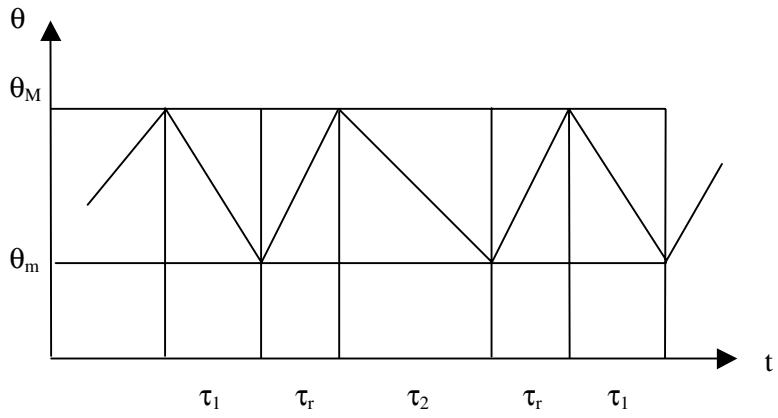



**Fig. 1.** Heating and refrigeration times

Clearly, if $\tau_r \geq T$ (temperature rises at a rate slower than the time of recovery of the rods), then the shutdown state is unreachable. However, this can be a far too strong condition for avoiding the undesired state. Inspecting Fig. 1 we find a weaker condition :

$$2\tau_r + \tau_1 \geq T \ \wedge \ 2\tau_r + \tau_2 \geq T \qquad (\mathbf{7})$$

i.e., if the time between two insertions of the same rod is greater than the time of recovery of the rod, the shutdown state is not reachable. We will prove that this is a sufficient condition for avoiding the undesired *state*3. We therefore make the general assumption that relation (**7**) is true.

In order to prove that formula *I* is an invariant of our system, it is sufficient to prove that

$$true \ \{| \ \overline{S}_0; \ N \ |\} \ I \qquad (\mathbf{8}a)$$

and

$$I \wedge \overline{g}_i \{ | \overline{S_i}; \mathrm{N} | \} I, \ \mathrm{i} = 1,\ldots,5 \qquad (8b)$$

where

$$\mathrm{N} =_{\mathrm{def}} \begin{cases} start := now; \\ now := \min \{t' \geq now \mid gg.t'\}, \end{cases}$$

assigns variable *start* to *now* first, and afterwards sets *now* to the next time instance when the disjunction of the guards of the actions in $\overline{TCS}$, $(gg = \overline{g}_1 \vee \overline{g}_2 \vee \overline{g}_3 \vee \overline{g}_4 \vee \overline{g}_5)$ holds (i.e., the next moment when at least one action is enabled).

### 5.1 Expressing the Invariant with the State chart

Finding the right invariant for proving a safety property is far from being trivial. Therefore, we start by generating the state chart of the temperature control system to get a first approximation of the invariant. Then, we keep adding information to the system's states in order to figure out an invariant strong enough to ensure safety.

The following state chart shows the states that the system can be in, and the properties that hold in each state. It is essentially a hybrid automaton view of the temperature control system.

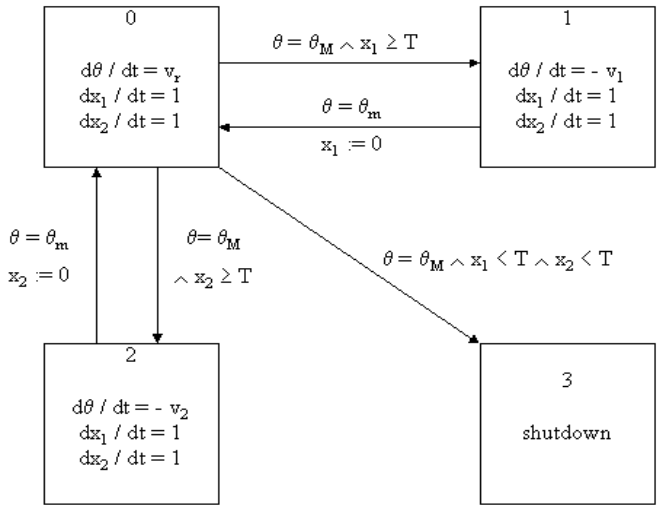



**Fig. 2.** The Temperature Control System's Hybrid Automaton

This figure describes a first invariant of the system. The invariant is the following (expressed in terms of $\overline{\mathrm{TCS}}$):

$$
\begin{aligned}
I =_{\text{def}} (\forall\ t \in [start, now) \bullet\ & \\
& state.start = 0 \Rightarrow state.t = 0\ \wedge \\
& \qquad \mathrm{d}\theta / \mathrm{d}t = v_r \ \text{in}\ [start, now) \wedge \\
& \qquad \mathrm{d}x_1 / \mathrm{d}t = 1 \ \text{in}\ [start, now) \wedge \\
& \qquad \mathrm{d}x_2 / \mathrm{d}t = 1 \ \text{in}\ [start, now) \wedge \\
& \qquad \theta.start = \theta_m \wedge (x_1.start = 0 \vee x_1.start = 0) \\
& \wedge\ state.start = 1 \Rightarrow state.t = 1\ \wedge \\
& \qquad \mathrm{d}\theta / \mathrm{d}t = -v_1 \ \text{in}\ [start, now) \wedge \\
& \qquad \mathrm{d}x_1 / \mathrm{d}t = 1 \ \text{in}\ [start, now) \wedge \\
& \qquad \mathrm{d}x_2 / \mathrm{d}t = 1 \ \text{in}\ [start, now) \wedge \\
& \qquad \theta.start = \theta_M \wedge x_1.start \geq T \\
& \wedge\ state.start = 2 \Rightarrow state.t = 2\ \wedge \\
& \qquad \mathrm{d}\theta / \mathrm{d}t = -v_2 \ \text{in}\ [start, now) \wedge \\
& \qquad \mathrm{d}x_1 / \mathrm{d}t = 1 \ \text{in}\ [start, now) \wedge \\
& \qquad \mathrm{d}x_2 / \mathrm{d}t = 1 \ \text{in}\ [start, now) \wedge \\
& \qquad \theta.start = \theta_M \wedge x_2.start \geq T \\
& \wedge\ state.start = 3 \Rightarrow \theta.start = \theta_M \wedge x_1.start < T \wedge x_2.start < T)
\end{aligned}
\qquad (\mathbf{9})
$$

The invariant thus shows the basic continuous behavior that holds in each state, as well as the discrete transitions.

It is easy to check on the translated form of the temperature control system, that it really has the properties described by the above state chart. By inspecting in $\overline{\text{TCS}}$ the expressions of each action's guard and each action's body, proving that (**9**) holds becomes trivial.

### 5.2 Finding a Stronger Invariant

Although we have extracted a first form of the invariant, we need to find a stronger one, in order to be able to prove that *state*3 is unreachable.

Adding information to the basic state features encapsulated in relation (**9**), leads us to a new invariant. We can add property $\theta \leq \theta_M$ to each of the states 0,1 and 2.

We obtain the following stronger invariant, also expressed in terms of $\overline{\text{TCS}}$:

$$
\begin{aligned}
I'_{\text{def}} = (\forall\ t \in [start, now) \bullet\ & state.start = 0 \Rightarrow \theta.t \leq \theta_M \wedge \\
& \qquad state.t = 0\ \wedge \mathrm{d}\theta / \mathrm{d}t = v_r \ \text{in}\ [start, now) \wedge \\
& \qquad (\mathrm{d}x_1 / \mathrm{d}t = 1, \mathrm{d}x_2 / \mathrm{d}t = 1) \ \text{in}\ [start, now) \wedge \\
& \qquad \theta.start = \theta_m \wedge (x_1.start = 0 \vee x_1.start = 0) \\
& \wedge\ state.start = 1 \Rightarrow \theta.t \leq \theta_M \wedge \\
& \qquad state.t = 1 \wedge \mathrm{d}\theta / \mathrm{d}t = -v_1 \ \text{in}\ [start, now) \wedge \\
& \qquad (\mathrm{d}x_1 / \mathrm{d}t = 1, \mathrm{d}x_2 / \mathrm{d}t = 1) \ \text{in}\ [start, now) \wedge \\
& \qquad \theta.start = \theta_M \wedge x_1.start \geq T \\
& \wedge\ state.start = 2 \Rightarrow \theta.t \leq \theta_M \wedge \\
& \qquad state.t = 2 \wedge \mathrm{d}\theta / \mathrm{d}t = -v_2 \ \text{in}\ [start, now) \wedge \\
& \qquad (\mathrm{d}x_1 / \mathrm{d}t = 1, \mathrm{d}x_2 / \mathrm{d}t = 1) \ \text{in}\ [start, now) \wedge
\end{aligned}
$$

$$\theta.start = \theta_M \wedge x_2.start \geq T$$
$$\wedge\ state.start = 3 \Rightarrow \theta.start = \theta_M \wedge x_1.start < T \wedge x_2.start < T)$$

The enriched state chart is shown in the following figure:

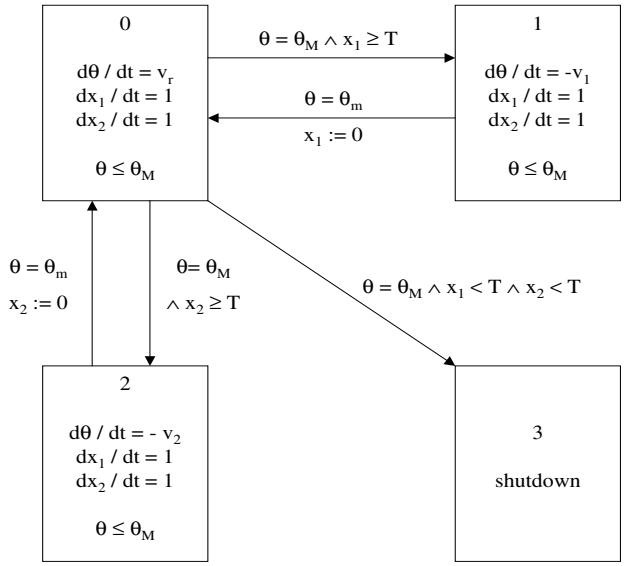



**Fig. 3**. The TCS state chart with the added property, $\theta \leq \theta_M$

Let us show that:

$$I_\theta' \text{ def } = (\forall\ t \in [start, now) \bullet$$
$$state.start = 0 \Rightarrow (\theta.t \leq \theta_M \wedge state.t = 0)$$
$$\wedge\ state.start = 1 \Rightarrow (\theta.t \leq \theta_M \wedge state.t = 1) \qquad (\mathbf{10})$$
$$\wedge\ state.start = 2 \Rightarrow (\theta.t \leq \theta_M \wedge state.t = 2))$$

is a property of the temperature control system.

We apply standard forward analysis technique on the translated model of the temperature control system. Thus, we have to prove that (**10**) is established by the initialization and then that it is preserved by every action. We show here the proofs for the initialization statement and for action 1 (cooling with rod1). The calculation of *gg* for the proofs is presented in the Appendix. We assume $v_1$, $v_2$, $v_r \in R_+\backslash\{0\}$, $\theta_m$ ,$\theta_M \geq 0$ and that the choice of the rod to use as coolant is demonically nondeterministic, in case both rods are available.

*Proof of* (**10**)

(**10**a) *Initialization*

We have to prove that *true* {| $\overline{S}_0$; N |} $I_\theta$' holds. The initialization statement $\overline{S}_0$; N establishes that

$now = 0;$
$state = (\lambda t \cdot 0);$
$c = (\lambda t \cdot t);$
$x_1 = (\lambda t \cdot T_1 + c.t);$
$x_2 = (\lambda t \cdot T_2 + c.t);$
$\theta = (\lambda t \cdot \theta_0 + v_r * t);$
$start = now;$
$now' = \min \{t' \geq now \mid gg.t'\}$

We have to prove that the partial invariant $I_\theta$' is satisfied by these assignments. Thus, we have that

$I_\theta'\ [now'/\ now]$
≡{definition of the invariant}
$(\forall\ t \in [start, now') \bullet\ (\lambda t \cdot 0).start = 0 \Rightarrow (\lambda t \cdot \theta_0 + v_r * t).t \leq \theta_M \wedge (\lambda t \cdot 0).t = 0$
$\wedge\ (\lambda t \cdot 0).start = 1 \Rightarrow (\lambda t \cdot \theta_0 + v_r * t).t \leq \theta_M \wedge (\lambda t \cdot 0).t = 1$
$\wedge\ (\lambda t \cdot 0).start = 2 \Rightarrow (\lambda t \cdot \theta_0 + v_r * t).t \leq \theta_M \wedge (\lambda t \cdot 0).t = 2)$
≡{$start = now = 0, now' = \min \{t' \geq 0 \mid gg.t'\}$}
$(\forall\ t \in [0, now') \bullet (\lambda t \cdot \theta_0 + v_r * t).t \leq \theta_M)$
≡{$now' = \min \{t' \geq 0 \mid \theta_0 + v_r * t' = \theta_M\}$}
$(\forall t \mid 0 \leq t < (\theta_M - \theta_0)/v_r \bullet v_r * t \leq (\theta_M - \theta_0))$
≡{logic}
true

Thus, we have showed that $I_\theta$' holds after the initialization, i.e., that it holds from the moment 0 until the first moment an action is enabled. Next, we shall compute the verification condition for the first action (cooling with rod1) and show that the invariant also holds after this action.

(**10***b*) *Cooling with rod1*

We assume that $I_\theta$' holds on [*start*, *now*), that $\overline{g}_1$ is true and that the local variables have been updated by the assignments of the body of action 1. Thus, we assume that

$I_\theta'$
$\wedge\ state.now = 0 \wedge \theta.now = \theta_M \wedge x_1.now \geq T$
$\wedge\ c' = c\ /\ now\ /\ (\lambda t \cdot t - now)$
$\wedge\ \theta' = \theta\ /\ now\ /\ (\lambda t \cdot \theta_M - v_1 * c.t)$
$\wedge\ state' = state\ /\ now\ /\ (\lambda t \cdot 1)$
$\wedge\ start' = now$
$\wedge\ now' = \min \{t' \geq now \mid gg.t'\}$

We have to prove that the added information in the invariant holds after these assignments. We have that

$I_\theta$' [$c$'/ $c$, $\theta$'/$\theta$, *state*'/ *state*, *start*'/ *start*, *now*'/ *now*]
<= {definition of $I_\theta$'}
($\forall t \mid start' \le t < now' \bullet state'.start' = 0 \Rightarrow (\theta'.t \le \theta_M \wedge state'.t = 0)$
$\wedge\ state'.start' = 1 \Rightarrow (\theta'.t \le \theta_M \wedge state'.t = 1)$
$\wedge\ state'.start' = 2 \Rightarrow (\theta'.t \le \theta_M \wedge state'.t = 2)$
<= {replacing updated variables *state*', $\theta$', $\lambda$-reduction, computing that $now' = now + \tau_1$}
($\forall t \mid now \le t < (now + \tau_1) \bullet 1 = 0 \Rightarrow (\theta_M - v_1 * (t - now) \le \theta_M \wedge 1 = 0)$
$\wedge\ 1 = 1 \Rightarrow (\theta_M - v_1 * (t - now) \le \theta_M \wedge 1 = 1)$
$\wedge\ 1 = 2 \Rightarrow (\theta_M - v_1 * (t - now) \le \theta_M \wedge 1 = 2)$
<= {logic}
($\forall t \mid now \le t < (now + \tau_1) \bullet \theta_M - v_1 * (t - now) \le \theta_M$)
<= {$\theta' = \theta_M - v_1 * (t - now)$ is decreasing starting from $\theta_M$, $v_1 > 0$, $(t - now) \ge 0$}
true

Therefore, we have proved that $I'(I \wedge I_\theta')$ holds after the discrete transition from *state*0 to *state*1 is taken. The proofs for the rest of possible safe transitions are similar, and are omitted here.

### 5.3. Expressing the Final Invariant and Proving the Safety Property of TCS

Our final goal is to provide sufficient assurance that the system evolves on the safe side. This is equivalent to proving that *state*3 can never be reached. Informally, the safety property reduces to proving that in any state, $\theta \le \theta_M$, and also that in *state*0 we always have (at least) one available rod, i.e., that $(x_1 \ge T \vee x_2 \ge T)$.

It follows that even though the invariant we have found is added with some new condition, it is still weak, i.e., it can not ensure safety. Thus, we need to keep on adding information until we reach a strong enough invariant. Clearly, the information that is missing regards the clocks $x_1$ and $x_2$, which measure the elapsed time since the last use of rod1 and rod2, respectively.

Reasoning on Fig. 1, we see that we can add more properties to the state chart in Fig. 3., properties that can provide us with enough information for proving that *state*3 doesn't belong to the reachable states of the system.

These properties are:

$$
\begin{aligned}
(\forall\ t \in [start, now) \bullet\ & state.t = 0 \Rightarrow (((x_1.t = t - start \wedge x_2.t \ge \tau_r + \tau_1 + t - start) \\
& \qquad \vee (x_2.t = t - start \wedge x_1.t \ge \tau_r + \tau_2 + t - start)) \\
& \qquad \wedge (now - start = \tau_r)) \\
& \wedge\ state.t = 1 \Rightarrow (x_2.t \ge \tau_r + t - start) \wedge (now - start = \tau_1) \\
& \wedge\ state.t = 2 \Rightarrow (x_1.t \ge \tau_r + t - start) \wedge (now - start = \tau_2) \\
& \wedge\ state.t = 3 \Rightarrow false)
\end{aligned}
\qquad (11)
$$

Beside the added properties regarding $x_1$ and $x_2$, we added some information regarding the time interval in between the initial moment (denoted by *start*) when the system enters a state and the final moment (denoted by *now*) of evolution of the system in the same state, moment that enables a transition to another (reachable) state. These properties also need to be proved, but we are skipping those proofs here.

We add all these new properties to the previous state chart, thus getting the new state chart in Fig. 4.

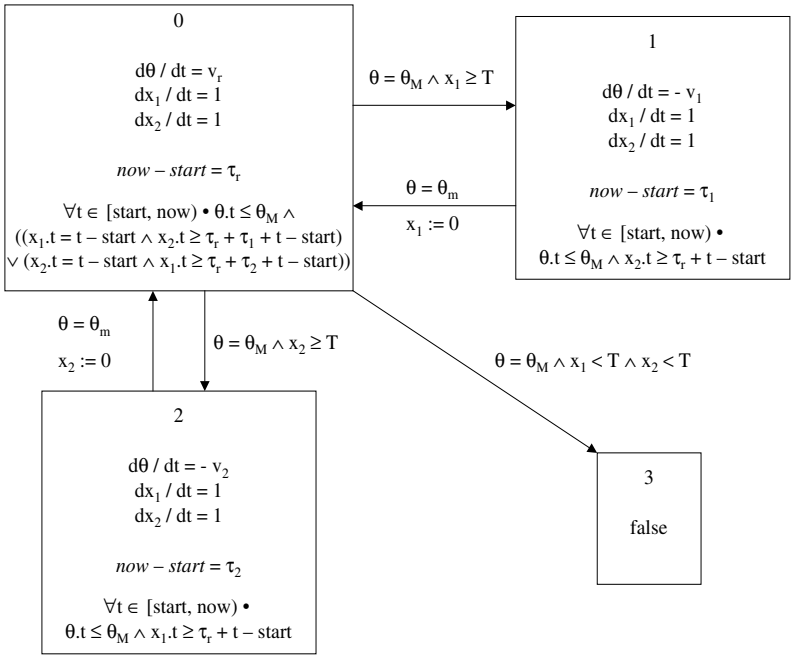



**Fig. 4**. The TCS State chart with added properties about $x_1$, $x_2$

It follows that the final invariant will be *I'* together with condition (**11**). One needs to choose the values of $T_1$ and $T_2$ so that this invariant will hold right from the start. This means that we can have either ($T_1 = \tau_r + \tau_2$ and $T_2 = 0$) or ($T_2 = \tau_r + \tau_1$ and $T_1 = 0$). Even without this choice, the invariant will hold after both rods have been used.

As an example, we are going to prove that condition (**11**) holds for releasing rod1, i.e., after the system has taken a discrete transition from *state*1 to *state*0, thus resetting $x_1$. The proofs for the initialization statement and for cooling with rod1 or rod2 are simpler, thus we are not presenting them in this paper.

We assume that the following properties hold:

$$
\begin{aligned}
&I' \\
&\wedge (\forall\, t \in [start, now) \bullet\ state.start = 0 \Rightarrow (((x_1.t = t - start \wedge x_2.t \geq \tau_r + \tau_1 + t - start) \\
&\qquad\qquad \vee (x_2.t = t - start \wedge x_1.t \geq \tau_r + \tau_2 + t - start)) \\
&\qquad\qquad \wedge (now - start = \tau_r)) \\
&\qquad \wedge state.start = 1 \Rightarrow (x_2.t \geq \tau_r + t - start) \wedge (now - start = \tau_1)
\end{aligned}
$$

$\wedge$ *state.start* = 2 => ($x_1.t \geq \tau_r + t - start$) $\wedge$ ($now - start = \tau_2$)
$\wedge$ *state.start* = 3 => *false*)
$\wedge$ *state.now* = 1 $\wedge$ $\theta.now = \theta_m$
$\wedge$ *c'* = *c* / *now* / ($\lambda t \cdot t$ - *now*)
$\wedge$ $x_1$' = $x_1$ / *now* / ($\lambda t \cdot t$ - *now*)
$\wedge \theta$ ' = $\theta$ / *now* / ($\lambda t \cdot \theta_m + v_r * c.t$)
$\wedge$ *state'* = *state* / *now* / ($\lambda t \cdot 0$)
$\wedge$ *start'* = *now*
$\wedge$ *now'* = min {*t'* $\geq$ *now* | *gg.t'*}

We need to prove that the new information holds after these assignments. We have that

($\forall$ $t \in$ [*start'*, *now'*) • *state'.start'* = 0 => (($x_1$'.*t* = *t* – *start'* $\wedge$ $x_2.t \geq \tau_r + \tau_1 + t - start'$)
$\vee$ ($x_2.t = t - start' \wedge x_1'.t \geq \tau_r + \tau_2 + t - start'$))
$\wedge$ *state'.start'* = 1 => ($x_2.t \geq \tau_r + t - start'$)
$\wedge$ *state'.start'* = 2 => ($x_1'.t \geq \tau_r + t - start'$)
$\wedge$ *state'.start'* = 3 => *false*)
≡ {*start'* = *now*, *state'.now* = 0}
($\forall$ $t \in$ [*now*, *now'*) • ($x_1'.t = t$ - *now* $\wedge$ $x_2.t \geq \tau_r + \tau_1 + t$ –*now*)
$\vee$ ($x_2.t = t$ - *now* $\wedge$ $x_1'.t \geq \tau_r + \tau_2 + t - now$))
≡{substituting for $x_1'.t = t - now$}
($\forall$ $t \in$ [*now*, *now'* ) • $x_2.t \geq \tau_r + \tau_1 + t - now$)
= {*state.start* = 1, so $x_2.t \geq (\tau_r + t - start)$ in [*start*, *now*), so $x_2.now \geq (\tau_r + now - start)$ =>
$x_2.now \geq \tau_r + \tau_1$, $dx_2/dt = 1$ in [*now*, *now'*)}
($\forall$ $t \in$ [*now*, *now'*) • ($x_2.t \geq \tau_r + \tau_1 + (t - now)$)
true

Therefore, the invariant holds after the system has entered state0 from state1.

Action 4 (release rod2) is symmetric to the second action, so the invariant holds after taking the transition from state2 to state0, following the same proof rule.

What is left is to prove that these properties mean that the last action is never enabled, i.e., that

$$I \wedge \bar{g}_5 = \text{false} \qquad (\mathbf{12})$$

As *state*1 and *state*2 are safe states, condition (**12**) reduces to proving that:

($\forall$ $t \in$ [*start*, *now*) • *state.t* = 0 => ((($x_1.t = t - start \wedge x_2.t \geq \tau_r + \tau_1 + t - start$)
$\vee$ ($x_2.t = t - start \wedge x_1.t \geq \tau_r + \tau_2 + t - start$))
$\wedge$ ($now - start = \tau_r$))
$\wedge$ (*state.now* = 0 $\wedge$ $\theta.now = \theta_M \wedge x_1.now < T \wedge x_2.now < T$)
≡ {($now - start = \tau_r$ in *state*0) $\wedge$ (($x_2.now \geq \tau_r + \tau_1 + (now - start) = 2\tau_r + \tau_1$) $\vee$ ($x_2.now \geq \tau_r + \tau_2$
+ ($now - start$) = $2\tau_r + \tau_2$)) $\wedge$ (assumption (**7**): $2\tau_r + \tau_1 \geq T \wedge 2\tau_r + \tau_1 \geq T$ ) => ($x_1.now \geq T \vee$
$x_2.now \geq T$)}
*false*

Therefore, conditions (**8**a,b) are met, implying that, under the $2\tau_r + \tau_1 \geq T \wedge 2\tau_r + \tau_2 \geq T$ assumption, the undesired shutdown state is not reachable.

## 6 Discussion

Following the discussion section in [10], we attempt to evaluate the formal framework we have used for modeling and verifying the safety-critical system studied here.

*Are the formal descriptions easy to understand*? The continuous action system model [3] offers an intuitive representation of a hybrid system, resembling an implementation of the system in a programming language. The requirements specifications look natural enough, when expressed as guarded statements, thus making the behavior of the system easily understandable. When shifting to the representation of the system where time is explicitly advanced, one does not have to rewrite the specifications, but just replace the variables with implicit time with the same variables with explicit time. This translated representation gives the semantics of the system. We used traditional forward analysis technique for verification purposes, rather than backward (weakest precondition) analysis, as the former can be easier to follow. We are not concerned here with termination properties but just with proving a safety property.

*How hard is it to construct a proof using this method*? Forward analysis as the verification method used in this paper, though not difficult, required some work. The hardest part was finding the right invariant for proving the safety property. Even though model-checking lets practitioners check automatically whether a given model of the system satisfies certain properties, it is not that powerful used alone, as it only verifies whether a subfamily of solutions satisfy those properties of interest. Carrying out a hand proof requires the ability to do formal proofs, but on the other hand the kind of proofs developed with the method used in this paper are fit for mechanical proof checking. In practice, interactive theorem provers (like HOL) are needed for automated support.

*Does the proof yield information other than just the fact that the model of the system is correct*? Yes. The invariant and the forward analysis itself as the verification method, even though require considerable effort, provide useful insight about the behavior of the system. Our approach allows references to historical values of the attributes in guards and expressions. In contrast, model-checking methods provide only an assertion that the implementation satisfies the desired properties [10].

*Does the formalism scale up to handle larger systems*? This is the big question. Quite a lot of automated support is needed for the method to be practical. Future work includes case studies done on larger systems. The complexity of the system adds complexity to the invariants. Parallel composition techniques can help in analyzing a larger system and further decomposition of the problem can simplify finding the proper invariant.

*How easy is it to modify the model of the system and the proofs*? Changing the specifications implies changing the proofs. Using an interactive theorem prover might help in uncovering the parts that need to be changed by rerunning the proofs quickly.

## 7 Conclusions and Related Work

Modeling and verification of hybrid systems require a rigorous formal framework, with proof techniques that are able to handle both discrete computing and continuous behavior. For more information about other formalisms developed for real-time systems, the reader is referred to [9].

In this paper we applied the generalized hybrid action system model to formalize and verify a hybrid system for temperature control, giving a formal proof for a safety property by using the same verification techniques as for ordinary action systems. Our approach, the hybrid action systems model [3], offers a homogenous and intuitive representation of a hybrid system, resembling an implementation of the system in a programming language. This makes the system's behavior easy to understand, in spite of its hybrid nature that might involve complex dynamics.

The way of reasoning presented in this paper starts from the continuous action system model and its translation into an ordinary action system. Then it continues with extracting a first approximation of the invariant from the state chart representation of the system (or its hybrid automaton) and afterwards it progressively adds properties for strengthening the invariant in order to ensure the system's safety. The verification method used is the classical forward analysis technique. Thus, the approach used for this case study lets us reason about both linear and nonlinear functions, without separating the discrete events from the continuous laws.

Reachability analysis for hybrid systems represents one of the most important and difficult problems to handle. In [1, 2], Alur, Henzinger et al. apply algorithmic analysis techniques using hybrid automata, techniques based on constructing the reachable region of linear hybrid systems, providing decidability and undecidability results for classes of linear hybrid systems (see also [13]). For general hybrid systems, the hybrid automata analysis methods can be applied with certain limitations. In [1], they perform symbolic model-checking for timed automata (introduced in [11]), illustrated on the temperature control system, using KRONOS to compute the characteristic set of state predicates, therefore using different values for the parameters. In this paper we give a general mathematical proof. As emphasized in section 6, the proof technique used in this paper, i.e, traditional forward analysis, is amenable to mechanical proof checking. In contrast to the model-checking technique used in [1], which provides only an assertion that the model of the temperature control system satisfies a safety property, our approach, i.e., proving an invariance property, offers useful key insights about the behavior of the system. In practice, this feature can contribute to the design stage of a new system, as adding information to the system's states might suggest adding design details.

In cases when the reachability construction fails, the reachability verification method is applied [12]. The user has first to guess (heuristically) the reachable region and then verify that the guess is correct. The method is almost fully automated (there are no automated guess heuristics), but in case that the guessed region is not directly inductive, new variables and constraints have to be added, making the method more complicated.

Our approach, the generalized action system formalism [3] is suited for modeling and verification of mission-critical systems, as it allows for the explicit failure of the system (modeled by the "abort" statement) and also allows references to historical values of the attributes in guards and expressions. These make our formalism more expressive than hybrid automata [1].

The hybrid constraint (language) approach (Hybrid cc) [8] requires the user to be able to express as constraints various aspects of the given hybrid automaton. This means that a

constraint system is needed that is expressive enough, a requirement sometimes difficult to satisfy.

The verification methodology based on abstracted automata, developed by Puri and Varaiya [14], which has simpler dynamics, faces the inconvenience that the abstractions that are created depend on the property to be proved. Different specifications may require different abstractions, so proving different properties of the same hybrid system implies creating different abstractions.

The strong point of our approach is that it allows almost any type of function in the dynamic laws characterizing the continuous behavior, compared to Rönkkö's and Li's linear hybrid action systems [15], where only smooth functions (without discontinuities) can be handled. Due to atomicity, the kind of action systems that were considered in [15] cannot model hybrid systems where continuous steps have nondeterministic ending time. In the presented approach, by defining the hybrid action system as a collection of piecewise time functions, we are allowed to also reason about functions with nondeterministic ending time. The approach introduced in [15] doesn't have an implicit notion of time and it is not intended to model real-time systems, whereas our model facilitates the description of real-time systems.

Future work involves looking at refinement of hybrid systems, based on the refinement calculus techniques [5], and analyzing hybrid action systems with interactive control.

## Acknowledgements

The authors would like to thank Luigia Petre, Ivan Porres Paltor and Mauno Rönkkö for their valuable comments on this case study.

## References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The Algorithmic Analysis of Hybrid Systems. Theoretical Computer Science, (1995) 138:3–34

2. Alur, R., Henzinger, T.A., Ho, P.-H.: Automatic Symbolic Verification of Embedded Systems. In IEEE Transactions on Software Engineering, (1996) 22:181-201

3. Back, R.J., Petre, L., Porres Paltor, I.: Generalizing Action Systems to Hybrid Systems. Turku Centre for Computer Science Technical Reports, No. 307, (1999)

4. Back, R.J.R, Kurki-Suonio, R.: Decentralization of Process Nets with Centralized Control. In the 2nd Symposium on Principles of Distributed Computing, Lecture Notes in Computer Science, Vol. 873. ACM SIGACT-SIGOPS, (1983) 131-142

5. Back, R.J.R., von Wright, J.: Refinement Calculus – A Systematic Introduction. Springer - Verlag, Berlin Heidelberg New York (1998)

6. Dijkstra, E. W.: A Discipline of Programming. Prentice-Hall International, (1976)

7. Gries, D.: The Science of Programming. Springer-Verlag, New York (1981)

8. Gupta, V., Jagadeesan, R., Saraswat, V., Bobrow, D.: Programming in Hybrid Constraint Languages. In Hybrid Systems II, Lecture Notes in Computer Science, Vol. 999. Springer - Verlag, Berlin Heidelberg New York (1995)

9. Heitmeyer, C., Mandrioli, D.: Formal Methods for Real-Time Computing. Balachander Krishnamurthy (ed.), John Wiley & Sons, Chichester New York Brisbane Toronto Singapore (1996)

10. Heitmeyer, C., Lynch, N.: The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems. In Proceedings of the IEEE Real-Time Systems Symposium, San Juan, Puerto Rico (1994) 120-131

11. Henzinger, T., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic Model-Checking for Real-Time Systems. In proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science, (1992) 394-406

12. Henzinger, T., Rusu, V.: Reachability Verification for Hybrid Automata. In Proceedings of the First International Workshop on Hybrid Systems: Computation and Control (HSCC 98), Lecture Notes in Computer Science, Vol. 1386. Springer-Verlag, Berlin Heidelberg New York (1998) 190-204

13. Kopke, P., Henzinger, T., Puri, A., Varaiya, P.: What's Decidable About Hybrid Automata? In Proceedings of the 27th Annual ACM Symposium on Theory of Computing, STOC'95 (1995) 373-382

14. Puri, A., Varaiya, P.: Verification of Hybrid Systems using Abstractions. Hybrid Systems II, Lecture Notes in Computer Science, Vol. 999. Springer-Verlag, Berlin Heidelberg New York, (1995) 359-369

15. Rönkkö, M., Li, H.: Linear Hybrid Action Systems. Turku Centre for Computer Science Technical Reports, No. 245, (1999)

## Appendix Calculation of *gg* in Proving Relations (10a), (10b) and (11)

$gg.t'$
$\equiv (\lambda t \cdot state.t = 0 \wedge \theta.t = \theta_M \wedge x_1.t \geq T).t' \vee$
$(\lambda t \cdot state.t = 1 \wedge \theta.t = \theta_m).t' \vee$
$(\lambda t \cdot state.t = 0 \wedge \theta.t = \theta_M \wedge x_2.t \geq T).t' \vee$
$(\lambda t \cdot state.t = 2 \wedge \theta.t = \theta_m).t' \vee$
$(\lambda t \cdot state.t = 0 \wedge \theta.t = \theta_M \wedge x_1.t < T \wedge x_2.t < T).t'$

*In Proof of* (**10**a)

$gg.t'[\theta := (\lambda t \cdot \theta_0 + v_r * t)]$
$\equiv$
$gg[\theta := (\lambda t \cdot \theta_0 + v_r * t)].t'$
$\equiv (state.t' = 0 \wedge (\lambda t \cdot \theta_0 + v_r * t).t' = \theta_M \wedge x_1.t' \geq T) \vee$
$(state.t' = 1 \wedge (\lambda t \cdot \theta_0 + v_r * t).t' = \theta_m) \vee$
$(state.t' = 0 \wedge (\lambda t \cdot \theta_0 + v_r * t).t' = \theta_M \wedge x_2.t' \geq T) \vee$
$(state.t' = 2 \wedge (\lambda t \cdot \theta_0 + v_r * t).t' = \theta_m) \vee$
$(state.t' = 0 \wedge (\lambda t \cdot \theta_0 + v_r * t).t' = \theta_M \wedge x_1.t' < T \wedge x_2.t' < T)$
$gg[\theta := (\lambda t \cdot \theta_0 + v_r * t), state := (\lambda t \cdot 0)].$
$\equiv \{\lambda\text{-reduction}\}$
$(0 = 0 \wedge \theta_0 + v_r * t' = \theta_M \wedge x_1.t' \geq T) \vee$
$(0 = 1 \wedge \theta_0 + v_r * t' = \theta_m) \vee$
$(0 = 0 \wedge \theta_0 + v_r * t' = \theta_M \wedge x_2.t' \geq T) \vee$
$(0 = 2 \wedge \theta_0 + v_r * t' = \theta_m) \vee$
$(0 = 0 \wedge \theta_0 + v_r * t' = \theta_M \wedge x_1.t' < T \wedge x_2.t' < T)$
$\equiv \{\text{logic}\}$
$(0 = 0 \wedge \theta_0 + v_r * t' = \theta_M)$
$\equiv \{\text{logic}\}$
$\theta_0 + v_r * t' = \theta_M$

*In Proof of* (**10**b)

$gg$ [state $:= (\lambda t \cdot 1)].t'$
$\equiv ((\lambda t \cdot 1).t' = 0 \wedge \theta.t' = \theta_M \wedge x_1.t' \geq T) \vee$
$((\lambda t \cdot 1).t' = 1 \wedge \theta.t' = \theta_m) \vee$
$((\lambda t \cdot 1).t' = 0 \wedge \theta.t' = \theta_M \wedge x_2.t' \geq T) \vee$
$((\lambda t \cdot 1).t' = 2 \wedge \theta.t' = \theta_m) \vee$
$((\lambda t \cdot 1).t' = 0 \wedge \theta.t' = \theta_M \wedge x_1.t' < T \wedge x_2.t' < T)$
$\equiv \{\lambda\text{-reduction}\}$
$(1 = 0 \wedge \theta.t' = \theta_M \wedge x_1.t' \geq T) \vee$
$(1 = 1 \wedge \theta.t' = \theta_m) \vee$

$(1 = 0 \wedge \theta.t' = \theta_M \wedge x_2.t' \geq T) \vee$
$(1 = 2 \wedge \theta.t' = \theta_m) \vee$
$(1 = 0 \wedge \theta.t' = \theta_M \wedge x_1.t' < T \wedge x_2.t' < T)$
$\equiv$ {logic}
$\theta.t' = \theta_m$

*In Proof of* (**11**)

$gg$ [state : = $(\lambda t \cdot 0)$].$t'$
$\equiv$ {similarly to (**10**b)}
$(0 = 0 \wedge \theta.t' = \theta_M \wedge x_1.t' \geq T) \vee$
$(0 = 1 \wedge \theta.t' = \theta_m) \vee$
$(0 = 0 \wedge \theta.t' = \theta_M \wedge x_2.t' \geq T) \vee$
$(0 = 2 \wedge \theta.t' = \theta_m) \vee$
$(0 = 0 \wedge \theta.t' = \theta_M \wedge x_1.t' < T \wedge x_2.t' < T)$
$\equiv$ {logic}
$\theta.t' = \theta_M$

# Modelling and Analysing a Railroad Crossing in a Modular Way

Dirk Beyer, Claus Lewerentz and Heinrich Rust

Software and Systems Engineering Team, Technical University Cottbus
Postfach 10 13 44
D-03013 Cottbus, Germany
Email:{db|cl|rust}@informatik.tu-cottbus.de

**Abstract.** One problem of modelling hybrid systems with existing notations of hybrid automata is that there is no modular structure in the model. We introduce an extended modelling notation which allows the modelling of a system as a hierarchical structure of modules. The modules are capable of communicating through the elements of an explicitly defined interface. The interface consists of signals and variables declared with different access modes. This paper describes a model of the railroad crossing example and how to verify it. The current version of a tool for reachability analysis using the double description method to represent symbolically the sets of reachable configurations is presented.

**Keywords.** Automata, continuous systems, formal verification, real-time systems

## 1 Introduction

The programming of embedded systems which have to fulfil hard real-time requirements is becoming an increasingly important task in different application areas, e.g. in medicine, in transport technology or in production automation. The application of formal methods, i.e. of modelling formalisms and analysis methods having a sound mathematical basis, is expected to lead to the development of systems with less defects via a better understanding of critical system properties (c.f. [Lev95]).

Beyer and Rust presented the modelling notation CTA which allows to model hybrid systems in a modular way [BR98]. It builds on the theoretical basis used in tools like UppAal [BLL+96], Kronos [DOTY96] and HyTech [HHWT95]. In these formalisms and tools, finite automata are used to model the discrete control component of an automaton. Analogous variables which may vary continuously with time are used to model the non-discrete system components of a hybrid system. Component automata of a larger system communicate via CSP-like synchronisation labels (cf. [Hoa85]) and via common variables. Algorithms for the analysis of these kinds of models have been presented in [ACD93] and [HNSY94].

To provide features for modelling and verifying modular hybrid systems we need a new formalism and tool which includes, in difference to the existing formalisms and tools, the following concepts:

- Compositional semantics: By using two predicates for the transition assignments in our formalism we preserve the information we need to define the semantics of a CTA module on the basis of the semantics of its parts.
- Hierarchy: Subsystem descriptions can be grouped. Interfaces and local components are separated.
- Explicit handling of different types of communication signals: It is possible to express explicitly that an event is an input signal for an automaton, an output signal, or a multiply restricted signal.
- It is possible to express explicitly that an analogous variable is accessed by an automaton as output, as input, or that it is multiply restricted, which means that each module has read and write access to that variable.
- Automatic completion of automata for input signals. Input signals are events which an automaton must always admit. If there are configurations of an automaton in which the reaction to an input signal is not defined, it is understood that the automaton enters an error state.
- Replicated subsystem components do not have to be multiply defined. They are instantiated from a common module type.

The differentiation between different roles of signals in an automaton has been used in the definition of IO-automata [LT87] and extended to hybrid systems [LSVW96]. [AH97] use different access modes in interfaces to describe modular hybrid systems, too. They build on reactive modules [AH96] and extend them with continuously changing variables.

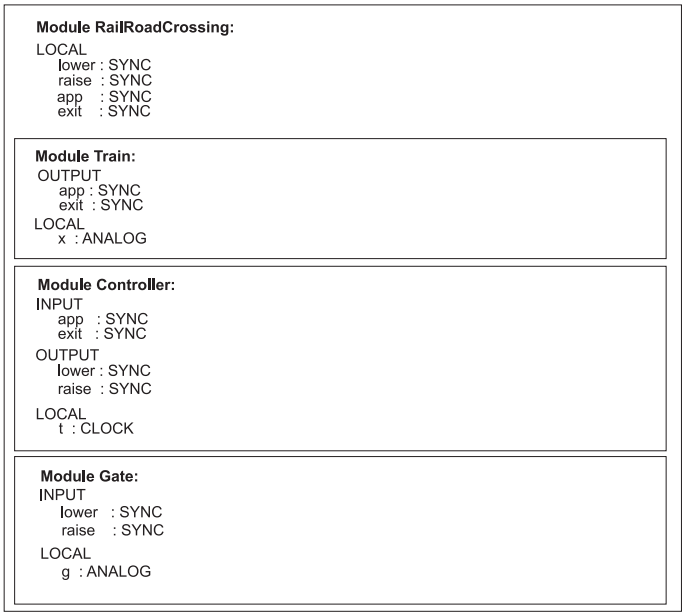



**Fig. 1.** Interfaces of components of the railroad crossing model

The CTA method provides a complete modelling notation and a tool for reachability analysis of those models regarding the features mentioned above.

In section 2 the example of the railroad crossing is described. Section 3 gives an informal description of the formalism. Section 4 gives an overview on the syntax of the modelling language and the analysis commands. Section 5 gives an overview on the CTA tool. The last section explains open questions to be dealt with.

## 2 Model for railroad crossing

### 2.1 The problem

The railroad crossing example deals with the following situation: The gate for a railroad crossing must be closed when the train reaches the crossing. There are two sensors on the railroad: One switching on if the train is at position 1000 m before the gate, and the other sensor is situated 100 m after the gate which indicates that the train has passed the gate. The gate can be moved between the angle 0 degree (which means the gate is closed) and the angle 90 degree (open) by a motor. The problem is to create a controller which fulfils the safety condition that the gate is closed before the train is nearer than in a 250 m distance to the crossing.

### 2.2 A modular, automaton-based approach

Our model consists of three subsystems: The train, the gate, and the controller. The controller coordinates the actions of the other components. The environment of the controller consists of one part for the train behaviour and one part for the gate behaviour.

Fig. 1 displays the structure of the modules: The main module 'RailRoadCrossing' has four synchronisation signals, which are locally defined because the system is modelled as a closed system. 'app' models the sensor at the 1000 m-position and 'exit' models the sensor at the 100 m-position for communication between the train and the controller. The signals 'lower' and 'raise' are used by the controller to control the action of the gate. They all have to be declared in this module because they are used by more than one contained module. The other modules contained in the 'RailRoadCrossing' are described in the following paragraphs.

**Train.** The train component (Fig. 2) models the position of the train (local variable x) and its speed (the time derivation of x). Because the sensor on the railroad which switches on when the train reaches

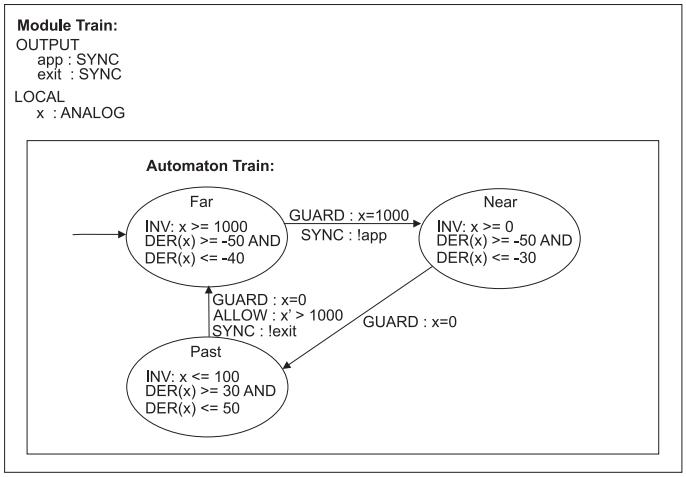



**Fig. 2.** The train model

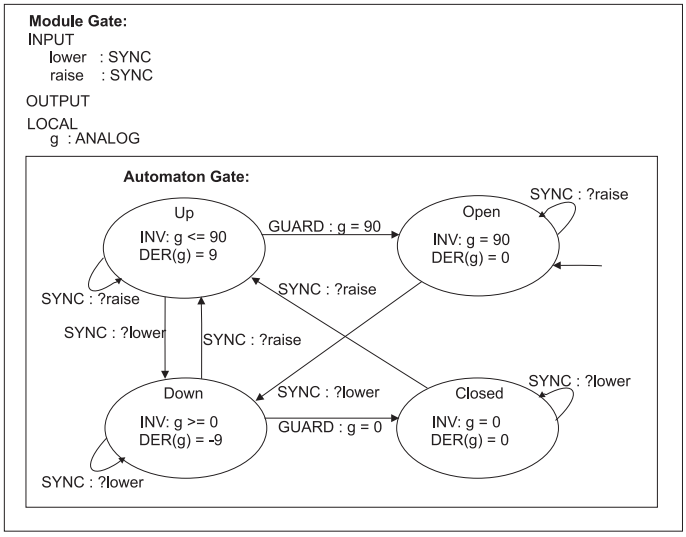



**Fig. 3.** The gate model

the position at 1000 meters distance to the gate is controlled by this module the signal 'app' is declared as 'OUTPUT'. This means that all modules in the environment are only allowed to read the signal. The same is true for the signal 'exit'.

The contained hybrid automaton has three discrete states: To model that the train comes from far away, if the distance to the gate is greater than 1000 m we have the state 'Far'. The invariant of this state is that the position of the train is greater than or equal to 1000. The derivation of 'x' has to be between -50 and -40 to model the possible speed of the train. When the train is passing the first sensor (x=1000) it switches to the state 'Near' by synchronising with the label 'app'. Following the CSP concept it means that all other automata which know this label must also take a transition with this label. The speed is modelled by the derivation of 'x' again and the invariant with the corresponding guard at the transition models that the train passes the gate, and the automaton switches to the state 'Past'. After another 100 m the invariant and the guard of the next transition forces the automaton to leave the state by firing this transition by forcing synchronising with the output signal 'exit'. It also sets the variable 'x' to a value greater than 1000 to be ready for the next cycle in the state 'Far'.

**Gate.** An illustration of the gate model is given in Fig. 3. This component models the angle of the gate by the analogous variable 'g'. The module has to react on the two input signals 'lower' and 'raise'. The

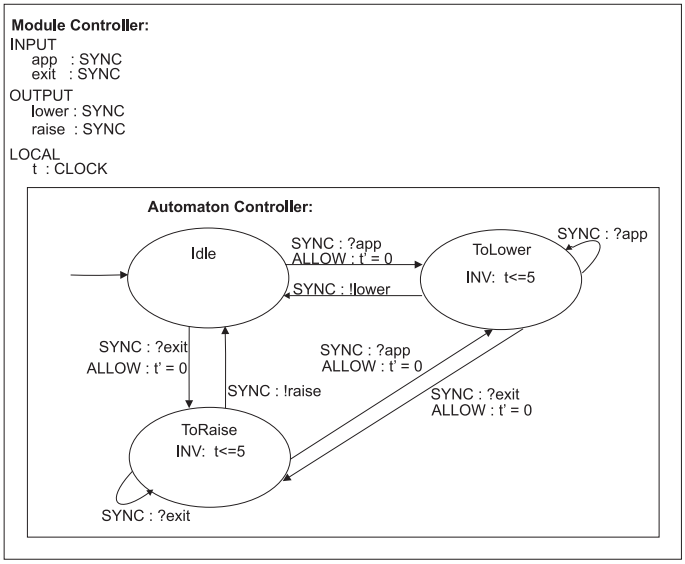



**Fig. 4.** The controller model

automaton starts in state 'Open' (g=90). When it receives the signal 'lower' it switches to the state 'Down', which models the situation that the gate is lowering but has not finished the task. The speed of the angle change is modelled by the derivation of 'g'. If the angle of 0 degrees is reached the automaton has to switch to state 'Closed'. In this state it waits for the signal 'raise', which forces to take the transition to the state 'Up' for starting the opening process. If the angle 90 is reached the automaton switches to the state 'Open' and it can start from beginning.

**Controller.** The controller models the component (the electronic device or the controller software) for the coordination of the other parts of the system (Fig. 4). It has two input signals to react on signals from the 'Train' module and two output signals to control the module 'Gate'. The local clock variable 't' is used to measure the reaction time of the controller.

The automaton starts in the 'Idle' state waiting for the train-is-coming signal from the train (1000 m-position). This signal forces the controller automaton to switch to the state 'ToLower' and the transition resets the clock 't'. That state models the situation that the controller consumes the signal, needs some time to react, and then to perform an action. The upper bound of the reaction time '$\alpha$' (in the example it is set to 5) is an important parameter for satisfying the safety condition of the system. At least after time $\alpha$ it must send the 'lower' signal to the gate and goes back to the 'Idle' state. If the train has passed the 100 m position after the gate then it sends the 'exit' signal to the controller. After some reaction time the controller sends the raise-signal to the gate and after this it is waiting again in state 'Idle'.

## 3 Theoretical background

The following section contains a short description of the CTA formalism. The complete formal definition of the CTA semantics is given in [BR99].

### 3.1 Formalism for modelling

A CTA system description consists of a set of modules. One of them is designated as the top module. It models the whole system. The other modules are used as templates. They can be instantiated several times in other modules. This makes it possible to express a hierarchical structure of the system, and to define replicated components of a system just once.

Each module consists of the following components:

– An **identifier.** A system description might contain several modules. Identifiers are used to name them.

– An **interface.** The interface consists of declarations of variables and signals used by the components of the module. It defines the access modes for the variables and signals and the data types for variables.
  - **Signals.** Signals are used for communication between modules running in parallel. Signals are modelled as CSP-like events.
  - **Variables.** Variables are used to model the (predominantly) continuously changing components of a hybrid system. CTA variables are real valued, they may change continuously with time, and they may change discretely.
– A **hybrid automaton.** This automaton consists of the following components:
  - $\mathbf{S}$: A finite set of discrete states.
  - $\mathbf{G}$: A finite set of signals.
  - $\mathbf{V}$: A finite set of analogous variables.
    Variables are used in **value assignments**. A value assignment for a set of variables is a member of the set of functions $A(V) = V \to \mathbb{R}$.
    At every point in time the situation of an automaton is described by its current discrete state and the current value assignment of all variables of the automaton. A **configuration** $c$ of a CTA is defined as the pair $c = (s, a)$, where $s$ is the current discrete state and $a$ is the current value assignment ($c \in C = S \times A(V)$). A **region** $r \in R = \mathcal{P}(C)$ is a set of configurations.
  - $\mathbf{I} \subseteq \mathbf{C}$: An initial condition, described as a region.
  - $\mathbf{T}$: A finite set of transitions.
  - **inv** : $S \to 2^{A(V)}$: A function associating an invariant to each state. The invariant is a set of value assignments. As long as the invariant of a state is true, the system may stay in the state. It may leave the state earlier, but the latest moment is just after the invariant has become false.
  - **deriv** : $S \to 2^{A(V')}$: A function associating a set of admissible derivatives to each state. In the finite set of variables $V'$ there is, for each variable $v \in V$, a corresponding element $v'$ which is used to define admissible time derivatives of the variable $v$. While the automaton remains in a state, the continuous changes of a variable $v$ are defined by their first time derivative $v'$.
  - **trans** : $T \to S \times S$: A function associating a starting state and a target state to each transition.
  - **guard** : $T \to 2^{A(V)}$: A function associating a guard with each transition. The guard is a set of value assignments. One condition for the transition to be taken is that the guard is true.
  - **sync** : $T \to G \cup \{*\}$: A function associating a signal or no signal to each transition. $*$ is not a signal; it is the value of sync$(t)$ for transitions without a signal.
  - **allowed** : $T \to (A(V) \to 2^{A(V)})$: A function associating with each transition a function which transforms a value assignment into one of a set of value assignments. The aim of this function is to restrict changes which may occur in any environment.
  - **initiated** : $T \to (A(V) \to 2^{A(V)})$ with $\forall t \in T, a \in A(V)$ : initiated(t)(a) $\subseteq$ allowed(t)(a): A function associating with each transition a function which transforms a value assignment into one of a set of value assignments. In contrast to 'allowed' the aim of this function is to restrict changes which occur *without* an environment.
    For each $t \in T$ and $a \in$ guard$(t)$, the set initiated$(t)(a)$ must be nonempty. This condition ensures that the 'initiated' or 'allowed' component can not inhibit a discrete transition to be taken.
    A further restrictions is: For each $s \in S$, there is an element $t_s$ of $T$ with the following properties:
    * trans$(t_s) = (s, s)$
    * guard$(t_s) =$ true
    * sync$(t_s) = *$
    * $\forall a \in A(V)$ : initiated$(t_s)(a) = \{a\}$
    * $\forall a \in A(V)$ : allowed$(t_s)(a) = A(V)$

    These transitions are no-op transitions. The subset of $T$ consisting of all no-op transitions is referred to by 'noop'. The 'allowed' function of no-op transitions does not exclude any resulting value, and 'initiated' defines that no variable value changes. □
– **Instances.** A module may contain instances of previously defined modules. This is used to model systems containing subsystems, and it is especially helpful if a subsystem occurs several times in a system. An instance consists of the following components:
  - An **identifier** is used to give a name to the instance.
  - A reference to a **module** defines which module is instantiated.
  - An **unification** of interface components of the instantiated module with declared components of the containing module defines how the instance is connected to the containing module. This may connect interface signals and interface variables of the instantiated module to signals and variables of the containing module.

**Notational convention:** A typical case to use the 'initiated' predicate is to restrict variables which are not restricted by any parallel transition. If there is a variable which does not occur ticked in at least one inequation, then this does not mean that the whole range of $\mathbb{R}$ is possible. For a variable which does not occur ticked in an inequation the meaning is that this transition does not change the value of the variable. Transitions of environmental automata are allowed to restrict the variable. But if no automaton restricts the variable $x$ in its transition in the same point in time, then we use the information of the 'initiated' set which contains typically the additional restriction $x' = x$. To express that the whole range of $\mathbb{R}$ is possible for $x$ after a transition, one might use the clause $x' > 0$ OR $x' \leq 0$.

In our notation we only use the typical case described above. Perhaps there are other useful aspects for a more general use of the 'initiated' set, but we did not yet find them, and thus we restrict our notation to have an easy to use syntax. Thus, in a syntactical INITIATE clause would be only restrictions of the form $x' = x$ for each variable $x \in X$ (set of variables known by the automaton), if $x$ does not occur ticked in any inequation of the syntactical clause for 'allow'. The consequence for us is to generate the 'initiated' set automatically, i. e. we have not a syntactical clause for 'initiate' in our notation.

Beside the additional concepts, the main difference to existing formalisms are the two assignments 'allowed' and 'initiated' and the semantics of the invariant. The splitting of the assignments leads to a compositional semantics. The fact that an invalid invariant not leads to a forbidden state but that a discrete transition is forced has the following advantage: If the invariant of a state becomes invalid because another automaton changes the values of the variables then the automaton can immediately fire a discrete transition to another state.

In the CTA formalism each of the interface components has an **restriction type** to control the access on the component. There are four different restriction types for variables and signals:

- **INPUT** The declaration of a variable as input variable for a module means that this module can only read this variable:
  - The derivation for an input variable may not be restricted in the deriv-set of any state of the automaton.
  - The value of an input variable after a transition may not be restricted in the allowed-set of any value assignment in a transition.
  - In difference to 'allowed', input variables can be restricted in 'initiated' to reflect that the value of the input variable is not changed by this automaton.

  For a signal the declaration as input means the following: For each input signal and each state of the automaton, some transition labelled with the signal can always be taken. In this way the automaton does not restrict the input signal and thus it is not to blame for a time deadlock. Thus, it is a guarantee for the environment that the module do not change that component .
- **OUTPUT** the declaration of a variable or signal as OUTPUT is an assumption, that the variable or signal is used only as INPUT in all other modules in the environment.
- **MULTREST** The multiply restricted components are available for all access modes. A module as well as the environment for which a signal or variable is declared as multiply restricted can restrict the component in any way.
- **LOCAL** The declaration of a variable or signal as LOCAL means that it is not visible outside the module and thus no other module can access such a variable or signal.

**Variable restrictions in transitions.** To express nondeterminism the value of a variable after a transition has been performed is selected from a set of possible values. These possible values are described by linear expressions over the variables, which can be denoted with the names for the value before the assignment and with the ticked name for the value of the variable after the assignment (for example $x' \geq 3 * x + 7$).

The **product automaton** construction uses the standard technique for composition of automata, with CSP synchronisation as described in [HHWT95]. We construct the product automaton of two hybrid automata in the following way:

- The new set of **states** is the cross product of the state sets from the two automata.
- The new set of **variables** and the set of **signals** are the union of the corresponding sets from the automata.
- The **initial condition** for the new automaton is the conjunction of initial conditions of the automata.
- The **transition** set is the subset of the cross product from the automata, which consists of the combination of two transitions with the same signals or one of the combined transition is a no-op transition, which do not change the values of the variables and has no synchronisation label.

- **Invariants** and **derivatives** are intersected.
- **Guards** are intersected with the additional condition that the set of allowed assignments must not be empty.
- The new **signal** of a transition is the signal used by one of the parallel transitions.
- The new set of **allowed assignments** is the intersection of the two sets of allowed assignments.

The **semantics of CTA** is understood as a labelled transition system.

**Notation.** For a real $u$ and two value assignments $a$ and $a' \in A(V)$, let $u * a$ denote the function $\lambda(v : V) : u * a(v)$, and let $a + a'$ denote the function $\lambda(v : V) : a(v) + a'(v)$.

time : $C \times \mathbb{R} \times A(V) \to C$ is a function describing how the passage of some time $u$ changes a configuration $(s, a)$ when a time derivative $d \in A(V)$ for the variable values is fixed:

$$\text{time}((s, a), u, d) = (s, a + u * d)$$

□

A hybrid automaton can perform time transitions and discrete transitions.

**Definition 1. (Time transitions and discrete transitions of a hybrid automaton)** Let $\mathcal{H}$ be a hybrid automaton.

time($\mathcal{H}$) is the set of **time transitions of** $\mathcal{H}$. It is defined as the following set:

$$\begin{array}{l}\{\quad((s, a_1), (s, a_2)) \in C \times C \\ \quad\mid \exists d \in \text{deriv}(s), u \in \mathbb{R}, u > 0 : \\ \qquad(\quad (s, a_2) = \text{time}((s, a_1), u, d) \\ \qquad\wedge\quad \forall(u' : 0 \le u' \le u) :\quad \text{time}((s, a_1), u', d) \in \text{inv}(s) \\ \qquad) \\ \}\end{array}$$

discrete($\mathcal{H}$) is the set of **discrete transitions of** $\mathcal{H}$. It is defined as the following set:

$$\begin{array}{l}\{\quad((s_1, v_1), (s_2, v_2)) \\ \quad\mid \exists(t : T) : \\ \qquad(\quad \text{trans}(t) = (s_1, s_2) \\ \qquad\wedge\quad v_1 \in \text{guard}(t) \\ \qquad\wedge\quad v_2 \in \text{initiated}(t)(v_1) \\ \qquad) \\ \}\end{array}$$

□

*Note.* For discrete transitions the invariant is irrelevant. An invariant which is identically false can be used to construct urgent states, i. e. states in which time cannot pass.

The state component of the configuration may not change in a time transition.

In the definition of the set of discrete transitions, the signals and the 'allowed' function are not used. Their meaning is defined later, when we consider the parallel composition of automata. □

We will define a transition system semantics for hybrid automata.

**Definition 2. (Transition system)** A **transition system** consists of the following components:

- A (possibly infinite) set $S$ of states, with a subset $S_0$ of initial states.
- A set $T \subseteq S \times S$ of transitions.

□

**Notation.** We use the point notation $A.x$ to address the component $x$ of $A$. □

The transition system corresponding to a hybrid automaton is defined in the following way:

**Definition 3.** Let $\mathcal{H}$ be a hybrid automaton. The **transition system** ts($\mathcal{H}$) corresponding to $\mathcal{H}$ is defined in the following way:

```
Region reachable := r;
WHILE (post(reachable) \ reachable ≠ ∅)
    reachable := reachable ∪ post(reachable);
```

**Fig. 5.** Algorithm for fixed point computation of reachable regions

- $\text{ts}(\mathcal{H}).S =_{\text{def}} \mathcal{H}.C$.
- $\text{ts}(\mathcal{H}).S_0 =_{\text{def}} \mathcal{H}.I$.
- $\text{ts}(\mathcal{H}).T =_{\text{def}} \text{time}(\mathcal{H}) \cup \text{discrete}(\mathcal{H})$.

□

*Note.* The state space of the transition system consists of the configurations of the hybrid automaton. The set of starting states in the transition system is defined via the initial condition of the hybrid automaton. The transitions of the transition system are all time transitions and all discrete transitions of the transition system. □

### 3.2 Reachability analysis

System properties which are to be proved in the verification are described as expressions containing reachability operators.

The most important operators for reachability analysis are the operator $post(c)$ for the region of all the configurations reachable from configuration $c$ by using a time transition or discrete transition, and the operator $pre(c)$ for the configurations from which $c$ is reachable in one time transition or one discrete transition. To handle regions as arguments, $post(r)$ is defined for a region $r$ as $\bigcup_{c \in r} post(c)$ and $pre(r)$ is defined as $\bigcup_{c \in r} pre(c)$.

Using the algorithm in Fig. 5 we can define another operator $reachable(r)$ as notation for the fixed point of collecting reachable configurations starting with the region $r$. For backward reachability one can define an analogous reachability operator for $pre(c)$.

For the railroad crossing in the example the safety condition which the modelled system has to fulfil is: '$error \cap reachable(initial) = \emptyset$', where *error* is the region where the gate is not closed and the train is within the 250 m distance to the gate, and *initial* is the starting region of the system where the train is far away (and x>=1000), the controller is in the idle-state, and the gate is open (and g=90).

In general, the algorithm for the fixed point computation does not necessarily terminate after a finite number of steps, but in the example it does so.

### 3.3 Abstraction and implementation relation

The CTA modelling language is capable of modelling a modular system by using a composition hierarchy. Several different levels of such a hierarchy can be used to express different abstraction levels of the system.

In each refinement step of a modelling process working in a top-down manner the more abstract modules are successively replaced by more specific modules with a deeper hierarchy or a more specific behaviour. In Fig. 6 an embedded system consists of an environment and a controller. There are two different versions of each component: an abstract module and a detailed module.

The aim of the modularity concept is not only to have a nice modelling technique but to have a technique for 'modular verification', too. Many real software systems are very large, so that a reachability analysis even with use of symbolic representation is not possible because of time and space complexity.

One solution for this problem is to use modular proving methods. For example, there is a system implementation which consists of two system components named CONTROLLER_IMPLEMENTATION and ENVIRONMENT_IMPLEMENTATION (denoted as CONTR_IMPL || ENV_IMPL) and we have to prove the safety property P. If the whole system is too complex for automatic analysis, it might be possible to prove the properties with the system CONTR_IMPL || ENV_ABST where ENV_ABST is a more abstract model of the environment than ENV_IMPL. Now, we can use for proving that the safety property P of the system CONTR_IMPL || ENV_ABST is valid if the following two proofs are valid:

- System CONTR_IMPL || ENV_ABST has safety property P and

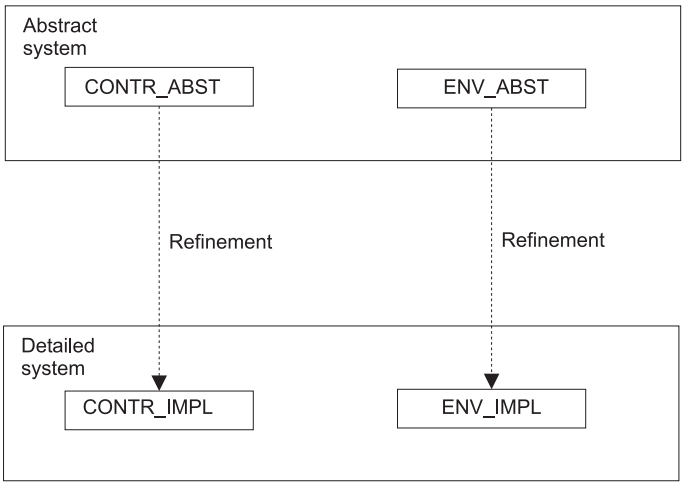



**Fig. 6.** Modelling by Refinement

– ENV_IMPL implements ENV_ABST.

It is expected that these two steps are easier to compute than the complete proof in one step if the abstractions are selected sensibly. For the first step we use reachability analysis and for the second step we use the method described in the following paragraph.

The intuition behind our implementation concept is an assumption/guarantee principle. We describe it with respect to our formalism: An implementation relation ($m_2$ implements $m_1$) for hybrid modules $m_1$ and $m_2$ has to fulfil the following properties ($G$ set of signals, $V$ set of variables, $I$ input, $O$ output, $MR$ multiply restricted):

– $m_1.GI \subseteq m_2.GI$. The occurrence of a signal $g$ as input in a module $m$ means that $m$ guarantees that $g$ is not restricted in $m$. This clearly is a guarantee. Thus each input signal of the specification should be an input signal of the implementation. The same is sensible for input variables $m.VI$.
– $m_2.GO \subseteq m_1.GO$. The occurrence of a signal $g$ as output in a module $m$ means that $m$ assumes that $g$ is not restricted in the environment. The implementation should not make more assumptions than the specification, thus each output signal of the implementation should also be an output signal in the specification. The same is sensible for the output variables $m.VO$.
– $m_1.G - m_1.GL = m_2.G - m_2.GL$. The signals of a module $m$ can be partitioned into a set of interface signals ($m.GI \cup m.GO \cup m.GMR$), and a set of local signals ($m.GL$). Interface signals are those via which $m$ can communicate with the environment. The same is sensible for the variables $m.V$.
– The external trace set $S_2$ (defined below) of the transition system generated by $m_2$ is a subset of the external trace set $S_1$ of the transition system generated by $m_1$. We use the set theoretical conceptualisation of implementation of Abadi and Lamport [AL91]. They use an implementation relation for sets of traces. They consider a set of traces $S_2$ to be an implementation of the set of traces $S_1$ if and only if $S_2 \subseteq S_1$. Their intuition is that the occurrence of a trace $t$ in $S_1$ means that $S_1$ *allows* the system behaviour $t$, and they consider that the implementation should not allow more behaviours than the specification.

Let $L = m.G \cup \mathbb{R}$ be the set of transition labels of the transition system of a module $m$. $m.G$ is the set of synchronisation labels of the module $m$. $\mathbb{R}$ is used for the time values. A **trace** of a given transition system is an infinite alternating sequence $< c_0, l_1, c_1, \ldots >$ of configurations and elements of $L$, starting with an initial configuration $c_0$. For each $(c_i, l_{i+1}, c_{i+1})$ the following must hold: If $l_{i+1}$ is a synchronisation label then a discrete transition of the transition system leads from $c_i$ to $c_{i+1}$; otherwise $l_{i+1}$ is an element of $\mathbb{R}$, the time width between the two configurations, and a time transition of the transition system of that time was taken.

To introduce the notion of an external trace for the behaviour visible outside the module we have to define a function $hide : C \rightarrow E$ which maps a configuration which contains also the information about local variables and the discrete state to a non-local configuration which contains only the value assignments

```
MODULE RailRoadCrossing {
  LOCAL
    app  : SYNC;
    exit : SYNC;
    lower: SYNC;
    raise: SYNC;
  INST Process_Train FROM Train WITH {
    app AS app;
    exit AS exit;
  }
  INST Process_Gate FROM Gate WITH {
    lower AS lower;
    raise AS raise;
  }
  INST Process_Controller FROM Controller WITH {
    app AS app;
    exit AS exit;
    lower AS lower;
    raise AS raise;
  }
}
```

**Fig. 7.** The model of the rail road crossing system

of the interface variables (input, output, and multiply restricted) by hiding the local components ($C = S \times A(VI \cup VO \cup VMR \cup VL); E = A(VI \cup VO \cup VMR)$).

An **external trace** of a given transition system is an infinite alternating sequence $< e_0, l_1, e_1, \ldots >$ of non-local configurations and elements of $L$ which is constructed by the $hide$ function from a trace of the transition system:
$< \ldots, e_i, l_{i+1}, e_{i+1}, \ldots >=< \ldots, hide(c_i), l_{i+1}, hide(c_{i+1}), \ldots >$.

## 4 System description

An introduction to the CTA languages is given in this section. The first subsection describes the language to define the model and the second one describes the language for the analysis commands.

### 4.1 Modelling language

A module contains four components: declaration of variables and signals, an initial configuration of the module, a set of instantiations and a set of hybrid automata.

One of the modules defined in a system description is the so called 'top module' which contains all the stuff needed to model the system. In the top module of the example model all the variables are declared as LOCAL because it is a closed system. (Open system have at least one variable or signal of restriction type INPUT or MULTREST.)

To provide an intuition for the notation, the textual CTA version of the example system is shown in some figures. The top module is the module 'RailRoadCrossing' (Fig. 7), which models the system consisting of the train (Fig. 8), the gate (Fig. 9), and the controller (Fig. 10).

**Declaration of the interface** The modelling language of CTA has two orthogonal type classifications for identifiers. There are four different restriction types to distinguish between different access modes:

- Local variables or signals are not visible outside the current module. They are used for communication between submodules and the automaton of the current module.
- Input variables or signals may not be restricted in the current module. This means for signals that the current module in every configuration must be able to react on such a signal.
- Output components may be restricted in the current module, but not outside the module. This means for signals that the decision when this signal appears is only taken in the current module, not in any other one.
- Multiply restricted variables may be restricted both in the current module and outside it (like normal global components).

```
MODULE Train {
  OUTPUT
    app: SYNC;
    exit: SYNC;
  LOCAL
    x: ANALOG;    // Distance of the train.
  INITIALIZATION {
    STATE(Train) = Far AND x >= 1000; }
  AUTOMATON Train {
    STATE Far {
      INV { x >= 1000; }
      DERIV { DER(x) >= -50 AND DER(x) <= -40; }
      TRANS { GUARD { x = 1000; } SYNC !app; GOTO Near}
    }
    STATE Near {
      INV { x >= 0; }
      DERIV { DER(x) >= -50 AND DER(x) <= -30; }
      TRANS { GUARD { x = 0; } GOTO Past}
    }
    STATE Past {
      INV { x <= 100; }
      DERIV { DER(x) >= 30 AND DER(x) <= 50; }
      TRANS { GUARD { x = 100; } SYNC !exit; DO { x' > 1000; } GOTO Far}
    }
  }
}
```

**Fig. 8.** The train model

```
MODULE Gate {
  LOCAL
    g: ANALOG;    // Degree of the gate.
  INPUT
    lower: SYNC;  // Lower the gate.
    raise: SYNC;  // Raise the gate.
  INITIALIZATION {
    STATE(Gate) = Open AND g = 90; }
  AUTOMATON Gate {
    STATE Open {
      INV { g = 90; }
      DERIV { DER(g) = 0; }
      TRANS { SYNC ?raise; GOTO Open }
      TRANS { SYNC ?lower; GOTO Down }
    }
    STATE Up {
      INV { g <= 90; }
      DERIV { DER(g) = 9; }
      TRANS { SYNC ?raise; GOTO Up }
      TRANS { SYNC ?lower; GOTO Down }
      TRANS { GUARD { g = 90;} GOTO Open }
    }
    STATE Down {
      INV { g >= 0; }
      DERIV { DER(g) = -9; }
      TRANS { GUARD { g= 0;} GOTO Closed }
      TRANS { SYNC ?raise; GOTO Up }
      TRANS { SYNC ?lower; GOTO Down }
    }
    STATE Closed {
      INV { g = 0; }
      DERIV { DER(g) = 0; }
      TRANS { SYNC ?raise; GOTO Up }
      TRANS { SYNC ?lower; GOTO Closed}
    }
  }
}
```

**Fig. 9.** The gate model

For example, the controller in Fig. 10 has two input signals to communicate with the train. They should be declared as input because the train forces the controller to react on the train actions. The controller must be able to synchronise at each point in time with such input signals. For the communication with the gate there are two output signals to control the gate. The declaration of a signal as output means that the environment must be able to react on it. The clock 't' is used to measure the time to react on a signal in the controller.

```
1  MODULE Controller {
2    LOCAL
3      t: CLOCK;    // Timer for the controller.
4    INPUT
5      app: SYNC;
6      exit: SYNC;
7    OUTPUT
8      lower: SYNC;  // Lower the gate.
9      raise: SYNC;  // Raise the gate.
10   INITIALIZATION {
11     STATE(Controller) = Idle; }
12   AUTOMATON Controller {
13     STATE Idle {  // Waiting for a signal from train.
14       TRANS { SYNC ?app;  DO { t' = 0; } GOTO ToLower}
15       TRANS { SYNC ?exit; DO { t' = 0; } GOTO ToRaise}
16     }
17     STATE ToLower {
18       INV { t <= 5; }
19       TRANS {SYNC ?app; GOTO ToLower }
20       TRANS { SYNC !lower; GOTO Idle }
21       TRANS { SYNC ?exit; DO { t' = 0; } GOTO ToRaise }
22     }
23     STATE ToRaise {
24       INV { t <= 5; }
25       TRANS { SYNC ?exit; GOTO ToRaise }
26       TRANS { SYNC !raise; GOTO Idle }
27       TRANS { SYNC ?app; DO { t' = 0; } GOTO ToLower }
28     }
29   }
30 }
```

**Fig. 10.** The controller model

To provide an expressive syntax, there are five different data types, as in HyTech:

- CONST: A variable with a fixed value.
- DISCRETE: A variable which can be used to store a value. The derivation of a discrete variable is always zero.
- CLOCK: A continuous variable with the fixed time derivation one.
- STOPWATCH: A clock which can be stopped. This means the time derivation can be zero or one.
- ANALOG: A continuous variable without restrictions for the time derivation.

The controller model of the railroad crossing has one clock as continuous variable and four signals.

**Instantiations** To get a model for the whole system, the module 'RailRoadCrossing' contains three instantiations of the other modules. Each of the interface variables and signals from the template module has to be identified with one of the variables or signals from the containing module. Local variables or signals are not identified with one of the containing module.

**Automata, states and transitions** A hybrid automaton consists of a set of states. For each state, there is a restriction to set the first time derivations of all the variables and another restriction to define an invariant which must be fulfilled while the automaton stays in the state.

The transitions are syntactically contained in the source state of the transition. A transition consists of a guard, a synchronisation label, a restriction to determine the values of the variables after the transition, and the follower state. Such a discrete transition can only fire if the guard (a restriction over the variables) is fulfilled and all other automata which also use the synchronisation label perform a transition with this label, too.

**Initialisation** The starting state of an automaton and the initial value assignment for the variables are set by the INITIALISATION clause.

**Linear restrictions** Linear restrictions are sets (disjunctions) of inequality systems to restrict the value assignments of the variables. They consist of conjunctions of constraints which use the operators $<$, $<=$,

```
REGION CHECK RailRoadCrossing {
  VAR
    initial, error, reached : REGION;
  COMMANDS
    // We use the initial regions from the modules.
    initial := INITIALREGION;
    error   :=   COMPLEMENT( STATE(Process_Gate.Gate) = Process_Gate.Closed )
                 INTERSECT
                   Process_Train.x < 250;
    reached := REACH FROM initial FORWARD;
    IF( EMPTY(error INTERSECT reached) )  {
      PRINT "Safety requirement satisfied.";
    }
    ELSE {
      PRINT "Safety requirement violated.";
      PRINT "The resulting region:" reached " !";
    };
}
```

**Fig. 11.** The analysis section for the railroad crossing model

$>$, $>=$, $=$, and $<>$. The expressions contained in the constraints must be linear, which means that the expression must be an additive combination of numbers or variables which can be multiplied with a number.

A variable x can occur in three forms in linear restrictions:

- **x** addresses the value of x. In an assignment of a transition it means the value of x before the transition is executed.
- **x'** is used only in assignments of transitions to address the value of the variable after execution.
- **DER(x)** is used only in the DERIV-clause of a state to address the first time derivation of a variable.

**Name spaces** Each module has its own name space. All names of variables and signals are only known to the module. To allow communication between different modules the interface components can be identified with some components of the containing module, respecting consistency conditions. Local components must not occur in an identification of interface components.

A module contains several different name spaces: Automata names, state names, variable and signal names, instantiation names. For the module names only one global name space exists.

### 4.2 Verification language

The analysis language provided by the tool is described in the following section. It is similar to HyTech's analysis language [HHWT95], but extended to handle the different name spaces coming from the hierarchical name space structure.

As in the previous section the example is used for illustration. In Fig. 11 the analysis section for the verification of the railroad crossing model is displayed.

The main idea is to compute with regions. There are variables for storing regions which are needed in the further verification process, and statements to say how to compute the next step.

**Declaration of region variables** As illustrated in Fig. 11 three regions are declared: 'initial' to have the region for the starting point of reachability analysis, 'reached' for the region representing the state space which is reachable from 'initial', and the region variable 'error' for the region which should not appear in a correct model.

**Region expressions** A region expression consists of operations to build regions either from existing regions or from linear restrictions, state restrictions, or region variables.

**Region restrictions** To define a region one can use linear restrictions over the variables of the module on the one hand, e.g. 'Process_Gate.g $>= 0$' to define a region of all configurations which fulfil the condition that the variable 'g' of the instantiation 'Process_Gate' is greater than zero. On the other hand a region can be defined by a state restriction, e.g. 'STATE(Process_Gate.Gate) = Process_Gate.Open' to define the region where the state of the automaton 'Gate' of the instantiation 'Process_Gate' is the state 'Open'.

**Reachability operations** For computation of regions which can be reached by using discrete and time transitions from a given region there are four reachability operations: 'POST(<region>)' and 'PRE(<region>)' to compute the follower (or precursor) region using one time or discrete transition, and 'REACH FROM <region> FORWARD' and 'REACH FROM <region> BACKWARD' to compute the fixed point region using iterative 'POST' (or 'PRE') operations.

**Set operations** To build new regions by set operations the language provides three binary operators and one unary operator: 'INTERSECT' for the intersection of two regions, 'UNION' for the union of two regions, 'DIFFERENCE' for the difference of two regions, and 'COMPLEMENT' for the complement of a region.

**Region variable** A region also can be defined by an existing region which is already computed and stored in a region variable.

**Initial region** To access the initial region defined in the section 'INITIALISATION' of the module description can be used in the analysis section by the keyword 'INITIALREGION'.

**Convention for access to components.** Because of the different name spaces, we use the dot notation to address a special component in a 'Region restriction' in a 'REGION CHECK' section. In the analysis section all variables and all states are available. The current name space is the name space of the top model used for analysis, i.e. to access a component 'x' of the module 'Process_Train' the absolute name $Process\_Train.x$ is used, and the discrete state 'Far' of the automaton 'Train' contained in the module instance 'Process_Train' is accessible by $Process\_Train.Far$.

**Boolean expressions** A Boolean expression is used as a predicate over regions for statements which use a condition to decide what they have to compute. Such a boolean expression consists of comparisons between and checks of regions as well as boolean combinations of such expressions.

**Comparisons** To evaluate set relations between two regions there are the operators '=' which returns true if both regions are the same sets, and the operator 'CONTAINS' which returns true if the first region contains the second one. To check whether a region is empty, one can use the keyword 'EMPTY'.

**Combinations of boolean expressions** Boolean expressions can be combined by the usual operators 'AND', 'OR', and 'NOT'.

**Statements** To formulate the verification tasks different statements can be used. Possible statements are assignments, if-then-statements, loops, and printing states.

**Assignment** One can assign a computed region to a previously declared region variable for further use with an assign statement '<regvar> := <regexpr>'.

**Conditional statement** With the keywords 'IF', 'THEN' and 'ELSE' one can evaluate boolean expressions and depending on the result execute some statements.

**Iteration** To define iterative verification processes a statement 'WHILE' is provided with the expected meaning.

**Output of results** To put messages out on the screen, the statement 'PRINT' is possible with different arguments. The argument of the print statement can be a string, a region expression or a region expression between two strings. A region is printed out as a set of inequalities over the variables and conditions about the discrete states. The print statement of the code in Fig. 11 illustrates it.

The statements in the example define three regions: the initial configurations, the error region (which the model has to avoid), and the region which can be reached starting from the initial region. The 'IF' statement evaluates whether the computed region is empty or not.

## 5 CTA tool environment

The first version of the CTA tool is implemented in C++ and consists of the following components:

- A compiler front-end for the **system description** to build the hierarchy of hybrid modules in memory.
- A library which provides the data structures and algorithms for the double description method (DDM) **symbolic representation of the regions** [FP96].

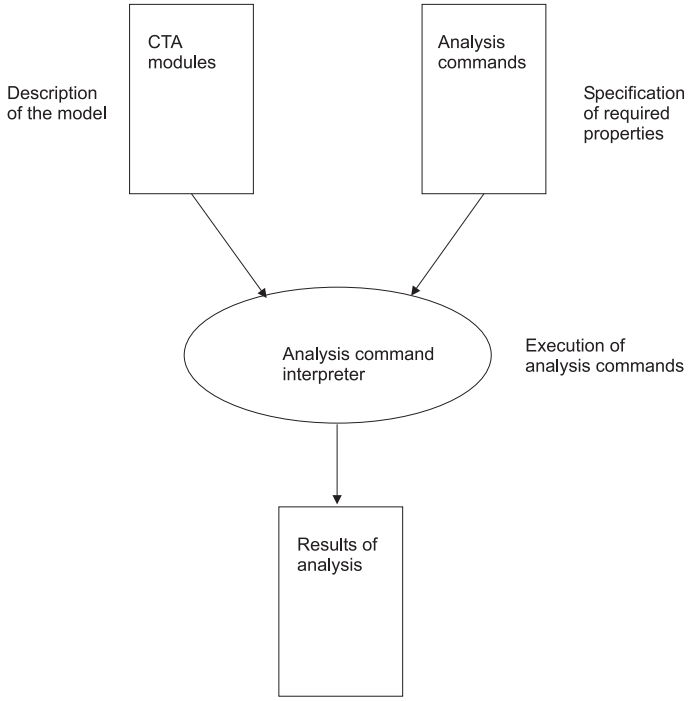



**Fig. 12.** CTA Tool

– An interpreter for the analysis language which provides several analysis commands for **reachability analysis**. It transforms parts of the model needed in the verification into a DDM representation and invokes the analysis tasks on the data structures.

The tool works in the way depicted in Fig. 12: There are two inputs, a description of the model as CTA modules and a specification of required properties as analysis commands. First, the tool analyses the model regarding context conditions like consistency and compatibility of combined modules.

After that, the analysis command interpreter executes the analysis commands and writes out the results of the verification process. To avoid problems with the name spaces, the module which will be used for analysis has to be declared in each region section.

At least for reachability analysis we can only use one single hybrid automaton. From this it follows that we have to transform our hierarchically structured system of communicating modules to a flattened normal form. This normal form of CTA consists of a set of hybrid automata without scopes, special data types and restriction types of variables/signals as well as without abstraction layers as a 'flat' system. This is done with the help of our tool after context check and some additional analysis. The second step is to produce a product automaton from the set of hybrid automata.

If we have not modelled all necessary transitions for reacting on input signals in an automaton, the tool adds these transitions automatically, but they lead to a special error state 'INPUT_ERROR'. If such a state exists, we can analyse if this state is reachable. If it is reachable then we have modelled a situation where an input signal is restricted, and thus we have a modelling mistake.

### 5.1 Region representation

From the real-valuedness of the variables follows the infinity of the set of configurations of any automaton with at least one analogous variable. Thus, for the analysis the tool has to use a symbolic representation of configuration sets. In the CTA tool a region is represented by sets of pairs $(s, a)$, where $s$ is a discrete state and $a$ is a set of convex polyhedra in $\mathbb{R}^{|V|}$, which are limited by hyper-planes. For operations over the polyhedra the tool uses the double description method. A region (as a set of configurations) is represented by a map which assigns to each state the corresponding set of polyhedra (if it is not empty).

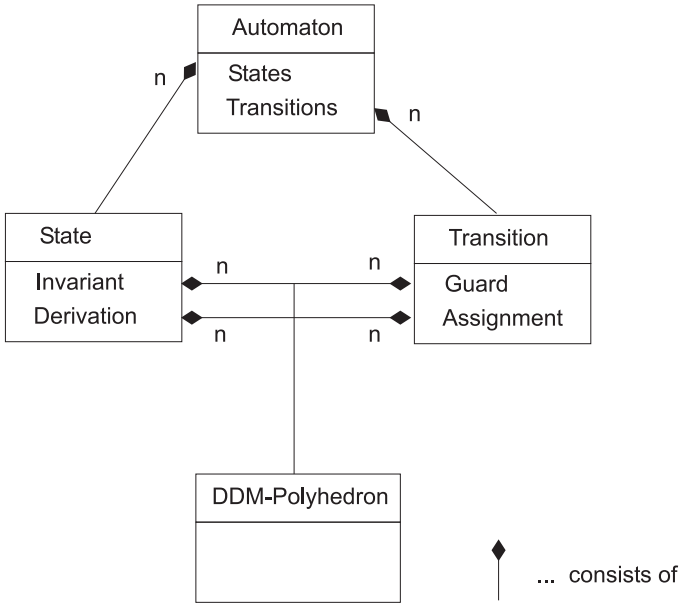



**Fig. 13.** Structure of automaton representation

Fig. 13 shows a rough overview of the internal representation with focus on the main data structure: The polyhedra. One automaton consists of a set of states and a set of transitions. Each state has two important components, one set of polyhedra for the state invariant and another set of polyhedra for the derivation. We have to use sets of polyhedra because the data structure with the efficient algorithms represents only convex polyhedra. Each transition has two such sets, the guard and the allowed assignments.

The component for the polyhedron representation is the most important one because all the algorithms of automaton components are based on the basic algorithms like intersection, unions and emptiness check for such polyhedra.

### 5.2 Results of the verification of the example

For analysis of the railroad crossing example we used the code from Fig. 11. Starting with the initial configuration of all automata the tool computes the region of reachable configurations. The region which violates the safety condition of the example is stored in another region. If the intersection of the reached region and the error region is nonempty, the model violates the condition. Using the model given in former sections the tool computes that the intersection is empty and thus the model fits the safety conditions. After execution of an reachability statement the tool gives a message how many steps were used to reach the fixed point.

The verification task of the example which is displayed in Fig. 11 fails if we increase the reaction time '$\alpha$' of the controller or if we increase the safety distance in which the train must not be before the gate is closed.

## 6 Open Questions

The verification of modelled systems with a lot of combined automata (i.e. more than five automata) is an open problem of the current version of the tool. As in other tools (i.e. HyTech), the technique used in the CTA-tool does not use symbolic representation for the discrete part of a configuration.

In our tool the region representation is separated from the rest of the tool through a clear interface. Thus, we are able to plug in other representation techniques. In the next step the research group is going to investigate the possibilities of BDD data structures for representing continuous state spaces and the discrete state space together.

## Acknowledgements

We thank the anonymous referees for their comments.

## References

[ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Computation*, 104:2–34, 1993.

[AH96] Rajeev Alur and Thomas A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pages 207–218, 1996.

[AH97] Rajeev Alur and Thomas A. Henzinger. Modularity for timed and hybrid systems. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR'97)*, LNCS 1243, pages 74–88, Berlin, 1997. Springer-Verlag.

[AL91] Martin Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[BLL+96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Petersson, and Wang Yi. Uppaal – a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III*, LNCS 1066, pages 232–243, Berlin, 1996. Springer-Verlag.

[BR98] Dirk Beyer and Heinrich Rust. Modeling a production cell as a distributed real-time system with cottbus timed automata. In Hartmut König and Peter Langendörfer, editors, *FBT'98: Formale Beschreibungstechniken für verteilte Systeme*, pages 148–159, June 1998.

[BR99] Dirk Beyer and Heinrich Rust. A formalism for modular modelling of hybrid systems. Technical Report 10/1999, BTU Cottbus, 1999.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III*, LNCS 1066, pages 208–219, Berlin, 1996. Springer-Verlag.

[FP96] Komei Fukuda and Alain Prodon. Double description method revisited. In *Combinatorics and Computer Science*, LNCS 1120, pages 91–111. Springer-Verlag, 1996.

[HHWT95] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A user guide to HyTech. In *Proceedings of the First Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 1019, pages 41–71. Springer-Verlag, 1995.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model-checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Hemel Hempstead, 1985.

[Lev95] Nancy G. Leveson. *Safeware*. Addison Wesley, Reading/Massachusetts, 1995.

[LSVW96] N. Lynch, R. Segala, F. Vaandrager, and H.B. Weinberg. Hybrid I/O automata. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III*, LNCS 1066, pages 496–510, Berlin, 1996. Springer-Verlag.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151. ACM, August 1987.

# A Formal Specification and Verification of a Safety Critical Railway Control System

S. Gnesi IEI – CNR [*]
D. Latella CNUCE – CNR [†]
G. Lenzini IEI – CNR [*]
C. Abbaneo ASF – GENOVA [‡]
A. Amendola ASF – NAPOLI [§]
P. Marmo ASF – NAPOLI [§]

March 7, 2000

**Abstract**

This paper describes an experiment in formal specification and validation performed in the context of an industrial joint project involving Ansaldobreda Segnalamento Ferroviario (ASF) and CNR Institutes - IEI and CNUCE - of Pisa. Within this project two formal models have been developed, describing different aspects of a wider safety-critical system for the management of medium-large railway networks. Validation of safety and liveness properties has been performed on both models. More specifically safety properties have been checked also in presence of byzantine behavior as well as other kinds of faults embedded in the models themselves. Liveness properties have been more focused on a communication protocol used within the system. Properties have been specified by means of assertions or temporal logical formulae. We used PROMELA as specification language, while the verification was performed using SPIN.

**Keywords**: safety-critical systems, dependable protocols, formal verifications, model checking.

[*]Istituto Elaborazione dell'Informazione CNR - Via S. Maria 46, 56123 PISA (Italy) tel:+39 050 593 489/485 - fax:+39 050 554 342, ({gnesi, lenzini}@iei.pi.cnr.it)

[†]CNUCE Institute CNR - Via S. Maria 40, 56123 PISA (Italy) tel:+39 050 593 230 - fax:+39 050 904 052, (d.latella@cnuce.cnr.it)

[‡]Ansaldobreda Segnalamento Ferroviario - Via dei Pescatori 35, 16129 GENOVA (Italy) tel:+39 010 655 2317 - fax:+39 101 655 2444, (cabbaneo@asf.atr.ansaldo.it)

[§]Ansaldobreda Segnalamento Ferroviario - Via Argine 425, 80147 NAPOLI (Italy) tel:+39 081 243 2621/2982, ({amendola, marmo})@atr.ansaldo.it

# 1 Introduction

The increasing request of safety and better performance in the field of modern railways and metropolitan networks has forced the introduction of sophisticated dependable control software and hardware in the automatic management of railways systems. These systems have a high degree of complexity, and require innovative validation techniques during both their design and developing phases. Traditional techniques, such as testing and simulation, could be insufficient when applied to this kind of systems. Exhaustive testing is usually ineffective because of the high number of running sequences to be analyzed, while the definition of of crucial tests is very hard due to the possibility of subtle behaviors. Simulation can provide useful information only on a limited (often already predicted) sequences and, due to the intrinsic lack of continuity, it is actually impossible to infer any global conclusion on the simulation paths not checked.

Despite of these technical difficulties industries are undoubtedly interested in discovering, in all their dependable applications, as many errors as possible before entering the production phase: during this last stage, in fact, the cost of correction per error increases enormously, while unpredictable errors can nullify months of hard work. In addition, usually governments, institutions and generally customers often require, as a guarantee of quality, accurate documentation about reliability features of a dependable system they are going to acquire. Finally, international standards have been studied to define precise safety and quality certificates (*e.g.* EN 50128 CENELEC Railways Applications [19], or IEC 65108 [12]). It worth pointing out that these standards strongly suggest Formal Methods (FM) for validation.

In fact, FM are widely recognized as fault avoidance techniques that can increase dependability by removing errors during the specification of requirements and during the design stages of development [3]. FM can be used to study the safety of a system by formally verifying that certain safety properties holds on a model of a system. In addition most FM approaches are well suitable to be mechanized and a great variety of tools for automatic validation are nowadays available.

On the other hand the use of FM introduces additional costs that cannot be ignored in industry management. In theory, the use of FM drastically lowers the developing cost of a system [15]; in practice, industries would carefully evaluate the cost/benefits ratio which can derive from changing a well established developing schedule. The integration of a new formal analysis step, in fact, requires either the intervention of specialized professionals, expert in the theory and tools of FM, or training of internal engineering teams. At the end additional series of validation steps are required to validate the real

system implementation respect to its formal specification, or more in general, the consistency of the implementation of design with its formal model.

In the last decade many industries, like Ansaldobreda Segnalamento Ferroviario (ASF), started pilot projects [8, 14, 17, 2] directed to evaluate the impact of FM on their production costs. As a result, positive experiences [1, 4] have shown how, for railway control systems, it could be possible to formalize significant models and to perform verification using the model checking [6, 20, 5] approaches.

In this paper we describe the results of a real project jointly carried out by ASF and CNR Institutes - IEI and CNUCE - in the context of the Pisa Department Computer Center of Consorzio Pisa Ricerche. The whole project consisted of two distinct parts: (a) designing a formal model of the behavior of a critical control system used in medium-large scale railway stations (b) verifying specific *safety* properties under the hypothesis of byzantine behavior [13] of one of the system component, and verifying general *liveness* properties on a dependable communication protocol developed by ASF. In this paper we focus more on the major results of the validation effort, only recalling the main modeling issues. Further details on the latter can be found in [7].

Industrial choices internal to ASF induced us to use PROMELA [10] to specify distributed asynchronous systems and express general correctness requirements. As a verification tool we used the SPIN [11] model checker.

The paper is organized as follows: in Section 2 we briefly and informally describe the system and all its component units; in Section 3 we recall the most important features of PROMELA and SPIN; in Section 5 and Section 7 we explain the PROMELA processes we defined and used as formal models, and the properties verified with some significant results; finally in Section 8 we critically discuss on the whole experience.

# 2 System Description

The railway system considered in this work was the *Computerized Central Apparatus* (ACC) [16], a control system - developed by ASF - responsible of the supervision of railway stations. The ACC is an highly programmable centralized control system, which constitutes the core of a node in a distributed architecture specifically designed to manage a large railway network. The ACC can be instantiated onto particular railway realities, and it is devoted to the control of a medium-large railway station, or a line section with small stations, or a complete low traffic line with simple interlocking logic. Its architecture (see Figure 1) consists in two subsystems that independently

perform *management* and *vital* functions. In particular:

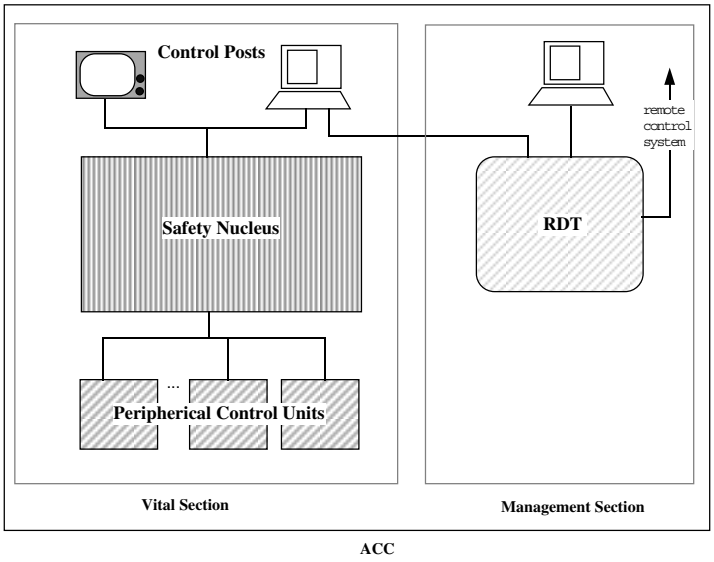



Figure 1: The Computerized Central Apparatus architecture and its environment.

- the vital section (VS) controls train movements and wayside equipment. It consists of a *Safety Nucleus* (SN), of input/output *Peripheral Control Units* (PCUs), and of *Control Posts*.
- the management section, also called RDT (Recording, Diagnosis and data Transmission), is dedicated to auxiliary functions, such as data recording, diagnostic management and remote control interface.

In our experiment we considered only the VS and in particular we focused mainly on the SN and the PCUs. In particular, the SN has been designed for the execution of safe operations, and for control and safety purposes. It is interposed between the Control Posts, from which human operator can digit commands to be sent to the periphery, and the PCUs that, in turn, execute the commands. These commands are considered *critical* because they affect critical machinery such as railway semaphores, rail points, or crossing levels. The SN achieves its purpose of safely delivering critical commands to the periphery by running monitor tasks on the state of the ACC system. In case of software/hardware faults in some components the SN tries to recover a consistent state or to exclude the faulty component. The SN is based on a triple modular redundant [21] configuration of computers which independently run different versions of the same program.

# 3 PROMELA and SPIN

PROMELA (Process Meta Language) [10] is a modeling language of general applicability introduced to describe distributed systems, communication protocols and, in general, asynchronous process systems. A PROMELA specification, usually called *a model*, consists in one or more *process templates* (called also *proctype*) and in at least one process instantiation. In a proctype a user defines the behavior of a process as an imperative program in a C-like syntax. The language is extended with nondeterministic control constructs, and with communication primitives, in a CSP [9] style. Processes can communicate via asynchronous message passing through buffered channels or shared memory. Rendezvous is modeled by buffers of length zero. In addition any running process can instantiate further asynchronous processes using proctypes.

SPIN [11] is an efficient formal verification tool for checking the logical consistency of a specification given in PROMELA, and it can generate an optimized on-the-fly verification program from a PROMELA model. Technically SPIN translates each PROMELA process template given as input, into a finite automaton. Conceptually a global automaton of a system behavior is obtained by the interleaving product (referred as the *space state*) of all the automata of the processes composing the system. In practice, efficient representations of the state space are used. PROMELA has been defined in such a way a model is necessary bound and has only finite state behavior. Then, in theory all the correctness properties become formally decidable. In practice, users needs to cope with limitations set by the state size and by computational resources.

SPIN accepts correctness claims specified either in the syntax of standard Linear Temporal Logic (LTL) [18], or as process invariants (using assertions) expressing *safety* and *liveness* properties. It can be used as an efficient on-the-fly verifier to check for deadlock presence, assertions violation, progress cycles, unreachable code, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes. Used as a LTL model checker, SPIN supports all correctness requirements expressible in this logic, either directly in the syntax of next-time free LTL, or indirectly as Büchi Automata[1].

[1] Further information on PROMELA and SPIN can be find at the official URL: `http://cm.bell-labs.com/cm/cs/what/spin/`

# 4 Formal Specification and Verification

In the following sections we introduce the general structure of our formalization work and the verification properties checked on it. About the formalization we developed two PROMELA models, we called respectively TMR and TMR-PCUS, each describing different views of the SN-PCUs system[2]. In particular:

1. the TMR model has been designed to describe in detail the SN. In this model the PCUs behavior is described more abstractly, and all the details regarding on the communication protocol between SN and PCUs have been omitted. The TMR model has been reserved to verify safety properties on the triple modular redundant mechanism of the SN in presence of byzantine behavior of one of its components;

2. the TMR-PCUS model has been designed to describe in detail the SN-PCUs communication protocol, and the PCUs themselves. Aspects of the SN behavior not related to the communication protocol have been left out. The TMR-PCUS model has been reserved to verify liveness properties on the SN-PCUs protocol, and safety properties on the same protocol in presence of *specified* hardware faults in the communication buses or in some of the PCUs.

# 5 The TMR model

The TMR model describes in detail the triple modular redundant mechanism of the SN. In Figure 2 we report a scheme of the general architecture of the system. We want to point out: (1) the three identical *central module*, called A, B and C, implementing the triple modular redundancy; (2) a special module called *exclusion logic*, devoted to checking the consistency of the three modules, and able to disconnect a module; (3) the *interconnections* between the modules (three symmetric channels), the modules and the exclusion logic (three symmetric channels), and the modules and the PCUs (a single bus); (4) the PCUs composed by $n$ control units[3]. Our PROMELA model reflects quite faithfully this general architecture: we reserved a process for each of the central module, a process for the exclusion logic and a process for each PCUs. We defined three symmetric channels of length zero between the

[2]Indeed a whole model was first considered, but we successively decided to split it because of serious state space dimension problems.

[3]In our study we have considered $n = 2$

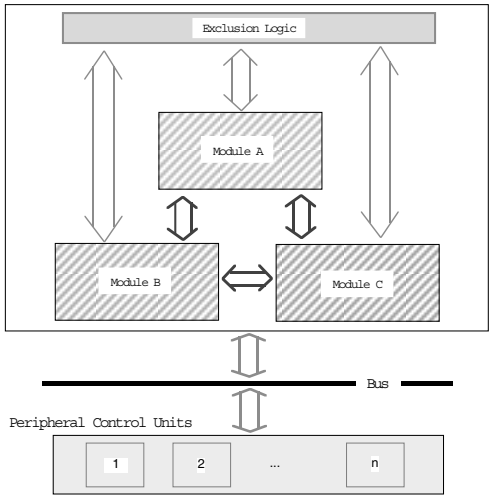



Figure 2: The TMR architecture

central modules, between the central modules and the exclusion logic. Finally we defined a bus between the central modules and the PCUs.

To have an idea of the difficulties faced within the formalization work and to understand the properties verified on it, we now briefly describe the algorithm run by a central module and the algorithm run by a peripheral unit. We first give a high level description of these algorithms and successively we report some meaningful part of the corresponding PROMELA code.

## 5.1 A central module

The behavior of each module consists of repeated sequences of *phases*, as described in the following pseudo-code.

```
 loop
 *  <synchronization>
 *  <data exchange with the other modules>
    <distribute voting>
 *  <communication to exclusion logic>

{communication with the PCUs}
    for i = 1 to n do
      if <is my turn> then
*         <synchronization>
*         <send command to the ith PCU>
      endif
*    <receive acknowledge from the ith PCUs>
    endfor
endloop
```

During each phase a central module runs local computations or communicates with other components of the system (these latter phases are stressed with an *). In particular, in the **synchronization** phase each module sends to and receives from the other two modules, with time-out[4], a synchronization message. This phase is used to collect information about the activity state of the other modules, and in particular: (1) if a time-out occurs, it is interpreted by the receiver as a sign of inactivity; (2) if two time-out occur the receiver switches in a safe shut-down. In the **data exchange** phase each module sends to the other modules a message containing its local state. Symmetrically it receives from the other modules, with time out, information about their local states. In the **distribute voting** each module performs a majority voting using the information received in the previous phase. In the **communication with the exclusion logic**, the result of the voting is sent to the exclusion logic which can disconnect a module considered potentially faulty. In the **communication with the PCUs**, a module communicates with the PCUs. At each loop only two modules are selected to send a command to the PCUs: a tournament distributed procedure assures a cyclic selection of the two modules communicating with the periphery.

In developing the PROMELA code we had to solve an important problem: the simulation of a time-out in the communication. In fact PROMELA does not deal with time. As a general solution we defined a particular EMPTY message, whose presence in a channel must be interpreted, by the receiver, as absence of any message it was waiting for. Then, whenever we had a *receive* action we introduced additional code finalized to discern, depending on the message content, if a time-out has been expired. The *send* action changed too: it has been implemented a non deterministic choice between either transmitting the "real" message or transmitting the EMPTY message. In the following we report a synthesis of the PROMELA code implementing the synchronization phase for the module A:

```
/***       In the global environment           ***/

#define EMPTY 0 // the empty message
#define SYNCH 1 // the synchronization message

                /* 1 = active */
d_step{
activeB = 1;   // local (for the module A) state of the module B
activeC = 1;   // local (for the module A) state of the module C
}
```

[4]Most of communications in the ACC are with time-out; moreover because all the channels are supposed to have no memory, a message sent in delay is to be considered lost.

```
/*==         Synchronization phase            ==*/
/* - inB,inC:   input channels from module B,C  */
/* - outB,outC: output channel to module B,C    */

d_step{
sentB = 0;    /* flag "sent" to module B */
recvB = 0;    /* flag "received" from modules B */
sentC = 0;    /* flag "sent" to module C */
recvC = 0;    /* flag "received" from modules C */
}

atomic{
do
 :: (!sentB) ->
     if
     // send the synch message if module B is active */
     :: true -> outB(SYNCH && activeB);
     // send the empty message
     :: true -> outB(EMPTY);
     fi;
     sentB = 1;

 :: (!recvB && inB?[synB]) ->
      inB?[synB];
      recvB = 1;

 :: (!sentC) ->
      if
      // send the synch message if module C is active */
      :: true -> outC(SYNCH && activeC);
      // send the empty message
      :: true -> outC(EMPTY);
      fi;
      sentC = 1;

 :: (!recvC && inC?[syn1]) ->
      inC?[synC];
      recvC = 1;

 :: (sentB && sentC && recvB && recvC) ->
      if
      :: synB == SYNCH -> activeB = 1;
      // if a time-out occurred the module
      // is considered not active
      :: else -> activeB = 0;
      fi;

      fi
```

```
      :: syncC == SYNCH -> activeC = 1;
      // if a time-out occurred the module
      // is considered not active
      :: else -> activeC = 0;
      fi;
od;
}

/* Conditional jump to the code implementing a safe shutdown*/
atomic{
if
/* if the other modules are recognized not active */
:: !activeB && !activeC ->
       global_activeA = 0;  \\ global state of module A
       goto SHUTDOWN
:: else -> skip
fi;
}
```

The PROMELA code implementing the other phases is similar to the one of synchronization, except for the type of messages involved or for some local computation.

More interesting is the implementation of a byzantine behavior, in order to model a situation in which the failure in one module may cause conflicting information to be sent to the other modules. In this context, byzantine behavior is to be intended as it was in Lamport et al. [13]: a byzantine module runs the same algorithm of a loyal module, but it can arbitrarily fail in executing it, and in particular it may send wrong messages, or send a message delayed respect to a synchronization, or send no message at all. In this interpretation of byzantine behavior, we focused the attention on communication events: all the communication phases (tagged with a * in the pseudo-code) have been realized in a *byzantine* version, where a communication error in sending a message may be possible. A communication error has been modeled as either a communication of a corrupted message, or as a delayed communication, or as no communication at all. In the following we report the PROMELA code used to implement the synchronization phase of module C, supposed to be affected by byzantine errors:

```
/***       In the global environment           ***/
#define EMPTY 0 // the empty message
#define SYNCH 1 // the synchronization message

                /* 1 = active */
d_step{
activeA = 1;   // local (for the module C) state of the module A
```

```
activeB = 1;   // local (for the module C) state of the module B
}

/*==         Synchronization phase            ==*/
/* - inA,inB:   input channels from module A,B  */
/* - outA,outB: output channel to module A,B    */

d_step{
sentA = 0;     /* flag "sent" to module A */
recvA = 0;     /* flag "received" from modules A */
sentB = 0;    /* flag "sent" to module B */
recvB = 0;    /* flag "received" from modules B */
}

atomic{
do
 :: (!sentA) ->
     if
     // send the synch message, if module A is active */
     :: true -> outA(SYNCH && activeA);
     // send the wrong message */
     :: true -> outA(-SYNCH);
     // send the empty message
     :: true -> outA(EMPTY);
     fi;
     sentA = 1;

 :: (!recvA && inA?[synA]) ->
      inA?[synA];
      recvA = 1;

 :: (!sentB) ->
      if
      // send the synch message, if module B is active */
      :: true -> outB(SYNCH && activeB);
      // send the wrong message */
      :: true -> outB(-SYNCH);
      // send the empty message
      :: true -> outB(EMPTY);
      fi;
      sentB = 1;

 :: (!recvB && inB?[syn1]) ->
      inB?[synB];
      recvB = 1;

 :: (sentA && sentB && recvA && recvB) ->
      if
      :: synA == SYNCH -> activeA = 1;
```

```
      // if a time-out or an error occurred
      // the module is considered not active*/
      :: else -> activeA = 0;
      fi;

      fi
      :: synB == SYNCH -> activeB = 1;
      // if a time-out or an error occurred
      // the module is considered not active*/
      :: else -> activeB = 0;
      fi;
od;
}

/* Eventually safe shutdown */
atomic{
if
/* if the other modules are considered not active */
:: !activeA && !activeB ->
       globlal_activeC = 0; // global state of module C
       goto SHUTDOWN
:: else -> skip
fi;
}
```

## 5.2 A peripheral control unit

On TMR model the behavior of a peripheral control unit is quite simple: it consists in waiting a commands from two modules, and in returning an acknowledgment back to all the modules. The PROMELA code implementing this simple communication protocol, for one of the peripheral unit (PCU1), is the following:

```
#define PCU1 <value>

/* loop */
do
::
count == 0; // number of commands received within the current loop

atomic{
  do
  :: (count < 2) && (bus?[PCU1, sender, cmd]) ->

     // message = (<PCU name>, <module name> <msg>)
     bus?PCU1, sender, cmd;
     count ++;
     /* send acknowledgment to all the modules */
```

```
      bus!PCU1, A, ACK;
      bus!PCU1, B, ACK;
      bus!PCU1, C, ACK;

    :: count == 2 -> break
    od;
    }
  od;
  /* endloop */
```

## 5.3 Formal Verification on TMR

In this section we list some of the most meaningful properties verified on the TMR model and the most meaningful results. Some properties have been formalized as LTL formulae, while the others with PROMELA *assertions*[5]. We used assertions for those properties that could be expressed as an invariant on all the run sequences (i.e., as the modal formula *always* $p$). In the following we assume the module C can show byzantine behavior, while modules A and B are loyal.

(TMR1) after a communication phase it is always true that if two modules do not receive any reply from the third module, this latter module will be eventually disconnected by the exclusion logic;

```
[] (p1 -> [] (q1 -> <> r1) )
```

In the previous formula `p1` stands for "the module A does not receive any message from C", `q1` stands for "the module B does not receive any message from C" and `r1` "C is disconnected".

(TMR2) after a communication phase, it is always true that if one module does not receive any reply from the other two modules, it will switch eventually in a safe shut-down state;

```
[] (p2 -> [] (q2 -> <> r2) )
```

In the previous formula `p2` stands for "the module A does not receive any message from C", `q2` stands for "the module $A$ does not receive any message from B" and `r2` "A jumps to the SHUTDOWN entry label".

[5] An assertion in PROMELA is a statement including a boolean expression, which is evaluated each time the statement is executed. If the expression evaluates `false` a violation of the correctness requirement is reported.

(TMR3) after a distributing voting phase, it is always true that if two modules, in reciprocal agreement on the global state knowledge, recognize that a third module is not in agreement with them, this latter module will be eventually disconnected by the exclusion logic;

```
[] (p3 -> [] (q3 -> <> r3 && t3) )
```

In the previous formula `p3` stands for "the module A and module B agree on their local states", `q3` stands for " module C local states differs from module A local state", `r3` "C is disconnected" and `t3` "A and B are active".

(TMR4) after a communication phase, every module has sent and received a message (eventually the empty message) from the other modules;

```
assert{(recvB + recvC == 2) && (sentB + sentC == 2)}
```

The previous assertion has been posed after each communication phase.

(TMR5) if a module is in safe shut-down state then necessarily the other two have caused a time-out in a previous communication phase;

```
assert{activeB + activeC == 0}
```

The previous assertion has been posed after the SHUTDOWN entry label.

The verification runs have been performed on different computers (for scheduling needs) by using different optimization options[6] In Table 1 we report on hardware characteristics of our computational resources and Table 2 we report on the results of the verifications.

| I | II | III | IV |
|---|---|---|---|
| SPARC St. | SPARC St. 20 | AMD K6 200 | PC 486 133 |
| 40Mb RAM | 64Mb RAM | 64Mb RAM | 32Mb RAM |
| SunOS 4.1.4 | SunOS 5.5.1 | LINUX RedHat 5.0 | LINUX Debian 2.0.27 |

Table 1: Resources used in the verification work

[6]In particular we used the following compiler options of SPIN: COLLAPSE (CO) to compress the state vector and MA to obtain a minimal automaton encoding.

| *property* | *PC* | *state vector* | *options* | *RAM (bytes)* | *depth* | *result* |
|---|---|---|---|---|---|---|
| TMR1 | III | 192 | CO+MA | 22.098 | 5266 | success |
| TMR2 | I | 192 | CO+MA | 14.898 | 5266 | success |
| TMR3 | IV | 196 | CO+MA | 21.029 | 45273 | fail |
| TMR4 | IV | 188 | MA | 25.170 | 297515 | success |
| TMR5 | I | 188 | CO | 9.515 | 6808 | success |

Table 2: Verification results on the TMR model

# 6 Discussion

We briefly discuss the failed properties TMR3. In analyzing the counter-example we noted that the byzantine module C can cause a module B to be disconnected by the exclusion logic. In fact module C causing a time-out in communicating with module B, makes module B to believe that module C is not active. Successively in the distribute voting module B is found in disagreement with A and C and then module B (and not module C) is disconnected by the exclusion logic. This situation depicts a weakness in the exclusion logic mechanism, already found by Ansaldo using traditional simulation techniques.

# 7 The TMR-PCUs model

The TMR-PCUS describes in detail the SN-PCUs communication protocol, and the PCUs behavior. Target of the protocol is to deliver critical commands also in presence of faults in the communication media. A scheme of TMR-PCUS architecture is reported in Figure 3. We want to stress: (1) the three identical *central modules*, called *module A*, *B* and *C*, implementing part of the SN; (2) the PCUs composed by $n$ control units[7], each constituted by a configuration of two computers, called A and B; (3) the *interconnections* among the modules (three symmetric channels), the ones between the modules and the PCUs (two busses).

With the TMR-PCUS model we were interested to verify:

1. *liveness properties* of SN-PCUs communication protocol in an error-free environment hypothesis. This protocol is implemented as a distributed algorithm designed to assure a cyclic use of the buses and a cyclic selection of the two modules demanded to to send the commands.

[7] In our study we considered $n$=2

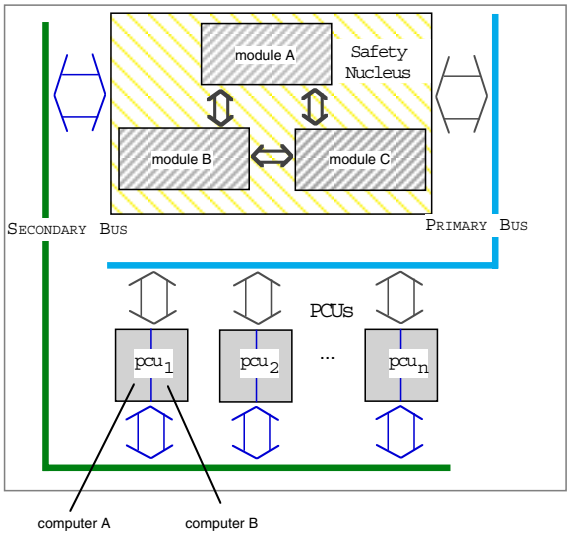



Figure 3: The TMR-PCUS architecture

2. *safety properties* of SN-PCUs communication protocol in case of some hardware faults. In particular we were interested in injecting faults in the interconnection buses and in the computers component a peripheral unit.

In our PROMELA code we reserved a process for each central module, and a process for each peripheral unit. We defined a symmetric channel of length zero between each pair of central modules, and two busses between the SN and the PCUs.

We now briefly describe the algorithm run by a central module and the one run by a peripheral unit. The algorithm of the central module is given only in a high level description, while the PROMELA code relative of the algorithm run by a peripheral unit is reported.

## 7.1 A central module

The behavior of a module can be described as a repeated sequences of *phases*, as in the following pseudo-code:

```
loop
<synthesis>

{communication with PCUs}
for i=1 to 2 do
 for j=1 to n do
```

```
  <synchronization>
  <diagnostic>
  <message elaboration>
  if <is my turn> then
     <send message to i-th computer of the
       j-th PCUs using the selected bus>
  endif
  <receive acknowledge from the i-th computer of
     the j-th PCUs using the selected bus>
  endfor
endfor

endloop
```

In the **synthesis** phase we have synthesized a possible outcome from phases 1 to 5 of the TMR model: substantially we decide if a module is active, or not active. Before communicating with each computer of each peripheral unit a module tries to infer information about the global state of the system. In particular, in the **synchronization** phase a module tries to know the other modules activity state, by exchanging a synchronization message. This information is used in the tournament procedure to decide what two modules are selected to send message to the periphery. In the **diagnostic** phase (which is quite complex in reality), by considering global information collected in a previous loop, a module tries to infer the global state of the PCU computers and of the two buses. Information collected is used to decide which buses to use. In addition, in the **message elaboration** phase depending on the state of peripheral computers, either the *effective peripheral command*, or a *special diagnostic message* is prepared.

## 7.2 A peripheral unit

In TMR-PCUS the PCUs model is realized in a deeper detail. Its behavior can be synthesized with the following pseudo-code:

```
loop
 <decide the state of each of the two busses>
 <decide the state of each of the two computers>

{communication with the safety nucleus}
parallel for i=1 to 2 do
<computer[i] receives a message from bus1
    and sends acknowledgments to all the modules>
<computer[i] receives a message from bus2
    and sends acknowledgments to all the modules>
endfor
endloop
```

In the **decide the state** phase, a non-deterministic choice is made to decide on the functional state of the buses and of the computers of the peripheral unit. In case of state set to "fault" every communication via the faulty bus or coming from the faulty computer resulted in an expiration time-out until the end of the loop. In the following we report the PROMELA code used to implement the previous algorithm.

```
#define DONE recvA1+recvA2+recvB1+recvB2==4
/* loop */
do
::

/* ===             Initialization of the variables:             === */

/* recvA1: counter of messages received by the computer A via bus1*/
/* recvB1: counter of messages received by the computer B via bus1*/
/* recvA2: counter of messages received by the computer A via bus2 */
/* recvB2: counter of messages received by the computer B via bus2 */
/* stateBUS1: the state of bus1 */
/* stateBUS2: the state of bus2 */
/* stateA: the state of computers A */
/* stateB: the state of computers B */

/* decide the state */
   d_step
   {
     if
     /* fault in the 1st computer */
     :: stateA = 0
     /* 1st computer is ok */
     :: stateA = 1
     /* fault in the 2nd computer */
     :: stateB = 0
     /* 2nd computer is ok */
     :: stateB = 1
     /* fault in the 1st bus */
     :: stateBUS1 = 0
     /* 1st bus is ok */
     :: stateBUS1 = 0
     /* fault in the 2nd bus */
     :: stateBUS2 = 0
     /* 2nd bus is ok */
     :: stateBUS2 = 1
     /* no fault */
     :: else -> skip
     fi
    };
RECEIVING:skip;
```

```
atomic{
i = 0;
/* A1 : computer A - BUS 1 */
/* A2 : computer A - BUS 2 */
/* B1 : computer B - BUS 1 */
/* B2 : computer B - BUS 2 */
do
  :: !DONE && A1in?[PCU1, senderA1, msg] ->
               A1in?PCU1, senderA1, msg;
        if
        /* if it is a diagnostic message */
        :: msg == HEADER -> skip;
        /* if it is a command message */
        :: else -> msg[i] = msg; i++;
        fi;

        /* acknowledgment to all the module */
        A1out!PCU1,A,(stateA && stateBUS1);
        A1out!PCU1,B,(stateA && stateBUS1);
        A1out!PCU1,C,(stateA && stateBUS1);
        recvA1++;

  :: !DONE && A2in?[PCU1, senderA2, msg] ->
               A2in?PCU1,senderA2,msg;
        if
        /* if it is a diagnostic message */
        :: msg == HEADER -> skip;
        /* if it is a command message */
        :: else -> msg[i] = msg; i++;
        fi;

        /* acknowledgment to all the module */
        A2out!PCU1,A,(stateB && stateBUS1);
        A2out!PCU1,B,(stateB && stateBUS1);
        A2out!PCU1,C,(stateB && stateBUS1);
        recvA2++;

  :: !DONE && B1in?[PCU1, senderB1, msg] ->
               B1in?PCU1,senderB1,msg;
        if
        /* if it is a diagnostic message */
        :: msg == HEADER -> skip;
        /* if it is a command message */
        :: else -> msg[i] = msg; i++;
        fi;

        /* acknowledgment to all the module */
        B1out!PCU1,A,(stateA && stateBUS1);
        B1out!PCU1,B,(stateA && stateBUS1);
```

```
            B1out!PCU1,C,(stateA && stateBUS1);
            recvB1++;

        :: !DONE && B2in?[CDA1, senderB2, msg] ->
                    B2in?PCU1,senderB2,msg;
            if
            /* if it is a diagnostic message */
            :: msg == HEADER -> skip;
            /* if it is a command message */
            :: else -> msg[i] = msg; i++;
            fi;

            /* acknowledgment to all the module */
            B2out!PCU1,A,(stateB && stateBUS2);
            B2out!PCU1,B,(stateB && stateBUS2);
            B2out!PCU1,C,(stateB && stateBUS2);
            recvB2++;

        :: DONE -> break;
      od
      };

  RECEIVED: skip
  /* endloop */
  od;
```

## 7.3 Formal Verification on TMR-PCUs

In this section we informally list some of the properties verified on the TMR-PCUS model, and the most meaningful results. The properties can be informally described as:

(PCUS1) correctness of the communication protocols, in absence of faults;

We verified these two properties checking for absence of deadlock. With the term *correctness* we mean a a general correctness of the diagnostic and of the tournament algorithm run by a central module. In this case we slightly modified the PROMELA code of the PCUs in such a way to force a peripheral unit to receive messages according to the right cyclic use of the busses. An incorrect use of it by one of the central module will have caused a deadlock.

The following properties has been verified in presence of faults.

(PCUS2) when two or more modules are active each peripheral unit eventually receives exactly two messages, in a single loop;

```
([]p2) -> (( []<> q2 ) && [] (q2 -> (<> r2) ))
```

In the previous formula `p2` stands for "at least two modules are active", `q2` stands for "PCU1 is in RECEIVING" and `r2` "PCUS1 is in RECEIVED and it has received exactly two messages".

(PCUS2') in presence of byzantine errors in one module, when two or more modules are active each peripheral unit eventually receives exactly two messages, in a single loop;

This properties is the same of PCUS2, but was verified with one module running a byzantine synchronization phase.

(PCU3) when two or more modules are active each peripheral unit eventually receives exactly two message via different buses, in a single loop;

```
([]p3) -> (( []<> q3 ) && [] (q3 -> (<> r3 && s3) ))
```

In the previous formula `p3` stands for "at least two modules are active", `q3` stands for "PCU1 is in RECEIVING", `r3` "PCUS1 is in RECEIVED and it has received exactly two messages", and `s3` "the messages received come from different buffers".

(PCU4) when two or more modules are active each computer of every peripheral units receives exactly one message, in a single loop.

```
([]p4) -> (( []<> q4 ) && [] (q4 -> (<> r4 && s4) ))
```

In the previous formula `p4` stands for "at least two modules are active", `q4` stands for "PCU1 is in RECEIVING", `r4` "PCUS1 is in RECEIVED and it has received exactly two messages", and `s4` "each computer has received at most one message".

In the Table 3 we report some of the most significant results.

| *property* | *PC* | *state vector* | *options* | *RAM (bytes)* | *depth search* | *output* |
|---|---|---|---|---|---|---|
| 1 | IV | 352 | C0 | 60.702 | 44047 | success |
| 2 | II | 284 | CO+MA | 23.808 | 25465 | success |
| 2' | IV | 284 | CO+MA | 33.491 | 1295 | fail |
| 3 | II | 284 | CO+MA | 23.808 | 25465 | success |
| 4 | IV | 284 | CO+MA | 33.553 | 405178 | success |

Table 3: Verification results on the TMR-PCUs model

### 7.4 Discussion

We briefly discuss the result of properties PCU2'. We wanted to prove safety properties of the tournament algorithm in the hypothetic situation of a byzantine behavior. In fact the fail was due to the byzantine behavior of a module, and analyzing the counter-example, we noticed that three modules (and not two) send a message to the periphery. It worth to point out that exclusion logic, we have omitted in this model, should have disconnected potentially byzantine module before entering in the communication with the periphery phase. Indeed this is what happens in the real system, as proved by ASF on the real system.

## 8 Conclusions

The work described in this paper, related to a real and wider project, consisted of the verification effort performed on a formal model of a safety-critical system developed by Ansaldobreda Segnalamento Ferroviario, to manage medium-large scale railway networks. The real system, actually running at one of the main Italian railway stations, has been validated also by Ansaldobreda Segnalamento Ferroviario. During the formal verification we found some interesting erroneous situations some of them due to imprecision in the requirements, while others due to weakness in the system itself. In particular the fail entry in Table 2 pointed out a situation in which a tricky combination of byzantine communications distorted the exclusion logic mechanism; in Table 3 the fail entry refers to a misunderstanding in the requirements of the dependable protocol, successively fixed. We would underline that, by using the results obtained in the formal analysis, all implementation errors were also confirmed by Ansaldobreda by using traditional verification techniques.

Although briefly described in this paper, many formalization problems has been faced during the modeling phase, primarily due to the missing of any concept of time in the PROMELA language, and secondly to an inappropriate (respect to our needs) treatment of the termination of a processes in its run time support. These weaknesses obliged us both to abstract, in our models, any reference to the time (for example in the communication with time out), and to realize a safe-shutdown state as active process that participates in all the active communications. Those modeling choices have had a substantial impact in the formalization effort and, indirectly, in the state dimension of the model realized. To face with this last problem we need to design *ad hoc* abstraction strategies. All these formalization issues in a companion paper [7]. On the contrary the contribution of this paper relies on

describing the bulk of the verification work, consisting prevalently in defining two different models each depicting different aspects of the control system at a different degree of abstraction.

# 9 Acknowledgment

This work was partly supported by the CNR/GMD cooperation project DECOR and by Progetto speciale CNR "Strumenti Automatici per la Verifica Formale nel Progetto di Sistemi Software ".

# References

[1] C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi, and D. Romano. A Formal Verification Environment for Railway Signaling System Design. *Formal Methods in System Design*, 2(12):139–161, 1998.

[2] A. Borälv. A Case Study: Formal Verification of a Computerized Railway Interlocking. *Formal Aspect of Computing*, 10(4):338–360, 1998.

[3] J. P. Bowen and M. G. Hinckey. Seven More Myths of Formal Methods. *IEEE Software*, 12:34–41, 1995.

[4] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso. Formal Verification of a Railway Interlocking System using Model Checking. *Formal Aspect of Computing*, 10(4):361–380, 1998.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specification. *ACM Transaction on Programming Languages and Systems*, 8:244–263, 1986.

[6] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, 1981. Springer-Verlag.

[7] S. Gnesi, D. Latella, G. Lenzini, C. Abbaneo, A. Amendola, and P. Marmo. A formal specification and validation of a critical system in presence of byzantine error. submitted to TACAS'2000.

[8] J. F. Groote, S. F. M. van Vlijemn, and J. W. C. Koorn. The Safety Guaranteeing System at Station Hoorn-Kersenboogerd in Propositional Logic. In *Proceedings of 10th Annual Conference on Computer Assurance (COMPASS'95)*, pages 57–68, 1995.

[9] C. A. R. Hoare. *Communicating Sequential Processes.* Prantice-Hall International, 1991.

[10] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentise Hall, 1991.

[11] G. J. Holzmann. The Model Checker SPIN. *IEEE Transaction on Software Engineering*, 5(23):279–295, 1997.

[12] IEC 61508 IEC. Functional safety of electrical/electronic/programmable electronic safety-related systems.

[13] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transaction on Programming Languages and Systems*, 4(3):382–401, 1982.

[14] P. G. Larsen, J. Fitzgerald, and T. Brookers. Applying Formal Specification in Industry. *IEEE Software*, 13(7):48–56, 1996.

[15] P. Liggersmeyer, M. Rothfelder, M. Rettelbach, and T. Ackermann. Qualitätssincherung Software-basierter Technischer Systeme - Problembereiche und Lösungsänsatze. *Informatik Spektrum*, 21:249–258, 1998. in German.

[16] G. Mongardi. *Dependable Computing for Railway Control System*, chapter 3. Springer-Verlag, 1993.

[17] M. J. Morely. Safety-Level Communication in Railway Interlockings. *Science of Communication*, 29:147–170, 1997.

[18] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, 1977. IEEE, IEEE Computer Society Press.

[19] pr EN 50128 CENELEC. Railways Applications: Software for Railway Control and Protection Systems.

[20] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings 5th International Symposium on Programming*, Lecture Notes in Computer Science, Vol. 137, pages 337–371. SV, Berlin/New York, 1982.

[21] Neil Storey. *Safety Critical Computer Systems.* Addison-Wesley, 1996.

# A Case Study in Formal Methods: Specification and Validation of the OM/RR Protocol

Tim Willemse[1], Jan Tretmans[2], and Arjen Klomp[3]

[1] Eindhoven University of Technology, Formal Methods Group,
Department of Mathematics and Computing Science, P.O.Box 513, 5600 MB Eindhoven,
The Netherlands, timw@win.tue.nl
[2] University of Twente, Formal Methods and Tools group, Faculty of Computing Science, P.O.Box 217,
7500 AE Enschede, The Netherlands, tretmans@cs.utwente.nl
[3] CMG Den Haag B.V. Division Advanced Technology, P.O.Box 187, 2501 CD The Hague,
The Netherlands, arjen.klomp@cmg.nl †

**Abstract.** This paper reports on the results of the application of formal methods in the development of an industrial, mission-critical system, called the *Operator Support System*. A critical communication protocol of this system, the *OM/RR Protocol*, and its corresponding service were formalised using the formal specification language LOTOS. The resulting specifications have been validated using the tool set LITE and models of the specifications, obtained by making abstractions, have been verified using the tool EUCALYPTUS. Whereas the use of formal methods is usually motivated by their ability to allow for unambiguous and precise system descriptions amenable to mathematical reasoning, it turned out that in this project most benefits were obtained by the sheer process of *formalising* the informal protocol description, revealing many omissions and ambiguities. The results and experiences obtained during formalisation, validation, abstraction and verification are discussed on a non-formal basis in this paper.

**Keywords & Phrases:** application of formal methods, industrial case study, LOTOS, communication protocol.

## 1 Introduction

*Formal methods* have always been envisioned to be applied to systems that require unambiguous, mathematically precise descriptions for their correctness. The use of formal methods in the industry has been prophesied since their very dawn, however, reality learns that their acceptance is still not very wide-spread. One way of increasing the acceptance is by applying formal methods in case studies of real industrial systems. This paper reports on such a case study: the *OM/RR protocol*. The project was conducted between 1997 and 1998.

The *OM/RR protocol*, which is a short-hand for *Object Management/Request Response protocol*, is a communication protocol used in the *Operator Support System* (OSS). OSS is a complex, *mission-critical system* which is being developed to integrate different *Motor-way Management Systems* (MMSs). CMG Den Haag B.V. takes part in this development which is commissioned by the Traffic and Transportation Department of the Dutch Ministry of Transport, Public Works and Water Management.

In the past years, various MMSs have been developed, such as fog detection systems, congestion detection systems and variable message signs for controlling traffic. For historical reasons, many MMSs have their own specification and implementation and they cannot interact. The control of an MMS takes place at an Operating Centre. The current situation in The Netherlands is that

† Current affiliation: CMG Eindhoven B.V. Sector Trade, Transport & Industry
Luchthavenweg 57, P.O.Box 7089, 5605 JB Eindhoven, The Netherlands

there are several Operating Centres throughout the country. Each Operating Centre has its own domain, consisting of several MMSs. Since these domains are piecewise disjoint, this severely limits the — nowadays much needed — cooperation between different Operating Centres.

In recent years this awareness has grown, leading to the description of the OSS. This system is intended to integrate the independent Operating Centres and MMSs, and thereby to increase the power of the system regarded as a whole. The idea is to enable one Operating Centre to take over duties from another Operating Centre, or to perform tasks that at some point go beyond the Operating Centre's own domain, thus offering maximal flexibility in regulating and controlling traffic.

The cooperation between the Operating Centres is supported by communication and authorisation protocols. The communications protocol is intended to allow both communication between Operating Centres and between an Operating Centre on the one hand and an MMS on the other hand. The authorisation protocol is intended to restrict all possible communications allowed by the communications protocol, avoiding possibly chaotic situations, in which control over MMSs can be lost. Both protocols display complex behaviour and are critical to the well-functioning of OSS, so they are sources of potential hazards to the functioning of the OSS.

The main motivation for CMG to incorporate formal methods in the development of the OSS was the mission critical aspect of the OSS. Moreover, CMG had positive experiences with the use of formal methods in another mission critical system: the BOS system, the decision support system which controls and operates the storm surge barrier in the Nieuwe Waterweg near Rotterdam – a movable dam which should protect Rotterdam from being flooded [10, 20].

The main goals of applying formal methods to the OM/RR protocol were to check its feasibility and suitability for OSS, and to judge the completeness, preciseness and consistency of its (informal) specification. It was acknowledged that, based on the documentation available for the OSS, no firm conclusions about these facts could be drawn. It was expected that formal methods could help in giving an answer to these questions. A secondary goal for CMG was to increase their knowledge of and their experience with formal methods and associated tools.

In this paper, we describe the successful application of formal methods to the analysis of the OM/RR protocol – successful at least from the formal methods point of view, not from the OM/RR protocol point of view, as will be seen. By carefully translating the informal documentation for this protocol, taking into account the generally believed functionality of this protocol, a formal specification is constructed. This process allows one to determine omissions, as well as ambiguities in the informal documentation. Simulation of the formal specification is used to validate the formal specification against informal specifications and design requirements. Confidence in the correctness of the formal specification is increased by showing proper, formal relations between *models* of the formal service specification and *models* of the protocol specifications.

The concepts of *service* and *protocol*, used in this project, are defined in [21] and stem from the context of the OSI Reference Model [16]. It is illustrated in Fig. 1. A communication service is provided by a *service provider*, which is considered as a blackbox. The *service users* communicate with the service provider via shared interaction points, so-called *Service Access Points* (SAPs). The layer-$n$ *service specification* defines the possible communications between the layer-$n$ service users by using the layer-$n$ service provider. The layer-$n$ *protocol specification* describes how the layer $n$-protocol entities should communicate via the $n-1$ service provider to establish the $n$-service.

The rest of this paper is structured as follows. Section 2 discusses the communications protocol: its basic, informal documentation, its ambiguities, its omissions and its characteristics, and the choices that have been made in the process of formalising specifications are sketched. The validation and verification aspects of the formal specifications are the topic of Section 3, and concluding remarks and an evaluation of the use of formal methods in this project are discussed in Section 4. Full technical details of this project can be found in [23].

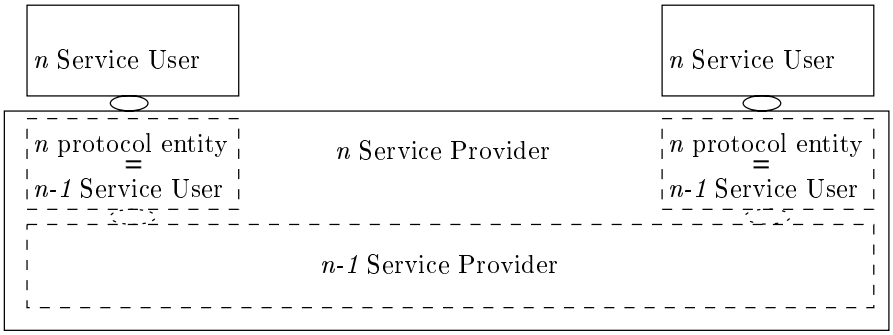



**Fig. 1.** *The Concept of Service*

## 2 The OM/RR Protocol

This Section describes the informal and formal OM/RR protocol: the documentation for the OM/RR protocol is discussed in Section 2.1. Subsequently, Section 2.2 discusses some of the problems and omissions of this document. Then Section 2.3 describes briefly the process of formalising the documentation for the OM/RR protocol.

### 2.1 The OM/RR Documentation

The starting point for the formalisation was the description of the OM/RR protocol in [7], which was developed outside the current project. According to this document, the OM/RR protocol is loosely modelled after the OSI CMIS [5], the OSI CMIP [6], some parts of Systems Management [1] and the OSI Remote Operations [4] standards.

The document [7] is, including appendices, a little over 75 pages, describing the OM/RR protocol and Association Management. Several appendices are added to explain short-hand notations and conventions, taking up 15 pages of the document. A few pages are dedicated to a more global, informal, description of the OM/RR protocol and its relation with the OSS, creating a context for the OM/RR protocol.

The contents of [7] consists mainly of information about data, described in the notation ASN.1 [19, 3]. Using this notation, the *service primitives* (i.e. primitives for communication between service users and a service provider) and the *protocol data units* (i.e. units of data, handled by a protocol entity) of the OM protocol entities and the RR protocol entities are specified. Although ASN.1 allows one to write data in a concise and platform independent way, it does not allow for specifying dynamic behaviour. The OSI standards CMIS and CMIP also do not encompass any dynamic behaviour, as they leave this part up to the users of these standards. Other issues, discussed in [7] deal with the decomposition of the OM/RR protocol in layers in an OSI/RM manner (see Fig. 2).

### 2.2 Omissions

During careful, thorough analysis and study of the informal OM/RR document with the intent of developing a formal description a lot of incompleteness, impreciseness, ambiguity and inconsistency was discovered.

Although OSI/RM-like layering of Fig. 2 is visually very compelling, additional information is required to understand the interaction mechanisms between the different protocol entities and layers. However, the functionalities of each layer are discussed only informally, ambiguously, or not revealing the inter-dependencies between the protocol entities. This lack of information about the entanglement of these protocol entities is regarded as a major omission.

Another major omission is the absence of descriptions for the dynamic behaviour (e.g. a service specification), except for a transition diagram described in one appendix and a scenario describing

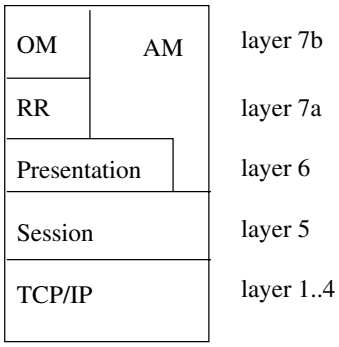



**Fig. 2.** *The protocol stack*

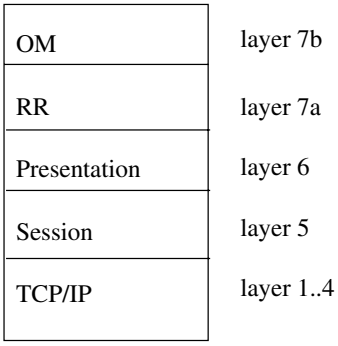



**Fig. 3.** *The alternative protocol stack*

aspects of communication for association management. The status of this diagram and the scenario, however, is not explained. Service specifications are vital for users of a protocol, since the service specification accurately describes the behaviour to be expected by a service user. The (formal or informal) protocol specifications are necessary for guiding a programmer to a correct program. Absence of both means that the documentation is not complete.

The dynamic behaviour is always described based on certain *design requirements*. Using these requirements, it should be possible to design a service specification and protocol specifications. Most design requirements, however, are not described in [7]. The design requirements that are documented in [7] are mostly concerned with performance aspects instead of functionality. As such, the documentation for the OM/RR protocol fails to meet its purpose.

Conversations with the developers revealed that they regarded the specifications of the service primitives and the protocol data units as a sufficient basis for implementing the OM/RR protocol. Specifying dynamic behaviour was regarded to be a non-issue, since it was, as they believed, "trivial". Within CMG, though, the opposite was believed. This feeling was confirmed, since the trajectory of formalisation revealed that various issues proved more complex than the developers had expected.

At this point in the project, one of the goals of the formalisation process had, in fact, already been achieved: the OM/RR protocol document [7] was not stable, complete and precise enough to form an unambiguous basis for further development of conforming implementations.

### 2.3 Formalisation of the OM/RR Protocol

Analysis of the informal OM/RR protocol document [7] revealed that this document is not suitable to be considered a basis for formalising the protocol, see Section 2.2. Despite this fact, we decided to continue the formalisation process to see to what extent the other goal – checking the feasibility and suitability of the design of the OM/RR protocol for OSS – could be met. An attempt has been made to design formal specifications for the OM/RR protocol. Using the few design requirements, documented in [7], information obtained in conversations with the developers of [7] and information within CMG from people working on the OSS project, design requirements have been formulated. These design requirements then served as the basis for formalising the OM/RR protocol. The focus in the formalisation process has been on the dynamic behaviour, since the data was already fully specified using ASN.1. It should be noted that now we not only formalised the existing protocol, but also were actually designing parts of it.

Subsequently, in the process of formalisation, a choice for a suitable formalism for describing the specifications had to be made. This choice was made by considering two criteria to which the formalism should adhere:

1. The formalism should allow for a concise and unambiguous specification of dynamic behaviour;

2. The formalism should be supported by tools able to analyse, validate and verify descriptions in this formalism.

Even though a number of formalisms adhere to these requirements, a slight preference for a process algebraic formalism led to LOTOS [8, 2]. The tool support used in this project for LOTOS consists of the tool-sets LITE [12] and EUCALYPTUS [14, 13] (various 1998-beta-versions of CADP 97b "Liège").

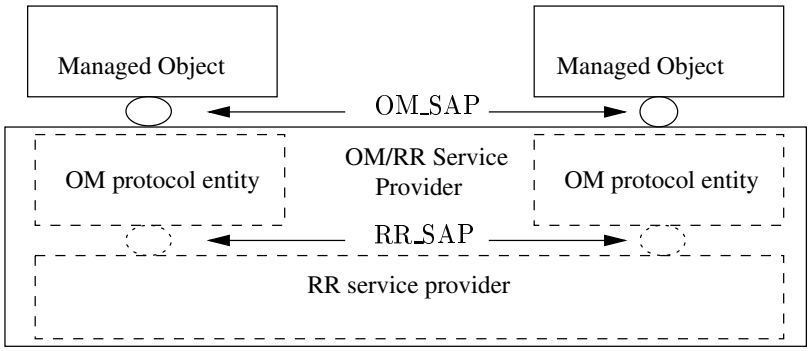



**Fig. 4.** *The Relation between the OM/RR Service, the OM Protocol and the RR Service*

The service specification of the OM/RR system – actually the OM service – is based on the design requirements formulated for OM/RR. Subsequent decomposition of this service specification leads to a lower-level service specification and a protocol specification. Visually, the decomposition is expressed in Fig. 4. This decomposition also has its impact on the OM/RR protocol stack (see Fig. 3 on page 4). The connection between the generic service/protocol structure in Fig. 1 and Fig. 4 is easily made. The service the OM/RR protocol, i.e. the *OM/RR Service Provider*, provides to its service users, i.e. *Managed Objects*, is described in LOTOS. The decomposition of the OM/RR Service Provider leads to specifications for the *OM Protocol Entities* and the *RR Service Provider*. These are also described in LOTOS. The correctness of this decomposition step can now easily be expressed in LOTOS as follows, where `OM_RR_Service_Provider[OM_SAP]`, `OM_Protocol_Entity[OM_SAP,RR_SAP]`, `RR_Service_Provider[RR_SAP]` are LOTOS processes for the OM/RR Service Provider, OM Protocol Entity and RR Service Provider, respectively; `OM_SAP` and `RR_SAP` are the Service Access Points of the OM layer and RR layer, respectively; and $\approx$ is a suitable semantic relation between processes.

```
OM_RR_Service_Provider[OM_SAP]  ≈                                    (2.1)
   HIDE  RR_SAP  IN    ((    OM_Protocol_Entity[OM_SAP,RR_SAP]
                          ||| OM_Protocol_Entity[OM_SAP,RR_SAP] )
                      |[RR_SAP]|  RR_Service_Provider[RR_SAP] )
```

Basically, the service specification of the OM/RR protocol describes communications between the service users of the OM/RR protocol, which are according to [7] *Managed Objects*. The service users impose, by definition, no restrictions on the OM/RR protocol. Managed Objects communicate with one another via *associations*, which are considered to be binary relations. Two Managed Objects connected with each other via an association are assigned *roles* based on the initiative in establishing the association. One takes the role of a *Manager* and one takes the role of an *Agent*. These roles have their impact on the set of service primitives a Managed Object is allowed to use in an association with another Managed Object. The order in which service primitives are used to establish, communicate and release an association, is described in LOTOS. The view a Managed Object has on an association is described by means of a state-transition diagram (see Fig. 5). This state-transition diagram served as a framework for writing the LOTOS specifications.

The specification of the OM protocol entities describes formally how the OM/RR service specification can be met, using the service specification of the RR protocol. The service specification of the RR protocol describes a connection-less service, offering reliable data transmissions between two service users. It is not clear to us yet, and in [7] it is also not explained, why the developers used a TCP/IP network below the RR layer (see Fig. 2 and 3).

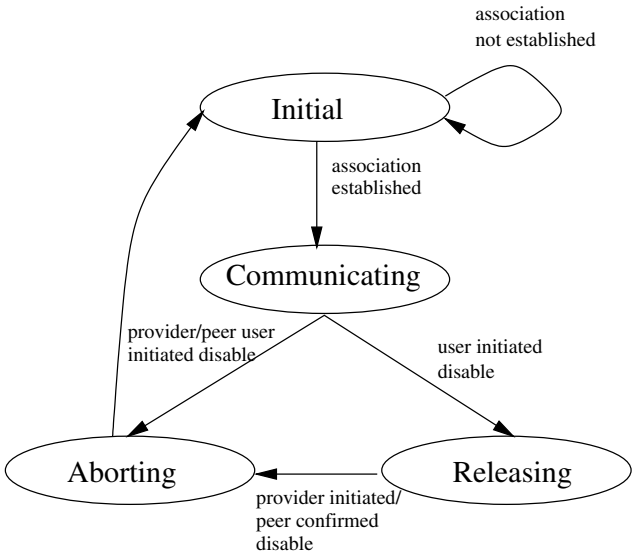



**Fig. 5.** *A Managed Object's view on an Association*

| Specification | Lines of LOTOS code |
|---|---|
| OM/RR service specification | 1000 |
| OM/RR protocol specification | 1600 |

**Table 1.** *Lines of* LOTOS *Code*

To give an indication into the complexity of the final specifications, an approximation of the number of lines of LOTOS code are mentioned in Table 1. In these specifications data not affecting the dynamic behaviour (e.g. data for the service users), is abstracted from. Clearly, the OM/RR service specification is specified more easily than the OM/RR protocol specification, which is an argument in favour of using the concept of service as a step in protocol development.

## 3 Validation, Verification and Analysis

A part of the objectives in our project has been to reveal the (lack of) information contained in documents concerning the OM/RR communications protocol. This part has been discussed in Section 2. Another objective has been to show how formal methods can be applied to write unambiguous specifications in an industrial project. Obtaining confidence in such specifications is also assisted by formal methods. The formal specifications written for the OM/RR protocol have been validated and verified using the tool-sets LITE [12] and EUCALYPTUS [14, 13]. These validation and verification efforts are discussed in the subsequent sections.

### 3.1 Validation and Analysis using LITE

The validation and analysis of the formal specifications of the communication protocol has been tackled in different stages. The first stage consisted of checking the syntax and static semantics of the LOTOS-specifications. For this, the syntax and static-semantic analyser of LITE were used. Although necessary for performing subsequent analysis, fixing the problems discovered in this stage only provided more insight into the description language rather than in the specification itself. Given the amount of errors that were reported in this stage, one may consider the FDT LOTOS either unnatural or hard to learn. The main problems encountered in this stage were of the nature of functionality of a process (i.e. `exit` versus. `noexit`), which at first seem very counter-intuitive.

The second stage involved checking the dynamic behaviour of the specifications. The design requirements and the state transition diagram (see Fig. 5) have been used as a guideline in this stage. The LOTOS simulator (animator) SMILE, which is part of the tool-set LITE, has been used for this. At first, a simulation based approach was taken, using SMILE to perform single stepping through the specifications. Single stepping is a simulation in which actions are presented that can occur *immediately after* choosing an action that has to occur. This analysis, however, is troubled by the fact that the overall behaviour of the system is "contaminated" with internal actions that are present in the specification. To overcome this problem, one can look at the *action prefix* to make an educated guess about which action leads to the desired observable action. Although in theory this can be used to perform some thorough simulations, it is also a very cumbersome method to use.

Another option that has subsequently been taken is by making use of test cases, in which certain behaviour of the system is expressed, i.e. so called *may* test cases [11]. In contrast to *must* test cases, in which the test case must specify behaviour which is *always* possible, may test cases specify only behaviour which *in some cases* is possible. Test cases control the behaviour of the specification into the direction expressed in the test cases. In this way (un)expected behaviour can be checked. The construction of the test cases that have been used, was guided by the structure of the state-transition diagram of Fig. 5. All test cases tested traces that started in the *Initial* state, and specified different routes back to the *Initial* state again.

Although the execution of test cases usually leads to a more restricted dynamic behaviour, in our case much of the behaviour was still blurred by the internal actions. Yet, many cases of faulty behaviour were revealed using the test cases. Several rounds of improvement and re-testing were necessary to mend the errors discovered at this stage. Note that in many cases, the translation of the ideas expressed in the natural language to the formal LOTOS-descriptions was incorrect, instead of the basic intuition behind these ideas.

By using the static analysers and the simulator SMILE from the tool-set LITE the confidence in the correctness of the LOTOS specifications was increased. However, the possibilities for further validation and analysis using tools from LITE are limited. This means that, in order to perform, for instance, checks for absence of deadlock, or establish the relation expressed by equation (2.1) formally, other tool-sets have to be used. The subsequent sections will go into more detail about the additional analysis and validation.

### 3.2 Validation, Verification and Analysis using EUCALYPTUS

The tool-set EUCALYPTUS [14, 13] is a collection of several tools for analysing and verifying LOTOS descriptions. However, the tools that come with this tool-set all have a major drawback in that they pose restrictions on the LOTOS grammar: not the full set of LOTOS operators is supported. The positive side is that these restrictions allow for more powerful tools, such as checking for deadlock, livelock, (bi)similarity and preorder relations. These problems are in general not computable for the full grammar of LOTOS. The use of this tool-set on the complete LOTOS specification as it is, however, is not possible due to this lack of support for the complete LOTOS grammar. In order to use the tools, simplifications and abstractions have to be made: *models* have to be constructed from the specifications. These models should then focus on some part of the vital behaviour of the

communication protocol. A second reason for developing models from the complete specifications is complexity. Although, with the restrictions on the LOTOS grammar the above mentioned checking problems are in principle computable, the size of the models, e.g. the number of system states, may prevent their effective computation by EUCALYPTUS. Also for this reason simplifications and abstractions are needed.

Although the use of models in the industry is fully accepted, various implications must be kept in mind. A model is a simplification of a real system, and as such it is often impossible to fully analyse a system. Also, the construction of a model can, on the one hand, introduce errors not present in the original specification, while, on the other hand, it may hide errors that are present in the original specification.

Models for the OM/RR protocol were developed from the formal specifications by abstraction and simplification. These models, and not the complete formal specifications, were the subject of verification with EUCALYPTUS. The justification for their development is given in Section 3.3, while Section 3.4 gives a brief documentation of the results obtained in the validation, verification and analysis of the models.

### 3.3 From Specifications to Models

The LOTOS constructs that are not supported by tools of the tool-set EUCALYPTUS are the following: no process recursion is allowed on the left and right hand part of the parallel-operator `|[..]|`, nor on the left hand part of the enable-operator `>>` and the disable operator `[>`. Furthermore, data type definitions are restricted to define only finite and enumerable sorts, instead of possibly infinite sorts. Most of these problems can be overcome by considering only a subset of the behaviour of the specifications. Design requirements in the case of the OM/RR protocol that led to a LOTOS description using unsupported LOTOS constructs are the following:

- An unbounded number of concurrent associations should be supported.
- An unbounded number of messages can be in transit in the communicating state at any moment in time.
- Each association can have infinitely many *sessions*, where a session is the period between and including the successful establishment and release of an association.

In the LOTOS specification, each association is described as an independent entity, not affecting (and aware of) the other associations. The unbounded number of associations is specified using process recursion combined with an interleaving operator (see Equation 3.2). Process `associations` represents the dynamic behaviour of *all* associations. Each particular association, specified by process `association`, is identified by an element of the (possibly infinite) set `AIdSet`.

```
PROCESS associations[OM_SAP] ( A : AIdSet ) =                          (3.2)
    CHOICE aa : AId [aa isin A] ->
    (   association[OM_SAP] (aa)
    ||| associations[OM_SAP] (remove (aa,A) ) )
ENCPROC (* associations *)
```

The use of the interleaving operator means there is no synchronisation on service access points between different associations, hence they are specified as being truly independent entities; therefore, it is fair to assume that considering only *one* association does not affect the validity of results for an unbounded number of associations. Thus, the model considers only process `association(aa)` for some `aa` $\in$ `AIdSet`. The same reasoning holds for the number of messages in the communicating state that can be in transit at any moment in time. In order to obtain some of the effects of the message reordering that has to be allowed by the OM/RR protocol, only a few messages need to be considered. Since a distinction has been made between user confirmed and unconfirmed service elements, which represent classes of service elements with equal numbers and types of service primitives, it is only fair to consider at least one message of each of these two

classes to represent *all* messages in transit. As far as the first two problem areas in the specification are concerned a reasonable solution exists. The third point of attention, i.e. the infinitely many sessions an association can have, is a more profound problem. The OM-service layer requires that sessions cannot be distinguished from one-another, and it is up to the OM-protocol entities and the RR-service layer to take care that this is actually the case. Yet, within an OM-protocol entity, a distinction *has* to be made towards different sessions, in order to eliminate the influence past sessions might have on a current session. Here, no real solution can be provided. Although considering only one or two sessions does not satisfactorily abstract from the infinite number of sessions, due to complexity reasons, no other options exist.

Concrete choices that have been made in writing down the dynamic behaviour of the models are listed below.

- Both the OM-service and the OM-protocol entities are restricted to one association only.
- Both the OM-service and the OM-protocol entities support at most three communication messages that can be in transit during the communicating state (see Fig. 5) at any point in time.
- The OM-protocol entities are restricted to *at most* two sessions.
- For the RR-service, the number of messages that can be in transit is reduced to five.

Further simplifications that have been made concerned the data types, written as ACT-ONE abstract data types in LOTOS. In general, the number of service elements was reduced by considering only the different *classes* of service elements instead of every single service element. The tool CÆSAR.ADT was used to compile the abstract data types. In compiling the data types, an anomaly in this tool was discovered. Instead of being able to enumerate the following (enumerable) sort `OSP`:

```
...
SORTS OSP
OPNS enablereq  (*! constructor *) : AID, ID -> OSP
     enableind  (*! constructor *) : AID, ID -> OSP
     enableconf (*! constructor *) : AID, ID, BOOL -> OSP
...
```

CÆSAR.ADT reports a warning about a theoretical limitation stating that no enumeration of this type definition can be made. This is against the intuition since in our case the sorts `AID`, `ID` and `BOOL` are all enumerable and finite. Indeed, an enumeration of the sort `OSP` does exist!

### 3.4 Verification and Analysis of the Models

The models that were obtained by applying the restrictions in Section 3.3 were analysed with tools from the tool-set EUCALYPTUS, in particular, with CÆSAR, CÆSAR.ADT and ALDÉBARAN. CÆSAR.ADT compiles the data types into a processable form, CÆSAR generates a *labelled transition system* (LTS) from a LOTOS model, which can then be analysed with ALDÉBARAN. The goal was in this case two-fold:

- show absence of deadlock in both the OM-service and in the decomposition consisting of the OM-protocol entities and the RR-service;
- find a formal relation (if one exists!) between the OM-service and its decomposition consisting of the OM-protocol entities and the RR-service, that is as strong as possible, i.e. verify equation (2.1).

The model for the OM-service does not distinguish between different sessions, whereas the model for the OM-protocol does. Essentially, the model for the OM-service describes an unbounded number of sessions and the model for the OM-protocol describes a bounded number of sessions. Hence, only a preorder relation between the two models can be expected.

The first activity was to validate the models against the original LOTOS specifications, using the simulation tool XSIMULATOR, which is comparable to SMILE. Alongside, each time changes were

made to one of the models, absence of deadlock was checked. The final versions of the OM-service model and the OM-protocol model – with, due to complexity reasons, the additional restriction of considering only *one* session – could be proved to have absence of deadlock using ALDÉBARAN. At that point, simulations of the OM-service and the OM-protocol did no longer reveal any strange behaviour. The restriction of the RR-service model to model five messages in transfer proved to be minimal: allowing only four messages in transfer introduced a deadlock in the OM-protocol.

The second objective was achieved by showing that a *safety preorder* [9] exists between the OM-service and the OM-protocol decomposition (again, due to complexity reasons, restricted to only *one* session). This was proved using the tool ALDÉBARAN. This also is the strongest relation that can be shown to exist with ALDÉBARAN between the OM-service and the OM-protocol, as ALDÉBARAN produced counter-examples for stronger relations. Basically, the safety preorder says that the protocol shows only behaviour that is allowed by the service.

A few comments and statistics for the generation of the LTSs are in order. Basically, two methods for generating an LTS from a LOTOS-source are possible: the compositional and the non-compositional method. The non-compositional method is easiest to use, since it requires no insight in the LOTOS-source. However, the drawback may be that the LTSs generated this way are too big to be calculated on most machines. The compositional method on the other hand, allows one to generate LTSs of orthogonal parts of the LOTOS-source and combine these LTSs (or the LTSs that have been reduced to their strong bisimulation normal form) in a later stage. This yields smaller LTSs in usually less time. Decomposing the LOTOS-source into these orthogonal parts, however, can be very difficult and may not always be the most efficient way. In generating LTSs for our models, we used both strategies. The results (obtained on a Linux based Intel Pentium II-300 Mhz, 64 Mb with 128 Mb swap) are listed in Table 2.

| | # States | # Transitions | # $\tau$-Transitions | Reduction |
|---|---|---|---|---|
| OM-service | 420,611 | 2,404,442 | n.a. | none |
| non-compositional | n.a. | n.a. | n.a. | strong |
| OM-service | 97,528 | 492,714 | 150,948 | none |
| compositional | 52,538 | 283, 512 | 80,720 | strong |
| $\text{OM-protocol}_1$ | n.a. | n.a. | n.a. | none |
| non-compositional | n.a. | n.a. | n.a. | strong |
| $\text{OM-protocol}_1$ | 47,525 | 209,190 | 25,238 | none |
| compositional | 5,180 | 21,272 | 14,942 | strong |
| $\text{OM-protocol}_2$ | n.a. | n.a. | n.a. | none |
| non-compositional | n.a. | n.a. | n.a. | strong |
| $\text{OM-protocol}_2$ | n.a. | n.a. | n.a. | none |
| compositional | n.a. | n.a. | n.a. | strong |

**Table 2.** *Statistics for the generation of the LTSs for the* LOTOS*-sources. The subscripts* 1 *and* 2 *indicate the number of considered sessions.*

In Table 2, n.a. stands for *not available*. This means that the results of these operations could not be computed due to the limitations on either the hardware, or the available time to do the calculation. For instance, the time spent on calculating the LTS for $\text{OM-protocol}_2$ – the LTS for the combination of RR-service and two OM-protocol entities with two sessions – using a compositional method, was aborted after running for more than five days, since CÆSAR reported that still only approximately 65% of the total state space was generated and this figure had been stable for four days. An attempt was made to calculate the LTS for $\text{OM-protocol}_2$ on a more powerful machine, however, no significant improvement over the Linux machine was noted.

To illustrate the gain in time that can be achieved using the compositional method over the non-compositional method, consider that the results for the OM-service in a non-compositional

strategy took more than three hours, whereas the compositional strategy took less than 30 minutes on the same machine.

### 3.5 Specifications and Models

During formalisation, verification and analysis of the OM/RR protocol we made a distinction between a formal *specification* and a formal *model*. They turn out to be related, but different uses of formal methods, which are both useful.

Specifications are meant to prescribe the behaviour of a system. They serve as the basis for implementation, coding and testing. Consequently, specifications should be as complete as possible, taking into account all possibilities of behaviour. On the other hand, models are meant to focus on one particular aspect of a system while abstracting from other aspects.

Specifications also abstract from many details, but these details concern properties which are not considered at all. For a given level of abstraction, or for a given family of properties under consideration, specifications should be complete. Models, however, also abstract from properties which are, in principle, under consideration. Consider, for instance, real-time properties of the OM/RR protocol: they are not considered at all in our analysis and they appear neither in our specifications nor in our models. On the other hand, in principle, we are interested in the behaviour of protocol entities in the presence of 5 other protocol entities and communicating via 9 associations. This behaviour is described in the formal specification, but for the models we made simplifications and restricted them to one other protocol entity communicating via only one association.

A specification is often too complex and contains too much detail for checking of properties, e.g. model checking. This is especially true if property checking is performed using a tool; state of the art tools, like EUCALYPTUS, cannot deal with large and complex formal descriptions; restrictions and simplifications are necessary. Hence, for checking of properties a formal model is developed. A model is a simplification of the system in which a lot of detail has been removed and only those aspects which are deemed important for the property at hand, are formally expressed. But, because of this, a model cannot be used, e.g. as the basis for testing of implementations: since some allowed behaviours may have been removed for simplification it cannot be decided for actions executed by an implementation whether they are allowed or not.

Making models is an intricate process in which the right level of abstraction must be chosen to achieve a good compromise between simplicity and completeness. As a consequence, verifying a property for a model does not give certainty about the system: any verification is only as good as the validity of the model on which it is based. Validity of the model is usually assumed or informally, using hand waving arguments, reasoned upon.

For the OM/RR protocol we first developed complete specifications (with respect to the properties under consideration), see Section 2.3. These specifications were analysed using tools from LITE. These tools have low functionality but do not impose restrictions on the formal specifications. Based on the specifications, simplifications were made such as restricting the number of associations, sessions, protocol entities, different kinds of messages, etc., see Section 3.3. This led to models which were amenable for the EUCALYPTUS tools. Properties were checked on these models but no certainty is obtained that any checked property, e.g. deadlock freedom, also holds for the complete specification. Only informal and hand waving arguments were given in Section 3.3.

When constructing specifications and models, different approaches are possible. On the one hand, one can start by first constructing the complete specification and then try to validate this specification. This validation may involve constructing models from the specification and verify these against desired properties. Another possibility is to consider models of the vital behaviour first and prove them correct by means of verification, and gradually extending these models to become a specification. Finally, a mixture of both approaches can be used, by constructing models and a specification hand in hand. The first approach is the one used in this project. Although very suitable, it is felt that the third approach would have been more convenient in the end.

## 4 Conclusions

The results described in this paper deal with a case-study of an industrial *mission critical* system, called the *Operator Support System* (OSS). Various inherently complex communication protocols, part of the OSS, are potential hazards for the correct functioning of the OSS. Because of the mission critical aspects of the OSS, CMG Den Haag B.V. decided to include formal methods in the development of the OSS. This paper describes the formalisation, analysis and verification of one of these protocols, the OM/RR protocol, using the *Formal Description Technique* (FDT) LOTOS. The number of hours spent on this project roughly equals 9 man months. This time includes the learning time of the FDT LOTOS and familiarising with the OSS project and the tool-sets.

As we showed, the trajectory of formalising an informal document itself already reveals many omissions, ambiguities and errors; as such, formalising informal documents is beneficial, even without doing anything with the resulting formal specification. Due to the ambiguities and omissions, the real functionality of the OM/RR protocol is shrouded in mist. This makes it hard to judge the quality of the protocol itself. Since the OM/RR protocol serves as one of the backbones of the OSS, it forms a severe risk for the reliability of the OSS.

Despite the severe omissions and ambiguities in the informal document we tried to develop a formal specification of the OM/RR protocol. Some of the ambiguities could be eliminated by taking the design requirements into account. However, the general lack of documentation concerning the design requirements did not allow for such solutions in general. Instead, conversations with the developers of the protocol served to find out about the assumptions and design requirements that were made during the trajectory of writing the formal specifications. These conversations revealed that the developers considered the dynamic behaviour to be trivial. As such, it had not been included in official documents. Formalising the behaviour of the protocol, however, raised many questions. This implies the dynamic behaviour cannot be considered trivial.

The conversations with the developers can be characterised as conversations not so much about design and specification, as about implementation. Our level of abstraction of talking about the protocol differed from the level of the developers. Our interest mainly focussed on the *service* of the protocol, yet, the developers mainly focussed on how to overcome various problems when *implementing* the protocol in real-life. Retrieving information and getting to the essence of the matter proved harder than was expected.

Although some of the questions that were found in the trajectory of formalising the OM/RR protocol have been answered, many questions remained open. As a result, the formal specification of the OM/RR protocol can only be considered as a first attempt. There is no guarantee at all that our formal description corresponds with the intentions of the developers. Consequently, the greatest benefit that is obtained in this project is not in the formal specifications, but in the number of questions that have been raised in the process of formalising, and that can be a basis for a next version of the protocol. With respect to the OM/RR protocol, the conclusion of the formalisation project is that the current document as it is [7], is not a good basis for implementing the crucial OM/RR protocol; it must certainly be improved.

The FDT LOTOS turned out not to be of great help in clarifying various questions posed to the developers of [7]. The reason was that the LOTOS descriptions were not really understood by them. An explanation for this can be found in the style that was used to specify the system, i.e. a *constraint-oriented* specification style [22], and their inexperience with the FDT LOTOS. This style is very useful in specifying a system, however, reading a specification in this style puts an extra burden on the reader, since it requires a reader to combine various constraints found throughout the specification.

Another issue can be that a process algebraic approach might not be the most suited for use in an industrial environment when confronted with people with no clear background in formal methods. Case studies are required, in which various formal methods are used to tackle a problem, in order to determine the most suited method (both in apprehension and application) for various problems. It is expected that visual formal methods, e.g. SDL [17] or MSC [18], might be more suited when explaining ambiguities and omissions to people with less experience in formal methods.

It can also be very beneficial to present the results of validations in a visual formalism. For instance, the traces of LOTOS simulations could be presented as MSC. This was used successfully in the BOS project (cf. SPIN [15] and [10]). Complete specifications in visually oriented languages, however, are thought to be hampered by the same problem of complexity. An interesting exercise would be to investigate the time it takes for people to read and fully understand various specifications in various formalisms. While LOTOS was not easily understandable by the developers of the protocol, the people doing the formalisation were reasonably satisfied. There are some anomalies in the language, e.g. see Section 3.1, but as a whole LOTOS is useful.

Although a full-fledged specification is inevitable when designing protocols, a fully *formal* specification of dynamic behaviour might not be most useful. On the one hand, writing a very detailed formal specification is a very cumbersome task, taking a lot of time, and on the other hand, vital parts of a specification are less likely to be recognised within a very detailed specification. As such, formal models or partial formal specifications may be more suited.

The tool-sets used in our project, EUCALYPTUS and LITE, are useful for this kind of projects. Confidence in a specification is only achieved by validation and verification. For any realistically sized formal description this cannot be done manually; tools are indispensable. The simulator SMILE contained in LITE turned out to be useful, mainly in the initial phases of development, for analysing specifications, but its usage was rather cumbersome and laborious. CÆSAR and ALDÉBARAN contained in EUCALYPTUS were mainly used for checking properties on models. EUCALYPTUS was found to have an omission concerning the abstract data types in the tool CÆSAR.ADT, see Section 3.3. Moreover, a thorough investigation concerning the efficiency of this tool-set with respect to memory usage is needed. Memory overflow, caused by state-space explosion, is one of the best known and most studied problems restricting industrial usage of this kind of tools. It would be interesting to investigate the use of the compositional approach in order to be able to give some general guidelines for optimal usage of this approach, and ultimately automate this approach. Last but not least, feedback from tools is an important issue, e.g. a deadlock found in a labelled transition system, is hard to locate in the corresponding LOTOS model. A tool helping to bridge the gap between these two representations would have saved much time.

## Acknowledgements

The first author would like to thank Jos Baeten of the Eindhoven University of Technology for the supervision during this project. We thank the members of the SIG-MCS/FM group of CMG, especially Michel Chaudron and Bart Botma for the numerous discussions on any topic and Jan Friso Groote for proof-reading early versions of this paper. CMG Den Haag B.V. is thanked for the pleasant working atmosphere and the financial support during this project.

## References

1. ISO/IEC 10164. *Systems Management.* Geneva, 1990.
2. ISO/IEC 8807. *Information Processing Systems, Open Systems Interconnection, LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* Geneva, 1989.
3. ISO/IEC 8824. *Information Processing Systems - Open Systems Interconnection, Specification of Abstract Syntax Notation One (ASN.1).* Geneva, March 1988.
4. ISO/IEC 9072. *Remote Operations.* Geneva, 1990.
5. ISO/IEC 9595. *Common Management Information Service Definition.* Geneva, 1991.
6. ISO/IEC 9596. *Common Management Information Protocol Specification.* Geneva, 1991.
7. Adviesdienst voor Verkeer en Vervoer. *OM/RR and Association Management Specifications, version 1.2*, July 18th 1997.
8. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In *Computer Networks and ISDN Systems*, volume 14, pages 25–59, 1987.
9. A. Bouajjani, J. C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for Branching Time Semantics. In *18th ICALP*. Springer-Verlag, 1991.

10. M. Chaudron, J. Tretmans, and K. Wijbrans. Lessons from the Application of Formal Methods to the Design of a Storm Surge Barrier Control System. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – World Congress on Formal Methods in the Development of Computing Systems II*, volume 1709 of *Lecture Notes in Computer Science*, pages 1511–1526. Springer-Verlag, 1999.
11. R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.
12. H. Eertink. Executing LOTOS specifications: The SMILE tool. In T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors, *LOTOSphere: Software Development with LOTOS*, pages 221–234. Kluwer Academic Publishers, 1995.
13. H. Garavel. OPEN/CAESAR: An Open Software Architecture for Verification, Simulation, and Testing. In B. Steffen, editor, *Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84. Springer-Verlag, March 1998.
14. Hubert Garavel. An overview of the EUCALYPTUS toolbox. In Z. Brezočnik and T. Kapus, editors, *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design (Maribor, Slovenia)*, pages 76–88. University of Maribor, 1996.
15. G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
16. ISO. *Information Processing Systems, Open Systems Interconnection, Basic Reference Model.* International Standard IS-7498. ISO, Geneve, 1984.
17. ITU-T. *ITU-T Recommendation Z.100: Specification and Description Language (SDL).* ITU-T, Geneva, 1992.
18. ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC).* ITU-T, Geneva, 1996.
19. D. Steedman. *Abstract Syntax Notation One (ASN.1) - The Tutorial and Reference.* Isleworth: Technology Appraisals, 1990.
20. J. Tretmans, K. Wijbrans, and M. Chaudron. Software Engineering with Formal Methods: The Development of a Storm Surge Barrier Control System – Seven Myths of Formal Methods Revisited. In S. Gnesi and D. Latella, editors, *Fourth Int. ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'99) – Proceedings of the FLoC Workshop*, volume II, pages 225–237, Pisa, Italy, July 11–12 1999. Servizio Tecnografico Area di Ricerca del CNR.
21. C. A. Visser and L. Logrippo. The Importance of the Service Concept in the Design of Data Communications Protocols. In M. Diaz, editor, *Protocol Specification, Testing, and Verification V*, pages 3–17. Elsevier Science Publishers B.V. (North-Holland), 1986.
22. C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.
23. T.A.C. Willemse. The Specification and Validation of the OM/RR-Protocol. Master's thesis, Eindhoven University of Technology, July 1998.

# Automatically Verifying an Object-Oriented Specification of the Steam-Boiler System

Paulo J. F. Carreira and Miguel E. F. Costa
{pcarreira, ecosta}@oblog.pt

OBLOG Software S.A.,
Alameda António Sérgio 7, 1-A
2795-023 Linda-a-Velha, Lisboa - PORTUGAL
tel:+351-214146930 fax:+351-214144125

**Abstract.** Correctness is a desired property of industrial software systems. Although the employment of formal methods and their verification techniques in embedded real-time systems has started to be a common practice, the same cannot be said about object-oriented software. This paper presents an experiment of a technique for the automated verification of a subset of the object-oriented language OBLOG. In our setting, object-oriented models are automatically translated to LOTOS specifications using a programmable rule-based engine included in the Development Environment of the OBLOG language. The resulting specifications are then verified by model-checking using the CADP tool-box. To illustrate the concept we develop and verify an object-oriented specification of a well known case study—the Steam-Boiler Control System.

**Keywords:** Automatic Verification, Code Generation, LOTOS, Model-Checking, Object-Oriented Systems, Steam-Boiler.

## 1 Introduction

The employment of an automatic method for verifying properties about formal specifications known as *model-checking* [QS82,CES86,VW86,Kur90] experienced a dramatic growth. It has emerged as an effective way of finding errors and proving correctness of hardware, and, more recently, software systems.

However, the applicability of this technique depends on the existence of models for the specifications with a finite number of states. Specifications of real-world systems often have state-spaces that are infinite or so large that would disable their verification in an automated[1] way. Nevertheless, much effort has been put in additional techniques that, when used in a combined way, allow the exploration of the state-spaces of many real-world systems [CW96].

The lack of automated tools is only one of the reasons for the weak acceptance of formal methods for software development. Another lies in the fact that specification languages still require some degree of mathematical sophistication. Object-oriented graphical languages like UML [BJR97] and StateCharts

[1] By *automated* we mean fully automated, without user intervention.

[HLN+90] were proposed and advocated as means to overcome the above situation. However, producing complete specifications using graphical specification languages is a labor-intensive task. Such specifications often become overwhelming thus compromising the initial goal of being easier to read.

The object-oriented language OBLOG [OBL99] is being used in industry for the specification and deployment of critical parts of software systems [AS96]. OBLOG models can be developed by using both graphical and textual notations, making feasible the specification of complete systems with thousands of objects and classes.

In this paper we illustrate the applicability of model-checking technology in the verification of object-oriented software specifications. We present an experiment with a technique that allows fully automated verification of a subset of OBLOG specifications by applying model-checking to corresponding LTSs (Labelled Transition Systems). To obtain these LTSs the formal semantics for OBLOG should have been defined. However, since the language is still under development, only an intuitive semantics is available. We chose to base our approach on an intermediate translation to LOTOS [ISO88] specifications that are subsequently expanded to LTSs, thus bridging the gap between the intuitive semantics of OBLOG and the needed formal semantics over LTSs. Furthermore, the effort of implementing an algorithm to expand data non-determinism is greatly reduced by using Cæsar.Adt [Gar89], an abstract datatype compiler for LOTOS included in Cadp [FGK+96].

In order to test our ideas, we decided to work with a simplified version of the Steam-Boiler Control System, a well known example from literature [ABL96], which allowed a faster analysis of the problem and provided other results for comparison.

Our paper is organized as follows: In Section 2, we present the requirements of a simplified version of the Steam-Boiler Control System and its modeling with OBLOG. The translation mechanism for producing LOTOS code is detailed in Section 3. We present and verify a formalization of the system requirements in Section 4, and Section 5 draws the conclusions of this work.

### 1.1 Related Work

There have been other attempts to verify the Steam-Boiler System by model-checking but none of them, to the best of our knowledge, used a high level object-oriented language. In [WS96], Willig and Schieferdecker developed a Time-Extended LOTOS specification. The system was validated through simulation and verified for deadlock freedom using full state-space exploration techniques. They used Cadp on a restricted model without time and without failures.

A formalization of the problem into Promela without time is given by Duval and Cattel [DC96]. Their model also abstracts from communication failures and major properties of the system are reported to have been verified on a fully automated way using the Spin Model-Checker. Jansen et al. [JMMS98] report the verification of Amber specifications using a translation into Promela. This

translation allowed the use of SPIN in the automated verification of finite-state subsets of AMBER.

## 2 Modeling the Steam-Boiler Controller System

The Steam-Boiler Control system is composed of a Micro-Controller connected to a physical system apparatus consisting of an Operator Desk and a Steam-Boiler attached to a turbine. There is also a Pump to provide water to the Boiler, an Escape Valve to evacuate water from the Boiler and devices for measuring the level of water inside the Boiler and the quantity of steam coming out. The Boiler is characterized by physical limits M1 and M2, and a safety range between N1 and N2. When the system is operating, the water level can never go above M1 or below M2, otherwise the Boiler could be seriously damaged. The safety range establishes boundaries that, when reached, must cause a reaction from the Controller that reverts the increasing or decreasing tendency of the water level.

### 2.1 System requirements

The Controller has different modes of operation, namely: *stopped*, *initialization*, *normal* and *emergency stop*. Initially the Steam-Boiler is switched off and the Controller is in stopped mode. System operations start when the start button of the operator desk is pressed. However, before the Boiler can start, the Controller must ensure that the water inside the Boiler is at an adequate level (between N1 and N2). To do this, it enters the initialization mode in which it uses the Water Pump and the Escape Valve to regulate the water level. When a safe range is reached, the Controller switches to normal mode and the production of steam initiates. In normal mode the Controller guarantees a safe water level inside the Boiler by starting and stopping the Pump. If something goes wrong, and the operator pushes the stop button, the Controller enters emergency stop mode and shuts down the Steam-Boiler.

The system can be further characterized by a set of requirements that are summarized as follows:

1. When the start button is pressed and the system is stopped the Controller enters the initialization mode.
2. When the Controller is in the initialization mode and the water level is below N1, the Pump must be started.
3. When the Controller is in the initialization mode and the water level is above N2, the Valve must be opened.
4. When the Controller is in the initialization mode and the water level is in the range N1 to N2, the Controller switches to normal mode.
5. When the Controller switches to normal mode and the Valve is opened, the Valve must be closed.
6. When the Controller is in normal mode, the Pump is started and the water level is above N2, the Pump must be stopped.

7. When the Controller is in normal mode, the Pump is stopped and the water level is below N1, the Pump must be started.
8. When the stop button is pressed the Controller enters emergency stop mode.
9. When the water level of the Boiler is greater than N2, it will eventually become lesser than or equal to N2.
10. When the water level of the Boiler is less than N1, it will eventually become greater than or equal to N1.
11. If the Pump is started, the water will never reach a level above M2.
12. If the Boiler is started, the water will never reach a leveçl below M1.
13. The Valve can only be opened if the Controller is in initialization mode.

### 2.2 The OBLOG Model

OBLOG (OBject LOGic) refers both to a language and a development environment. The language OBLOG is a strongly-typed object-oriented specification language. Specifications are developed in a hierarchical fashion using *specification regions*. A specification region can be a class or an object encapsulating local declarations consisting of constants, attributes and operations as well as local specifications of datatypes and nested specification regions. Class and object operations can be implemented by several methods distinguished by corresponding enabling conditions.

In the original specification of the Steam-Boiler problem, the Controller interacts with the physical units through a single communication medium which has a specialized protocol defined for it. Our specification abstracts communication by modeling it with usual interaction between objects i.e., calls to object operations. However, we attempted to preserve the Controller's viewpoint by which the physical units are seen as a single entity composed of several other simpler entities.

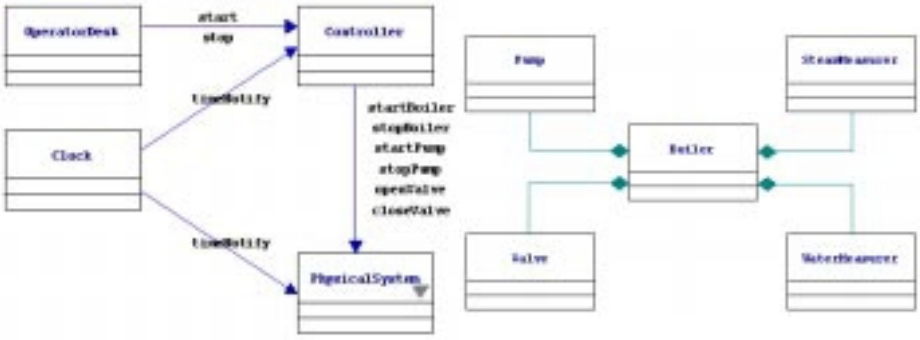



**Fig. 1.** a) Top-level objects; b) Objects in the `PhysicalSystem` specification region

At the top-level of our specification we have the `Controller` object which models the Controller software component and the `PhysicalSystem` object mod-

eling the unified composition of all the physical units comprising the Steam-Boiler apparatus. In the specification region of this object are models of those units, namely the `Boiler`, `Valve`, `Pump`, `WaterMeasurer` and `SteamMeasurer` objects. Finally, also at top-level, are the `OperatorDesk` object and the `Clock` object, which is used to model time evolution.

In OBLOG there are two ways of initiating activity, *signal reaction* operations (denoted with a prefixing `^`) and *self-fire* operations (denoted with a prefixing `!`). Reactions are triggered by signals sent by the external environment and we use them to model the events of pressing the start and stop buttons in the operator desk. Self-fire operations are used to model pro-active behavior. In our setting, since we do not have time constructs in OBLOG, time evolution was modeled with a self-fire operation of the `Clock` object named `!clockTic()`.

The `!clockTic()` operation notifies both the `PhysicalSystem` and the `Controller`. The `PhysicalSystem` forwards this notification to the `Boiler`, which computes the new water level based on the current water level, the state of the `Valve` and `Pump` objects and its own internal state[2]. When the `Controller` is notified, it takes the appropriate actions according to its current operation mode as detailed above in the requirements section.

When a signal corresponding to the action of pressing the start or stop button is sent to the system, it is caught by the `OperatorDesk` object which contains two corresponding signal reaction operations named `^startButton()` and `^stopButton()` respectively. When the `Controller` is in stopped mode and the `^startButton()` operation is triggered, the Controller is started. Similarly, when the `^stopButton()` operation is triggered, the `Controller` is sent to emergency stop mode.

## 3 Translating OBLOG Specifications into LOTOS

An OBLOG specification can be automatically translated to another language using an automatic code generation tool included in the OBLOG tool-set. Using this, we developed a translation of a sequential subset of the OBLOG language into LOTOS, which is a standard Formal Description Language for software systems. This language is composed of two specialized sub-languages for specifying data and control parts. The data part is specified using the language ActOne [EM85] which is based on the theory of abstract datatypes. The control part is specified using a process algebraic language that combines and extends features of both CSP [Hoa85] and CCS [Mil89].

### 3.1 Translation Framework

The current framework is an evolution from previous studies in emulating subsets of the OBLOG language with process algebraic approaches to allow automatic

[2] Recall that the `Valve` object can be either opened or closed, and both the `Boiler` and the `Pump` can be either started or stopped.

```
object Controller
declarations

  data types
    OperationMode = enum{
      Stopped,
      Initialization,
      Normal,
      Emergency,
    } default Stopped;

  attributes
    object
      mode : OperationMode
        := Stopped;
  operations
    object
      start();
      stop();
      timeNotify();

body
  methods
    start
    method start is
      if mode = Stopped
        set mode := Initialization;
      endif
    end

    timeNotify
    method tnStopped
    enabling
      mode = Stopped;
    is
      skip;
    end

    timeNotify
    method tnInit
    local
      waterLevel : Integer;
    enabling
      mode = Initialization;
    is
      call PhysicalSystem.
        getWaterLevel(waterLevel);
      if waterLevel < N1
        call PhysicalSystem.
          startPump();
      endif
      if waterLevel > N2
        call PhysicalSystem.openValve()
      endif
      if (N1 <= waterLevel) AND
        (waterLevel <= N2)
        call PhysicalSystem.closeValve();
        call PhysicalSystem.startBoiler();
        set mode := Normal;
      endif
    end

    timeNotify
    method tnNormal
    local
      waterLevel : Integer;
    enabling
       mode = Normal;
    is
      call PhysicalSystem.
        getWaterLevel(waterLevel);
      if waterLevel < N1
        call PhysicalSystem.startPump();
      endif
      if waterLevel > N2
        call PhysicalSystem.stopPump();
      endif
    end

    timeNotify
    method tnEmergency
    enabling
       mode = Emergency;
    is
      skip;
    end

    stop
    method stop is
      if (mode = Normal) OR
        (mode = Initialization)
        call PhysicalSystem.stopPump();
        call PhysicalSystem.closeValve();
        call PhysicalSystem.stopBoiler();
        set mode := Emergency;
      endif
    end

end object
```

**Fig. 2.** Specification code of the `Controller` object

verification [Car99]. These approaches are based on a translation that represents each object as a parallel composition of two recursively instantiated processes, one dedicated to the state and the other to the behavior of the object. The two processes synchronize through designated gates for reading and writing attribute values. In fact, this coding relies heavily on LOTOS gates, also using them for both operation calls and parameter passing, resulting in a high degree of non-determinism which causes the explosion of the state-space. In our framework, in order to produce a LOTOS specification that can be compiled and verified in sensible time, an attempt was made to reduce non-determinism as much as possible; thus, gates were used as least as possible.

The state attributes of all the objects were merged into a global system variable that undergoes transformations corresponding to the behavior of the objects. To support this, special abstract datatypes are defined, namely type $ObjType$ that for each object $Obj_i$ ($i$ ranging in the number of objects in the system) with attributes $A_1 : T_{A_1}, \ldots, A_n : T_{A_n}$ defines a sort named $ObjSort_i$, and type $SysState$ that provides a representation of the global system state using each of the $ObjSort_i$ sorts. The definition is as follows, where $n$ is the number of attributes of object $Obj_i$ and $m$ is the number of objects in the system:

**type** $ObjState$ **is** $T_{A_1}, \ldots, T_{A_n}$
  **sorts**
    $ObjSort_i$
  **constructors**
    $mkObj_i : T_{A_1} \times \cdots \times T_{A_n} \rightarrow ObjSort_i$
  **functions**
    $setObj_i A_1 : ObjSort_i \times T_{A_1} \rightarrow ObjSort_i$
    $getObj_i A_1 : ObjSort_i \rightarrow T_{A_1}$
    $\ldots$
    $setObj_i A_n : ObjSort_i \times T_{A_n} \rightarrow ObjSort_i$
    $getObj_i A_n : ObjSort_i \rightarrow T_{A_n}$
  **equations**
  $\forall x_1 : T_{A_1}, \ldots, x_n : T_{A_n}$
  $\forall y_1 : T_{A_1}, \ldots, y_n : T_{A_n}$
    $setObj_i A_1(mkObj_i(x_1, \ldots, x_n), y_1) = mkObj_i(y_1, x_2, \ldots, x_n)$
    $getObj_i A_1(mkObj_i(x_1, \ldots, x_n)) = x_1$
    $\ldots$
    $setObj A_n(mkObj_i(x_1, \ldots, x_n), y_n) = mkObj_i(x_1, \ldots, x_{n-1}, y_n)$
    $getObj_i A_n(mkbj(x_1, \ldots, x_n)) = x_n$
**endtype**

**type** $SysState$ **is** $ObjType$
  **sorts**
    $SysState$
  **constructors**
    $mkSys : ObjSort_1 \times \cdots \times ObjSort_m \rightarrow SysState$
  **functions**
    $setObj_1 : SysState \times ObjSort_1 \rightarrow SysState$
    $getObj_1 : SysState \rightarrow ObjSort_1$
    $\ldots$
    $setObj_m : SysState \times ObjSort_m \rightarrow SysState$
    $getObj_m : SysState \rightarrow ObjSort_m$
  **equations**
  $\forall u_1 : ObjSort_1, \ldots, u_m : ObjSort_m$
  $\forall v_1 : ObjSort_1, \ldots, v_m : ObjSort_m$
    $setObj_1(mkSys(u_1, \ldots, u_m), v_1) = mkSys(v_1, u_2, \ldots, u_m)$
    $getObj_1(mkSys(u_1, \ldots, u_m)) = u_1$
    $\ldots$
    $setObj_m(mkSys(u_1, \ldots, u_m), v_m) = mkSys(u_1, \ldots, u_{m-1}, v_m)$
    $getObj_m(mkSys(u_1, \ldots, u_m)) = u_m$
**endtype**

The main difference to previous approaches is that we do not use statements of the kind `G?s:SysState` in the LOTOS code, which are the main causes of the state-space explosion problem because they correspond to a non-deterministic choice ranging in the domain of the accepted variables.

In fact, no part of the system state is explicitly sent through any gate. Rather, when operations are called, the corresponding processes that encode them are instantiated taking the system state as a parameter. These processes are composed of subprocesses that correspond to the several methods of each operation, which

are further composed of other subprocesses implementing elementary actions—called *quarks* in OBLOG—like setting the value of an object attribute or calling other operations. Generally, a behavior component $bc$ (that can be an operation, a method or a quark) is translated to a process that receives the system state as a parameter, forwards it to the subprocesses or applies a transformation to it, returning a potentially altered version of the system state. The translation of $bc$, denoted by $\boldsymbol{proc}_{bc}$, renders the following:

```
proc_bc ≡
      process name_bc[G](s:SysState,in_bc) : exit(SysState,out_bc,Bool) :=
        action_bc
      where
        subprocs_bc
      endproc
```

where $\boldsymbol{G}$ is a set of gates, $\boldsymbol{name}_{bc}$ is a unique identifier for the behavior component, $\boldsymbol{action}_{bc}$ is the action taken by the behavior component and $\boldsymbol{subprocs}_{bc}$ is the declaration of subprocesses in the case of a compound behavior component. If $bc$ is an operation with input (resp. output) parameters, these will be included in the $\boldsymbol{in}_{bc}$ (resp. $\boldsymbol{out}_{bc}$) list. Moreover, if $bc$ is a method with local variables or a quark within a method with local variables, these will also be in $\boldsymbol{in}_{bc}$.

In OBLOG, a behavior component may result in failure in which case the `Bool` exit value of its corresponding LOTOS process is `true`. This is, however, not relevant in this report since the model we present does not allow failure in any case. This feature was only included in the framework for genericness sake.

The translation procedure can be summarized, in terms of behavior components, as follows:

**Operations** If $bc$ is an operation composed by methods $M_1, \ldots, M_n$, with input parameters $I_1 : T_{I_1}, \ldots, I_n : T_{I_n}$ and output parameters $O_1 : T_{O_1}, \ldots, O_m : T_{O_m}$ we have:

```
action_bc ≡
     name_M1(s,I1,...,In)
     []
     ...
     []
     name_Mn(s,I1,...,In)
```

$$\boldsymbol{subprocs}_{bc} \equiv \boldsymbol{proc}_{M_1} \cdots \boldsymbol{proc}_{M_n}$$

$$\boldsymbol{in}_{bc} \equiv I_1 : T_{I_1}, \ldots, I_n : T_{I_n}$$

$$\boldsymbol{out}_{bc} \equiv T_{O_1}, \ldots, T_{O_m}$$

**Methods** If $bc$ is a method such that: (1) its parent operation has inputs $I_1 : T_{I_1}, \ldots, I_n : T_{I_n}$ and outputs $O_1 : T_{O_1}, \ldots, O_m : T_{O_m}$ with default values $D_{O_1}, \ldots, D_{O_m}$; (2) has local variables $L_1 : T_{L_1}, \ldots, L_k : T_{L_k}$ with default values $D_{L_1}, \ldots, D_{L_k}$; (3) $Q$ is its implementation quark; we have:

$\boldsymbol{action}_{bc} \equiv$
$\quad \boldsymbol{name}_Q$(s,$I_1$,...,$I_n$,
$\quad\quad D_{O_1}$,...,$D_{O_m}$,$D_{L_1}$,...,$D_{L_k}$)
$\quad$>> accept s2:SysState,
$\quad\quad I'_1$:$T_{I_1}$,...,$I'_n$:$T_{I_n}$,
$\quad\quad O_1$:$T_{O_1}$,...,$O_m$:$T_{O_m}$,
$\quad\quad L'_1$:$T_{L_1}$,...,$L'_k$:$T_{L_k}$,
$\quad\quad$f:Bool
$\quad$in
$\quad\quad$exit(s2,$O_1$,...,$O_m$,f)

$\boldsymbol{subprocs}_{bc} \equiv \boldsymbol{proc}_Q$

$\boldsymbol{in}_{bc} \equiv I_1 : T_{I_1}, \ldots, I_n : T_{I_n}$

$\boldsymbol{out}_{bc} \equiv T_{O_1}, \ldots, T_{O_m}$

**Quarks** In the context of a quark, no distinction is made between input parameters, output parameters and method local variables. Instead, if *bc* is a quark, we say that it has a working set of variables declared as $V_1 : T_{V_1}, \ldots, V_n : T_{V_n}$ that subsume the previous declarations.

If *bc* is an operation call quark of the form **call** $op(!I_1 << V_{I_1}, \ldots, !I_n << V_{I_n}, !O_1 >> V_{O_1}, \ldots, !O_m >> V_{O_m})$ where $!I_i << V_{I_i}$ is an input binding associating input parameter $I_i$ to a local variable $V_{I_i}$, and $!O_i >> V_{O_i}$ is an output binding associating output parameter $O_i$ to a variable $V_{O_i}$, we have that:

$\boldsymbol{action}_{bc} \equiv$
$\quad \boldsymbol{name}_{op}$(s,$V_{I_1}$,...,$V_{I_n}$)
$\quad$>> accept s2:SysState,
$\quad\quad O_1$:$T_{O_1}$,...,$O_m$:$T_{O_m}$,
$\quad\quad$f:Bool
$\quad$in
$\quad\quad$exit(s2,$\boldsymbol{V}[O_i/V_{O_i}]$,f)

$\boldsymbol{in}_{bc} \equiv V_1 : T_{V_1}, \ldots, V_n : T_{V_n}$

$\boldsymbol{out}_{bc} \equiv T_{V_1}, \ldots, T_{V_n}$

where $\boldsymbol{V}$ represents the list of variables $V_1, \ldots, V_n$ and $\boldsymbol{V}[O_i/V_{O_i}]$ represents the list obtained from $\boldsymbol{V}$ by replacing each variable $V_{O_i}$ with its corresponding bound value $O_i$.

To verify the system requirements, these will later be translated to formulas using predicates on the state of the objects. The generation procedure is parameterized with the predicates that belong to a particular formula. The obtained LOTOS specification is such that when modifying an object attribute, if the assignment causes any of these predicates to become true, an appropriate gate is signaled.

Let $p_1, \ldots, p_n$ be predicates that involve an attribute $A$ that is modified and, for each $p_i$, let $p_i(s)$ designate the evaluation of the predicate in a given state $s$. The predicate checking procedure for attribute $A$ is defined by the following processes, where $i$ ranges in $1, \ldots, n$:

$\boldsymbol{check}_i \equiv$
$\quad$process checkP$_i$[$\boldsymbol{gate}_{p_1}$,...,$\boldsymbol{gate}_{p_n}$] (s1:SysState,s2:SysState) : exit :=
$\quad\quad$[NOT($p_i$(s1)) AND $p_i$(s2)] -> $\boldsymbol{gate}_{p_i}$, checkP$_{i+1}$[$\boldsymbol{gate}_{p_1}$,...,$\boldsymbol{gate}_{p_n}$](s1,s2)
$\quad\quad$[]
$\quad\quad$[$p_i$(s1) OR NOT($p_i$(s2))] -> checkP$_{i+1}$[$\boldsymbol{gate}_{p_1}$,...,$\boldsymbol{gate}_{p_n}$](s1,s2)
$\quad$endproc

$\boldsymbol{check}_{n+1} \equiv$
$\quad$process checkP$_{n+1}$[$\boldsymbol{gate}_{p_1}$,...,$\boldsymbol{gate}_{p_n}$] (s1:SysState,s2:SysState) : exit :=
$\quad\quad$exit
$\quad$endproc

where $s$ and $s'$ represent the state of the system respectively before and after the modification of the attribute, and $\boldsymbol{gate}_{p_i}$ is the corresponding gate for each $p_i$ predicate. If $bc$ is an attribute modification quark of the form **set** $A$ `:=` $exp$ where $A$ is an attribute of an object $Obj$ and $exp$ is an expression of the same type as $A$, we have:

$$
\begin{aligned}
&\boldsymbol{action}_{bc} \equiv \\
&\quad \texttt{checkP}_1\,\texttt{[}\boldsymbol{gate}_{p_1},\ldots,\boldsymbol{gate}_{p_n}\texttt{](s,set}Obj\texttt{(s,set}A\texttt{(get}Obj\texttt{(s),}exp\texttt{)))} \\
&\quad \texttt{>>} \\
&\quad \texttt{exit(set}Obj\texttt{(s,set}A\texttt{(get}Obj\texttt{(s),}exp\texttt{)),}\boldsymbol{V}\texttt{,false)}
\end{aligned}
$$

$$
\boldsymbol{subprocs}_{bc} \equiv \boldsymbol{check}_1 \cdots \boldsymbol{check}_{n+1} \qquad \boldsymbol{in}_{bc} \equiv V_1{:}T_{V_1}, \ldots, V_n{:}T_{V_n}
$$

$$
\boldsymbol{out}_{bc} \equiv T_{V_1}, \ldots, T_{V_n}
$$

Other quarks include the modification of local variables and the sequential and conditional quark compositions.

In order to prevent the state-space explosion, another important issue is where activity starts. Instead of allowing any operation to be initiated at any time, activity initiates at only a few well-determined points at a single top--level recursive process, corresponding to the triggering of self-fire operations and reactions to external events. In each instantiation of this scheduler process, every enabled self-fire operation and every reaction to received external signals is called. In this context, the reception of signals is modeled as a choice between receiving or not receiving them i.e., calling the corresponding reaction operations or not. As with predicates, we can also configure the translation procedure to include gates that are used as observers of receptions of signals.

On the first instantiation of the scheduler, the system state is initialized with the default values specified in the declaration of the objects. If an attribute of an object was not given a default value, we convention the corresponding initial value to be non-deterministically chosen in the range of the domain of that attribute. While not affecting the semantic mapping, this convention allows us to verify our properties for every possible initial scenario, in our case in particular, for every possibility of the water level inside the boiler at start-up.

### 3.2 Automatic generation

OBLOG language concepts are represented in an object-oriented Meta-Model as classes. An OBLOG repository can thus be regarded as a collection of instances of these classes.

The OBLOG Generator tool transforms repositories into actual implementations using transformation rules that map concepts described in the Meta-Model into constructs of a given target language. These transformation rules are written in RDL [OBL99] which is a scripting language executed in a specialized

rule-execution engine in the following way: rules can access properties and relationships of repository object; rules execute within a given object context; rules may consist of statements for producing side effects (e.g., outputting to a file), navigating in the repository and calling other rules; navigating in the repository can be done explicitly through the use of a context switching operator or implicitly by iterating through collections of objects; when a rule calls another, the calling rule implicitly passes its context to the called one.

## 4 Verification

Our ultimate goal is to demonstrate that the Controller operates correctly, i.e., that all the system requirements are guaranteed. A formal representation for each of the requirements must be produced and verified.

### 4.1 Requirements formalization

A natural way of expressing properties about object-oriented systems is using a logic that allows one to express properties about states and actions, e.g., *when the Controller is in* ***stopped mode****, the valve will never* ***open***. In our setting, states are caracterised by predicates like `Controller.mode` $= Stopped$ and actions can be signal receptions like `^StartButtonPressed` or calls to object operations like `Valve.close()`. The ACTL (Action CTL) temporal logic [NV90] is appropriate for formalizing the Steam-Boiler requirements being expressive enough for writing properties about states and actions. We selected a fragment of ACTL containing the following operators (besides usual logic connectors). Let $p$ be a predicate, $\alpha$ a set of action labels and $\Phi$ an ACTL formula:

$$\Phi ::= p \mid \langle\alpha\rangle\Phi \mid [\alpha]\Phi \mid \mathbf{A}[\Phi_\alpha \mathbf{U} \Phi'] \mid \mathbf{A}[\Phi_\alpha \mathbf{U}_{\alpha'} \Phi']$$

Informally, the semantics of $\langle\alpha\rangle\Phi$ and $[\alpha]\Phi$ is that "eventually" (respectively "always") we reach states satisfying $\Phi$ performing "one" (respectively "all") actions denoted by $\alpha$. The operator $\mathbf{A}[\Phi_\alpha \mathbf{U} \Phi']$ means that in all paths, $\Phi$ holds through $\alpha$ steps until it reaches $\Phi'$. The operator $\mathbf{A}[\Phi_\alpha \mathbf{U}_{\alpha'} \Phi']$ means that in all paths, $\Phi$ holds through $\alpha$ steps until it reaches $\Phi'$ through an $\alpha'$ step. We write $\mathbf{AG}(\Phi)$ as a shorthand for $\mathbf{A}[\Phi_{true} \mathbf{U} false]$, meaning that all paths consist of states satisfying $\Phi$.

The system requirements can thus be formalized as:

1. $\mathbf{AG}(\texttt{Controller.mode} = Stopped \Rightarrow$
   $[\texttt{\^{}StartButtonPressed}]\mathbf{A}[true_{true} \mathbf{U} \texttt{Controller.mode} = Initialization])$
2. $\mathbf{AG}(\texttt{Controller.mode} = Initialization \wedge \texttt{Boiler.waterLevel} < N1 \Rightarrow$
   $\mathbf{A}[true_{true} \mathbf{U}_{\texttt{Pump.start()} \vee \texttt{\^{}StopButtonPressed}} true])$
3. $\mathbf{AG}(\texttt{Controller.mode} = Initialization \wedge \texttt{Boiler.waterLevel} > N2 \Rightarrow$
   $\mathbf{A}[true_{true} \mathbf{U}_{\texttt{Valve.open()} \vee \texttt{\^{}StopButtonPressed}} true])$
4. $\mathbf{AG}(\texttt{Controller.mode} = Initialization \wedge N1 \leqslant \texttt{Boiler.waterLevel} \leqslant N2 \Rightarrow$
   $\mathbf{A}[true_{true} \mathbf{U} (\texttt{Controller.mode} = Normal \vee \langle \texttt{\^{}StopButtonPressed} \rangle true)])$

5. $\mathbf{AG}(\texttt{Controller.mode} = Initialization \wedge$
   $N1 \leqslant \texttt{Boiler.waterLevel} \leqslant N2 \wedge \texttt{Valve.state} = ValveOpened \Rightarrow$
   $\mathbf{A}[true_{true}\mathbf{U}_{\texttt{Valve.close()}}true])$

6. $\mathbf{AG}(\texttt{Controller.mode} = Normal \wedge \texttt{Pump.state} = PumpStarted \wedge \texttt{Boiler.waterLevel} > N2 \Rightarrow$
   $\mathbf{A}[true_{true}\mathbf{U}(\texttt{Pump.state} = PumpClosed \vee \texttt{Controller.mode} = Emergency)])$

7. $\mathbf{AG}(\texttt{Controller.mode} = Normal \wedge \texttt{Pump.state} = PumpStopped \wedge \texttt{Boiler.waterLevel} < N1 \Rightarrow$
   $\mathbf{A}[true_{true}\mathbf{U}(\texttt{Pump.state} = PumpStarted \vee \texttt{Controller.mode} = Emergency)])$

8. $\mathbf{AG}(\texttt{Controller.mode} = Initialization \vee \texttt{Controller.mode} = Normal \Rightarrow$
   $[\hat{}\texttt{StopButtonPressed}]\mathbf{A}[true_{true}\mathbf{U}\texttt{Controller.mode} = Stopped])$

9. $\mathbf{AG}(\texttt{Controller.mode} \neq Stopped \wedge \texttt{Boiler.waterLevel} > N2 \Rightarrow$
   $\mathbf{A}[true_{true}\mathbf{U}(\texttt{Boiler.waterLevel} \leqslant N2 \vee \texttt{Controller.mode} = Emergency)])$

10. $\mathbf{AG}(\texttt{Controller.mode} \neq Stopped \wedge \texttt{Boiler.waterLevel} < N1 \Rightarrow$
    $\mathbf{A}[true_{true}\mathbf{U}(\texttt{Boiler.waterLevel} \geqslant N1 \vee \texttt{Controller.mode} = Emergency)])$

11. $\mathbf{AG}(\neg\texttt{Pump.state} = PumpStarted \wedge \texttt{Boiler.waterLevel} > M2)$

12. $\mathbf{AG}(\neg\texttt{Boiler.state} = BoilerStarted \wedge \texttt{Boiler.waterLevel} < M1)$

13. $\mathbf{AG}(\neg\texttt{Controller.mode} \neq Initialization \wedge \texttt{Valve.state} = ValveOpened)$

### 4.2 Requirements verification

To verify the above properties, we used the EVALUATOR Model-Checker included in the CADP tool-box [FGK+96]. CADP is a set of integrated tools for producing and analysing Labelled Transition Systems. LTSs can be obtained from low level descriptions, networks of communicating automata and high-level LOTOS specifications. Analysis functionalities include interactive simulation and verification through comparison of LTSs according to different simulation relations and model-checking.

However, this Model-Checker does not allow the evaluation of predicates, and observations on the system state must be included as actions in the model. As was mentioned before, the generated LOTOS code can be augmented with gates that are signaled when a given condition $p$ becomes true. The subsequent LTSs will be likewise enriched with transitions, labelled $\alpha_p$, that are taken when that predicate is verified. In view of this, we can reformulate the properties, to a form allowed by the Model-Checker, as follows:

1. $\mathbf{AG}([\alpha_{cond1A}]\mathbf{A}[true_{true}\mathbf{U}[\hat{}\texttt{StartButtonPressed}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond1B})}true]])$

2. $\mathbf{AG}([\alpha_{cond2}]\mathbf{A}[true_{true}\mathbf{U}_{(\texttt{Pump.start()} \vee \hat{}\texttt{StopButtonPressed})}true])$

3. $\mathbf{AG}([\alpha_{cond3}]\mathbf{A}[true_{true}\mathbf{U}_{(\texttt{Valve.open()} \vee \hat{}\texttt{StopButtonPressed})}true])$

4. $\mathbf{AG}([\alpha_{cond4A}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond4B}) \vee (\hat{}\texttt{StopButtonPressed})}true])$

5. $\mathbf{AG}([\alpha_{cond5}]\mathbf{A}[true_{true}\mathbf{U}_{(\texttt{Valve.close()})}true])$

6. $\mathbf{AG}([\alpha_{cond6A}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond6B})}true])$

7. $\mathbf{AG}([\alpha_{cond7A}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond7B})}true])$

8. $\mathbf{AG}([\alpha_{cond8A}]\mathbf{A}[true_{true}\mathbf{U}[\hat{}\texttt{StopButtonPressed}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond8B})}true]])$

9. $\mathbf{AG}([\alpha_{cond9A}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond9B}\vee\alpha_{cond9C})}true])$
10. $\mathbf{AG}([\alpha_{cond10A}]\mathbf{A}[true_{true}\mathbf{U}_{(\alpha_{cond10B}\vee\alpha_{cond10C})}true])$
11. $\mathbf{AG}(\neg\langle\alpha_{cond11}\rangle true)$
12. $\mathbf{AG}(\neg\langle\alpha_{cond12}\rangle true)$
13. $\mathbf{AG}(\neg\langle\alpha_{cond13}\rangle true)$

where:

$$
\begin{aligned}
cond1A &\equiv (\texttt{Controller.mode} = Stopped)\\
cond1B &\equiv (\texttt{Controller.mode} = Initialization)\\
cond2 &\equiv (\texttt{Controller.mode} = Initialization \wedge \texttt{Boiler.waterLevel} < N1)\\
cond3 &\equiv (\texttt{Controller.mode} = Initialization \wedge \texttt{Boiler.waterLevel} > N2)\\
cond4A &\equiv (\texttt{Controller.mode} = Initialization \wedge N1 \leqslant \texttt{Boiler.waterLevel} \leqslant N2)\\
cond4B &\equiv (\texttt{Controller.mode} = Normal)\\
cond5 &\equiv (\texttt{Controller.mode} = Initialization \wedge\\
&\quad N1 \leqslant \texttt{Boiler.waterLevel} \leqslant N2 \wedge \texttt{Valve.state} = ValveOpened)\\
cond6A &\equiv (\texttt{Controller.mode} = Normal \wedge \texttt{Pump.state} = PumpStarted \wedge\\
&\quad \texttt{Boiler.waterLevel} > N2)\\
cond6B &\equiv (\texttt{Pump.state} = PumpStopped \vee \texttt{Controller.mode} = Emergency)\\
cond7A &\equiv (\texttt{Controller.mode} = Normal \wedge \texttt{Pump.state} = PumpStopped \wedge\\
&\quad \texttt{Boiler.waterLevel} < N1)\\
cond7B &\equiv (\texttt{Pump.state} = PumpStarted \vee \texttt{Controller.mode} = Emergency)\\
cond8A &\equiv (\texttt{Controller.mode} = Initialization \vee \texttt{Controller.mode} = Normal)\\
cond8B &\equiv cond9C \equiv cond10C \equiv (\texttt{Controller.mode} = Emergency)\\
cond9A &\equiv (\texttt{Controller.mode} \neq Stopped \wedge \texttt{Boiler.waterLevel} > N2)\\
cond9B &\equiv (\texttt{Boiler.waterLevel} \leqslant N2)\\
cond10A &\equiv (\texttt{Controller.mode} \neq Stopped \wedge \texttt{Boiler.waterLevel} < N1)\\
cond10B &\equiv (\texttt{Boiler.waterLevel} \geqslant N1)\\
cond11 &\equiv (\texttt{Pump.state} = PumpStarted \wedge \texttt{Boiler.waterLevel} > M2)\\
cond12 &\equiv (\texttt{Boiler.state} = BoilerStarted \wedge \texttt{Boiler.waterLevel} < M3)\\
cond13 &\equiv (\texttt{Controller.mode} \neq Initialization \wedge \texttt{Valve.state} = ValveOpened)
\end{aligned}
$$

The verification yielded the following results, using a CADP installation on a 500MHz Intel machine with 128Mb of RAM running the Linux operating system:

| *Requirement number* | *Lines of* LOTOS *code* | LOTOS *compilation timings* | *Number of states* | *Number of transitions* | *Verification time* |
|---|---|---|---|---|---|
| 1 | 1762 | 00'53.27" | 215191 | 221763 | 00'22.22" |
| 2 | 1786 | 00'49.68" | 159815 | 164725 | 00'16.32" |
| 3 | 1786 | 00'50.02" | 159765 | 164675 | 00'16.26" |
| 4 | 1806 | 00'56.00" | 251418 | 259112 | 00'28.39" |
| 5 | 1808 | 00'51.71" | 160687 | 165597 | 00'16.08" |
| 6 | 1838 | 00'55.69" | 252625 | 260346 | 00'28.28" |
| 7 | 1838 | 00'57.15" | 253569 | 261263 | 00'28.43" |
| 8 | 1762 | 00'51.77" | 216362 | 222904 | 00'22.42" |
| 9 | 1832 | 00'54.56" | 252911 | 260605 | 00'46.43" |
| 10 | 1804 | 00'57.16" | 252896 | 260590 | 00'41.33" |
| 11 | 1774 | 00'52.67" | 159029 | 163939 | 00'16.35" |
| 12 | 1774 | 00'49.10" | 159526 | 164436 | 00'15.80" |
| 13 | 1763 | 00'47.35" | 140117 | 144421 | 00'13.73" |

Each requirement corresponded to the generation of a single LOTOS specification from an OBLOG source file with 548 lines of code. All specifications were compiled and verified with a restriction on the integer domain to a range between 0 and 50.

## 5 Conclusions

Writing specifications using a high-level object-oriented language can be highly desirable. Typically, in many problem domains, using them for writing specifications is much easier. This promotes their use by domain experts wanting to skip the mathematical background needed by traditional specification languages.

We have seen how to verify properties of a subset of object-oriented specifications in a completely automated way. Our approach is based on a translation to LOTOS, which allowed us to establish a verification framework for the OBLOG language taking advantage of existing verification tools.

In the formalization of the system requirements, expressing apparently simple properties resulted initially in complex specification patterns. This seems to confirm [DAC98] that formalization in temporal logic can be quite error prone, although this effort increased our understanding of the problem through the analysis of the counter-examples provided by the Model-Checker. Indeed, some errors in our model were found and corrected.

Concerning the overhead of using an intermediate language, it can be claimed that a direct translation from OBLOG to LTSs could avoid many undesired transitions resulting from the LOTOS compilation. This direct translation can be enhanced through connecting to the API provided with the Open/Cæsar environment for generation and on-the-fly exploration of LTSs. However, by analyzing the obtained LOTOS specifications as high level representations of LTSs, we were able to isolate sources of non-determinism and devise strategies to optimize our initial translation.

This work is a contribution to a broader project that aims to the verification of OBLOG specifications. For the moment we are leaving out features like *dynamic creation of objects*, *dynamic references* and *exception handling* which can result in infinite state-spaces. To cope with this, we are planning to incorporate techniques based on *abstraction* [CGL94], in particular we are looking at recent developments in the combined use of abstraction and program analysis techniques [DHZ99,SS98].

A formal semantics document for OBLOG is currently being organized. It will allow us to extend the supported subset of specifications and verify the correctness of this translation framework.

## References

[ABL96] J. Abrial, E. Böger, and H. Langmaack, editors. *Formal Methods for Industrial Applications – Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[AS96] Luis F. A. Andrade and Amilcar Sernadas. Banking and Management Information System Automation. In *Proceedings of the 13th world congress of the International Federation of Automatic Control (San Francisco, USA)*, volume L, pages 113–136. Elsevier-Science, 1996.

[BJR97] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1997.

[Car99] Paulo J. F. Carreira. Automatic Verification of OBLOG Specifications. Master's thesis (to be published), Faculdade de Ciências da Universidade de Lisboa, Departamento de Informática, 1700 Campo Grande - Lisboa, 1999.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications. volume 8(2) of *ACM Transactions on Programming Languages and systems*, pages 244–263. 1986.

[CGL94] E.M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. volume 16(5) of *ACM Transactions on Programming Languages and Systems*, pages 834–871. 1994.

[CW96] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. volume 28(4es) of *ACM Computing Surveys*. December 1996.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-state Verification. In *Proceedings of the 2nd Workshop on Formal Methods in Software Practice*, Clearwater Beach, Florida, USA, March 1998.

[DC96] Gregory Duval and Thierry Cattel. Specifying and Verifying the Steam Boiler Problem with SPIN. In Jean-Raymond Abrial, Egon Böger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications – Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 203–217. Springer-Verlag, 1996.

[DHZ99] Matthew B. Dwyer, John Hatcliff, and Hongjun Zheng. Slicing Software for Model Construction. In *ACM SIGPLAN Partial Evaluation and Program Manipulation*. January 1999.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*, volume I. Springer-Verlag, 1985.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *LNCS*, pages 437–440. Springer-Verlag, August 1996.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[HLN+90] David Harel, H. Lachover, A. Naamad, Amir Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. volume 4 of *IEEE Transactions on Software Engineering*, pages 403–414. 1990.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[ISO88] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[JMMS98] Wil Janssen, Radu Mateescu, Sjouke Mauw, and Jan Springintveld. Verifying Business Processes using SPIN. In *Proceedings of the 4th International SPIN Workshop (Paris, France)*, 1998.

[Kur90] Robert P. Kurshan. Analysis of Discrete Event Coordination. In W. P. de Rover J. W. de Bakker and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 414–453, Berlin, 1990. Springer-Verlag.

[Mil89] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[NV90] R. De Nicola and F. W. Vaandrager. Action versus State Based Logics for Transition Systems. volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag, 1990.

[OBL99] OBLOG. The OBLOG Technical Information. Technical report, OBLOG Software S.A., www.oblog.com/tech, 1999.

[QS82] J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CAESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, Berlin, 1982. Springer-Verlag.

[SS98] D. A. Schmidt and B. Steffen. Data-Flow Analysis as Model-Checking of Abstract Interpretations. In G. Levi, editor, *Proceedings of the 5th Static Analysis Symposium*, volume 1165 of *Lecture Notes in Computer Science*, Pisa, Italy, September 1998. Springer-Verlag.

[VW86] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Logic in Computer Science. IEEE Computer Society Press*, page 332. 1986.

[WS96] Andreas Willig and Ina Schieferdecker. Specifying and Verifying the Steam Boiler Control System with Time Extended LOTOS. In Jean-Raymond Abrial, Egon Böger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications – Specifying and Programming the Stem Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 473–492. Springer-Verlag, 1996.

# Cooperative Information Systems Modelling and Validation Using the Co-nets Approach: The Chessmen Making Shop Case Study

Nasreddine Aoumeur* Gunter Saake
ITI, FIN, Otto-von-Guericke-Universität Magdeburg
Postfach 4120, D–39016 Magdeburg, Germany
E-mail: {aoumeur|saake}@iti.cs.uni-magdeburg.de

**Abstract**

The Co-nets approach that we are developing is an object oriented specification model based on a sound and complete integration of object oriented (OO) concepts and constructions into a variant of algebraic Petri nets. The Co-nets approach is interpreted in rewriting logic and is particularly suited for specifying and validating advanced information systems as autonomous, distributed, but yet cooperative components. This suitability is especially reflected by the following main Co-nets features. First, the approach is based on a perception of classes rather as modules with hidden (structural and behavioural) features enhancing autonomy and external features which may be exchanged with the environment and other classes. Second, the approach allows an incremental constructions of complex components, as a hierarchy of modules, through different forms of inheritance, object composition and aggregations; where each component behaves with respect to an appropriate intra-component evolution pattern supporting intra- as well as inter-object concurrency. Third, for interacting components, an appropriate inter-component interaction pattern is proposed promoting concurrency and keeping encapsulated all local features of communicating components. Fourth, due to their interpretation in rewriting logic, Co-nets modules can be rapid-prototyped using concurrent rewriting techniques.

Besides a didactic presentation of this approach, the objective of this paper is to enhance its practicability by handling a complex case study dealing with a Chessmen Making Shop(CMS) as a specific variant of a complex production system. In this system more than six autonomous components cooperate through explicit interfaces for achieving final (chessmen) products. Moreover, from a methodological point of view the paper shows how to specify and validate the corresponding Co-nets modules from their informal OO description, using well-known diagram notations, is a very straightforward way.

**Keywords:** Co-nets, production systems modelling, rewriting logic, components.

## 1 Introduction

Information systems, mostly characterized as reactive systems with large databases and application programs, have potentially benefited from the object-oriented (OO) structuring mechanisms. Indeed, thanks to this paradigm, all developing phases (i.e. analysis, specification / validation, design, and implementation) uniformly regard such systems as a community of interacting objects but with different levels of abstraction. Particularly, for the crucial specification / validation phase on which we are focusing, a very promising OO formalisms have been forwarded

*This work is partially supported by a DAAD scholarship and by the Deutsche Forschungsgemeinschaft (DFG) project "Integration of Software Specifications for Engineering Applications".

during the last decade; they are based on different frameworks going from the algebraic setting [EGS92, GD93] (just to cite some), temporal setting [DB95], to Petri nets [Lak95, SB94, BBG97].

Nevertheless, the ever-increasing in size and space of present-day organizations has shown some shortcomings of existing approaches particularly for coping with the intrinsic *distribution* of such systems and with their natural description as cooperating *components* and not as a whole community (of objects). This unsatisfactory state of affairs has recently yield an overwhelming need for investigating on more advanced conceptual models for specifying and validating such systems rather as fully distributed, autonomous but nevertheless cooperative components [PS98]. In such characterization, each component has to be (at least) regarded as a hierarchy of classes through different forms of inheritance, object composition and aggregation. On the other side, distribution in such components has to be reflected by a true intra- as well inter-object concurrency and by exhibition of different forms of communication including synchronous and asynchronous ones. Finally, autonomy that has to be coupled with close cooperation should be particularly reflected by encapsulation of proper features in each component and by the existence of explicit interfaces [AG97].

As a significant contribution towards the achievement of these challenges, we are developing a multi-paradigm advanced conceptual modeling approach that we argue fulfills as far as possible the mentioned requirements. The approach that we referred to as CO-NETS approach [AS99a, AS99b, AS00] is a formal and complete integration of OO concepts and constructions into an appropriate variant of algebraic Petri nets named ECATNETS [BM92]. The key features of CO-NETS approach in modeling cooperative systems are the following:

- A clean separation between *data* —specified algebraically[1] and used for describing object attributes (values and identifiers) and messages— and *objects* as indivisible units of structure and behaviour. Moreover, we regard a class rather as a *module* —in the spirit of [EM90] (at the syntactical level)—with a hidden part including structure as well as behaviour, and an observed part as interface for interacting with the environment and other modules.

- An incremental construction of components, as a hierarchy of modules[2], through simple and multiple inheritance (with redefinition, associated polymorphism and dynamic binding), object composition and aggregation. Such components behave with respect to an appropriate *intra-component* evolution pattern that naturally supports intra- as well as inter-object concurrency (i.e. without suffering from the inheritance anomaly problem).

- For interacting different components and thereby constructing more complex systems as cooperative components, an adequate *inter-component* interaction pattern is proposed; it enhances concurrency and preserves encapsulated local (i.e. hidden) features of each component.

- By interpreting the CO-Nets behaviour in rewriting logic, rapid-prototypes may be generated using rewrite techniques and current implementation of the MAUDE language [Mes93] particularly.

Besides a simplified presentation of this framework, the objective of the present paper is to assess the model against a real-life case study dealing with chessmen-making as a particular variant of production systems. In this production system more than six components behave autonomously and concurrently nevertheless cooperate for producing the desired chessman. Moreover, from a methodological point view, our framework allow a straightforward specification (and then validation) from a given informal OO description using well known OO diagrams.

[1] Using Ordered Sorted Algebra(OSA) [GD94].

[2] Throughout the rest of the paper, we use indifferently module or class.

The rest of this paper is organized as follows. In the second section, first, we informally introduce the Chessmen Making Shop case study. Then, using some appealing OO graphical notations, we describe in more detail each component of this application. In the main section, different features of the Co-nets approach are introduced and applied to a significant part (i.e. three components) of the CMS problem, while for sake of readability the other component specifications are given in the appendix. The last section presents some concluding remarks and future work.

# 2 The Chessmen Making Shop

In this section, first, we informally describe the CMS problem that we borrow from [DB95]. Then, we propose an object oriented description of different components of this case study.

## 2.1 CMS : Informal Description

The *Chessmen Making Shop* is a small manufacturing unit in charge of producing *chessmen* from wooden *cylinders*. It is composed of production *components* (automated units), and is supervised by a *manager* (as a human actor). The shop manager gives a *production order* to a component when a chessman has to be machined. The order specifies the type of chessman to be produced (king, queen, castle, bishop, knight or pawn) and a coded reference. The system is composed of the following components: *machine tools*, a *stock*, a *robot*, a *clamping system*, an *auto-guided vehicle* (AGV), and a *controller* (i.e. computer).

The cylinders and end-products are stored in the stock. Before they can be machined, cylinders have to be clamped on a pallet. This work is done by the clamping system. The robot is used to withdraw parts from the stock and furnish them for the clamping operation. Once clamped, parts are transported to one machine by the AGV. If additional machining is required, the AGV picks the partly-machined part and transports it to another machine (one or several times) until the desired chessman is obtained. The chessman is then transported to the clamping system to be unclamped and stored by the robot in the stock (at the location where the raw cylinder was withdrawn). All these activities are coordinated by a controller.

The stock is composed of a number of locations identified by addresses. Each location may contain at most one item (a cylinder or a chessman). Items can be entered in or withdrawn from the stock either manually by the shop manager or automatically by the robot. The manager is able to see if the stock is empty, and an alarm is sent to him by the stock each time it is full.

The robot is in charge of loading and unloading the clamping system. The grip of the robot is equipped with a sensor, so it can detect which item is stored at a given location of the stock when its arm reaches that location. The same holds for the contents of the clamping system. The robot receives commands from the controller to load the clamping system with an item and commands to unload it. The commands specify the item to be transported and the address of the stock concerned by the operation.

The clamping system role is to clamp and unclamp the items it receives. It is an automatic device: it automatically clamps any item put on it by the robot and automatically unclamps any item brought by the AGV. The clamping system may handle one item at a time and must be unloaded between two operations.

The AGV has two places where items can be put. But those places can never be occupied at the same time during transportation; they are only used to perform some exchange between an item already in the AGV and an item located on a device i.e. machine tool or clamping system. The AGV receives commands from the controller asking it to carry one item from one location to another. In reaction to these commands, it has to move to the origin (if it was not there yet),

pick up the item, move to the destination and deliver the item. The AGV notifies the devices when it picks up (resp. delivers) an item from (resp. to) it.

Machine-tools are used to apply basic transformations to raw materials or partly-machined items. Each machine-tool has some capabilities, i.e. the set of programs it is able to run. A machine-tool receives commands from the controller. A command, addressed to a given machine tool, specifies the program to be run. Machine-tools notify the controller when the program they had to run has been completed.

The controller is in charge of achieving orders it receives from the manager by issuing commands to a corresponding component. The software running in the controller contains a 'recipe book' which associate a recipe to each chessman type. A recipe consists in a sequence of production steps. Each production step is characterized by a machine and a program to be run on it. The controller keeps traces of the status of each order under processing and the status of each component.

## 2.2 CMS : Corresponding Object Oriented Description

Following OO methodologies [RBP+91] and the modeling of classes rather as modules, the different components of this case study are depicted in figure 1. In this (semi-) graphical description we have used intuitive notations borrowed from [Weg90] that we have enriched with explicit distinction between local attributes (included in each ellipse) and observed ones (outside the ellipse). On the other hand, we have also made a distinction between local messages (included in the internal box) and external ones (included in the external box) with a precise indication, using arrows, to which component(s) such (external) messages are sent (i.e. exported). More precisely, each component may be described as follows:

**The Stock component:** It is characterized by the following state components (i.e. attributes): *Contents* as a set of locations, each one is uniquely referenced by its address and it may contain either a cylinder or a chessman; this attribute is observed (by the robot class); *capacity* as a natural number fixing the number of locations; and as derived (boolean-valued) attributes we have *empty* and *full*. As messages there is: *Alarm-full* sent to other components (i.e. Manager component) when all locations are full. As imported messages we have, from the manager, *Remove* and *Store* of an item from/at a given address, and equivalent ones from the Robot component i.e. *Put* and *Get* respectively.

**The Robot component:** Internal attributes are: *Grip* that takes as value an item (i.e. cylinder or chessman) and *busy* as boolean attribute indicating if the grip contains an item or not. Observed is the *location* attribute that represents current position (i.e. address) of the robot. Local messages are : *load* (resp. *unload*) reflecting loading (resp. unloading) of an item from (resp. at) a given address; *Goto* to a location and *Move* from a location to another. Messages exported to other components are *Put* and *Get*. Finally, the robot receives orders from the controller component— as imported messages— corresponding to *load-clamp* and *unload-clamp* parameterized by a location and an item.

**The Clamping System component:** as observed attribute is *Contents* containing the item to be clamped, while as internal is a corresponding boolean attribute *Clamped*. Local messages are : *Clamp* and *Unclamp*. Exported messages (to the controller) are: *Clamping-done* and *Umclamping-done* signaling respectively the end of clamping and unclamping operations. Imported messages from the AGV component are: *Deliver* an item from one of the two places of the AGV and *Pick-up* an item from a given place. Imported from the Robot component are *Put* and *Get* messages.

**The Auto-guided Vehicle component:** Attributes of this class are both places *place1* and *place2* containing (alternatively) an item, the *Location* of the AGV, and a boolean attribute *Busy* evaluated to true when one of the place is occupied. Local messages are *Goto* to a given location, *Transport* an item from a given location to another, and *Move* from a location to another. Also, we have *load* an item in a given place (from a given location) as local message. Exported to other components are the messages: *Pick-up* an item in a given place and *Deliver* an item from a given place to the clamping system or robot component. Imported message (from the controller) is *Carry*.

**The Machine Tool component:** Attributes of this component are: *Contents* containing the item as a cylinder or an intermediate product to be machined or an end-product; this attribute has to be observed by the AGV component. *Busy* is an internal attribute. Also as attributes we have an abstract attribute *Capabilities* storing the names of programs that offer the concerned machine. Local message is *work*. Exported message to the Controller component is *work-done* parameterized by programs. Imported messages are: *Deliver* and *Pick-up* from the AGV and *Run* a given program on a given machine from the controller.

**The Controller component:** its attributes are: *Orders* as a list of orders indexed by a reference denoting the type of currently under processing chessmen, *Recipes* as a list indexed by different types of chessmen corresponds to the recipe that have to be followed to produce each type. *Proc* is a list of references denoting the orders currently handled by the machine tools, and *Status* is a list of status keeping current status (i.e. working or down) of each machine. *Agv-ord* is a set of references denoting the orders currently handled by the AGV. *Free-cyl* is set of stock addresses denoting the set of stock cells containing not booked cylinders. Local messages are: *Machining* and *Machining-step* parameterized by the type and the reference. Exported messages are: *Unload-clamp* and *Load-clamp* parameterized by the reference of the orders and a stock or a clamping address; both messages are sent to the Robot component; *Carry* parameterized by a reference of an order, an item and the initial and the final location to be carried to; *Run* parameterized by an order reference, a machine tool and a corresponding program; and *Alarm* message. Imported messages are: *Produce*, *Remove* and *Store* from the manager; *Clamping-done* and *Unclamping-done* from the Clamping-System; *Put* from the Robot; *Deliver* from the AGV; and *Work-done* from the Machine-tool.

**The Manager component:** This component has no attributes. Messages that are exported from this component are: *Produce* parameterized by a type and an associated order reference and is sent to the Controller. *Remove* and *Store* parameterized by an item and a corresponding address are to be sent to the Stock and the Controller components. Imported messages are: *Alarm* from the Controller and *Alarm-full* from the Stock.

## 3 The CO-NETS Approach : An Overview

The purpose of this section is to present this approach in an incremental, didactic way with the help of a formal specification, dealing with three components, of the CMS case study; but we note that the complete specification is given in the appendix. More precisely, first, we describe how the CO-NETS approach structurally and semantically captures notions of template as a description of kind of objects with explicit interfaces and a component (or a class) as a time varying collection of object-state (and message) instances. Second, we present how more abstraction mechanisms, particularly inheritance and interaction are specified using this approach.

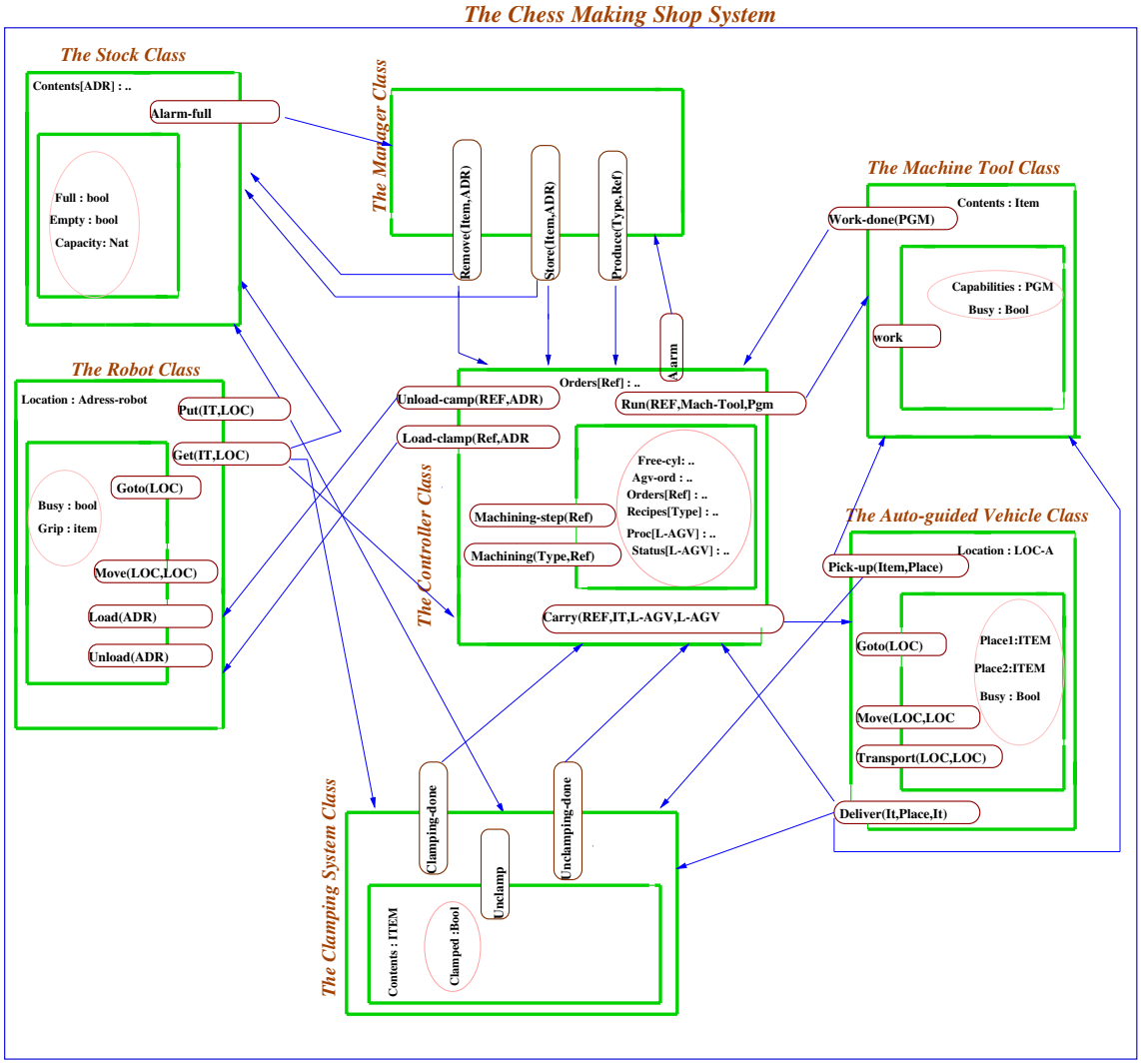



Figure 1: The Graphical illustration of different components and their Interaction

## 3.1 CO-NETS : Template and Class Specification

This section deals with the modelling of the basic concepts of the object oriented paradigm, namely objects, templates and classes. We first present the structure or what is commonly called the 'object' signature templates [EGS92], then we describe how 'specification' templates and classes are specified.

### 3.1.1 Template Signature Specification

A template signature defines the structure of object states and the form of operations that have to be accepted by such states. Basically, in the CO-NETS approach, we follow the general object signature proposed for MAUDE [Mes93]. That is to say, object states are regarded as terms —precisely as a tuple— and messages as operations sent or received by objects. However, apart from these general conceptual similarities, and in order to be more close to the aforementioned information system requirements, the OO signature that we propose can be informally described as follows:

- The object states are terms of the form $\langle Id | atr_1 : val_1, ..., atr_k : val_k, at\_bs_1 : val'_1, ..., at\_bs_{k'} : val'_s \rangle$; where $Id$ is an observed object identity taking its value from an appropriate abstract

data type $OId$; $atr_1, .., atr_k$ are the local, hidden from the outside, attribute identifiers having as current values respectively $val_1, .., val_k$. The observed part of an object state is identified by $at\_bs_1, ..., at\_bs_s$ and their associated current values are $val'_1, .., val'_s$. Also, we assume that all attribute identifiers (local or observed) range their values over a suitable sort denoted $AId$, and their associated values are ranged over the sort $Value$ with $OId < Value$ (i.e. $OId$ as subsort of $Value$) in order to allow object valued attributes.

- In contrast to the indivisible object state proposed in MAUDE that avoids any form of intra-object concurrency, we introduce a powerful axiom, called 'splitting / recombination' axiom allowing to split (resp. recombine) the object state out of necessity. As described in more detail later, this axiom can be described as follows: $\langle Id|attrs_1, attrs_2\rangle = \langle Id|attrs_1\rangle \oplus \langle Id|attrs_2\rangle$[3].

- In addition of conceiving messages as terms —that consist of a message name, identifiers of objects the message is addressed to, and, possibly, parameters—, we make a clear distinction between internal, local messages and the external as imported or exported messages. Local messages allow for evolving the object states of a given component, while external ones allow for interacting different components by exclusively using their observed attributes and external messages.

Following these informal description and some ideas from [Mes93], the formal description of object states as well as template structures, using an OBJ [GWM+92] notation, is depicted in what follows.

```
obj Object-State is
  sort AId .
  subsort OId < Value .
  subsort Attribute < Attributes .
  subsort Id-Attributes < Object .
  subsort Local-attributes External-attributes < Id-Attributes .
  protecting Value OId AId .
  op _:_ :  AId Value → Attribute .
  op _,_ :  Attribute Attributes → Attributes [associ. commu. Id:nil] .
  op ⟨_|_⟩ :  OId Attributes → Id-Attributes .
  op _⊕_ :  Id-Attributes Id-Attributes → Id-Attributes [associ. commu. Id:nil] .
  vars Attr:  Attribute ; Attrs1, Attrs2:  Attributes ; I:OId .
  eq1 ⟨I|attrs1⟩ ⊕ ⟨I|attrs2⟩ = ⟨I|attrs1, attrs2⟩ .
  eq2 ⟨I|nil⟩ = I
endo.

obj Class-Structure is
  protecting Object-state, s-atr1,...,s-atrn , s-arg11,1,.., s-argl1,l1,
             ...,s-argi1,1,...,s-argi1,i1 ...
  subsort Id.obj < OId .
  subsort Mesl1, Mesl2,...,Mesll < Local_Messages .
  subsort Mese1, Mese2,...,Meseе < Exported_Messages .
  subsort Mesi1, Mesi2,...,Mesii < Imported_Messages .
  sort Id.obj, Mesl1, .  .  .  ,Mesip
  (* local attributes *)
    op ⟨_|atr1 : _, ..., atrk : _⟩ :  Id.obj s-atr1 ... s-atrk
       → Local-Attributes.
  (* observed attributes *)
    op ⟨_|atrbs1 :, ..., atrbsk' : _⟩ :  Id.obj s-atbs1 ... s-atbsk'
       → External-Attributes.
  (* local messages *)
    op msl1 :  s-argl1,1 ... s-argl1,l1 → Mesl1 . ...
```

[3] $attr_i$ stands for a simplified form of $atr_{i1} : val_{i1}, ..., atr_{ik} : val_{ir}$.

```
  (* export messages *)
    op ms_e1 :  s-arg_e1,1 ... s-arg_e1,e1 → Mes_e1 . ...
  (* import messages *)
    op ms_i1 :  s-arg_i1,1 ... s-arg_i1,i1 → Mes_ip . ...
endo.
```

**Remark 3.1** Local messages in a given component $Cp$ have to include at least two usual messages: the message for creating a new object state and the message for the deletion of an existing object. We denote them respectively by $Ad_{Cp}$ and $Dl_{Cp}$. It is also worth mentioning that imported messages being part of a given component have to be declared as exported ones in (exactly one) another components. ♢

**Example 3.1** *Following this* Co-nets *general form of template signature, we present in what follows the description of the Stock, Robot, and Manager components.*

**Robot template signature:** *First we have to specify the data types that are used in this template signature for specifying attribute values and/or event parameters. For the attributes we have: Boolean for Busy , ITEM (i.e. constants composed of [cyl, King, Queen, ..]) for grip and a set of constants reflecting either stock (shortly, ST-ADR) or clamping (shortly, CL-ADR) addresses on which the Robot (i.e. its location attribute) may be in. We denote this (Robot location) by R-ADR. For events parameters the (same) sorts are used. All these data types will be referred to as R-DATA and specified using an OBJ notation as follows:*

```
obj R-DATA is
  protecting Bool .
  sort ITEM .
  subsort ST-ADR  CL-ADR < R-ADR .
  op Cyl, King, Queen, Castle, Bishop, Knight, Pawn :  → ITEM .
endo.
```

*Using these data types and respecting the informal description given in the previous section, the corresponding Robot template signature can be defined as follows:*

```
obj Robot-signature is
  protecting Object-state R-ADR .
  subsort Id.Robot < OId .
  subsort Local_Robot  External_Robot < Robot .
  subsort GOTO  MOVE  LOAD  UNLOAD < Local_Robot_Mes .
  subsort GET  PUT < Exported_Robot_Mes .
  (* local attributes *)
   op ⟨_|Busy : _, Grip : _⟩ :  Id.Robot  Bool  ITEM → Local_Robot.
  (* observed attributes *)
   op ⟨_|Location : _⟩ :  Id.Robot  R-ADR → External_Robot .
  (* local messages *)
   op Goto :  Id.Robot  R-ADR → Local_Robot_Mes .
   op Move :  Id.Robot  R-ADR  R-ADR → Local_Robot_Mes .
  (* export messages *)
   op Put, Get :  Id.Robot  OId  ITEM  R-ADR → Exported_Robot_Mes .
  (* Imported messages *)
   op Load-cp, Unload-cp:OId  Id.Robot  REF  ST-ADR → Imported_Robot_Mes (* from the Controler *).
  vars R : Id.Robot .(* these variables are used in the corresponding net *).
  vars L  L1 :  R-ADR .
  vars it :  ITEM .
endo.
```

**Stock template signature:** *Data types associated with this template are ST-ADR specifying different addresses of the stock; CONTENT as a list of pairs of addresses and corresponding contents (i.e.* **nil** *for empty,* **cyl** *for cylinder or* **Chm** *for an end product (the specific type of a given chessman is irrelevant in this class); capacity as a natural constant reflecting the capacity. These data types will be referred to as S-DATA.*

```
obj S-DATA is
  protecting Bool .
  sort Bool ITEM-ST NAT.
  sort ST-ADR .
  subsort ELT < CONTENT .
  op nil, cyl, Chm :  → ITEM-ST .
  op capacity :  → NAT .
  op [_._] :  ST-ADR  ITEM-ST → ELT .
  op _._ :  ELT  CONTENT → CONTENT [assoc.  commu] .
endo.
```

```
obj Stock-signature is
  protecting Object-state S-ADR .
  subsort Id.Stock < OId .
  subsort Local_Stock  External_Stock < Stock .
  subsort ALARM < Exported_Stock_Mes .
  (* local attributes *)
  op ⟨_|Full : _, Empty : _, Capacity : _⟩ :  Id.Stock  Bool  Bool  NAT → Local_Robot.
  (* observed attributes *)
  op ⟨_|Contents : _⟩ :  Id.Robot  CONTENT → External_Stock .
  (* export messages *)
  op Alarm :  Id.Stock  OId → Exported_Stock_Mes .
  (* Imported messages *)
  op Remove, Store :  OId  Id.Manager  ITEM  ST-ADR
     → Imported_Stock_Mes (* from the Manager Component *).
  op Produce :  OId  Id.Manager  TYPE  REF → Imported_Stock_Mes (* from the Manager Component *).
  op Put, Get :  OId  Id.Robot  ITEM  R-ADR → Imported_Stock_Mes (* from the Robot Component *).
  vars S : Id.Stock .(* these variables are used in the corresponding net *).
  vars Adr :  S-ADR .
  vars C Ls :  CONTENT .
endo.
```

**Manager template signature:** *This class just allows for coordinating some tasks and therefore it contains no attributes. Its corresponding signature takes the form:*

```
obj Manager-signature is
  protecting Object-state S-ADR REF .
  subsort Id.Manager < OId .
  subsort Local_Stock  External_Stock < Stock .
  subsort REMOVE  STORE  PRODUCE < Exported_Manager_Mes .
  (* export messages *)
  op Remove, Store :  Id.Manager  OId  ITEM  ST-ADR → Exported_Manager_Mes .
  op Produce :  Id.Manager  OId  TYPE  REF → PRODUCE .
  (* Imported messages *)
  op Alarm :  Id.Stock  Id.Manager → Imported_Manager_Mes (* from the Stock Component *).
  op Alarm-full :  Id.Controller  Id.Manager → Imported_Manager_Mes (* from the Controller Component *).
  vars M : Id.Manager .
endo.
```

### 3.1.2 Template and component specification

On the basis of the template signature, we define the notion of template specification as a Co-net and the notion of a component (here just a class with interface) as a marked Co-net. Informally the associated Co-net structure, with a given template signature, can be described as follows:

- The places of Co-net are precisely defined by associating with each message generator one place that we called 'message' place. Henceforth, each message place has to contain

message instances, of a specific form, addressed to objects (and not yet performed). In addition to these message places, we associate with the object sort one 'object' place that has to contain the current object states of this component. Note also that places associated with 'external' messages will be drawn with bold circles.

- The CO-NET transitions reflect the effect of messages on the object states to which they are addressed. Also, we make distinction between local transitions that reflect object states evolution and external ones modeling the interaction between different components. The requirements to be fulfilled for each transition form are given in the subsection below. Both input and output arcs are defined as multisets of terms respecting the type of their input and/or output places—the associated union operation is denoted by $\oplus$.

- Conditions may be associated with transitions. They involve attribute and/or message parameters variables.

**Example 3.2** *Following the above (informal) component and template definitions, the three* CO-NETS *that have to be associated with the afore described template signatures are as follows:*

**The robot** CO-NET **component:** *This net as depicted in figure 2 is composed of an object place denoted by* `ROBOT` *and containing the current (object) state of different robots*[4]*; and six message places corresponding to different messages declared in this template in addition to the two imported messages. The effect of each message is captured by an appropriate transition. For instance, the effect of the message* `Goto(R,L1)` *is captured by the transition* `GT` *that takes as input the current location of the robot (i.e.* $\langle R \mid Location : L \rangle$[5]*) and as result its change to the new location (i.e.* $L1$*) provided that it is different from* $L$*. The effect associated with* `Move(R,L1,L2)` *can be easily understood similarly. A more complex effect is for instance the effect of the* `LOAD` *message. This effect modeled by the transition* `LD` *can be explained as follows: First this message is existing only after sending of a* `Load-clamp` *message from the controller; this fact is captured by the transition* `LDCp`*. Second, the effect of* `Load(R,Rf,Adr)`*, as a loading by the robot* `R` *of an item from the (stock-) address* `Adr` *(with respect to a given reference* `Rf` *of an order) and its deposit at the clamping system, is equivalent to the following output messages: going to this address (i.e.* `Goto(R,Adr)`*) followed by getting the corresponding item (i.e.* `Get(R,Adr,-)`*) followed by going to the clamping system (with this item) (i.e.* `Goto(R,Clamping)`*) and finally putting this item in the clamping system. The same reasoning may applied to the effect associated with the* `Unload` *message, but here after receiving an* `Unload-clamp` *from the controller through the transition* `ULDCp` *the robot has just to go to the (stock-) address* `Adr` *and then put griped (end-product) item. The partial effect*[6] *of a* `Put(R,-,-,L)`*, as modeled by transition* `PUT` *consists in deposing the griped item it in the (stock-) address corresponding to* `L` *(i.e. where the robot is located). This means that the robot state before have to be* $\langle R | grip : it, Busy : True, Location : L \rangle$*, while its state after change will be* $\langle R | grip : nil, Busy : False, Location : L \rangle$*. The same reasoning may be applied to the effect of a Get message.*

[4]Although in this application just one robot have to used, using our approach it is quite possible to have more than one robot. However, in this later case some constraints have to be added in order to avoid conflict between them.

[5]Remark that due to the splitting axiom only the location which is the just concerned attribute in this effect is selected.

[6]Because both Put and Get message have to interact with either the stock of the clamping system, in addition to their local effect an observed effect as interaction with each one of these classes have to be modeled (see the interaction subsection).

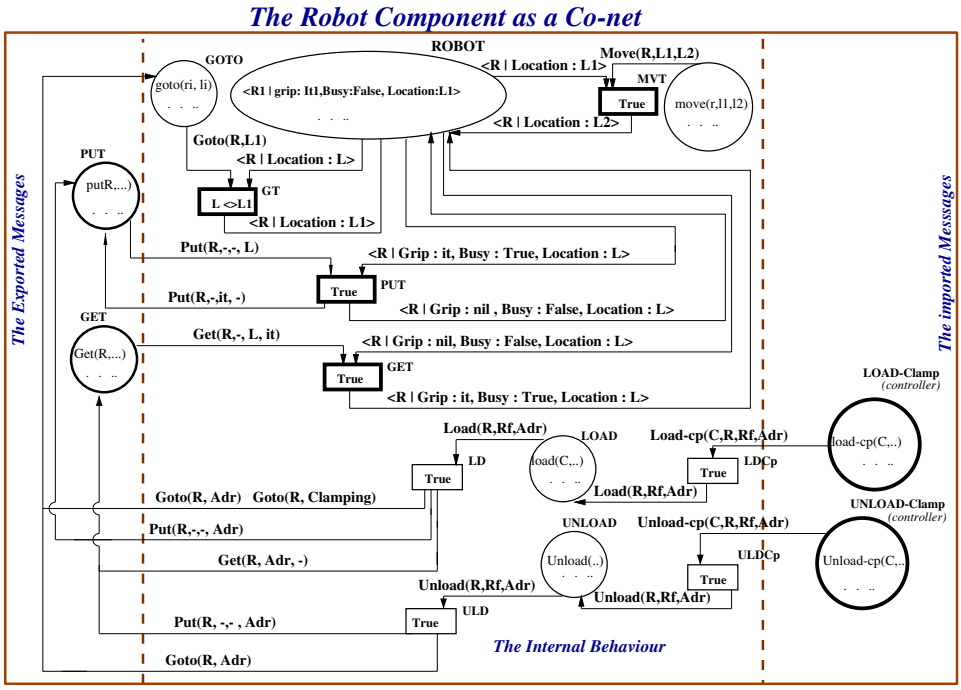



Figure 2: The Robot Component as a CO-NET.

**The stock component:** *as depicted in figure 3, in addition to the object place denoted by STOCK we have six places corresponding to different message generators. As local effect we have just the automatic sending of an alarm (to the Manager) if the capacity of the stock is reached.*

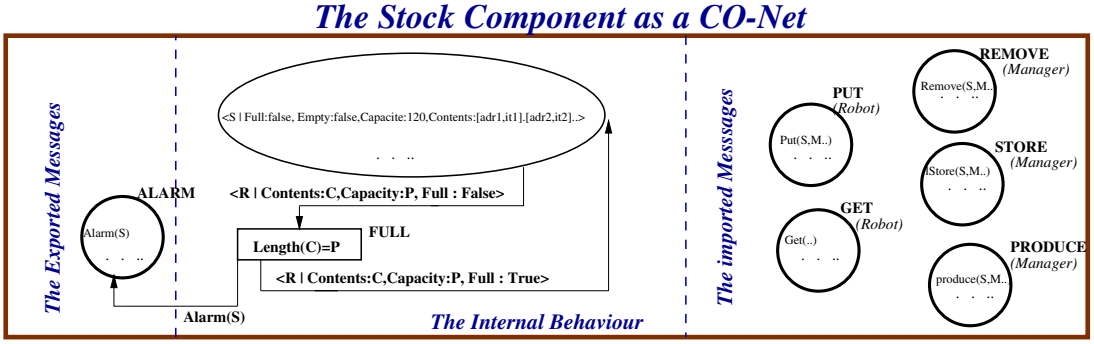



Figure 3: The Stock Component as a CO-NET .

**The manager class:** *In this component depicted in figure 4, there is no object place because this (functional) component has no attributes. So we have just places associated with the messages.*

## 3.2 CO-NETS : Semantical Aspects

After highlighting how CO-NETS templates and components are constructed, we focus herein on the behavioural semantics aspects of such components. That is, how to construct a *coherent* object society as a community of object states and message instances associated with each component, and how such a society evolves only into a *permissible* states. By coherence we

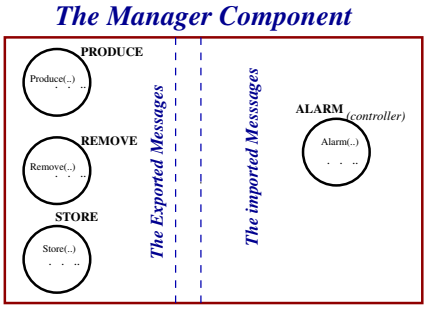



Figure 4: The Manager Component as a Co-net.

mainly mean the respect of the system structure, the uniqueness of object identities and the non violation of the encapsulation property.

### 3.2.1 Objects creation and deletion

In order to ensure the uniqueness of object identities, we propose the following conceptualization:

1. We associate with each marked Ob-net, modeling a component denoted $Cl$, a new place of sort $Id.obj$ ($< OId$) containing *actual object identifiers* of the objects in $Cp$.
2. For the creation of new objects, we introduce a new message sort denoted $Ad_{Cp}$ and an associated symbol operation (i.e. creation message) denoted $ad_{Cl}$ indexed by $Id.obj \times Ad_{Cp}$.
3. Each object state creation should be performed through the net depicted in the left hand side of figure 5. The intended semantics for the notation $\sim$ is that for firing the transition $NEW$ the identifier $Id$ should not already exist in the place $Id.obj$. After firing this transition, there is an addition of the new identifier $Id$ to the place $Id.obj$ and creation of a new object namely $\langle Id|atr_1 : in_1, ..., atr_k : in_k\rangle$, where $in_1, ..., in_k$ are optional initial attributes values. The deletion process given in the right hand of Figure 5 is conceived in the same way.

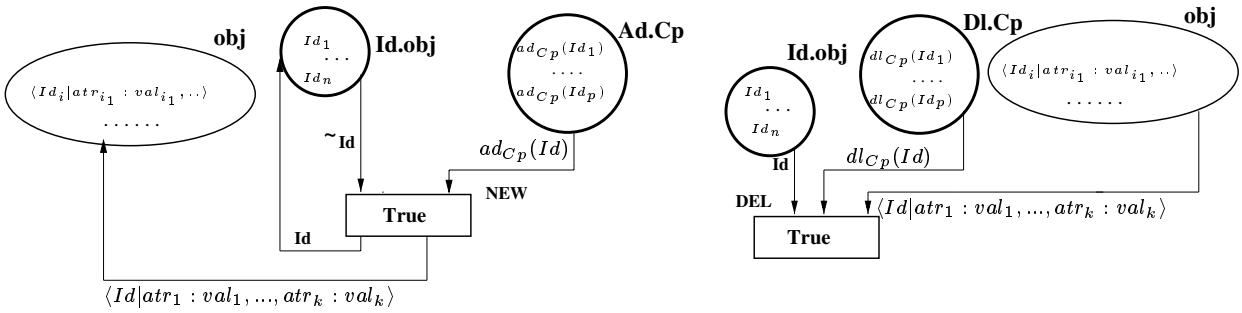



Figure 5: Objects Creation and Deletion Using Co-nets

### 3.2.2 Evolution of Object States in Classes

For evolving object states in a given class, we propose a general pattern that has to be respected in order to ensure the encapsulation property—in the sense that no object states or messages of other classes participate in this communication — as well as the preservation of the object identity uniqueness. Following such guidelines and in order to exhibit a maximal concurrency, this evolution schema is depicted in Figure 6, and it can be intuitively explained as follows: The contact of the just relevant parts of some object states of a given $Cl$—namely $\langle I_1|attrs_1\rangle$

;..; $\langle I_k|attrs_k\rangle$— with some messages $ms_{i1}, .., ms_{ip}$—declared as *local or imported* in this class— and under some conditions on the invoked attributes and message parameters results in the following effects. First, the messages $ms_{i1}, .., ms_{ip}$ vanish. Second, some (parts of ) object states participating in the communication, namely $I_{s1}, .., I_{st}$, may change (such change is symbolized by $attrs'_{s1}, .., attrs'_{st}$). The other (unchanged part of) object states are denoted by $attrs_{i_1}, .., attrs_{i_r}$ so that $\{i_1, ...i_r\} \cup \{s_1, .., s_t\} = \{1, .., k\}$[7]. Third, there may be a deletion of some objects using explicit sending of delete messages to such objects; and finally messages may be sent to objects of $Cl$, namely $ms_{h_1}, .., ms_{h_r}$.

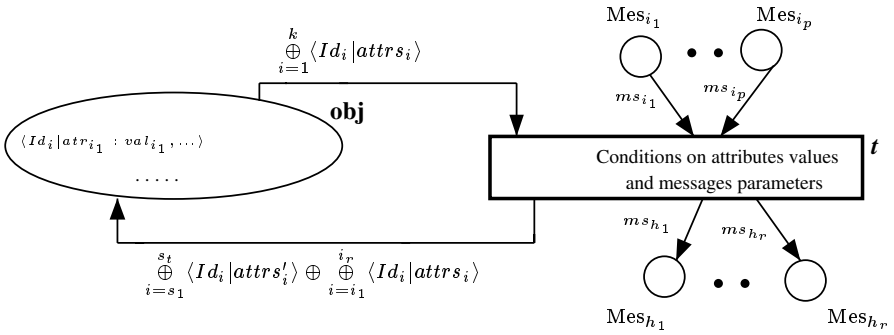



Figure 6: The intra-component evolution pattern in CO-NETS

### 3.2.3 Rewriting rules governing the CO-NETS behaviour

In the same spirit as in [BM92], each CO-NET transition is captured by an appropriate rewriting rule interpreted into rewrite logic. Following the intra-component evolution pattern in figure 6, the general form of rewrite rules that we associate with it takes the following form:

$$\textbf{t: } (Obj, \overset{k}{\underset{i=1}{\oplus}} \langle I_i|attrs_i\rangle) \otimes \overset{p}{\underset{k=1}{\otimes}} (Mes_{ik}, ms_{ik}) \Rightarrow (Obj, \overset{t}{\underset{k=1}{\oplus}} \langle I_{s_k}|attrs'_{s_k}\rangle \oplus \overset{r}{\underset{k=1}{\oplus}} \langle I_{i_k}|attrs_{i_k}\rangle) \otimes \overset{r}{\underset{k=1}{\otimes}} (Mes_{h_k}, ms_{h_k}) \textit{ if } Condition \wedge M(Ad.Cp) = \emptyset \wedge M(Dl.Cp) = \emptyset.$$

**Remark 3.2** The operator $\otimes$ is defined as a multiset union and allows for relating different places identifiers with their current marking. Moreover, we assume that $\otimes$ is distributive over $\oplus$ i.e. $(p, mt_1 \oplus mt_2) = (p, mt_1) \otimes (p, mt_2)$ with $mt_1$, $mt_2$ multiset of terms over $\oplus$ and $p$ a place identifier. The condition $M(Ad_{Cp}) = \emptyset$ and $M(Dl_{Cp}) = \emptyset$, in this rule, means that the creation and the deletion of objects have to be performed at first; In other words, before performing the above rewrite rule the markings in the $Ad_{Cp}$ as well of the $Dl_{Cp}$ places have to be empty. This allows particularly to avoid inconsistency like the manipulation of an object that is already logically deleted (i.e. a corresponding delete message was already sent) and not really or 'physically' deleted by firing its corresponding transition. Finally, please note that that the selection of just the *invoked parts* of object states, in this evolution pattern, is quite possible because of the splitting /recombination axiom—that has to be performed before and in accordance with each invoked state evolution. ♢

**Example 3.3** *By applying this general form of rule, it is not difficult to generate the rules governing the three component behaviour. These rules may be straightforwardly described as follows:*

**The robot component rewriting rules**

[7] In other words, there is no *implicit* creation or deletion of (part of) object states— that may lead to inconsistency w.r.t. the above described creation/deletion schema.

**GT**[8]**:** $(GOTO, Goto(R, L1)) \otimes (ROBOT, \langle R \mid Location : L\rangle))$
$\Rightarrow (ROBOT, \langle R \mid Location : L1\rangle)\ if\ (L \neq L1)$

**MVT:** $(MOVE, Move(R, L1, L2)) \otimes (ROBOT, \langle R \mid Location : L1\rangle)$
$\Rightarrow (ROBOT, \langle R \mid Location : L2\rangle)$

**PUT:** $(PUT, Put(R, -, -, L)) \otimes (ROBOT, \langle R \mid Grip : it, Busy : True, Location : L\rangle)$
$\Rightarrow (ROBOT, \langle R \mid Grip : nil, Busy : False, Location : L\rangle) Put(R, -, it, L)$

**GET:** $(GET, Get(R, -, it, L)) \otimes (ROBOT, \langle R \mid Grip : nil, Busy : False, Location : L\rangle)$
$\Rightarrow (ROBOT, \langle R \mid Grip : it, Busy : True, Location : L\rangle)$

**LDCp:** $(LOAD\text{-}Cp, Load\text{-}cp(C, R, Rf, Adr)) \Rightarrow (LOAD, Load(R, Rf, Adr))$

**LD:** $(LOAD, Load(R, Rf, Adr)) \Rightarrow (GOTO, Goto(R, Adr))\ ;\ (GET, Get(R, Adr, -))\ ;$
$(GOTO, Goto(R, Clamping))\ ;\ (PUT, put(R, I, -, Clamping))$

**LDCp:** $(UNLOAD\text{-}Cp, Unload\text{-}cp(C, R, Rf, Adr)) \Rightarrow (UNLOAD, Unload(R, Rf, Adr))$

**ULD:** $(UNLOAD, Unload(R, Rf, Adr)) \Rightarrow (GOTO, Goto(R, Adr))\ ;\ (GET, Get(R, Adr, -))$

**The stock class rewriting Rule**

**FULL** $:(STOCK, \langle S \mid Contents : S, Capacity : C, Full : False\rangle)$
$\Rightarrow (STOCK, \langle S \mid Contents : S, Capacity : C, Full : True\rangle) \otimes (ALARM, alarm(S, M))$
$if\ \ (length(S) = C)$

**Remark 3.3** In the above rewrite rules we have used an additional operator for *enforcing* an (sequential) order on which the messages should be performed. We denote this sequential operator as usual by ";"[9]. The appropriate semantics of this operator and other (process-algebra-like ones such as choice, synchronization, etc) operators have been already introduced by Wirsing et al. in [LLNW96, WNL95] for the MAUDE language. This complete rewriting algebra that allows for controlling the rewriting process like in process algebras, has been presented as follows.

```
mod MSG_Algebra is
  protecting CONFIGURATION
  op _+_        (* for specifying the choice between two rules *) .
  op _;_        (* for specifying the sequence of two rules *) .
  op _||_       (* for specifying parallel composition *) .
  vars m1, m2, n1, n2 : Msg .
  vars c, d, c1, c2, d1, d2, h : Configuration (as a multiset of object states and messages using the 'empty' binary operator _).
    (* The rewrite rules specifying their semantics *)
  rl (m1 + m2) c => d     if m1 c => d \/ m2 c => d.
  rl (m1 ; m2) c1 c2 => d1 d2     if m1 c1 => d1 h /\ m2 c2 h => d2.
  rl (m1 || m2) c1 c2 => d1 d2     if m1 c1 => d1 /\ m2 c2 => d2 .
endm
```

We limit ourselves here just to the adaptation of the choice operator to the CO-NETS setting; the other operators may be done following the same reasoning.

**choice** $:(P\_ms_1, ms_1)\ ;\ (P\_ms_2, ms_2) \otimes \underset{k}{\otimes}(P_k, [t_k]_\oplus) \otimes \underset{l}{\otimes}(P_l, [t_l]_\oplus) \Rightarrow \underset{k'}{\otimes}(P_{k'}, [t_{k'}]_\oplus) \otimes \underset{l'}{\otimes}(P_{l'}, [t_{l'}]_\oplus)$
$\texttt{if}\ (P\_ms_1, ms_1) \otimes \underset{k}{\otimes}(P_k, [t_k]_\oplus) \Rightarrow \underset{k'}{\otimes}(P_{k'}, [t_{k'}]_\oplus) \otimes \underset{h}{\otimes}(P_h, [t_h]_\oplus)\ \texttt{and}\ (P\_ms_2, ms_2) \otimes$
$\underset{l}{\otimes}(P_l, [t_l]_\oplus) \otimes \underset{h}{\otimes}(P_h, [t_h]_\oplus) \Rightarrow \underset{l'}{\otimes}(P_{l'}, [t_{l'}]_\oplus)$ $\diamondsuit$

Finally, given such rewrite rules and an initial community (of objects and messages) of each component, is is quite possible to derive in a true concurrent way any reachable community. Such computation using concurrent rewriting techniques and Maude [CDE+99] is simultaneously accompanied by the corresponding graphical animation which allows to detect missing and errors. However, due to space limitation we kindly advice the reader to consult our paper [ACS99] where a large banking account system is prototyped.

[9]We are grateful to one of the two referees for pointing out the inappropriateness of using in this case the operator $\otimes$ due to its commutativity.

### 3.3 CO-NETS : More Advanced Constructions

So far, we have presented only how the CO-NETS approach allows for conceiving independent classes. In what follows, we give how more complex systems can be constructed using advanced abstraction mechanisms, especially inheritance and interaction between classes. However, due to space limitation and to the straightforward extension of simple inheritance to multiple inheritance (with or without redefinition) we only present the former one.

### 3.4 Inheritance in CO-NETS

Giving a (super) class *Cl* modeled as a CO-Net, for constructing a subclass that inherits the structure as well as the behaviour of the superclass *Cl* and exhibits new behaviour involving additional attributes, we propose the following straightforward conceptualization.

- We define the structure of the new subclass by introducing the new attributes and messages. Structurally, the new attribute identifiers with their value sorts and the message generators are described using the *extending* primitive in the OBJ notation.
- As *object place* for the subclass we use the *same* object place of the superclass; which means that such place should contains now the object states of the superclass as well as the object states of the subclass. This is semantically sound because the sort of this object place is a supersort for objects including more attributes.
- As previously described, the proper behaviour of the subclass is constructed by associating with each new message a corresponding place and constructing its behaviour (i.e. transitions) with respect to the communication model of figure 3 under the condition that at least one of the additional attributes has to be involved in such transitions.

**Remark 3.4** Such conceptualization is only possible because of the splitting / recombination operation. Indeed this axiom permits to consider an object state of a subclass, denoted for instance as $\langle Id|attrs, attrs'\rangle$ with $attrs'$ the additional attributes (i.e. those proper to the subclass), to be also an object state of the superclass (i.e. $\langle Id|attrs\rangle$). Obviously, this allows a systematic inheritance of the and structure as well as the behaviour. ♢

**Example 3.4** *With respect to our case study, we assume that, besides the 'standard' robots more 'sophisticated' ones can be used— as a subclass of (standard) robot. In such special robots, for instance, the speed of the performed actions, as an additional attribute, may be changed (increased or decreased) when such special robots are not busy. Moreover, we assume that such robots are equipped with a second grip and so they can get or put two items at the same time. Of course, such a 'sophisticated' robot besides this proper behaviour behaves exactly as standard robot when it receives message like Goto, Move, ..ect; In other words it* **inherits** *all the structure and the behaviour of standard robots. More precisely, following the aforementioned steps, we present hereafter the structure as well as the associated* CO-NET *modeling this subclass.*

```
obj Special-Robot is.
  extending Robot .
  subsort Id-Sp.Robot < Id.Robot .
  subsort CHG-SPEED < Local_Sp-Robot_Mes .
  subsort GET-TWO  PUT-TWO < External_Sp-Robot_Mes .
  subsort Local_Sp-Robot < Local_Robot .
  subsort External_Sp-Robot < External_Robot.
  (* local attributes *)
   op ⟨_|Speed : _⟩ :  Id.Sp-Robot  Nat → Local_Sp-Robot.
  (* observed attributes *)
   op ⟨_|Grip2 : _⟩ :  Id.Sp-Robot  ITEM → External_Sp-Robot .
  (* local messages *)
```

```
  op Chg-speed :  Id-Sp-Robot  Nat → CHG-SPEED .
  (* export messages *)
  op Put-Two, Get-Two :  Id.Robot  OId  ITEM  ITEM  R-ADR → Exported_Sp-Robot_Mes .
endo.
```

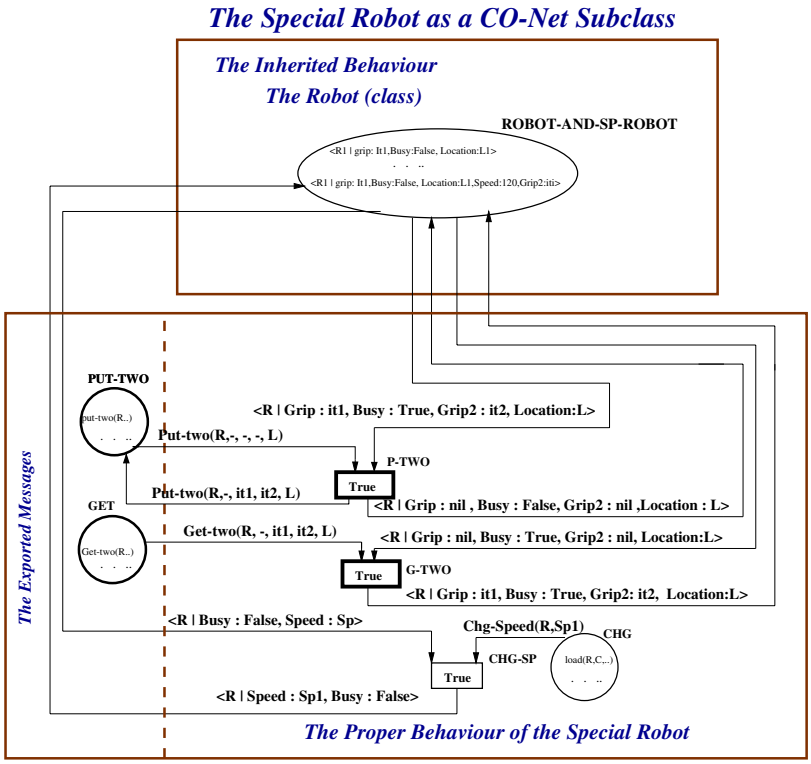



Figure 7: The Special Robot as a Subclass in the Robot component

## 3.5 Component interaction

Taking into account that object state evolution in components is ensured by the intra-component pattern specified in figure 6, and in order to ensure the encapsulation property which stipulate that the internal part of an object state as well as the local messages have to be hidden from the outside, we propose an appropriate inter-component interaction pattern for communicating these components by exclusively using their *explicitly* declared observed attributes and external messages.

As depicted in figure 8, the general schema of 'external' transitions and may be explicit as follows—by selecting just one object place from each component. The contact of some but only *external* parts of some objects states namely $\underset{i}{\oplus}\langle Id_{1_i}|attrs\_bs_{1_i}\rangle, \ldots \underset{j}{\oplus}\langle Id_{p_j}|attrs\_bs_{p_j}\rangle$ respectively belonging to components $Cp_1, ..., Cp_p$, with some external (i.e. declared as imported or exported) messages, namely $ms_{i_1}, .., ms_{i_r}, ms_{j_1}, .., ms_{j_h}$ from such components, and under some conditions on attributes values and parameters messages, results in the following: (1) messages $ms_{i_1}, .., ms_{i_r}$ being consumed; (2) states of some external parts of object participating in the communication change; and (3) new external messages (that may involve deletion/creation ones) are sent to objects of different in different components, namely $ms_{h_1}, .., ms_{h_r}$.

**Example 3.5** *In order to achieve the expected objective, that is, the production of different chessmen, the seven components have to interact through their imported / exported messages and observed attributes with respect to the general inter-component pattern described above. When*

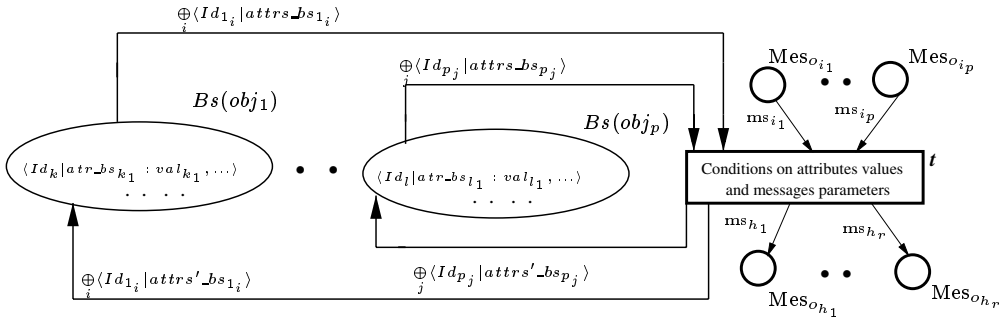



Figure 8: The Inter-component interaction pattern

*we restrict ourselves to the three components* `Robot`, `Stock` *and* `Manager`, *the corresponding interaction* CO-NETS *is depicted in figure 9. In this interaction, the effect of a* `Put` *message— that have afterwards to be consumed in the internal behaviour of the Robot— is the result of the interaction of the Stock and Robot component; they interact here only through their* `observed` *attributes Content and Location respectively. More precisely, to get an item from the stock, the robot should be at the corresponding location (i.e. Location = address-content) that have to be non empty. The same reasoning may be applied to the message Get. Equivalent to these messages are the* `Store` *and* `Remove` *messages by the manager into / from stock component.*

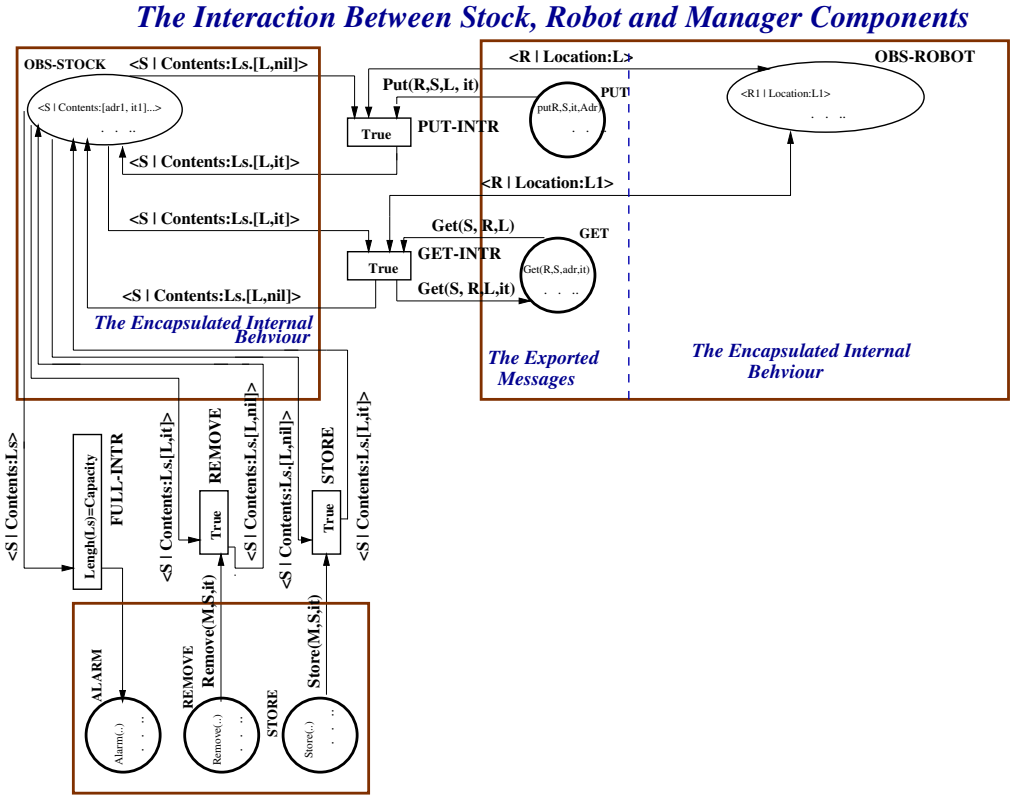



Figure 9: The Robot-Stock-Manager interaction behaviour.

The rewriting rules governing this interaction can be captured in a similar way, as done for the internal behaviour, from the effect of each transition.

# 4 Conclusion

We proposed an object Petri nets based conceptual model for specifying and validating information systems as a distributed, autonomous and cooperating components. The model called CO-NETS is a sound and complete combination of OO concepts and constructions in a variant of algebraic Petri nets. The semantics of the CO-NETS is expressed in rewriting logic allowing us to derive rapid-prototypes using concurrent rewriting. Some key features of the CO-NETS approach for specifying complex and distributed information systems include: (1) a straightforward modeling of simple and multiple inheritance with the possibility of overriding; (2) a characterization of two communication patterns—an intra-component model for evolving object states in a hierarchy of classes with the possibility of exhibiting intra-object as well as inter-object concurrency, and an inter-component communication model for interacting different components.

Different features of the approach have been presented through a non trivial case study dealing with a typical Chess Making Shop; where more than six interacting components cooperate through their explicit interfaces for achieving final chessmen product. This case study shows that the CO-NETS conceptual model, with its different abstractions mechanisms, is well suited for dealing with complex information and production systems conceived as cooperating components.

As nearest future work we plan to focus on the validation phase; where we have to confirm the relevance of CO-NETS graphical simulation with the symbolic computation (using rewriting techniques [DJ90]) in verifying main properties of the specified at hand. Moreover, although we have just sketched in this present paper, the control of the order in firing different transitions, either by adapting the message algebra proposed in [WNL95, LLNW96] or more elegantly by following rewriting logic reflection capabilities [CM96], deserves more deeper investigations.

### Acknowledgments

We acknowledge the two anonymous referees for their sharp comments and very detailed suggestions which resulted in a significant improvement of the paper.

# References

[ACS99] N. Aoumeur, S. Conrad, and G. Saake. Prototyping Object Specifications Using the CO-NETS Approach. In J. Desel and A. Oberweis, editors, *Proc. Sixth Workshop Algorithmen und Werkzeuge für Petrinetze, Frankfurt/Main, October 1999*, pages 7–17, 1999.

[AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. Technical report, School of Computer Science, Carnegie Mellon University, 1997.

[AS99a] N. Aoumeur and G. Saake. Towards a New Semantics for Mondel Specifications Based on the CO-Nets Approach. In J. Desel and K. Pohl and P. Schuerr, editor, *Proc. of Modellierung'99*, pages 107–122, Karlsrule, Germany, March 1999. B.G. Teubner-Verlag.

[AS99b] N. Aoumeur and G. Saake. Towards an Object Petri Nets Model for Specifying and Validating Distributed Information Systems. In M. Jarke and A. Oberweis, editors, *Proc. of the 11th Int. Conf. on Advanced Information Systems Engineering, CAiSE'99, Heidelberg, Germany*, volume 1626 of *Lecture Notes in Computer Science*, pages 381–395, Berlin, 1999. Springer-Verlag.

[AS00] N. Aoumeur and G. Saake. CO-NETS: A Formal OO Framework for Specifying and Validating Distributed Information Systems. Preprint Nr. 2, Fakultät für Informatik, Universität Magdeburg, 2000.

[BBG97] O. Biberstein, D. Buchs, and N. Guelfi. CO-OPN/2: A Concurrent Object-Oriented Formalism. In *Proc. of Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems(FMOODS)*, pages 57–72. Chapman and Hall, March 1997.

[BM92] M. Bettaz and M. Maouche. How to Specify Non Determinism and True Concurrency with Algebraic Term Nets. In M. Bidoit, and C. Choppy, editor, *Proc. of 8th Workshop on Abstract Data Types*, volume 655 of *Lecture Notes in Computer Science*, pages 164–180, 1992.

[CDE+99] M. Clavel, F. Duran, S. Eker, J. Meseguer, and M. Stehr. Maude : Specification and Programming in Rewriting Logic. Technical report, SRI, Computer Science Laboratory, March 1999. URL : http://maude.csl.sri.com.

[CM96] M. Clavel and J. Meseguer. Reflection and Strategies in rewriting logic. In G. Kiczales, editor, *Proc. of Reflection'96*, pages 263–288. Xerox PARC, 1996.

[DB95] P. Du Bois. *The Albert II Language: On the Design and the Use of a Formal Specification Language for Requirements Analysis.* PhD thesis, Computer Department, University of Namur, Namur(Belgique), September 1995.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.

[EGS92] H.D. Ehrich, M Gogolla, and A. Sernadas. Objects and Their Specification. In M. Bidoit and C. Choppy, editors, *Proc. of 8th Workshop on Abstract Data Types*, volume 655 of *Lecture Notes in Computer Science*, pages 40–66. Springer-Verlag, 1992.

[EM90] H. Ehrig and B. Mahr. Fundamentals of Algebraic Specifications 2 : Module Specifications and Constraints. *EATCS Monographs on Theoretical Computer Science*, 1990.

[GD93] J.A. Goguen and R. Diaconescu. Towards an Algebraic Semantics for the Object Paradigm. In *Proc. of 10th Workshop on Abstract Data types*, 1993.

[GD94] J. Goguen and R. Diaconescu. An Oxford Order Sorted Algebra. *Mathematical Structures in Computer Science*, 4(4), 1994.

[GWM+92] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, 1992.

[Lak95] C. Lakos. From Coloured Petri Nets to Object Petri nets. In *Proc. of 16th Application and Theory of Petri Nets*, volume 935 of *Lecture Notes in Computer Science*, pages 278–287. Springer-Verlag, 1995.

[LLNW96] U. Lechner, C. Langauer, F. Nickel, and M. Wirsing. (Objects + Concurrency) & Reusability – A Proposal to Circumvent the Inheritance Anomaly. In *ECOOP'96 - Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 232–248. Springer Verlag, 1996.

[Mes93] J. Meseguer. A Logical Theory of Concurrent Objects and its Realization in the Maude Language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*, pages 314–390. The MIT Press, 1993.

[PS98] M. P. Papazoglou and G. Schlageter, editors. *Cooperative Information Systems : Trends and Directions.* Academic Press, Boston, 1998.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, 1991.

[SB94] C. Sibertin-Blanc. Communicative and cooperative nets. In E. Astesiano, R. Reggio, and A. Tarlecki, editors, *Proc. of the 15th International Confernce on the application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Weg90] P. Wegner. Concepts and paradigms of Object-Oriented Programming. *OOPS Messenger*, 1:7–87, 1990.

[WNL95] M. Wirsing, F. Nickel, and U. Lechner. Concurrent Object-Oriented Specification in Spectrum. In Y. Inagaki, editor, *Workshop on Algebraic and Object-Oriented Approaches to Software Science, Nagoya/Japan*, Electronic Notes in Theoretical Computer Science, pages 39–70. Nagoya University, 1995.

# A a Complete Co-nets Chess Making Shop Specification

**The controller template specification:** The data types that are associated with the Controller signature are: RECIPE as a list of pairs: programs (shortly, PGM) and their corresponding machines MACH-TOOL. The ORDER as triplet consisting of: Loc as a set of pairs of AGV identities and their corresponding location i.e. LOC-AGV; Dest as final destination corresponding to a LOC-AGV; and Tdl as the corresponding Recipe of this orders. TYPE for capturing different types of chessmen i.e. Cyl, King, Queen, Castle, Bishop, Knight and Pawn. STATUS for representing different states of the machines i.e. Free, Booked, Working, Done and Down. Finally, Place of the AGV corresponding to Place1 or Place2. The corresponding specification of these data types can be done as follows:

```
obj CONTROL-DATA is
  protecting ROBOT-DATA .
  sorts REPICE  ORDER  STATUS.
  subsorts REPICE-ELT < RECIPE .
  subsorts LOC-ELT < LOC .
  op Cyl  King  Queen  Castle  Bishop  Knight  Pawn:  → TYPE .
  op Free  Booked  Working  Done  Down :  → STATUS .
  op Place1  Place2 :  → PLACE .
  op [_,_]:  MACH-TOOL  PRG → RECIPE-ELT .
  op __:  RECIPE-ELT  RECIPE → RECIPE .
  op [_,_]:  AGV  LOC-AGV → LOC-ELT .
  op _,_:  LOC-ELT  LOC → LOC [assoc.  comm.]  .
  op [_,_,_]:  LOC  LOC-AGV  RECIPE → ORDER .
endo.
```

Using these data types and respecting the structure of the Controller as described in figure 1, corresponding template signature and associated Co-net are given in what follows.

```
obj Controller-signature is
  protecting Object-state, CONTROL-DATA .
  subsort Id.Controler < OId .
  subsort Local_Controller  External_Controller < Controller .
  subsort MACHINING-STEP MACHNING < Local_Controller_Mes .
  subsort UNLOAD-CAMP  LOAD-CAMP  RUN  CARRY
            ALARM< Exported_Controller_Mes .
  (* local attributes *)
  op ⟨_|Free − cyl : _, Agv − ord : _, Recipe[Type] : _, Proc[LOC-AGV] : _,
  STATUS[LOC − AGV] : _⟩ :  Id.Controller  ST-ADRS  REF  RECIPE
     REF  STATUS → Local_Controller.
  (* observed attributes *)
  op ⟨_|Orders[Ref] : _⟩ :  Id.Controller  ORDER → External_Controller .
  (* local messages *)
  op Machning-step  Machining → Local_Controller_Mes .
  (* export messages *)
  op Load-clamp, Unload-clamp :  Id.Controller  OId  REF  ST-ADR → Exported_Controller_Mes .
  op Run :  Id.Controller  OId  REF  MACH-TOOL  PGM → RUN .
  op Carry :  Id.Controller  OId  REF  ITEM  LOC-AGV  LOC-AGV → CARRY .
  op Alarm :  Id.Controller  OId → ALARM .
  (* Imported messages *)
  op Put :  OId  Id.Controller  ITEM  LOC → Imported_Controller_Mes (* from the Robot Component *).
  op Clamping-done, Unclamping-done :  OId  Id.Controller
     → Imported_Controller_Mes (* from the Clamping-system Component *).
  op Remove, Store :  OId  Id.Controller  ITEM  ST-ADR
     → Imported_Controller_Mes (* from the Manager Component *).
  op Produce :  OId  Id.Controller  TYPE  REF
     → Imported_Controller_Mes (* from the Manager Component *).
  op Work-done :  OId  Id.Controller  PGM
     → Imported_Controller_Mes (* from the Machine Tool Component *).
  op Deliver :  OId  Id.Controller  ITEM  PLACE  ITEM
```

```
      → Imported_Controller_Mes (* from the Auto-Guided Vehicle Component *).
endo.
```

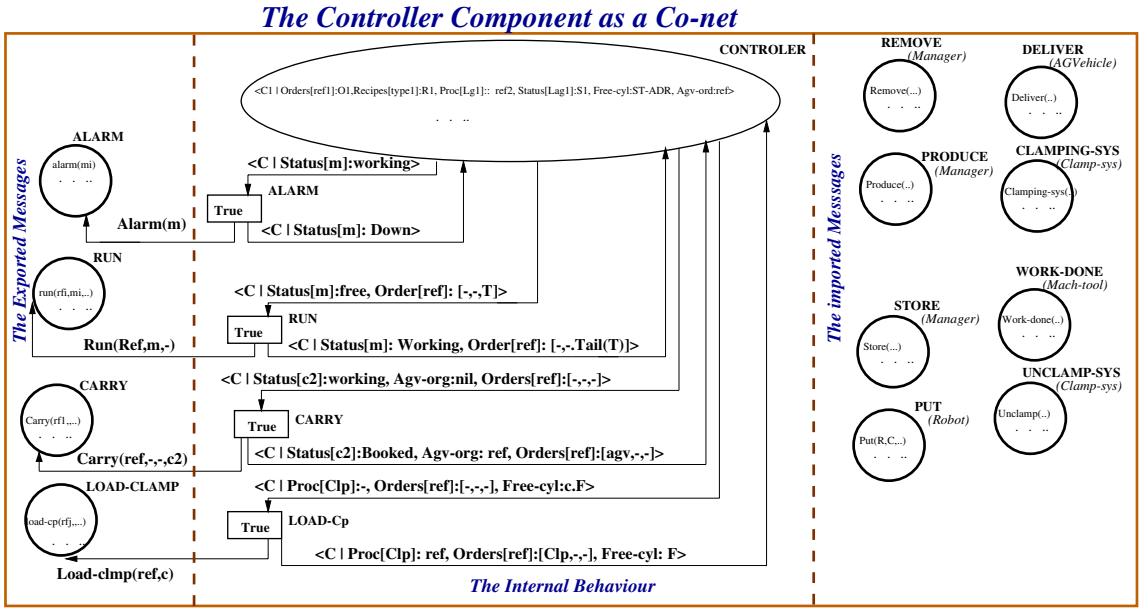



Figure 10: The Controller Component as a Co-NET .

**The auto-guided vehicle specification:**

```
obj AGV-signature is
  protecting Object-state  CONTROL-DATA .
  subsort Id.AGV < OId .
  subsort Local_AGV  External_AGV < AGV .
  subsort MOVE LOAD  GOTO  TRANSPORT < Local_AGV_Mes .
  subsort PICK-UP  DELIVER  RUN < Exported_AGV_Mes .
  (* local attributes *)
   op ⟨_|Place1 : _, Place2 : _, Recipe[Type] : _⟩ :  Id.AGV  PLACE  PLACE → Local_AGV.
  (* observed attributes *)
   op ⟨_|Location : _⟩ :  Id.AGV  Clamping → External_AGV .
  (* local messages *)
   op Goto :  Id.AGV  Clamping → GOTO .
   op Load :  Id.AGV  ITEM  PLACE → LOAD .
   op Move :  Id.AGV  Clamping  Clamping → MOVE .
   op Transport :  Id.AGV  ITEM  Clamping  Clamping → TRANSPORT .
  (* export messages *)
   op Pick-Up  Id.AGV  OId  ITEM  R-ADR→ Exported_AGV_Mes .
   op Deliver  Id.AGV  OId  ITEM  Clamping→ Exported_AGV_Mes .
  (* Imported messages *)
   op Carry :  Id.Controller  OId  REF  ITEM  LOC-AGV  LOC-AGV
      → Imported_AGV_Mes (* from the Controller Component *).
endo.
```

**The machine tool component specification:**

```
obj MACHINE-DATA is
  protecting R-DATA .
  subsorts PGM < LIST-PGM .
  op __:  PGM  LIST-PGM → LIST-PGM [assoc.  comm.]  .
endo.

obj Machine-signature is
  protecting Object-state  MACHINE-DATA .
```

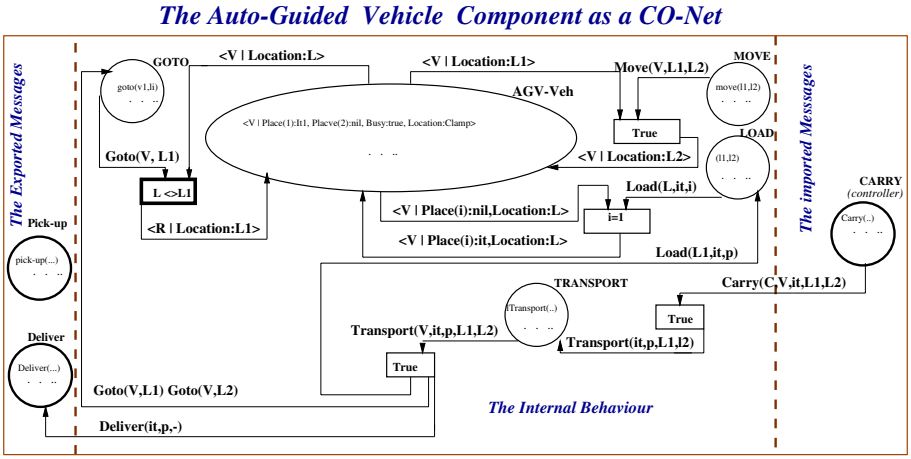



Figure 11: The Auto-Guided Vehicle Component Co-Net.

```
  subsort Id.Machine < OId .
  subsort Local_Machine  External_Machine < Machine .
  subsort WORK < Local_Machine_Mes .
  subsort WORK-DONE < Exported_Machine_Mes .
  (* local attributes *)
   op ⟨_|Busy : _, Capabilities : _⟩ :  Id.Machine  Bool  LIST-PGM → Local_Machine .
  (* observed attributes *)
   op ⟨_|Contents : _⟩ :  Id.Machine  ITEM → External_Machine .
  (* local messages *)
   op Work :  Id.Machine  PGM → Local_Machine_Mes .
  (* export messages *)
   op Work-done :  Id.Machine  OId  PGM → Exported_Message_Mes .
  (* Imported messages *)
   op Run :  Id.Controller  OId  REF  MACH-TOOL  PGM
      → Imported_Machine_Mes (* from the Controller Component *).
endo.
```

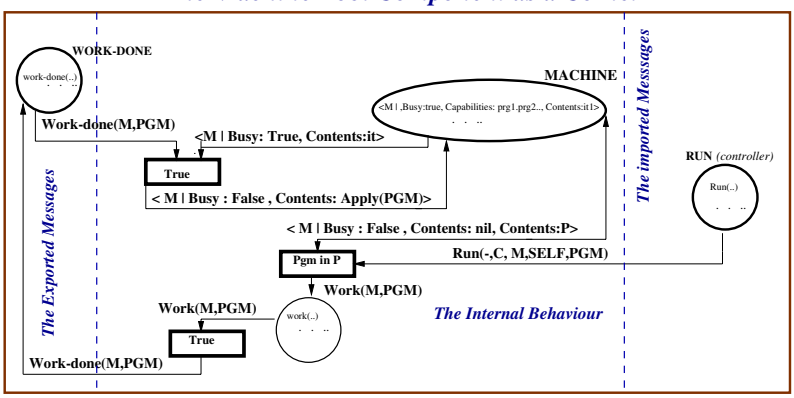



Figure 12: The Machine Tool Component Co-Net.

**The clamping system component specification:**

```
obj Clamping-signature is
  protecting Object-state  S-DATA .
  subsort Id.Clamping < OId .
  subsort Local_Clamping  External_Clamping < Clamping .
  subsort UNCLAMP  CLAMP < Local_Clamping_Mes .
```

```
  subsort UNCLAMPING CLAMPING < Exported_Clamping_Mes .
  (* local attributes *)
  op ⟨_|Clamped : _⟩ : Id.Clamping Bool → Local_Clamping.
  (* observed attributes *)
  op ⟨_|Contents : _⟩ : Id.Clamping ITEM → External_Clamping .
  (* local messages *)
  op Uclamp Clamp : Id.Clamp → Local_Clamping_Mes .
  (* export messages *)
  op Clamp-done Unclamp-done: Id.Clamping OId → Exported_Clamping_Mes .
  (* Imported messages *)
  op Put, Get : OId Id.Controller ITEM LOC
     → Imported_Clamping_Mes (* from the Robot Component *).
  op Pick-up, Deliver : OId Id.Controller ITEM PLACE →
     Imported_Clamping_Mes (* from the Auto-Guided Vehicle Component *).
endo.
```

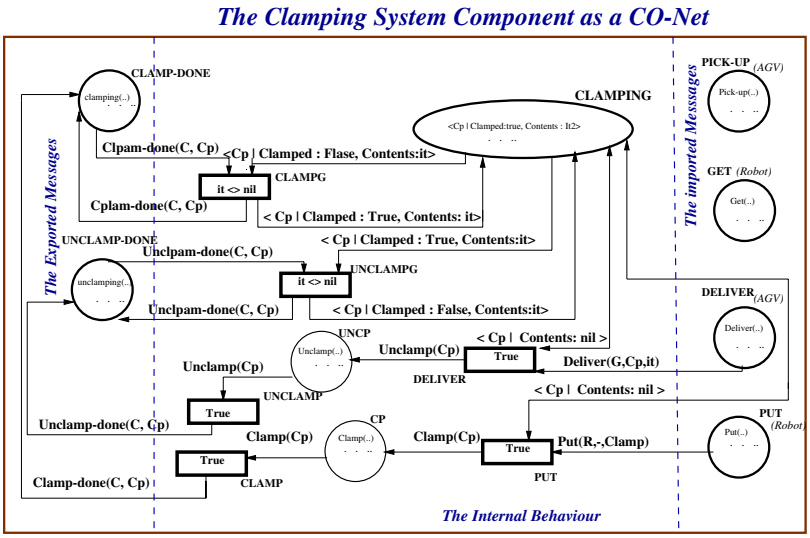



Figure 13: The Clamping Component Co-Net.