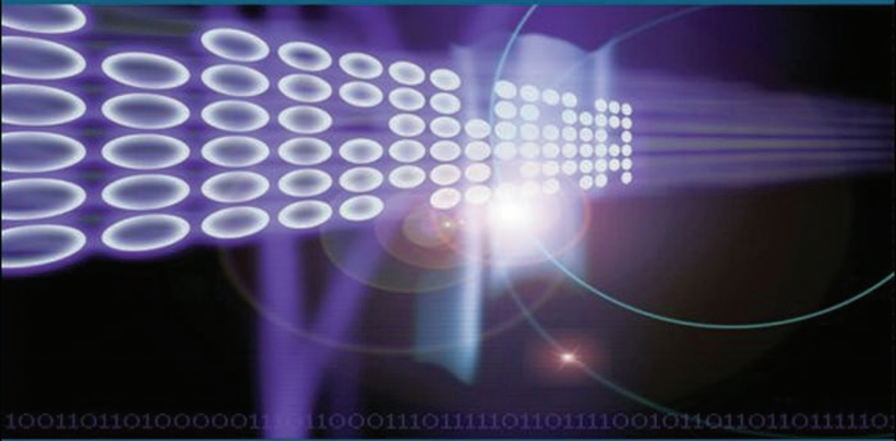


Modeling and Verification of Real-Time Systems

Edited by
Stephan Merz and Nicolas Navet

iSTE

WILEY

Modeling and Verification of Real-Time Systems

Modeling and Verification of Real-Time Systems

Formalisms and Software Tools

Edited by
Stephan Merz
Nicolas Navet

ISTE WILEY

First published in Great Britain and the United States in 2008 by ISTE Ltd and John Wiley & Sons, Inc.

ISTE Ltd
6 Fitzroy Square
London W1T 5DX
UK

www.iste.co.uk

John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030
USA

www.wiley.com

Library of Congress Cataloging-in-Publication Data

Modeling and verification of real-time systems : formalisms and software tools / edited by Nicolas Navet, Stephan Merz.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-1-84821-013-4
1. Real-time data processing. 2. Computer software--Verification. 3.
Formal methods (Computer science) I. Navet, Nicolas. II. Merz, Stephan.
QA76.54.M635 2008
004.01'51--dc22

2007045063

British Library Cataloguing-in-Publication Data
A CIP record for this book is available from the British Library
ISBN: 978-1-84821-013-4

Printed and bound in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire.

Contents

Preface

The study of real-time systems has been recognized over the past 30 years as a discipline of its own whose research community is firmly established in academia as well as in industry. This book aims at presenting some fundamental problems, methods, and techniques of this domain, as well as questions open for research.

The field is mainly concerned with the control and analysis of dynamically evolving systems for which requirements of timeliness are paramount. Typical examples include systems for the production or transport of goods, materials, energy or information. Frequently, controllers for these systems are "embedded" in the sense that they are physically implemented within the environment with which they interact, such as a computerized controller in a plane or a car. This characteristic imposes strong constraints on space, cost, and energy consumption, which limits the computational power and the available memory for these devices, in contrast with traditional applications of computer science where resources usually grow exponentially according to Moore's law. The design of real-time systems relies on techniques that originate in several disciplines, including control theory, operations research, software engineering, stochastic process analysis and others.

Software supporting real-time systems needs not only to compute the correct value of a given function, but it must also deliver these values at the right moment in order to ensure the safety and the required performance level of the overall system. Usually, this is implemented by imposing constraints (or *deadlines*) on the termination of certain activities. The verification techniques presented in this volume can help to ensure that deadlines are respected.

Chapter written by Stephan MERZ and Nicolas NAVET.

The chapters of this book present basic concepts and established techniques for modeling real-time systems and for verifying their properties. They concentrate on functional and timing requirements; the analysis of non-functional properties such as schedulability and Quality of Service guarantees would be a useful complement, but would require a separate volume. Formal methods of system design are based on mathematical principles and abstractions; they are a cornerstone for a "zero-default" discipline. However, their use for the development of real-world systems requires the use of efficient support tools. The chapters therefore emphasize the presentation of verification techniques and tools associated with the different specification methods, as well as the presentation of case studies that illustrate the application of the formalisms and the tools. The focus lies on model checking approaches, which attempt to provide a "push-button" approach to verification and integrate well into standard development processes. The main obstacle for the use of model checking in industrial-sized developments is the state-explosion problem, and several chapters describe techniques based on abstraction, reduction or compression that stretch the limits of the size of systems that can be handled.

Before verification can be applied, the system must be modeled in a formal description language such as (timed) Petri nets, timed automata or process algebra. The properties expected of a system are typically expressed in temporal logic or using automata as observers. Two main classes of properties are *safety* properties that, intuitively, express that nothing bad ever happens, and *liveness* properties that assert that something good eventually happens. The third step is the application of the verification algorithm itself to decide whether the properties hold over the model of the system or not; in the latter case, model checking generates a counter-example exhibiting a run of the system that violates the property.

Beyond *verification*, which compares two formal objects, the model should also be *validated* to ensure that it faithfully represents the system under development. One approach to validation is to decide healthiness properties of the model (for example, ensure that each component action can occur in a system run), and model checking is again useful here. In general, it is helpful to narrow the gap between the system description and its formal model, for example by writing a model in a high-level executable language or in a notation familiar to designers such as UML. The chapters of this book, written by researchers active in the fields, present different possible approaches to the problems of modeling, verification and validation, as well as open research questions.

Chapter 1, written by Bernard Berthomieu, Florent Peres and François Vernadat, explains the analysis of real-time systems based on timed Petri nets. It illustrates the high expressiveness of that formalism and the different, complementary verification techniques that are implemented in the Tina tool.

In Chapter 2, Camille Constant, Thiery Jéron, Hervé Marchand and Vlad Rusu describe an approach that combines verification and conformance testing (on the actual implementation platform) of input/output symbolic transition systems. Disciplined approaches to testing are indeed a very valuable complement to formal verification for ensuring the correctness of an implementation. This is true in particular when the complexity of the models makes exhaustive verification impossible.

Chapters 3 and 4 are devoted to the presentation of model checking techniques. Starting with the canonical example of a lift controller, Stephan Merz presents the basic concepts and techniques of model checking for discrete state transition systems: temporal logics, the principles of model checking algorithms and their complexity, and strategies for mastering the state explosion problem. Patricia Bouyer and François Laroussinie focus on model checking for timed automata, the main semantic formalism for modeling real-time systems. They describe the formalism itself, timed modal and temporal logics, as well as some extensions and subclasses of timed automata. Finally, algorithms and data structures for the representation and verification of timed automata are introduced, and the modeling and verification environment Uppaal is described in some detail.

Using a model of an industrial drilling station as a running example, Radu Mateescu presents in Chapter 5 the functionalities of the CADP toolbox for modeling and verification. CADP is designed to model arbitrary asynchronous systems whose components run in parallel and communicate by message passing. The toolbox accepts models written in different formalisms, including networks of communicating automata or higher-level models written in Lotos. It implements a set of model transformations, simulation and verification algorithms, and offers the possibility to generate conformance tests for the implementation.

Chapter 6, written by Pascal Raymond, is devoted to the verification of programs written in the synchronous language Lustre with the help of the model checker Lesar. Synchronous languages enjoy ever more success for the development of reactive systems, of which real-time systems are a particular instance. Based on mathematical models of concurrency and time, synchronous languages provide a high-level abstraction for the programmer and are well-suited to formal verification.

In Chapter 7, Paul Caspi, Grégoire Hamon and Marc Pouzet go on to describe the language Lucid Synchrone that extends Lustre with constructs borrowed from functional languages, further augmenting expressiveness. The authors start by asking why synchronous languages are relevant for the design of critical systems. They give an account of the development of the Lucid language, and present in detail its primitives and the underlying theoretical concepts, illustrating them by several examples.

One of the most exciting developments over the past 15 years has been the emergence of techniques for the verification of probabilistic systems, intimately coupled with work on stochastic processes carried out in the realm of performance evaluation. Probabilistic models are very useful because they add quantitative information above the non-deterministic representation of the behavior of system components and the environment. They can also be used to determine system parameters such as queue sizes, as a function of the desired failure guarantees. Marta Kwiatkowska, Gethin Norman, David Parker and Jeremy Sproston lay the bases in Chapter 8 by defining probabilistic timed automata and extending the model checking algorithms for ordinary timed automata to handle probabilistic models. The case study of the IEEE FireWire Root Contention Protocol illustrates the application of these techniques. In Chapter 9, Serge Haddad and Patrice Moreaux give an overview of verification techniques for probabilistic systems: discrete and continuous-time Markov chains, stochastic Petri nets, Markov decision processes and associated temporal logics. They also cover some of the main tools used in this domain, including GreatSPN, ETMCC and Prism.

Chapter 10, written by Marius Bozga, Susanne Graf, Laurent Mounier and Iulian Ober, presents the IF toolset, a tool environment for modeling and verifying real-time systems centered around a common internal description language based on communicating timed automata. User-level specifications written in languages such as SDL or UML are translated into this internal representation and can be subject to analysis using algorithms of static analysis, reduction and model checking. An extended case study from the aerospace domain based on joint work with EADS concludes the chapter.

Chapter 11, written by Anne-Marie Déplanche and Sébastien Faucou, is dedicated to the architecture description language AADL, originally designed and standardized for the avionic and aerospace domains, but which is an excellent candidate for arbitrary real-time systems. Architectural descriptions can serve as a reference for all actors involved in system design; they contain the information needed for simulation, formal verification and testing. The authors examine the specific requirements for describing real-time systems and then present the AADL and its support tools. Their use is illustrated with the help of a case study of a closed-loop control system.

We would like to express our gratitude to all of the authors for the time and energy they have devoted to presenting their topic. We are also grateful to ISTE Ltd. for having accepted to publish this volume and for their assistance during the editorial phase.

We hope that you, the readers of this volume, will find it an interesting source of inspiration for your own research or applications, and that it will serve as a reliable, complete and well-documented source of information on real-time systems.

Stephan MERZ and Nicolas NAVET
INRIA Nancy Grand Est and LORIA
Nancy, France

Chapter 1

Time Petri Nets – Analysis Methods and Verification with TINA

1.1. Introduction

Among the techniques proposed to specify and verify systems in which time appears as a parameter, two are prominent: Timed Automata (see Chapter 4) and Time Petri nets, introduced in [MER 74].

Time Petri nets are obtained from Petri nets by associating two dates $\min(t)$ and $\max(t)$ with each transition t. Assuming t became enabled for the last time at date θ, t cannot fire (cannot be taken) before the date $\theta+\min(t)$ and must fire no later than date $\theta+\max(t)$, except if firing another transition disabled t before then. Firing a transition takes no time. Time Petri nets naturally express specifications "in delays". By making explicit the beginnings and ends of actions, they can also express specifications "in durations"; their applicability is thus broad.

We propose in this chapter a panorama of the analysis methods available for Time Petri nets and discuss their implementation. These methods, based on the technique of state classes, were initiated in [BER 83, BER 91]. State class graphs provide finite abstractions for the behavior of bounded Time Petri nets. Various abstractions have been proposed in [BER 83, BER 01, BER 03b], preserving various classes of properties. In this chapter, we will discuss in addition the practical problem of the verification of formulae (*model checking*) of certain logics on the graphs of state classes available. Using these techniques requires software tools, both for the construction of the state

Chapter written by Bernard BERTHOMIEU, Florent PERES and François VERNADAT.

space abstractions and for the actual verification of the properties. The examples discussed in this chapter are handled with the tools available in the `Tina` environment [BER 04].

The basic concepts of Time Petri nets are reviewed in section 1.2. Sections 1.3 to 1.5 introduce various state class graph constructions, providing finite abstractions of the infinite state spaces of Time Petri nets. Sections 1.3 and 1.4 present the constructions preserving the properties of linear-time temporal logics such as LTL; in addition to traces, the construction of section 1.3 preserves markings while that exposed in section 1.4 also preserve states (markings and temporal information). Section 1.5 discusses preservation of the properties expressible in branching-time temporal logics. Section 1.6 discusses the analysis of firing schedules and presents a method to characterize exactly the possible firing dates of transitions along any finite sequence. The toolbox `Tina`, implementing all state space abstractions reviewed and a model checker, is presented in section 1.7. The following sections are devoted to the verification of LTL formulae on the graphs of state classes. Section 1.8 presents the logic selected, $SE{-}LTL$, an extension of the LTL logic, and the implementation of a verifier for that logic, the module `selt` of `Tina`. Section 1.9 discusses two application examples and their verification.

1.2. Time Petri nets

1.2.1. *Definition*

Let $\mathbf{I}^+$ be the set of non-empty real intervals with non-negative rational endpoints. For $i \in \mathbf{I}^+$, $\downarrow i$ denotes its left endpoint and $\uparrow i$ its right endpoint (if i is bounded) or ∞ (otherwise). For all $\theta \in \mathbf{R}^+$, $i \stackrel{\bullet}{-} \theta$ denotes the interval $\{x - \theta \mid x \in i \wedge x \geqslant \theta\}$.

DEFINITION 1.1.– *A* Time Petri net *(TPN for short) is a tuple* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s\rangle$*, in which* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0\rangle$ *is a Petri net and* $I_s : T \rightarrow \mathbf{I}^+$ *is a function called the* Static Interval *function.*

Application I_s associates a temporal interval $I_s(t)$ with each transition of the net. The rationals $Eft_s(t) = \downarrow I_s(t)$ and $Lft_s(t) = \uparrow I_s(t)$ are called the static earliest firing time, and static latest firing time of transition t, respectively. A Time Petri net is represented in Figure 1.1.

1.2.2. *States and the state reachability relation*

DEFINITION 1.2.– *A state of a Time Petri net is a pair* $e = (m, I)$ *in which* m *is a marking and* $I : T \rightarrow \mathbf{I}^+$ *a function that associates a temporal interval with each transition enabled at* m*.*

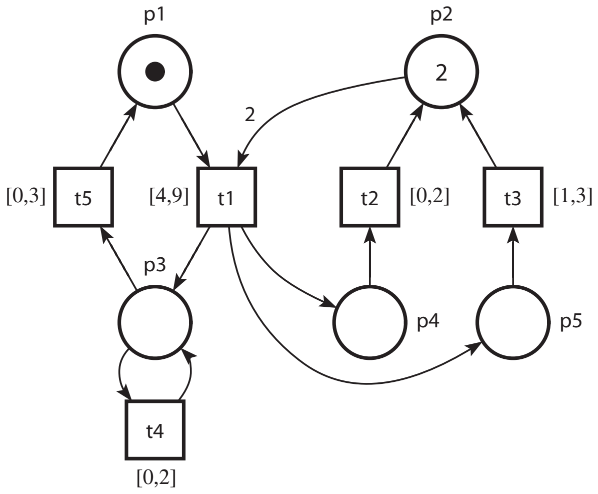


Figure 1.1. *A Time Petri net*

The initial state is $e_0 = (m_0, i_0)$, where I_0 is the restriction of I_s to the transitions enabled at m_0. Any transition enabled must fire in the time interval associated with it.

Firing t at date θ from $e = (m, I)$ (or, equivalently, waiting θ units of time then firing t instantly) is thus allowed if and only if:

$$m \geqslant \mathbf{Pre}(t) \wedge \theta \in I(t) \wedge (\forall k \neq t)(m \geqslant \mathbf{Pre}(k) \Rightarrow \theta \leqslant \uparrow(I(k))).$$

The state $e' = (m', I')$ reached from e by firing t at θ is determined by:

1) $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$ (as in Petri nets).

2) For each transition k enabled at m':

$$I'(k) = \text{if } k \neq t \text{ and } m - \mathbf{Pre}(t) \geqslant \mathbf{Pre}(k) \text{ then } I(k) \dot{-} \theta \text{ else } I_s(k).$$

Let us note by $\xrightarrow{t@\theta}$ the timed reachability relation defined above, and let $e \xrightarrow{t} e'$ stand for $(\exists\theta)(e \xrightarrow{t@\theta} e')$. A *firing schedule* is a sequence of timed transitions $t_1@\theta_1 \cdots t_n@\theta_n$. It is firable from e if the transitions in sequence $\sigma = t_1 \cdots t_n$ are successively firable at the relative dates they are associated with in the schedule. σ is called the *support* of the schedule. A sequence of transitions is firable if and only if it is the support of some firable schedule.

Let us note that, in Time Petri nets, and contrarily to Timed Automata, the elapsing of time can only increase the set of firable transitions from a state, but can in no case reduce it. Relation $\xrightarrow{t}$ characterizes exactly the "discrete" behavior of a TPN (bisimilarity after abstraction of time) when interpreting time-elapsing as non-determinism. Alternatively, time-elapsing could be interpreted as for Timed Automata. In that case, we should add to the transitions of relation $\xrightarrow{t@\theta}$ those representing the elapsing of time, of form $(m, I) \xrightarrow{r} (m, I \dot{-} r)$ where, for any k, r is not larger than $\uparrow (I(k))$.

Finally, let us note that the concept of state introduced in this section associates exactly one temporal interval with each enabled transition, whether or not that transition is multi-enabled (t is multi-enabled at m if there is an integer $k > 1$ such that $m \geqslant k.\mathbf{Pre}(t)$). An alternative interpretation of multi-enabledness is discussed in [BER 01], in which several temporal intervals may be associated with transitions; this interpretation will be briefly discussed in section 1.3.4.1.

1.2.3. *Illustration*

The states can be represented by pairs (m,D), in which m is a marking and D is a set of vectors of dates called a firing domain. The i^{th} projection of domain D is the firing interval $I(t_i)$ associated with the i^{th} enabled transition. Firing domains can be described by systems of linear inequalities with one variable per enabled transition (noted like the transitions).

The initial state $e_0 = (m_0, D_0)$ of the net represented in Figure 1.1 is written:

$$\begin{aligned} m_0 &: p_1, p_2 * 2 \\ D_0 &: 4 \leqslant t_1 \leqslant 9 \end{aligned}$$

Firing t_1 from e_0 at relative time $\theta_1 \in [4, 9]$ leads to state $e_1 = (m_1, D_1)$ given by:

$$\begin{aligned} m_1 &: p_3, p_4, p_5 \\ D_1 &: 0 \leqslant t_2 \leqslant 2 \\ &\quad 1 \leqslant t_3 \leqslant 3 \\ &\quad 0 \leqslant t_4 \leqslant 2 \\ &\quad 0 \leqslant t_5 \leqslant 3 \end{aligned}$$

Firing t_2 from e_1 at relative time $\theta_2 \in [0, 2]$ leads to $e_2 = (m_2, D_2)$, where:

$$\begin{aligned} m_2 &: p_2, p_3, p_5 \\ D_2 &: \max(0, 1 - \theta_2) \leqslant t_3 \leqslant 3 - \theta_2 \\ &\quad 0 \leqslant t_4 \leqslant 2 - \theta_2 \\ &\quad 0 \leqslant t_5 \leqslant 3 - \theta_2 \end{aligned}$$

Since θ_2 may take any real value in $[0, 2]$, state e_1 admits an infinity of successors.

1.2.4. *Some general theorems*

A Petri net or Time Petri net is *bounded* if, for some integer b, the marking of each place is smaller than b. Let us recall an undecidability result.

THEOREM 1.1.– *The problems of marking reachability, of state reachability, of boundedness and of liveness are undecidable for Time Petri nets.*

Proof. It is shown in [JON 77] that the marking reachability problem for $TPNs$ is reducible to that, undecidable, of the termination of a two-counter machine. Undecidability of the other problems follows.

Representing the behavior of a Time Petri net by its state reachability graph (as the behavior of a Petri net is represented by its marking reachability graph) is in general impossible: the transitions being able to fire at any time in their firing interval, states typically admit an infinity of successors. The purpose of the state classes defined thereafter are precisely to provide finite representation for this infinite state space, when the network is bounded, by grouping certain sets of states. However, there are two remarkable subclasses of Time Petri nets admitting finite state graphs if and only if they are bounded. □

THEOREM 1.2.– *Consider a* TPN $\langle P, T, \mathbf{Pre}, \mathbf{Post}, M_0, I_s \rangle$. *If all transitions* $t \in T$ *have static interval* $[0, \infty[$, *then the state graph of the net is isomorphic with the marking graph of the underlying Petri net.*

Proof (by induction). If each transition carries interval $[0, \infty[$, then the firing conditions for the transitions are reduced to that in Petri nets (without temporal constraints). In addition, the firing rule preserves the shape of firing intervals. □

THEOREM 1.3.– *Consider a* TPN $\langle P, T, \mathbf{Pre}, \mathbf{Post}, M_0, I_s \rangle$. *If* I_s *associates with each transition a punctual interval (reduced to one point), then:*
(i) the state graph of the net is finite if and only if the net is bounded;
(ii) if, in addition, all transitions bear equal static firing intervals, then its state graph is isomorphic with the marking graph of the underlying Petri net.

Proof (by induction). (i) By the firing rule, if all static intervals are punctual, then a state has at most as many successor states as the number of transitions enabled at it. In addition, the firing rule preserves the punctual character of firing intervals. For (ii), note that the firing condition reduces then to that in Petri nets. □

Theorems 1.2 and 1.3 make it possible to interpret Petri nets as particular Time Petri nets, in various ways. The most frequent interpretation is to regard them as Time Petri nets in which each transition carries static interval $[0, \infty[$.

1.3. State class graphs preserving markings and LTL properties

1.3.1. *State classes*

The set of states of a Time Petri net may be infinite for two reasons: on one hand because a state may admit an infinity of successors and, on the other hand, because a TPN can admit schedules of infinite length going through states whose markings are all different. The second case will be discussed in section 1.3.3. To manage the first case, we will gather certain sets of states into state classes. Several grouping methods are possible; the construction reviewed in this section is that introduced in [BER 83, BER 91].

For each firable sequence σ, let us note by C_σ the set of states reached from the initial state by firing schedules of support σ. For any such set C_σ, let us define its marking as that of the states it contains (all these states have necessarily the same marking) and its firing domain as the union of the firing domains of the states it contains. Finally, let us note by $\cong$ the relation satisfied by two sets C_σ and $C_{\sigma'}$ when they have equal markings and equal firing domains. If two sets of states are related by $\cong$, then any schedule firable from some state in one of these sets is firable from some state in the other set.

The graph of state classes of states of [BER 83], or SCG, is the set of sets of states C_σ, for any firable sequence σ, considered modulo relation $\cong$, and equipped with the transition relation: $C_\sigma \xrightarrow{t} X$ if and only if $C_{\sigma.t} \cong X$. The initial state class is the equivalence class of the singleton set of states containing the initial state.

The SCG is built as follows. State classes are represented by pairs (m, D), where m is a marking and D a firing domain described by a system of linear inequalities $W\underline{\phi} \leqslant \underline{w}$. The variables $\underline{\phi}$ are bijectively associated with the transitions enabled at m. The equivalence $(m, D) \cong (m', D')$ holds if and only if $m = m'$ and $D = D'$ (i.e. the systems describing D and D' have equal solution sets).

ALGORITHM 1.1.– *Construction of the SCG, state classes*
For any firable sequence σ, let L_σ be the class computed as explained below. Compute the smallest set C of classes including L_ϵ and, whenever $L_\sigma \in C$ and $\sigma.t$ is firable, then $(\exists X \in C)(X \cong L_{\sigma.t})$:

– The initial class is $L_\epsilon = (m_0, \{Eft_s(t) \leqslant \underline{\phi}_t \leqslant Lft_s(t) \mid t \in T \wedge m_0 \geqslant \mathbf{Pre}(t)\})$.

– If σ is firable and $L_\sigma = (m, D)$, then $\sigma.t$ is firable if and only if:

(i) $m \geqslant \mathbf{Pre}(t)$ (t is enabled at m) and

(ii) the system $D \wedge \{\underline{\phi}_t \leqslant \underline{\phi}_i \mid i \in T \wedge i \neq t \wedge m \geqslant \mathbf{Pre}(i)\}$ is consistent.

– If $\sigma \cdot t$ is firable, then $L_{\sigma.t} = (m', D')$ is computed as follows from $L_\sigma = (m, D)$:

$m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$,

D' obtained as follows:

(a) the above firability constraints (ii), of t from L_σ, are added to D;

(b) for each k enabled at m', a new variable $\underline{\phi}'_k$ is introduced, obeying:

$\underline{\phi}'_k = \underline{\phi}_k - \underline{\phi}_t$, *if $k \neq t$ and $m - \mathbf{Pre}(t) \geqslant \mathbf{Pre}(k)$,*

$Eft_s(k) \leqslant \underline{\phi}'_k \leqslant Lft_s(k)$, *otherwise;*

(c) variables $\underline{\phi}$ are eliminated.

L_σ is the equivalence class by relation $\cong$ of the set C_σ of states reached from s_0 by firing schedules of support σ. Equivalence $\cong$ is checked by putting the systems representing firing domains into canonical forms. These systems are systems of differences. Computing canonical form for them reduces to a problem of "all-pairs shortest path", solved in polynomial time using, for example, Floyd/Warshall's algorithm.

Remark: two sets C_σ and $C_{\sigma'}$ can be equivalent by $\cong$ while having different contents in terms of states. The notation of the state classes by a pair (m, D) canonically represents an equivalence class of sets of states for relation $\cong$, but we cannot tell from such a class if it contains some given state.

1.3.2. *Illustration*

As an illustration, let us build some state classes of the TPN represented in Figure 1.1. The initial class c_0 is described in the same way as the initial state e_0 (see section 1.2.3). Firing t_1 from c_0 leads to a class c_1 described like state e_1 (since the target state does not depends on the time at which t_1 fired). Firing t_2 from c_1 leads to $c_2 = (m_2, D_2)$, with $m_2 = (p_2, p_3, p_5)$ and D_2 computed in three steps:

(a) D_1 is augmented with the firability conditions for t_2, given by system:

$$\begin{array}{l} t_2 \leqslant t_3 \\ t_2 \leqslant t_4 \\ t_2 \leqslant t_5 \end{array}$$

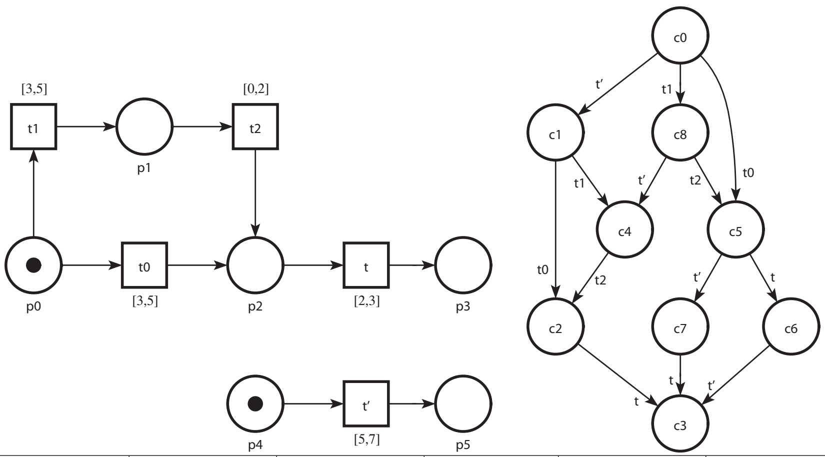
(b) No transition is newly enabled, and t_3, t_4, t_5 remain enabled while t_2 fires. So we simply add equations $t'_i = t_i - t_2$, for $i \in \{3, 4, 5\}$.

(c) The variables t_i are eliminated, yielding system:

$$\begin{array}{ll} 0 \leqslant t'_3 \leqslant 3 & t'_4 - t'_3 \leqslant 1 \\ 0 \leqslant t'_4 \leqslant 2 & t'_5 - t'_3 \leqslant 2 \\ 0 \leqslant t'_5 \leqslant 3 & \end{array}$$

The graph of state classes of the net in Figure 1.1 admits 12 classes and 29 transitions, which can be found in [BER 01].

Figure 1.2 shows another Time Petri net, which will be used to compare the various state class graph constructions, together with its SCG graph.



class	$c0$	$c1$	$c2$	$c3$	$c4$
marking	$p0,\ p4$	$p0,\ p5$	$p2,\ p5$	$p3,\ p5$	$p1,\ p5$
firing domain	$5 \leqslant t' \leqslant 7$	$0 \leqslant t0 \leqslant 0$	$2 \leqslant t \leqslant 3$		$0 \leqslant t2 \leqslant 2$
	$3 \leqslant t0 \leqslant 5$	$0 \leqslant t1 \leqslant 0$			
	$3 \leqslant t1 \leqslant 5$				
class	$c5$	$c6$	$c7$	$c8$	
marking	$p2,\ p4$	$p3,\ p4$	$p2,\ p5$	$p1,\ p4$	
firing domain	$2 \leqslant t \leqslant 3$	$0 \leqslant t' \leqslant 2$	$0 \leqslant t \leqslant 3$	$0 \leqslant t' \leqslant 4$	
	$0 \leqslant t' \leqslant 4$			$0 \leqslant t2 \leqslant 2$	

Figure 1.2. *A TPN and its SCG. The variable $\underline{\phi}_t$ is written t*

1.3.3. *Checking the boundedness property on-the-fly*

It remains to examine the conditions under which the set of state classes of a TPN is finite. Let us recall that a Petri net or Time Petri net is bounded if the marking of any place admits an upper bound; boundedness implies finiteness of the set of reachable markings. It was proven in [BER 83] that the set of firing domains of a Time Petri net is finite; its SCG is thus finite if and only if the net is bounded. This property is undecidable for arbitrary TPN (see Theorem 1.1), but sufficient conditions can be applied. The simplest such sufficient condition is that the underlying Petri net is bounded (which is decidable), but weaker conditions can be stated. The following theorem reviews some of them.

THEOREM 1.4.– *[BER 83] A Time Petri net is bounded if its SCG does not contain a pair of classes $c = (m, D)$ and $c' = (m', D')$ such that:*

i) c' is reachable from c,

ii) $m' \ngeqslant m$,

iii) $D' = D$,

iv) $(\forall p)(m'(p) > m(p) \Rightarrow m'(p) \geqslant \max_{(t \in T)}\{\mathbf{Pre}(p,t)\})$.

Properties (i) to (iv) are necessary for the boundedness property, but not sufficient. This theorem, for example, ensures that the nets in Figures 1.1, 1.3 (left) and 1.3 (middle) are bounded, but it cannot be used to prove that the net represented in Figure 1.3 (right) is bounded, even though this net only admits 48 classes. If we omit (iv), then boundedness of the net in Figure 1.3 (middle) cannot be proven anymore. Omitting (iii), in addition, we cannot infer boundedness for the net in Figure 1.3 (left). The condition obtained then ((i) and (ii)) is similar to the boundedness condition for the underlying (untimed) Petri net [KAR 69].

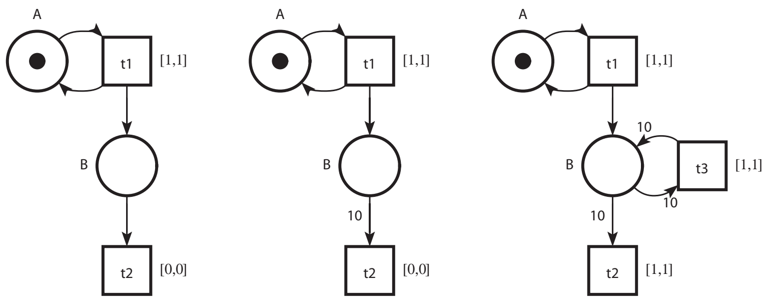


Figure 1.3. *Three bounded Time Petri nets*

1.3.4. *Variations*

1.3.4.1. *Multiple enabledness*

A transition t is multi-enabled at a marking m if there is some integer $k > 1$ such that $m \geqslant k.\mathbf{Pre}(t)$. In the classes of the graph introduced in section 1.3, each enabled transition is associated with exactly one temporal variable, whether or not the corresponding transition is multi-enabled; the various enabling dates for a transition are systematically identified with the largest of these dates.

In some applications, this interpretation may be found to be too restrictive; we may want to distinguish the different enabledness instances. The issue is investigated in [BER 01], where several operational interpretations of multi-enabledness are discussed. We can naturally see the enabledness instances as independent, or as ordered according to their creation dates. In any case, Algorithm 1.1 is easily adapted to produce state class graphs "with multi-enabledness" [BER 01].

1.3.4.2. *Preservation of markings (only)*

It is possible to compact further the state class graph SCG by not storing a class when it is included in an already built class. More precisely, let $L_\sigma = (m, D)$ and $L_{\sigma'} = (m', D')$ be two classes, and $L_\sigma \sqsubseteq L_{\sigma'}$ if and only if $m = m'$ and $D \subseteq D'$. Then, rather than proceeding as in Algorithm 1.1, we build a set C of classes such that, when $L_\sigma \in C$ and $\sigma \cdot t$ is firable, then $(\exists X \in C)(L_{\sigma \cdot t} \sqsubseteq X)$.

Intuitively, if such a class X exists, any schedule firable from a state of $L_{\sigma \cdot t}$ is also firable from a state in X. We will thus not find new markings by storing $L_{\sigma \cdot t}$. Of course, this construction does not preserve the firing sequences, and therefore LTL properties; it only preserves markings, but it typically produces smaller graphs. This construction is convenient when correctness requirements can be reduced to marking reachability properties or to absence of deadlocks.

1.4. State class graphs preserving states and LTL properties

As already mentioned, the classes built using Algorithm 1.1 canonically represent equivalence classes of sets of states by $\cong$, but not the sets C_σ defined in section 1.3.1. A consequence is that the SCG cannot be used to prove or disprove reachability of a particular state of a TPN.

The *strong state classes* (also called *state zones* by some authors) reviewed in this section, introduced in [BER 03b], coincide exactly with these sets of states C_σ. The graph of strong state classes ($SSCG$ for short) preserves LTL properties and states in a way we will make clear.

1.4.1. *Clock domain*

To build the SSCG, it is necessary to represent the sets of states C_σ in a canonical way. An adequate representation is provided by clock domains.

With any firing schedule we can associate a clock function γ as follows: with any transition enabled after firing the schedule, function γ associates the time elapsed since that transition was last enabled. Clock functions can also be seen as vectors $\underline{\gamma}$, indexed by the enabled transitions.

The set of states described by a marking m and a clock system $Q = \{G\underline{\gamma} \leqslant \underline{g}\}$ is the set $\{(m, \phi(\underline{\gamma})) \mid \underline{\gamma} \in Sol(Q)\}$, where $Sol(q)$ is the solution set of system Q, and the firing domain $\phi(\underline{\gamma})$ is the set of solutions in $\underline{\phi}$ of system:

$$\underline{0} \leqslant \underline{\phi}\ ,\ \underline{e} \leqslant \underline{\phi} + \underline{\gamma} \leqslant \underline{l} \ \text{ where } \ \underline{e}_k = Eft_s(k) \text{ and } \underline{l}_k = Lft_s(k).$$

DEFINITION 1.3.– *Let m and m' be two markings and Q and Q' two clock systems. Then let $\equiv$ be the relation such that $(m, Q) \equiv (m', Q')$ holds when the pairs (m, Q) and (m', Q') describe equal sets of states in the above sense.*

In general, distinct clock vectors may describe the same firing domain, and thus distinct clock systems may describe equal sets of states. However, equivalence $\equiv$ is easily established in the following particular case.

THEOREM 1.5.– *[BER 03b] Let $c = (m, Q=\{G\underline{\gamma} \leqslant \underline{g}\})$ and $c' = (m', Q'=\{G'\underline{\gamma}' \leqslant \underline{g}'\})$. If all transitions enabled at m or m' have bounded static firing intervals, then: $c \equiv c' \Leftrightarrow m = m' \wedge Sol(Q) = Sol(Q')$*

The general case will be discussed later. Note that when a transition k is firable at any date beyond its earliest firing time e, all clock vectors whose component k is at least equal to e, and differing only by this component, describe the same state.

1.4.2. *Construction of the* $SSCG$

Strong state classes are represented by pairs (m, Q), where m is a marking and Q a clock domain described by a system of linear inequalities $G\underline{\gamma} \leqslant \underline{g}$. The variables $\underline{\gamma}$ are bijectively associated with the transitions enabled at m.

ALGORITHM 1.2.– *Construction of the $SSCG$, strong state classes*
For any firable sequence σ, let R_σ be the class computed as explained below. Compute the smallest set C of classes including R_ϵ and, whenever $R_\sigma \in C$ and $\sigma \cdot t$ is firable, then $(\exists X \in C)(X \equiv R_{\sigma \cdot t})$.

– The initial class is $R_\epsilon = (m_0, \{0 \leqslant \underline{\gamma}_t \leqslant 0 \mid t \in T \wedge m_0 \geqslant \mathbf{Pre}(t)\})$.

– If σ is firable and $R_\sigma = (m, Q)$, then $\sigma \cdot t$ is firable if and only if:

(i) $m \geqslant \mathbf{Pre}(t)$ (t is enabled at m) and

(ii) the following system is consistent (θ is a new variable):

$$Q \wedge \{0 \leqslant \theta\} \wedge \{Eft_s(t) - \underline{\gamma}_t \leqslant \theta \leqslant Lft_s(i) - \underline{\gamma}_i \mid i \in T \wedge m \geqslant \mathbf{Pre}(i)\}$$

– If $\sigma \cdot t$ is firable, then $R_{\sigma \cdot t} = (m', Q')$ is computed as follows from $R_\sigma = (m, Q)$:

$m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$,

Q' obtained as follows:

(a) a new variable is introduced, θ, constrained by conditions (ii) above;

(b) for each k enabled at m', a new variable $\underline{\gamma}'_k$ is introduced, obeying:

$\underline{\gamma}'_k = \underline{\gamma}_k + \theta$, *if $k \neq t$ and $m - \mathbf{Pre}(t) \geqslant \mathbf{Pre}(k)$,*

$0 \leqslant \underline{\gamma}'_k \leqslant 0$, *otherwise;*

(c) variables $\underline{\gamma}$ and θ are eliminated.

The temporary variable θ describes the possible firing times of transition t from the states in the source class. There is an arc labeled t between classes R_σ and X if and only if $X \equiv R_{\sigma.t}$. Like the systems describing the firing domains in the SCG classes, the clock systems of the $SSCG$ classes are difference systems.

If the net satisfies the restriction introduced in Theorem 1.5 (all static intervals are bounded), then the set of distinct clock systems we can build with Algorithm 1.2 is finite and $\equiv$ is checked in polynomial time by putting clock systems into canonical forms. If this condition is not met then, to ensure termination of the construction, we must add to Algorithm 1.2 a step (d) of relaxation of the state class. A possible implementation of this step is explained in [BER 03b], denoting the set of states C_σ by the largest set of clock vectors describing the same set of states. This largest set is not in general convex, but it can always be described by a finite union of convex sets. In practice, the representation of the classes and relation $\equiv$ need not be changed, but a class will be allowed to have several following classes by the same transition.

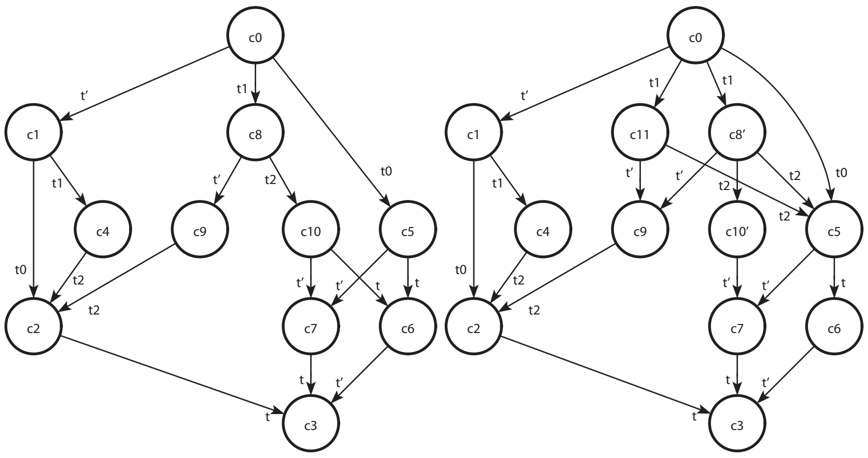
Compared with the SCG, the $SSCG$ also preserves the markings, traces and maximal traces of the net and thus its properties that can be expressed in temporal logic LTL, but, in addition, it makes it possible to check reachability of a state. Assume that we want to check if state $s = (m, I)$ is reachable. Then, it is always possible to compute a clock vector $\underline{\Gamma}$ such that, for any transition k enabled by m, we have $I(k) = I_s(k) \dot{-} \underline{\Gamma}_k$. The properties of the $SSCG$ ensure that s is reachable if and only if there is a strong class (m', q) in the $SSCG$ of the net such that $m = m'$ and $\underline{\Gamma} \in Q$.

The $SSCG$ of the net represented in Figure 1.2 admits 11 strong state classes and 16 transitions; it is represented in Figure 1.4 (left). By comparing it with the SCG of this net, represented in Figure 1.2, the reader will notice that the sets of states $C_{t'.t_1}$ and $C_{t_1.t'}$ are distinguished in the $SSCG$ (represented by the classes c_4 and c_9, respectively), whereas they were equivalent by $\cong$ in the SCG. The same applies for the sets $C_{t_1.t_2}$ (c_10) and C_{t_0} (c_5).

1.4.3. *Variants*

As for Algorithm 1.1, computing the SCG, the verification of sufficient conditions for the boundedness property can easily be added to Algorithm 1.2.

Finally, as for the SCG again (see section 1.3.4.2), and as proposed in [BOU 04], a typically smaller variant of the $SSCG$ preserving only the states of the net but not its firing sequences can be built, by not storing a class included in an already built class.



class	$c0$	$c1$	$c2$	$c3$
marking	$p0,\ p4$	$p0,\ p5$	$p2,\ p5$	$p3,\ p5$
clock domain	$0 \leqslant t' \leqslant 0$ $0 \leqslant t0 \leqslant 0$ $0 \leqslant t1 \leqslant 0$	$5 \leqslant t0 \leqslant 5$ $5 \leqslant t1 \leqslant 5$	$0 \leqslant t \leqslant 0$	
class	$c4$	$c5$	$c6$	$c7$
marking	$p1,\ p5$	$p2,\ p4$	$p3,\ p4$	$p2,\ p5$
clock domain	$0 \leqslant t2 \leqslant 0$	$0 \leqslant t \leqslant 0$ $3 \leqslant t' \leqslant 5$	$5 \leqslant t' \leqslant 7$	$0 \leqslant t \leqslant 3$
class	$c8$	$c9$	$c10$	$c8'$
marking	$p1,\ p4$	$p1,\ p5$	$p2,\ p4$	$p1,\ p4$
clock domain	$3 \leqslant t' \leqslant 5$ $0 \leqslant t2 \leqslant 0$	$0 \leqslant t2 \leqslant 2$	$0 \leqslant t \leqslant 0$ $3 \leqslant t' \leqslant 7$	$3 < t' \leqslant 5$ $0 \leqslant t2 \leqslant 0$
class	$c10$	$c11$		
marking	$p2,\ p4$	$p1,\ p4$		
clock domain	$0 \leqslant t \leqslant 0$ $5 < t' \leqslant 7$	$3 \leqslant t' \leqslant 3$ $0 \leqslant t2 \leqslant 0$		

Figure 1.4. *SSCG and ASCG of the net in Figure 1.2.*
The variable $\underline{\gamma}_t$ *is written t*

1.5. State class graphs preserving states and branching properties

The branching properties are those expressed in branching-time temporal logics like CTL, or in modal logics like HML or the μ-calculus. Neither the SCG nor the $SSCG$ preserve these properties of the state graph. Let us consider the net represented in Figure 1.2 with its graph of classes SCG. An example of HML formula whose truth value is not preserved by this graph is $<t_1>[t_2]<t>\mathbf{T}$, expressing that, via t_1, it is possible to reach a state from which firing t_2 leads to a state from which t may fire. This formula is true on the SCG, but false on the state graph of the net: indeed, after firing the schedule $(t1@5.t2@2)$, the transitions t' and t are both enabled, but only t' is firable, at time 0.

In the absence of silent transitions, branching properties are captured by the concept of bisimulation; any graph of classes bisimilar with the state graph (omitting temporal annotations) preserves its branching properties. The first construction for such a graph of classes was proposed in [YON 98], for the subclass of $TPNs$ in which the static intervals of all transitions are bounded. A specialization of this construction, preserving only the formulae of logic $ACTL$, was proposed in [PEN 01]. Rather than these constructions, we will recall that of [BER 03b], which is applicable to any TPN and generally yields smaller graphs.

The standard technique for computing bisimulations is that of the "minimal partition refinement" introduced in [PAI 87]. Let $\rightarrow$ be a binary relation on a finite set U. For any $S \subseteq U$, let $S^{-1} = \{x \mid (\exists y \in S)(x \rightarrow y)\}$. A partition P of U is stable if, for any pair (a, b) of blocks of P, either $A \subseteq B^{-1}$ or $A \cap B^{-1} = \emptyset$ (A is said to be stable with respect to B). Computing a bisimulation from an initial partition P is computing a stable refinement Q of P [KAN 90]. An algorithm is as follows [PAI 87]:

```
Initially: $Q = P$
While: there are blocks $A, B \in Q$ such that $\emptyset \not\subseteq A \cap B^{-1} \not\subseteq A$
Do: replace $A$ by $A_1 = A \cap B^{-1}$ and $A_2 = A - B^{-1}$ in $Q$.
```

Our context is significantly different from that intended above because our sets of states are not finite, but the method can be adapted. The strong classes of the $SSCG$ are an adequate initial partition (contrary to the classes of SCG, since equivalence $\cong$ does not preserve atomicity). The fact that this set is a cover rather than a partition implies however that the final result will be generally non-minimal in number of blocks, and non-unique.

Following [YON 98], let us call *atomic* a class stable with respect to all its following classes, i.e. such that each state of the class has a successor in each of the following classes. The graph of atomic state classes of a net (or $ASCG$) will be obtained by refinement of its $SSCG$. The partition technique is explained in detail in [BER 03b]. Intuitively, for each transition of the current graph, we compute from the target class

c' the set of (possible clocks of) states having a successor in c'. The intersection of this set of states with those captured in the source class c defines a partition of c. The atomic state classes are considered modulo equivalence $\equiv$, like strong state classes. If the static interval of at least one transition is not bounded, then the $ASCG$ must be built taking as initial cover the "relaxed" $SSCG$ (see section 1.4.2).

For example, the $ASCG$ of the net represented in Figure 1.2 admits 12 atomic state classes and 19 transitions, as represented in Figure 1.4. The only non-atomic class of its $SSCG$, $c10$, was initially split into class c'_{10} and a class equivalent by $\equiv$ to c_5. The class c_8 of the $SSCG$ then became non-atomic, and was in turn split into c'_8 and c_{11}. It can be checked on the $ASCG$ that it preserves the truth value of our HML formula example: $<t_1>[t_2]<t>\mathbf{T}$.

1.6. Computing firing schedules

1.6.1. *Schedule systems*

For each firable sequence σ, the possible dates at which the transitions in the sequence may fire can be characterized by a system of linear inequalities $P\underline{\theta} \leqslant \underline{p}$ in which the i^{th} component of vector $\underline{\theta}$ describes the dates on which the i^{th} of θ may fire.

To compute such systems, we proceed as for the calculation of strong state classes by Algorithm 1.2, but omitting elimination of the auxiliary variables θ. This method is summarized by Algorithm 1.3 hereafter.

ALGORITHM 1.3.– *Characterizing all firing schedules of support* σ
Let us note σ^i *the prefix of length* i *of sequence* σ*, and* σ_i *the* i^{th} *transition of* σ.

We compute by the method below a series of pairs Z^i*, for* $i \in \{0, \dots, k\}$*, consisting of a marking and a system of inequalities of the form* $P(\underline{\theta} \mid \underline{\gamma}) \leqslant \underline{p}$*. Intuitively, if* $Z^i = (m, K)$*, then* m *is the marking reached after firing* σ^i *and system* K *simultaneously characterizes the possible firing times of the transitions along* σ^i *and the clock values of the transitions enabled at* m.

The required system is that obtained by eliminating the variables from clock $\underline{\gamma}$ *in the system of inequalities of the last pair computed,* Z^k.

– Initially, $Z^0 = (m_0, \{0 \leqslant \underline{\gamma}_t \leqslant 0 \mid t \in T \wedge m_0 \geqslant \mathbf{Pre}(t)\})$*. This pair has the desired form, even though no "path variable" appears in the inequality system.*

– Let $t = \sigma_i$ *and* $Z^i = (m, K)$*. If* σ^i *is firable, then* $\sigma^i \cdot t$ *is firable if and only if:*

(i) $m \geqslant \mathbf{Pre}(t)$ *(*t *is enabled at* m*) and,*

(ii) the system below is consistent (θ *is a new "path variable"):*

$$K \wedge \{0 \leqslant \theta\} \wedge \{Eft_s(t) - \underline{\gamma}_t \leqslant \theta \leqslant Lft_s(i) - \underline{\gamma}_i \mid i \in T \wedge m \geqslant \mathbf{Pre}(i)\}$$

– *The pair* $Z^{i+1} = (m', K')$ *is computed as follows from the pair* $Z^i = (m, K)$*:*

$m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$,

K' *obtained by:*

(a) introducing a new variable, θ*, constrained as in (ii) above;*

(b) for each j *enabled at* m'*, a new variable* $\underline{\gamma}'_j$ *is introduced, obeying:*

$\underline{\gamma}'_j = \underline{\gamma}_j + \theta$, *if* $i \neq t$ *and* $m - \mathbf{Pre}(t) \geqslant \mathbf{Pre}(j)$,

$0 \leqslant \underline{\gamma}'_j \leqslant 0$, *otherwise;*

(c) eliminating variables $\underline{\gamma}$*.*

The exact implementation of Algorithm 1.3 will not be detailed here. Several optimizations are essential for good performances over long sequences. In particular, these systems are often constituted by independent subsystems, a property that we should exploit. Alternatively, rather than by the algorithm above, schedule systems can be built from the sequence of state classes or strong state classes (without relaxation) built for sequence σ.

These systems allows us to solve certain "quantitative" problems expressed in terms of firing schedules, particularly that of the fastest or slowest schedule over a given support sequence, or the existence of schedules of given support satisfying particular conditions about the firing times of the transitions.

The systems $P\underline{\theta} \leqslant \underline{p}$ produced for σ by Algorithm 1.3 have a particular structure: each inequality has one of the following forms, in which a and b are integer constants and I is an interval of integers in $\{1, \ldots, |\sigma|\}$. Note that the variables implied in the sums necessarily relate contiguous transitions of σ:

$$\begin{array}{ll} a \leqslant \theta & \theta \leqslant b \\ a \leqslant \sum_{i \in I}(\theta_i) & \sum_{i \in I}(\theta_i) \leqslant b. \end{array}$$

1.6.2. *Delays (relative dates) versus dates (absolute)*

The systems built by Algorithm 1.3 characterize relative firing dates (delays). It is possible to reformulate this algorithm in order to compute systems in "absolute" rather than relative firing times (dates). In this case, we will introduce an additional variable, $start$, which represents the date of initialization of the net.

The firing date of the i^{th} transition of σ is the initial date $start$ increased by the relative firing dates of the i first transitions of σ. We thus pass easily from a system "in delays" (θ_i) to a system "in dates" (ψ_i), or conversely, by the following transformation (all variables being non-negative):

$\psi_i - \psi_{i-1} = \theta_i$ (assuming $\psi_0 = start$).

Note that, applied to a system "in delays", and taking into account the general form of these systems, this transformation produces a system of differences.

1.6.3. *Illustration*

As an illustration, any schedule of support $t1.t2.t.t'$ in the net of Figure 1.2 has the form $t1@\theta_0.t2@\theta_1.t@\theta_2.t'@\theta_3$, the variables θ_i obeying the following inequalities:

$$\begin{array}{l}
3 \leqslant \theta_1 \leqslant 5 \\
3 \leqslant \theta_1 + \theta_2 \leqslant 7 \\
0 \leqslant \theta_2 \leqslant 2 \\
5 \leqslant \theta_1 + \theta_2 + \theta_3 \leqslant 7 \\
2 \leqslant \theta_3 \leqslant 3 \\
5 \leqslant \theta_1 + \theta_2 + \theta_3 + \theta_4 \leqslant 7 \\
0 \leqslant \theta_4.
\end{array}$$

Expressed as dates, rather than delays, we would obtain the following system, in which $start$ is the date of initialization of the net, and ψ_i the date on which the i^{th} transition from σ fires:

$$\begin{array}{l}
3 \leqslant \psi_1 - start \leqslant 5 \\
3 \leqslant \psi_2 - start \leqslant 7 \\
0 \leqslant \psi_2 - \psi_1 \leqslant 2 \\
5 \leqslant \psi_3 - start \leqslant 7 \\
2 \leqslant \psi_3 - \psi_2 \leqslant 3 \\
5 \leqslant \psi_4 - start \leqslant 7 \\
0 \leqslant \psi_4 - \psi_3.
\end{array}$$

1.7. An implementation: the `Tina` environment

`Tina` (*TIme Petri Net Analyzer*)[1] is a software environment allowing the manipulation and analysis of Petri nets and Time Petri nets. The various tools constituting the environment can be combined or used independently. These tools include the following:

`nd` (*NetDraw*): `nd` is a tool for editing Time Petri nets and automata, in graphic or textual form. It also integrates a "stepper" simulator operating on graphic or textual representations for Time Petri nets and allows us to invoke the tools described below without leaving the editor.

1. http://www.laas.fr/tina.

`tina`: this tool builds state space abstractions for Petri nets or Time Petri nets. For Petri nets (untimed), `tina` provides the traditional constructions (marking graphs, covering graphs) but also some abstract state space constructions based on partial order techniques such as stubborn sets and covering steps. These constructions (as described in [BER 04]) preserve certain classes of properties, such as the absence of deadlocks, the properties of certain logics, or test equivalence. For Time Petri nets, `tina` offers all of the constructions of state class graphs discussed in this chapter. These graphs can be produced in various formats: in verbal form (for teaching purposes), in a compact interchange format targeted at the other tools in the environment, or in formats accepted by some external model checkers such as MEC [ARN 94], for the verification of formulae of the μ-calculus, or the tools $CADP$ [FER 96] for the verification of preorders or equivalences of behaviors.

When building these abstractions, `tina` can check on-the-fly some "general" properties such as boundedness (see section 1.3.3), the presence or absence of deadlocks – expected in some applications (which must terminate) but undesired in some others (e.g. reactive systems) – the pseudo-liveness property (absence of transitions that never fire), allowing us to detect "dead code", or the liveness property that ensures that all transitions (or a system state) remains reachable from any state of the system.

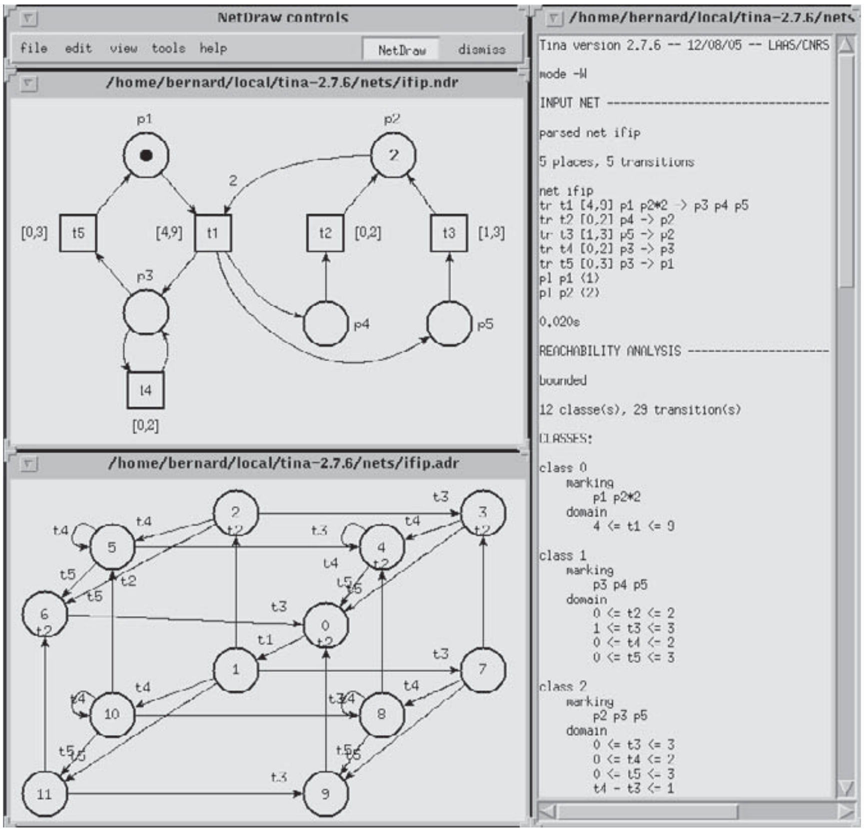
`plan`: this tool implements the computation of schedule systems, as explained in section 1.6. It produces on request either a characterization of all schedules over a sequence (as an inequality system), or an arbitrary solution of this system, in delays or dates, put into canonical form or not. It is also able to compute the fastest and slowest firing schedules over a sequence.

`struct`: this is a tool for structural analysis, not described here. It computes generating sets for the flows or semi-flows of the net, over the places or transitions.

`selt`: in addition to the general properties checked by `tina`, it is generally essential to assert specific properties related to the modeled system. The tool `selt` is a model-checker for the formulae of an extension of temporal logic $SE-LTL$ (State/Event LTL) of [CHA 04]. It operates on the state space abstractions produced by `tina`, or, through a conversion tool, on the behavior graphs produced by other tools such as the $CADP$ tools [FER 96]. This tool will be described at length in section 1.8.2, together with some examples of use.

`ndrio`, `ktzio`: these are format conversion tools for nets (`ndrio`) and transition systems (`ktzio`).

These various tools can cooperate through files in documented interchange formats. A screen capture of a `Tina` session is reproduced in Figure 1.5, with a Time Petri net being edited, a state class graph in textual format and its graphical representation.


NetDraw controls
file edit view tools help
NetDraw dismiss
/home/bernard/local/tina-2.7.6/nets/ifip.ndr
p1
p2
2
[0,3] t5
[4,9] t1
t2 [0,2]
t3 [1,3]
p3
p4
p5
t4
[0,2]
/home/bernard/local/tina-2.7.6/nets/ifip.adr
/home/bernard/local/tina-2.7.6/nets

```
Tina version 2.7.6 -- 12/08/05 -- LAAS/CNRS

mode -W

INPUT NET ---------------------------------

parsed net ifip

5 places, 5 transitions

net ifip
tr t1 [4,9] p1 p2*2 -> p3 p4 p5
tr t2 [0,2] p4 -> p2
tr t3 [1,3] p5 -> p2
tr t4 [0,2] p3 -> p3
tr t5 [0,3] p3 -> p1
pl p1 (1)
pl p2 (2)

0.020s

REACHABILITY ANALYSIS --------------------

bounded

12 classe(s), 29 transition(s)

CLASSES:

class 0
    marking
        p1 p2*2
    domain
        4 <= t1 <= 9

class 1
    marking
        p3 p4 p5
    domain
        0 <= t2 <= 2
        1 <= t3 <= 3
        0 <= t4 <= 2
        0 <= t5 <= 3

class 2
    marking
        p2 p3 p5
    domain
        0 <= t3 <= 3
        0 <= t4 <= 2
        0 <= t5 <= 3
        t4 - t3 <= 1
```

Figure 1.5. *Screen capture of a* Tina *session*

1.8. The verification of $SE\!-\!LTL$ formulae in `Tina`

This section focuses on the verification of specific properties of nets and presents the techniques and tools available in `Tina` to carry out this task. The verification of specific properties first requires a formal language in which the correctness properties are stated. We will use for that purpose an extension of the linear-time temporal logic $SE\!-\!LTL$. Next, we need a description or abstraction of the behavior of the net preserving the properties we can express in that logic, the SCG and $SSCG$ constructions provided by `tina` are adequate for this purpose. Finally, we need a tool to evaluate the truth values of the formulae expressing correctness on the behavior abstraction produced, this is the task of the `selt` module.

1.8.1. *The temporal logic* $SE\!-\!LTL$

The LTL logic extends propositional calculus by specific expressions about the execution sequences of a system. $SE\!-\!LTL$ is a recent variant of LTL [CHA 04], that treats in a uniform way both state properties and transition properties. $SE{-}LTL$ formulae are interpreted over labeled Kripke structures, also called Kripke transition systems.

DEFINITION 1.4.– *Kripke transition systems*

A Kripke transition system is a tuple $KTS = \langle S, Init, AP, \nu, T, \Sigma, \epsilon\rangle$ *in which:*

– S *is a finite set of states;*

– $Init \subset S$ *is the set of initial states;*

– AP *is a finite set of atomic state propositions;*

– $\nu : S \longrightarrow 2^{AP}$ *is a labeling of states with state propositions;*

– $T \subseteq S \times S$ *is a finite set of transitions;*

– Σ *is a finite set of* events *or* actions*;*

– $\epsilon : T \longrightarrow 2^{\Sigma}$ *is a labeling of transitions with actions.*

$s \overset{E}{\rightarrow} q$ stands for $(s,q) \in T \wedge \epsilon(s,q) = E$. A run, or execution, of a KTS is an infinite sequence $\pi = \langle s_1, a_1, s_2, a_2, \ldots\rangle$ alternating states and transitions such that $(\forall i \geqslant 1)(s_i \in S \wedge a_i \in \Sigma \wedge s_i \overset{a_i}{\rightarrow} s_{i+1})$. The pair (s_i, a_i) will be called step i. Classically, the reachability relation is assumed to be total: all states admit some successor. $\Pi(KTS)$ will denote the set of all runs having their origin in one of the states $s \in Init$. Finally, for a run $\pi = \langle s_1, a_1, s_2, a_2, \ldots\rangle$, π^i denotes the suffix of the run π starting at state s_i.

DEFINITION 1.5.– $SE{-}LTL$

$SE{-}LTL$ *logic is defined over the sets* AP *and* Σ *of a Kripke transition system.* p *denotes a variable of* AP *and* a *a variable of* Σ*.*

The formulae Φ *of* $SE{-}LTL$ *obey the following rules:*

$$\Phi ::= p \mid a \mid \neg\Phi \mid \Phi \vee \Phi \mid \bigcirc\, \Phi \mid \Box\, \Phi \mid \Diamond\, \Phi \mid \Phi\, U\, \Phi.$$

Their semantics is inductively defined, as follows:

$KTS \models \Phi$ *iff* $\pi \models \Phi$ *for any run* $\pi \in \Pi(KTS)$

$\pi \models p$ *iff* $p \in \nu(s_1)$ *(where* s_1 *is the first state of* π*)*

$\pi \models a$ *iff* $a \in \epsilon(a_1)$ *(where* a_1 *is the first transition of* π*)*

$\pi \models \neg\Phi$ *iff* $\pi \not\models \Phi$

$\pi \models \bigcirc\, \Phi$ *iff* $\pi^2 \models \Phi$

$\pi \models \Box\, \Phi$ *iff* $(\forall i \geqslant 1)(\pi^i \models \Phi)$

$\pi \models \Diamond\, \Phi$ *iff* $(\exists i \geqslant 1)(\pi^i \models \Phi)$

$\pi \models \Phi_1\, U\, \Phi_2$ *iff* $(\exists i \geqslant 1)(\pi^i \models \Phi_2 \wedge (\forall 1 \leqslant j \leqslant i-1)(\pi^j \models \Phi_1))$

EXAMPLE 1.1.– *Some example formulae of $SE-LTL$:*

	In any run:
P	*P true at the start (at the initial step),*
$\bigcirc P$	*P true at the next step,*
$\Box P$	*P true all along the run,*
$\Diamond P$	*P true at some step along the run,*
$P \; U \; Q$	*Q true at some step and P true until then,*
$\Box \Diamond P$	*P true infinitely often,*
$\Box (P \Rightarrow \Diamond Q)$	*Q follows P.*

1.8.2. *Preservation of LTL properties by* `tina` *constructions*

The graphs of state classes described in sections 1.3 to 1.5 preserve the markings, traces and maximum traces of the state graph of the nets, and thus the truth values of all $SE–LTL$ formulae when taking AP as the presence of a token in a place and Σ as the set of transitions of the net. The Kripke structures associated with these graphs are these graphs themselves with their nodes labeled by the place marked and their edges labeled by the transition fired.

Since the static temporal intervals associated with the transitions of a Time Petri net are not necessarily bounded, it is possible that control forever remains in certain states of the Kripke structure, even though no loop on this node materializes this arbitrary latency. This information can be retrieved from the temporal information captured by the corresponding state classes, however. When building state space abstractions with `tina`, the user has the choice of materializing these arbitrary latencies or not. If they are taken into account, a loop on the relevant nodes of the graph will be added, as well as a specific *div* property (for temporal divergence).

In addition, the reachability relation in Kripke structures being supposedly total, a loop should also be added to all nodes not having any successor; `tina` does not add these loops, but the tool checking the formulae does it transparently when loading a state space abstraction, and adds to those nodes a specific *dead* property (for deadlock).

1.8.3. `selt`*: the $SE-LTL$ checker of* `Tina`

1.8.3.1. *Verification technique*

In this section, we describe the main functions of the `selt` module, the model-checker for the $SE-LTL$ logic part of the `Tina` toolbox. `selt` makes it possible to check $SE-LTL$ formulae on the Kripke transition system obtained as explained above from a graph of state classes. Traditionally, checking a formula is done in three steps (in practice, the last is performed on-the-fly):

1) build a Büchi automaton accepting the words which do not satisfy the $SE–LTL$ formula to be checked; this phase is carried out in a transparent way for the user by invoking `ltl2ba`, a tool developed at LIAFA by Paul Gastin and Denis Oddoux [GAS 01];

2) build the synchronized product of the Kripke transition system obtained from the state class graph and the Büchi automaton obtained from the negation of the formula. The states of this product automaton are pairs capturing a state of the Kripke transition system and a state of the Büchi automaton;

3) find in this product automaton a strongly connected component including an accepting state of the Büchi automaton. If no such component is found, then the formula is satisfied, otherwise the infinite sequence containing that state defines a counter-example.

In the event of non-satisfaction of a formula, `selt` can provide a sequence counter-example, either in clear or under a format loadable by the stepper simulator of `Tina`, so that it can be replayed. Note that the counter-example sequence obtained by the above procedure is untimed. When the input model is timed, it is thus necessary to compute from it a timed counter-example; this can be done by the module `plan` of `Tina` described in section 1.6.

In some cases, in particular for timed systems, the counter-example sequence of a formula can be very long and thus not easily exploitable by the user. For this reason, `selt` may optionally produce compacted counter-examples, in a symbolic form. The counter-example is then presented as a sequence of symbolically defined sequences (each printed as a single transition), materializing only the essential state changes (in practice those corresponding to actual state changes in the Büchi automaton).

1.8.3.2. *The* `selt` *logic*

The Kripke transition systems referred to here are obtained from Petri nets or Time Petri nets. In this context, AP corresponds to the set P of the places of the network and σ to the set T of its transitions. The interpretation of $SE-LTL$ formulae given in Definition 1.5 does not then distinguish two states that mark exactly the same place, independently of the actual token counts. If this approach is exact in "safe" net (i.e. when places are marked with one token at most), it in general incurs a significant loss of information.

For this reason, `selt` supports an enrichment of $SE-LTL$, interpreted on enriched Kripke transition systems taking into account the actual token counts in markings. The only difference with Definition 1.4 is the replacement of the ν and ϵ applications by a ν' application that associates with each state S a multiset on AP (i.e. an application of $AP \mapsto \mathbb{N}$, or a marking, in our context) and an application ϵ' that associates with each transition of T a multiset builds on σ. Considering transition properties as multisets is not necessary in the examples we will discuss in this chapter, but it is useful for other `tina` constructions not discussed here, such as the *covering steps* construction in which several transitions of the net may fire simultaneously.

An enrichment of the language of formulae corresponds to the enrichment of Kripke transition systems. To exploit the added information, the Boolean propositional calculus parametrizing the logic will be replaced by a multi-valued calculus, and the query language of `selt` is extended with logico-arithmetic operators.

For $p \in AP$, $a \in \Sigma$, $c \in \mathbb{N}$, the formulae ϕ, propositions r, and expressions e of `selt` obeys the following rules:

$$\begin{array}{ll}
\Phi ::= r \mid \neg\Phi \mid \Phi \vee \Phi \mid \bigcirc\, \Phi \mid \Box\, \Phi \mid \Diamond\, \Phi \mid \Phi\, U\, \Phi & \\
r ::= e \mid e\Delta e & (\Delta \in \{=, <, \leqslant, >, \geqslant\}) \\
e ::= p \mid a \mid c \mid e\nabla e & (\nabla \in \{+, -, *, /\})
\end{array}$$

For any run π, the interpretation $\lceil r \rceil(\pi)$ of a proposition r and the integer value $\langle e \rangle(\pi)$ of an expression e are defined as follows:

$$\begin{array}{rcl}
\lceil e \rceil(\pi) & = & \langle e \rangle(\pi) \geqslant 1, \\
\lceil r_1 \Delta r_2 \rceil(\pi) & = & \lceil r_1 \rceil(\pi) \Delta \lceil r_2 \rceil(\pi), \\
\langle c \rangle(\pi) & = & c, \\
\langle p \rangle(\pi) & = & \nu'(s)(p), \text{ where } s \text{ is the first state of } \pi, \\
\langle a \rangle(\pi) & = & \epsilon'(t)(p), \text{ where } t \text{ is the first transition of } \pi, \\
\langle e_1 \nabla e_2 \rangle(\pi) & = & \langle e_1 \rangle(\pi) \nabla \langle e_2 \rangle(\phi).
\end{array}$$

The semantic of formulae is defined as with $SE{-}LTL$ (see Definition 1.5), except for the rules relative to p and a, replaced by:

$$\pi \models e \text{ iff } \lceil e \rceil(\pi).$$

Finally, `selt` allows the user to declare derived operators abbreviating some patterns of existing operators. A small number of commands are also provided to control the printing format of results or loading formula libraries. The reader is referred to the `selt` manual for details.

EXAMPLE 1.2.– *Some* `selt` *formulae concerning the net in Figure 1.1*

$t_1 \wedge p_2 \geqslant 2$	*initially* $m_0(p_2) \geqslant 2$ *and any run starts with* t_1*,*
$\Box\ (p_2 + p_4 + p_5 = 2)$	*a linear marking invariant,*
$\Box\ (p_2 * p_4 * p_5 = 0)$	*a non-linear marking invariant,*
$infix\ q\ R\ p = \Box\ (p \Rightarrow \Diamond\ q)$	*declares operator "follows", written R,*
$t_1\ R\ t_5$	t_1 *follows* t_5.

1.9. Some examples of use of `selt`

1.9.1. *John and Fred*

1.9.1.1. *Statement of problem*

This example is borrowed from [DEC 91], in the field of artificial intelligence and constraint satisfaction. The problem is stated as follows:

– John goes to work using his car (which takes him between 30 to 40 minutes) or the bus (taking him at least 60 minutes),

– Fred goes to work using his car (which takes him between 20 to 30 minutes) or a carpool (taking him 40 to 50 minutes),

– today John left home between 7:10 and 7:20 and Fred arrived at work between 8:00 and 8:10. In addition, John arrived at work between 10 to 20 minutes after Fred left home.

Three questions must be answered:

1) Are the temporal constraints in this scenario consistent?

2) Is it possible that Fred took the bus and John the carpool?

3) At which time could Fred have left home?

A Time Petri net model for this problem is shown in Figure 1.6. The subnets drawn in black correspond to the usual behaviors of John and Fred, respectively. The gray subnets are "observers" materializing the temporal constraints specific to the particular scenario. The left observer materializes the time of departure of John and hour of arrival of Fred. The relative constraint stipulating that "John arrived between 10 and 20 minutes after Fred left home" is materialized by the second observer.

1.9.1.2. *Are the temporal constraints appearing in this scenario consistent?*

The SCG of this net, computed by `tina` using Algorithm 1.1, admits 3,676 classes and 7,578 transitions. As this graph is without circuits, the answer is positive if there exists some run satisfying the expressed constraints such that John and Fred eventually arrive at work or, in other words, satisfying the following formula ϕ_1:

$$\begin{array}{rll}
\phi_1 = & \Diamond\,(john\_left \wedge 7h10\_7h20) & \text{John left between 7:10 and 7:20} \\
\wedge & \Diamond\,(fred\_arrives \wedge 8h00\_8h10) & \text{Fred arrived between 8:00 and 8:10} \\
\wedge & \Diamond\,(john\_arrives \wedge 10\_20) & \text{John arrived 10 to 20 minutes after Fred left} \\
\wedge & \Diamond\,(john\_working \wedge fred\_working) &
\end{array}$$

Since LTL formulae are interpreted over all runs, we will check with `selt` the negation $\neg\phi_1$ of the formula. The answer is negative: not all executions satisfy $\neg\phi_1$,

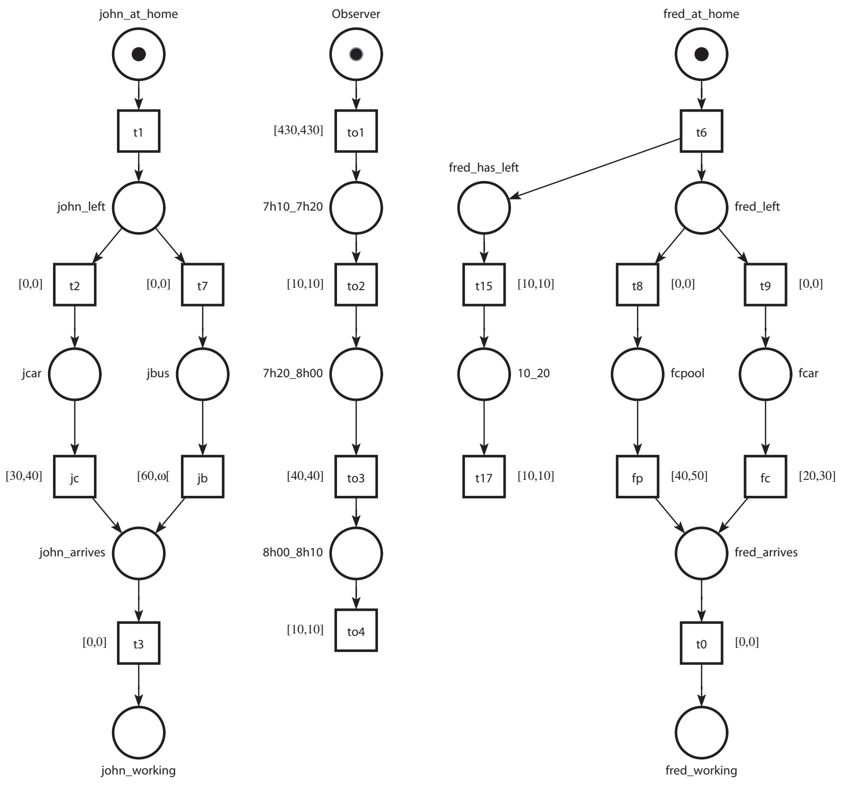


Figure 1.6. *A Time Petri net model for the John and Fred problem*

therefore one at least satisfies its negation ϕ_1. `selt` provides simultaneously a counter-example of $\neg\phi_1$, that is a sequence showing that the scenario is consistent. The tool `plan` discussed in section 1.6 allows us to compute all schedules, or just one, having this sequence as support. The consistent scenario found by `selt`, in which John used his car and Fred the carpool, is the following:

```
        state 0: Observer fred_at_home john_at_home
-t1->   state 1: Observer fred_at_home john_left
-to1->  state 1694: fred_at_home john_left 7h10_7h20
-t2->   state 1883: fred_at_home jcar 7h10_7h20
-t6->   state 1884: fred_has_left fred_left jcar 7h10_7h20
-t8->   state 1885: fcpool fred_has_left jcar 7h10_7h20
-to2->  state 1886: fcpool fred_has_left jcar 7h20_8h00
-t15->  state 1887: 10_20 fcpool jcar 7h20_8h00
-jc->   state 1888: 10_20 fcpool john_arrives 7h20_8h00
-t17->  state 1901: fcpool john_arrives 7h20_8h00
```

```
-t3->  state 1902: fcpool john_working 7h20_8h00
-to3-> state 1903: fcpool john_working 8h00_8h10
-fp->  state 1904: fred_arrives john_working 8h00_8h10
-t0->  state 1909: fred_working john_working 8h00_8h10
-to4-> state 1910: dead fred_working john_working
```

1.9.1.3. *Is it possible that Fred took the bus and John the carpool?*

To answer this question, we will proceed in the same way as previously, except for adding to formula ϕ_1 the requirement that Fred traveled by bus and John took the carpool. So, such a scenario is possible if an execution at least satisfies $\phi_2 = \phi_1 \wedge (\Diamond\, jbus \wedge \Diamond\, fcpool)$.

`selt` asserts that the formula $\neg\phi_2$ is true and therefore no run satisfies ϕ_2. It is thus not possible to satisfy the temporal constraints when Fred takes the bus and John the carpool.

1.9.1.4. *At which time could Fred have left home?*

In contrast to the previous questions, this problem cannot simply be given a yes/no answer, as we expect a time range for the departure of Fred.

It could easily be checked if the scenario was consistent assuming Fred left home in some given time range, by associating this time range with transition t_6 of the net (materializing the departure of Fred) and applying the previous method. However, this does not answer the question. The actual question does not reduce to a verification problem, but it is indeed a synthesis problem.

The `plan` tool only partially addresses this synthesis problem. Given a firable firing sequence including transition t_6 (departure of Fred) and satisfying the temporal constraints, `plan` can determine the possible times at which t_6 in this sequence can fire. This can be done for any sequence, but only one at a time. So, to have all possible times at which t_6 may fire, it is necessary to perform the synthesis of the possible firing times of t_6 for each sequence satisfying the time constraint; the required result is then the union of the ranges computed. These sequences could be obtained *via* `selt`, by building the synchronized graph in full, thus including the sequences of all consistent scenarios.

1.9.2. *The alternating bit protocol*

Communications protocols make a broad use of temporal constraints: the reconfiguration mechanisms after message losses, for example, are typically implemented using time outs. The alternating bit protocol is certainly the simplest of these protocols. It is a stop and wait data transfer protocol: before sending a new message, the sender waits for the acknowledgment of the last message it sent.

Hypotheses on the behavior of the transmission medium are that messages or their acknowledgments may be lost or damaged while in transit (in this last case, they are simply rejected). To recover from these losses, a time-out is set when a message is sent and if its acknowledgment does not arrive in time the message is retransmitted. This basic mechanism is sufficient for recovering from losses of messages but does not prevent duplication-free reception: if an acknowledgment is lost, the receiver is unable to decide whether the next message it receives is a new message or another copy of the last message it received. To solve this problem, messages are numbered prior to transmission with modulo-2 sequence numbers and acknowledgments refer explicitly to these numbers, hence the name of the protocol.

Figure 1.7 shows a TPN model for the alternating bit protocol. For simplicity, damaged messages are considered as lost. Notice that the loss of messages and acknowledgments is simply represented with transitions that have no output places; there is no need here for any artificial mechanism relating the loss of messages to retransmisions. Estimates for the durations of all elementary actions of the protocols must be provided. Retransmission of a message occurs at a time comprised between 5 and 6 units after the last copy of the message has been sent. Equal estimates (between 0 and 1) are given for losses and receptions of messages and acknowledgments. No constraints, i.e. intervals $[0, \infty[$, are given for transmission of the first copies of the messages.

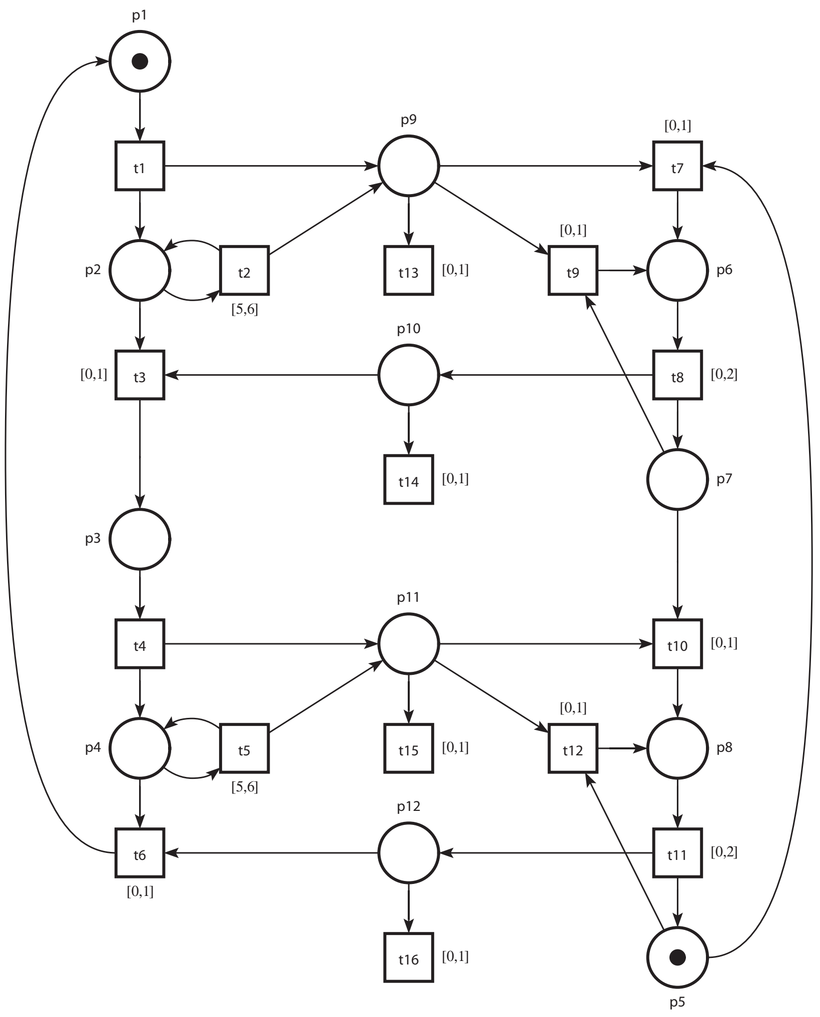
This TPN is bounded. Its graph of state classes, built by `tina` using Algorithm 1.1, admits 16 state classes.

Consider now the following properties, expressed in the language of `selt`:

$$
\begin{aligned}
&\phi_1 = \Box\,(p_9 + p_{10} + p_{11} + p_{12} \leqslant 1)\\
&\phi_2 = t_1\\
&\phi_3 = \Box\,(t_1 \Rightarrow \Diamond\, t_7)\\
&\phi_4 = \Box\,(t_1 \Rightarrow \Diamond\,(t_{13} \vee t_7))\\
&\phi_5 = -\Box\,\Diamond\,(t_{13} \vee t_{14}) \Rightarrow \Box\,(t_1 \Rightarrow \Diamond\, t_3)\\
&\phi_6 = \Box\; -dead
\end{aligned}
$$

ϕ_1 expresses that, at any time, at most one message or acknowledgment is in transit. This guarantees that the delay chosen for retransmission of a message is correct (long enough). ϕ_1 is satisfied over the SCG of the net.

ϕ_2 expresses that any execution starts with transition t_1. It is proven false: indeed, since it is not temporally constrained, t_1 can be delayed indefinitely and the initial state is temporally divergent. As already mentioned, the tool `tina` leaves to the user the choice to interpret the transitions with unbounded firing intervals as being able, or not, to be infinitely delayed. By default, the first interpretation is retained.



t_1:	Send packet 0	t_9:	Reject duplicate packet 0
t_2:	Resend packet 0	t_{10}:	Accept packet 1
t_3:	Receive acknowledgment 0	t_{11}:	Send acknowledgment 1
t_4:	Send packet 1	t_{12}:	Reject duplicate packet 1
t_5:	Resend packet 1	t_{13}:	Lose packet 0
t_6:	Receive acknowledgment 1	t_{14}:	Lose acknowledgment 0
t_7:	Accept packet 0	t_{15}:	Lose packet 1
t_8:	Send acknowledgment 0	t_{16}:	Lose acknowledgment 1

Figure 1.7. *A Time Petri net for the alternating bit protocol*

ϕ_3 expresses that any message sent is eventually received. As any message can be lost then retransmitted an infinity of times, it is not satisfied. The counter-example, shown below, is a sequence containing a circuit going through state 16 and in which the message is lost (t_{13}) then retransmitted (t_2). On the other hand, any message sent is either received or lost, as expressed by ϕ_4, which evaluates to true.

```
        state 0: L.div p1 p5
-t1->   state 16: p2 p5 p9
-t13->  state 17: p2 p5
-t2->   state 16: p2 p5 p9
```

ϕ_5 expresses that, assuming neither the message nor its acknowledgment is lost an infinity of times, the acknowledgment of delivery of any transmitted message is received. It evaluates to true.

Finally, formula ϕ_6 expresses the absence of deadlocks, and it evaluates to true. As already explained, deadlocks in the Kripke transition systems handled by `selt` have the specific *dead* state property.

1.10. Conclusion

The constructions reviewed in this chapter produce finite abstractions for the generally infinite state spaces of bounded Time Petri nets. Various abstractions are available, preserving various families of properties of the state space, from marking reachability properties to bisimulation. The traditional state class graph construction (SCG) has been used in many works, academic and industrial, concerning hardware or software, and has been integrated into many tools. All the techniques discussed in this chapter are supported by the `Tina` environment [BER 04]. Once a finite representation of the set of states is available, we can apply to it model checking techniques to prove properties of the state space. These possibilities were illustrated in the framework of temporal logic LTL.

These methods suffer some intrinsic limits that we should keep in mind. One is that no necessary and sufficient condition can be stated for the boundedness property of Time Petri nets, a property which is required for applying the verification techniques described. There are a number of convenient sufficient conditions for this property, though, behavioral or structural. Another limit is that the number of state classes of bounded Time Petri net can be very large. This number, which is hardly predictable, results from the interplay between the structure of the marking space of nets and the temporal constraints associated with the transitions. It is thus essential to investigate incremental verification techniques or any techniques helping to limit combinatorial explosion [RIB 05].

The works in progress are concerned with enrichment of the description model. Two aspects are particularly investigated: enriching the Time Petri net model with external data structures and operations on these structures, and the capability of suspending and resuming transitions (Stopwatch Time Petri nets), so that preemptive systems can be represented and analyzed [BER 05].

Finally, many projects, such as Topcased [FAR 04] or Cotre [BER 03a], showed the interest and the will of industrial companies to integrate formal description techniques and some verification in their design process. This constitutes a second prospective axis of our work. The integration of verification techniques in an industrial development workshop requires theoretical work (scalability of the methods) and development work (efficiency/ergonomy of tools), in interaction with the final users, to shorten the gap between the standards in use in the industry and the target models used for verification.

1.11. Bibliography

[ARN 94] ARNOLD A., BEGAY D., CRUBILLÉ P., *Construction and Analysis of Transition Systems with MEC*, Word Scientific Publishing Co, Singapore, 1994.

[BER 83] BERTHOMIEU B., MENASCHE M., "An enumerative approach for analyzing time Petri nets.", *IFIP Congress Series*, vol. 9, p. 41–46, Elsevier Science Publishers (North Holland), 1983.

[BER 91] BERTHOMIEU B., DIAZ M., "Modeling and verification of time dependent systems using time Petri nets.", *IEEE Transactions on Software Engineering*, vol. 17, num. 3, p. 259–273, 1991.

[BER 01] BERTHOMIEU B., "La méthode des classes d'états pour l'analyse des réseaux Temporels – Mise en œuvre, Extension à la multi-sensibilisation", *Proc. Modélisation des Systèmes Réactifs*, Toulouse, France, 2001.

[BER 03a] BERTHOMIEU B., RIBET P.-O., VERNADAT F., BERNARTT J., FARINES J.-M., BODEVEIX J.-P., FILALI M., PADIOU G., MICHEL P., FARAIL P., GAUFFILET P., DISSAUX P., LAMBERT J., "Towards the verification of real-time systems in avionics: the cotre approach", *Workshop on Formal Methods for Industrial Critical Systems (FMICS'2003)*, p. 201–216, 2003.

[BER 03b] BERTHOMIEU B., VERNADAT F., "State Class Constructions for Branching Analysis of Time Petri Nets", *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2003), Warsaw, Poland, Springer LNCS 2619*, p. 442–457, 2003.

[BER 04] BERTHOMIEU B., RIBET P.-O., VERNADAT F., "The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets", *International Journal of Production Research*, vol. 42, num. 14, p. 2741–2756, 2004.

[BER 05] BERTHOMIEU B., LIME D., ROUX O. H., VERNADAT F., "Modélisation des Systèmes Réactifs (MSR'05)", *Journal Européen des Systèmes Automatisés*, vol. 39 (1-2-3), p. 223–238, 2005.

[BOU 04] BOUCHENEB H., HADJIDJ R., "Towards optimal CTL^* model checking of Time Petri nets", *Proceedings of 7th Workshop on Discrete Events Systems*, Rheims, France, 2004.

[CHA 04] CHAKI S., E M., CLARKE, OUAKNINE J., SHARYGINA N., SINHA N., "State/Event-based software model checking", *4th International Conference on Integrated Formal Methods (IFM'04), Springer LNCS 2999*, p. 128–147, 2004.

[DEC 91] DECHTER R., MEIRI I., PEARL J., "Temporal constraint networks", *Artificial Intelligence*, vol. 49 (1-3), p. 61–95, 1991.

[FAR 04] FARAIL P., GAUFILLET P., FILALI M., MICHEL P., VERNADAT F., "Vérifications dans un AGL orienté modèles", *Génie Logiciel*, vol. 69B, p. 51–55, 2004.

[FER 96] FERNANDEZ J.-C., GARAVEL H., MATEESCU R., MOUNIER L., SIGHIREANU M., "Cadp, a protocol validation and verification toolbox", *8th Conference Computer-Aided Verification, CAV'2001, Springer LNCS 1102*, p. 437–440, 1996.

[GAS 01] GASTIN P., ODDOUX D., "Fast LTL to Buchi Automata Translation", *13th Conference Computer-Aided Verification, CAV'2001, Springer LNCS 2102*, p. 53–65, jul 2001.

[JON 77] JONES N. D., LANDWEBER L. H., LIEN Y. E., "Complexity of Some Problems in Petri Nets.", *Theoretical Computer Science 4*, p. 277–299, 1977.

[KAN 90] KANELLAKIS P. K., SMOLKA S. A., "CCS Expressions, Finite State Processes, and Three Problems of Equivalence", *Information and Computation*, vol. 86, p. 43–68, 1990.

[KAR 69] KARP R. M., MILLER R. E., "Parallel Program Schemata", *Journal of Computer and System Sciences 3*, vol. 3, num. 2, p. 147–195, 1969.

[MER 74] MERLIN P. M., *A Study of the Recoverability of Computing Systems.*, Irvine: Univ. California, PhD Thesis, 1974.

[PAI 87] PAIGE P., TARJAN R. E., "Three partition refinement algorithms", *SIAM Journal on Computing*, vol. 16, num. 6, p. 973–989, 1987.

[PEN 01] PENCZEK W., PÓŁROLA A., "Abstraction and partial order reductions for checking branching properties of Time Petri Nets", *Proc. 22nd International Conference on Application and Theory of Petri Nets (ICATPN 2001), Springer LNCS 2075*, p. 323–342, 2001.

[RIB 05] RIBET P.-O., "Vérification formelle de systèmes. Contribution à la réduction de l'explosion combinatoire", Report, PhD thesis of the Institut National des Sciences Appliquées, Toulouse (LAAS/CNRS Report 05631), 2005.

[YON 98] YONEDA T., RYUBA H., "CTL model checking of Time Petri nets using geometric regions", *IEEE Transactions on Information and Systems*, vol. E99-D, num. 3, p. 1–10, 1998.

Chapter 2

Validation of Reactive Systems by Means of Verification and Conformance Testing

In this chapter we describe a methodology integrating verification and conformance testing for the formal validation of reactive systems. A specification of the system (an extended input-output automaton, which may be an infinite-state) and a set of safety properties ("nothing bad ever happens") and possibility properties ("something good may happen") are assumed. The properties are first tentatively verified on the specification using automatic techniques based on approximated state-space exploration, which are sound, but, as a price to pay for automation, are not complete for the given class of properties. Because of this incompleteness and of state-space explosion, verification may not succeed in proving or disproving the properties. However, even if verification did not succeed, the testing phase can proceed and provide useful information about the implementation. Test cases are automatically and symbolically generated from the specification and the properties, and are executed on a black-box implementation of the system. The test execution may detect violations of conformance between implementation and specification; in addition, it may detect violation/satisfaction of the properties by the implementation and by the specification. In this sense, testing completes verification. The approach is illustrated on a bounded retransmission protocol.

2.1. Introduction

Formal verification and conformance testing are two well-established approaches for the validation of reactive systems. Both approaches consist of comparing two different views, or levels of abstraction, of a system:

Chapter written by Camille CONSTANT, Thierry JÉRON, Hervé MARCHAND and Vlad RUSU.

– formal verification compares a formal *specification* of the system with respect to a set of higher-level *properties* that the system should satisfy;

– conformance testing [ISO 92, BRI 90] compares the observable behavior of an actual black-box implementation of the system with the observable behavior described by a formal specification, according to a conformance relation. It is an instance of *model-based testing* where specifications, implementations and the conformance relation between them are formalized. Test cases are automatically derived from specifications, and the verdicts resulting from test execution on an implementation are proved to be consistent with respect to the conformance relation. This chapter adopts the (by now classical) testing theory based on IOLTS (*input-output labeled transition systems*) and the *ioco* conformance relation [TRE 96].

Verification operates on formal models and allows in principle for complete, exhaustive proofs of properties on specifications. However, for expressive models such as those considered in this chapter (symbolic, infinite-state transition systems), verification is undecidable or too complex to be carried out completely, and we have to resort to partial or approximated verification. For example, safety properties can be checked on over-approximations of the reachable states of the specification; if the property holds in the over-approximation then it also holds on the original specification, but nothing can be concluded if the property violates the over-approximation.

Unlike verification, conformance testing is performed on the real implementation of the system by means of interactions between the implementation and test cases derived from the specification. In general, testing cannot prove that the implementation conforms to the specification: it can only detect non-conformances between the two "views" of the system. Since the specification serves as a basis for conformance testing, it is quite clear that the specification itself should be verified against expected properties. Thus, the two formal validation techniques are complementary: verification proves that the specification is correct with respect to the higher-level properties, and then conformance testing checks the correctness of the implementation with respect to the specification. Also, the two approaches use the same basic techniques and algorithms; see, e.g., [GAR 99, JAR 04, BLO 04, HON 02].

However, simply applying verification followed by conformance testing is not fully satisfactory. The main problem with this approach is that it does not guarantee that the properties are being tested at all on the implementation. This is because test cases are typically generated from specifications only, without taking other information about the system (such as requirements or properties) into account. What is missing is a formal link between properties and tests during the test generation and execution phases. In this chapter we describe a methodology combining verification and testing that provides the missing link, ensuring that properties verified on the specification are also tested on the implementation:

– A specification of a system (an extended input-output automaton, which may be infinite-state) and a set of safety properties ("nothing bad ever happens") and possibility properties ("something good may happen") are assumed. In practice, the specification could be, for example, a UML statechart, and the properties could be a set of positive scenarios (for possibility properties) or negative scenarios (for safety properties).

– The properties are first tentatively verified on the specification using automatic techniques based on approximated state-space exploration, which are sound, but, as a price to pay for automation, are not complete for the given class of properties. Because of this incompleteness and state-space explosion, the verification may not succeed in proving or disproving the properties.

– However, even if verification did not succeed, the testing phase can proceed and provide useful information about the implementation. Test cases are automatically and symbolically generated from the specification and the properties, and are executed on a black-box implementation of the system. The test execution may detect violations of conformance between implementation and specification; in addition, it may detect violation/satisfaction of the properties by the implementation *and by the specification*. In this sense, testing completes verification.

From a methodological point of view, this means that it is not required that the (typically difficult) verification of the specification fully succeeds before testing of the final implementation can start; and that the methodology provides correct information about the consistency between properties, specification and implementation.

From a more theoretical point of view, a uniform presentation of verification and conformance testing is given. All properties are represented using *observers*, which are automata (possibly extended with variables) with a set of accepting locations. We also show how the specification can be transformed into an observer for non-conformance. Such an observer is then a *canonical tester* [BRI 88] for *ioco*-conformance with respect to a given specification, which also proves that *ioco*-conformance to a specification is a safety property. Then, *test generation* is essentially a synchronous product between the canonical tester and observers for the properties. It can be followed by a *test selection* operation, which consists of extracting a "part" of the product that specifically targets one or several of the properties.

The chapter is organized as follows: in section 2.2 we present the model of input-output symbolic transition systems (IOSTS). The semantics of IOSTS in terms of (usually infinite) labeled transition systems is also given. In section 2.3 we define three symbolic operations on IOSTS that are used for verification and conformance testing. Section 2.4 presents the notion of observer for possibility or safety properties, the verification of such properties, and the theory of conformance testing of [TRE 96] reformulated using the notion of canonical tester, which is basically an observer for non-conformance. In section 2.5 we describe test generation from a specification and a set of properties as a product between observers for the properties and the canonical tester. Then, test selection amounts to choosing one or several properties of interest, and targeting the test towards checking the given properties; in section 2.6 we show how this can be done by using approximated analysis techniques based on abstract interpretation [COU 77]. The methodology is illustrated on a bounded retransmission protocol [HEL 94]. The material presented in this chapter synthesizes some of our earlier publications [JAR 04, JÉR 02, JEA 05, RUS 04, RUS 05, CON 07].

2.2. The IOSTS model

The IOSTS model is inspired from I/O automata [LYN 99]. Unlike I/O automata, IOSTS do not require *input-completeness* (i.e., all input actions do not need to be enabled all the time).

2.2.1. *Syntax of IOSTS*

DEFINITION 2.1.– *An IOSTS is a tuple* $\langle D, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ *where:*

– D *is a finite set of typed* Data*, partitioned into a set* V *of* variables *and a set* P *of* parameters*. For* $d \in D$*,* $type(d)$ *denotes the type of* d*;*

– Θ *is the* initial condition*, a predicate on* V*;*

– L *is a non-empty, finite set of* locations *and* $l^0 \in L$ *is the* initial location;

– Σ *is a non-empty, finite* alphabet*, which is the disjoint union of a set* $\Sigma^?$ *of* input actions*, a set* $\Sigma^!$ *of* output actions*, and a set* Σ^τ *of* internal actions*. For each action* $a \in \Sigma$*, its* signature $sig(a) = \langle p_1, \ldots, p_k \rangle \in P^k$ *(*$k \in \mathbb{N}$*) is a tuple of parameters. The signature of internal actions is the empty tuple;*

– $\mathcal{T}$ *is a set of* transitions*. Each transition is a tuple* $\langle l, a, G, A, l' \rangle$ *made of:*

- *a location* $l \in L$*, called the* origin *of the transition;*
- *an action* $a \in \Sigma$ *called the* action *of the transition;*
- *a predicate* G *on* $V \cup sig(a)$*, called the* guard*;*
- *an* assignment A*, which is a set of expressions of the form* $(x := A^x)_{x \in V}$ *such that, for each* $x \in V$*, the right-hand side* A^x *of the assignment* $x := A^x$ *is an expression of* $V \cup sig(a)$[1]*;*
- *a location* $l' \in L$ *called the* destination *of the transition.*

In graphical representations, inputs are identified by the "?" symbol and outputs are identified by the "!" symbol.

1. We assume, without further formalization, that the assignments are "well typed", that is, each expression e_x in the right-hand side has a type that corresponds to that of the variable x in the left-hand side.

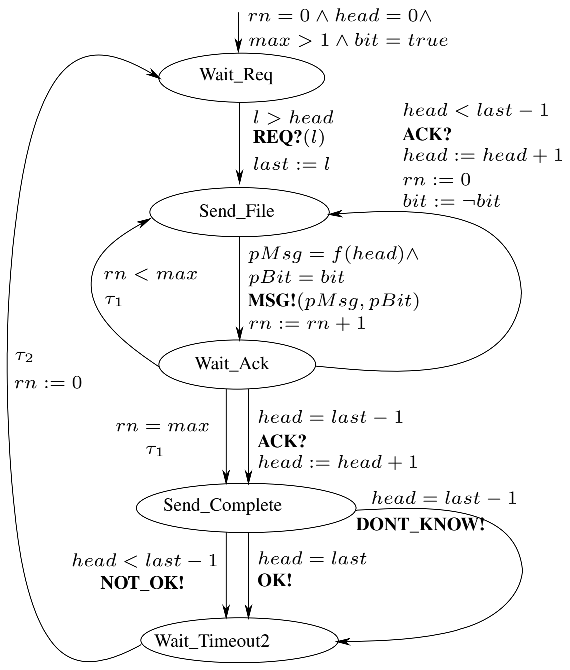


Figure 2.1. *Sender of the BRP*

EXAMPLE 2.1.– *We consider a bounded retransmission protocol (BRP) [HEL 94], whose role is to transmit data in a reliable manner over an unreliable network. We focus on the sender of the protocol. The specification of the sender is depicted in Figure 2.1.*

Execution starts by the reception of a **REQ?**(l) *input. The meaning of the parameter* l *is that the current session must transmit* $f(head), f(head + 1), \ldots, f(l - 1)$, *where* f *is the file to transmit, and the variable* $head$, *initialized to* 0, *is the index of the next "piece" of the file to be transmitted. Hence, the request makes sense only if* $l > head$.

Then, the sender proceeds by transmitting messages **MSG!**, *together with the data to transmit:* $f(head), f(head + 1), \ldots$ *and an alternating* bit *allows the receiver to distinguish between new messages and retransmissions. After each message, the sender waits for an acknowledgment* **ACK?** *after which it proceeds by sending the next message. If the acknowledgment does not arrive, the current message is retransmitted at most* $\max -1$ *times, where* $\max$ *is a symbolic constant of the protocol. The number of retransmissions is counted by the variable* rn. *Globally, the transmission terminates successfully when the last message has been transmitted and acknowledged. In* *this case, the sender confirms the success to its client by sending it an* **OK!** *confirmation. The transmission may also terminate unsuccessfully (confirmation* **NOT_OK!***) if some message (except the last one) is not acknowledged, or the outcome may be unknown (confirmation* **DONT_KNOW!***) if the last message is not acknowledged – either the message or its acknowledgment could have been lost.*

2.2.2. *Semantics of IOSTS*

The semantics of IOSTS is described in terms of input-output labeled transition systems.

DEFINITION 2.2.– An input-output labeled transition system (IOLTS) is a tuple $\langle S, S^0, \Lambda, \rightarrow\rangle$ where S is a possibly infinite set of *states*, $S^0 \subseteq S$ is a possibly infinite set of *initial states*, $\Lambda = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$ is a possibly infinite set of (input, output and internal) *actions*, and $\rightarrow \subseteq S \times \Lambda \times S$ is the *transition relation*.

The set $\Lambda^? \cup \Lambda^!$ is also called the set of *observable actions*. Intuitively, the IOLTS semantics of an IOSTS $\langle D = V \cup P, \Theta, L, q^0, \Sigma, \mathcal{T}\rangle$ "explores" the possible tuples of values (hereafter called *valuations*) of parameters P and variables V. Let $\mathcal{V}$ denote the set of valuations of the variables V, and Π denote the set of valuations of the parameters P. Then, for an expression E involving (a subset of) $V \cup P$, and for $\nu \in \mathcal{V}$, $\pi \in \Pi$, we denote by $E(\nu, \pi)$ the value obtained by substituting in E each variable by its value according to ν, and each parameter by its value according to π. For $P' \subseteq P$, we denote by $\Pi_{P'}$ the restriction of set Π of valuations to set P' of parameters.

DEFINITION 2.3.– *The semantics of an IOSTS* $\mathcal{S} = \langle D, \Theta, L, l^0, \Sigma, \mathcal{T}\rangle$ *is an IOLTS* $[\![\mathcal{S}]\!] = \langle S, S^0, \Lambda, \rightarrow\rangle$*, defined as follows:*

– the set of states is $S = L \times \mathcal{V}$*,*

– the set of initial states is $S^0 = \{\langle l_0, \nu\rangle \in S \mid \Theta(\nu) = true\}$*,*

– the set of actions $\Lambda = \{\langle a, \pi\rangle \mid a \in \Sigma, \pi \in \Pi_{sig(a)}\}$*, also called the set of* valued actions*, is partitioned into the sets* $\Lambda^?$ *of* valued inputs*,* $\Lambda^!$ *of* valued outputs*, and* Λ^τ *of* internal actions[2] *such that for* $\# \in \{?, !, \tau\}$*,* $\Lambda^\# = \{\langle a, \pi\rangle \mid a \in \Sigma^\#, \pi \in \Pi_{sig(a)}\}$.

– $\rightarrow$ *is the smallest relation in* $S \times \Lambda \times S$ *defined by the following rule:*

$$\frac{\langle l, \nu\rangle, \langle l', \nu'\rangle \in S \quad \langle a, \pi\rangle \in \Lambda \quad t = \langle l, a, G, A, l'\rangle \in \mathcal{T} \quad G(\nu, \pi) = true \quad \nu' = A(\nu, \pi)}{\langle l, \nu\rangle \overset{\langle a, \pi\rangle}{\rightarrow} \langle l', \nu'\rangle}$$

The rule says that the valued action $\langle a, \pi\rangle$ takes the system from state $\langle l, \nu\rangle$ to state $\langle l', \nu'\rangle$ if there exists a transition $t = \langle l, a, G, A, l'\rangle$ whose guard G evaluates to $true$

2. Since internal actions do not carry parameters, the sets Λ^τ and Σ^τ can be identified.

when the variables are evaluated according to ν and the parameters carried by action a evaluate according to π. Then, the assignment A maps the pair (ν, π) to ν'.

DEFINITION 2.4.– *An execution fragment is a sequence of alternating states and valued actions* $s_1\alpha_1 s_2\alpha_2 \cdots \alpha_{n-1}s_n \in S \cdot (\Lambda \cdot S)^*$ *such that* $\forall i = 1, n-1, s_i \xrightarrow{\alpha_i} s_{i+1}$. *An* execution *is an execution fragment starting in an initial state. We denote by* $Exec(\mathcal{S})$ *the set of execution fragments of an IOSTS* $\mathcal{S}$.

We often write $s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \cdots s_n \xrightarrow{\alpha_n} s_{n+1}$ as a shortcut for $\forall i = 1, n-1, s_i \xrightarrow{\alpha_i} s_{i+1}$. A state is *reachable* if it belongs to an execution. For a sequence $\sigma = \alpha_1\alpha_2 \cdots \alpha_n$ of valued actions, we write $s \xrightarrow{\sigma} s'$ for

$$\exists s_1, \ldots, s_{n+1} \in S, \quad s = s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \cdots s_n \xrightarrow{\alpha_n} s_{n+1} = s'.$$

For a set $S' \subseteq S$ of states of an IOSTS, we write $s \xrightarrow{\sigma} S'$ if there exists a state $s' \in S'$ such that $s \xrightarrow{\sigma} s'$. We say that s is *co-reachable* for S'.

DEFINITION 2.5.– *The* trace *trace*(ρ) *of an execution* ρ *is the projection of* ρ *on the set* $\Lambda^! \cup \Lambda^?$ *of observable actions. The set of traces of an IOSTS* $\mathcal{S}$ *is the set of all traces of all executions of* $\mathcal{S}$, *and is denoted by Traces*$(\mathcal{S})$.

Let $F \subseteq L$ be a set of locations of an IOSTS $\mathcal{S}$. An execution ρ of $\mathcal{S}$ is *recognized by* F if the execution terminates in a state in $F \times \mathcal{V}$. A trace is recognized by F if it is the trace of an execution recognized by F. The set of traces of an IOSTS $\mathcal{S}$ recognized by a set F of locations is denoted by *Traces*$(\mathcal{S}, F)$.

2.3. Basic operations on IOSTS

In this section we define a few basic operations on IOSTS that are used in verification and conformance testing based on this model.

2.3.1. *Parallel product*

The *parallel product* of two IOSTS $\mathcal{S}_1$, $\mathcal{S}_2$ will be used both in verification (for defining the set of traces of an IOSTS that are recognized by an observer) and in conformance testing (for modeling the synchronous execution of a test case on an implementation). This operation requires that $\mathcal{S}_1$, $\mathcal{S}_2$ share the same sets of input and output actions (with the same signatures), have the same set of parameters, and have no variable in common.

DEFINITION 2.6.– *The IOSTS* $\mathcal{S}_j = \langle D_j, \Theta_j, L_j, l_j^0, \Sigma_j, \mathcal{T}_j\rangle$ *(*$j = 1, 2$*) with* $D_j = V_j \cup P_j$, *and* $\Sigma_j = \Sigma_j^? \cup \Sigma_j^!$ *are* compatible *if* $V_1 \cap V_2 = \emptyset$, $P_1 = P_2$, $\Sigma_1^! = \Sigma_2^!$, *and* $\Sigma_1^? = \Sigma_2^?$.

DEFINITION 2.7.– *The parallel product* $\mathcal{S} = \mathcal{S}_1 || \mathcal{S}_2$ *of two compatible IOSTS* $\mathcal{S}_1, \mathcal{S}_2$ *(cf. Definition 2.6) is the IOSTS* $\langle D = V \cup P, \Theta, L, l^0, \Sigma, \mathcal{T}\rangle$ *defined by:* $V = V_1 \cup V_2$,

$P = P_1 = P_2$, $\Theta = \Theta_1 \wedge \Theta_2$, $L = L_1 \times L_2$, $l^0 = \langle l_1^0, l_2^0 \rangle$, $\Sigma^? = \Sigma_1^? = \Sigma_2^?$, $\Sigma^! = \Sigma_1^! = \Sigma_2^!$, $\Sigma^\tau = \Sigma_1^\tau \cup \Sigma_2^\tau$. The set $\mathcal{T}$ of symbolic transitions of the parallel product is the smallest set satisfying the following rules:

$$(1) \quad \frac{\langle l_1, a, G_1, A_1, l_1' \rangle \in \mathcal{T}_1, \quad a \in \Sigma_1^\tau, \quad l_2 \in L_2}{\langle \langle l_1, l_2 \rangle, a, G_1, A_1 \cup (x := x)_{x \in V_2}, \langle l_1', l_2 \rangle \rangle \in \mathcal{T}}$$

(*and symmetrically for* $a \in \Sigma_2^\tau$)

$$(2) \quad \frac{\langle l_1, a, G_1, A_1, l_1' \rangle \in \mathcal{T}_1 \quad \langle l_2, a, G_2, A_2, l_2' \rangle \in \mathcal{T}_2}{\langle \langle l_1, l_2 \rangle, a, G_1 \wedge G_2, A_1 \cup A_2, \langle l_1', l_2' \rangle \rangle \in \mathcal{T}}$$

The parallel product allows internal actions to evolve independently by Rule (1) and synchronizes on the observable actions (inputs and outputs) by Rule (2).

LEMMA 2.1.– *Traces$(\mathcal{S}_1 \| \mathcal{S}_2)$ = Traces$(\mathcal{S}_1) \cap$ Traces$(\mathcal{S}_2)$, and Traces$(\mathcal{S}_1 \| \mathcal{S}_2, F_1 \times F_2)$ = Traces$(\mathcal{S}_1, F_1) \cap$ Traces$(\mathcal{S}_2, F_2)$.*

2.3.2. *Suspension*

In conformance testing it is assumed that the environment may observe not only outputs, but also the *absence of outputs* (i.e., in a given state, the system is blocked in the sense that it does not emit any output for the environment to observe). The absence of output is called *quiescence* in conformance testing [TRE 99]. In a black-box implementation, quiescence is observed using timers: a timer is reset whenever the environment sends a stimulus to the implementation. It is assumed that the duration of the timer is large enough such that if no output occurs while the timer is running, then no output will ever occur. Then, when the timer expires, the environment "observes" quiescence.

Blocking and quiescence are not necessarily errors. A specification may state that the system is blocked, waiting for an input from the environment, or that the system's execution is terminated. In order to distinguish a quiescence that is accepted by a specification from one that is not, quiescence can be made explicit on specifications by means of a symbolic operation called *suspension*. This operation transforms an IOSTS $\mathcal{S}$ into an IOSTS $\mathcal{S}^\delta$, also called the *suspension IOSTS* of $\mathcal{S}$, in which quiescence is materialized by a new output action δ. For this, each location l of $\mathcal{S}^\delta$ contains a new self-looping transition, labeled with a new output action δ, which may be fired if and only if no other output or internal action may be fired in l.

DEFINITION 2.8.– *Given $\mathcal{S} = \langle D = V \cup P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ an IOSTS, with $\Sigma = \Sigma^! \cup \Sigma^? \cup \Sigma^\tau$ and $\delta \notin \Sigma$, the* suspension *IOSTS $\mathcal{S}^\delta$ is the tuple $\langle D, \Theta, L, l^0, \Sigma^\delta, \mathcal{T}^\delta \rangle$ where $\Sigma^\delta = \Sigma^{\delta !} \cup \Sigma^? \cup \Sigma^\tau$ with $\Sigma^{\delta !} = \Sigma^! \cup \{\delta\}$, $\mathcal{T}^\delta = \mathcal{T} \cup \{\langle l, \delta, G_{\delta,l}, (x := x)_{x \in V}, l \rangle \mid l \in L\}$ and*

$$G_{\delta,l} \triangleq \bigwedge_{a \in \Sigma^! \cup \Sigma^\tau} \neg G_{a,l} \quad \text{and} \quad G_{a,l} \triangleq \bigvee_{t = \langle l, a, G, A, l' \rangle \in \mathcal{T}} \exists sig(a) \cdot G. \tag{2.1}$$

We assume that the guards are expressed in a theory where existential quantifiers can be eliminated, such as, for example, Presburger Arithmetic. The IOSTS $\mathcal{S}^\delta$ is obtained from $\mathcal{S}$ by adding a new output action δ to $\Sigma^!$, and, for each location l of $\mathcal{S}$, a new self-looping transition, labeled with action δ, with identity assignments, and whose guard is $G_{\delta,l}$. This guard formalizes the conditions under which δ may be fired in location l: no output or internal action may be fired in l. Now, for $a \in \Sigma^! \cup \Sigma^\tau$, $G_{a,l}$ gives the conditions on the system's variables under which the action a is firable in l. Hence, $G_{\delta,l}$ is the *conjunction* of the *negations* of all formulae $G_{a,l}$, for all $a \in \Sigma^! \cup \Sigma^\tau$.

EXAMPLE 2.2.– *The suspended IOSTS* Sender$^\delta$ *corresponding to the* Sender *of the BRP is depicted in Figure 2.2. In location* Wait_Req*, the system is blocked waiting for input from the environment, hence, it has a self-loop labeled with the suspension action* $\delta!$*. The other locations where* $\delta!$ *is potentially firable are* Wait_Ack *and* Send_Complete*. For example, in* Wait_Ack*, the guard of the* $\delta!$*-labeled self loop is* $rn > \max$*, which is the complement of the two conditions (*$rn = \max$*,* $rn < \max$*) labeling transitions by which the system can leave the location without intervention of the environment.*

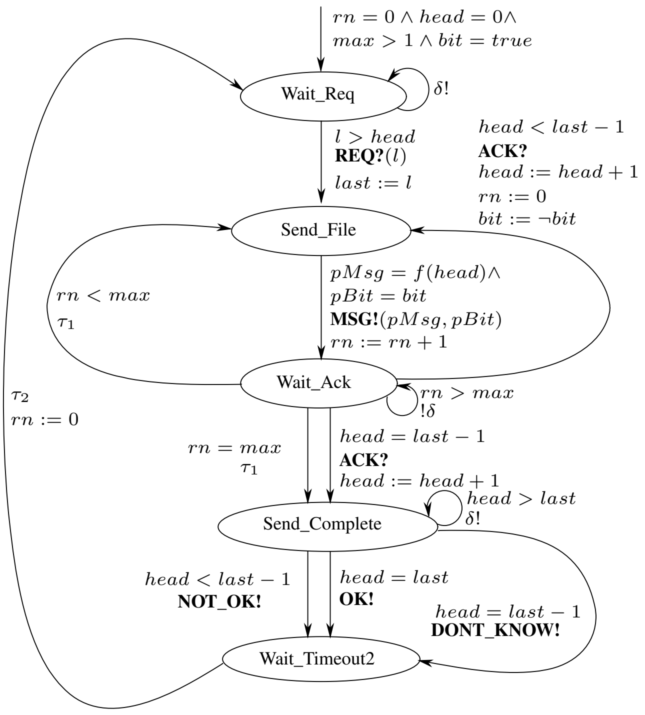


Figure 2.2. *Suspended BRP*

2.3.3. *Deterministic IOSTS and determinization*

Intuitively, an IOSTS is *deterministic* if each of its traces matches exactly one execution. For example, test cases in conformance testing satisfy this property, as they must always give the same verdict on the same interaction trace with an implementation.

DEFINITION 2.9.– *An IOSTS $\langle D, \Theta, L, l^0, \Sigma, \mathcal{T}\rangle$ is deterministic if*

– it has no internal actions: $\Sigma^\tau = \emptyset$;

– it has at most one initial state, i.e., the initial condition Θ is satisfied by at most one valuation ν^0 of the variables (then, the initial state is $\langle l^0, \nu^0\rangle$);

– for all $l \in L$ and for each pair of transitions with origin in l and labeled by the same action a: $\langle l, a, G_1, A_1, l_1\rangle$ and $\langle l, a, G_2, A_2, l_2\rangle$, the conjunction of the guards $G_1 \wedge G_2$ is unsatisfiable[3].

Determinizing an IOSTS $\mathcal{S}$ means computing a deterministic IOSTS with the same traces as $\mathcal{S}$. This operation consists of two steps: eliminating internal actions, and eliminating non-determinism between observable actions. We omit them here due to space limitations; these operations are presented in detail in [RUS 06, JÉR 06].

EXAMPLE 2.3.– *The determinized system* $\mathrm{det}(\mathrm{Sender}^\delta)$ *obtained from* Sender^δ *is depicted in Figure 2.3. Note that* $\mathrm{det}(\mathrm{Sender}^\delta)$ *is not, strictly speaking, deterministic, because its initial condition is satisfied by more than one (actually, by infinitely many) valuation of the symbolic constants* $\max$ *and f. However, any instance of* $\mathrm{det}(\mathrm{Sender}^\delta)$ *obtained by instantiating* $\max$ *and f to actual values is deterministic.*

We denote by $det(\mathcal{S})$ the deterministic IOSTS obtained from the IOSTS $\mathcal{S}$.

LEMMA 2.2.– *Traces*$(det(\mathcal{S})) = Traces(\mathcal{S})$.

2.4. Verification and conformance testing with IOSTS

2.4.1. *Verification*

Let us consider the following standard verification problem: given a reactive system modeled using an IOSTS $\mathcal{S}$ and property ψ on the system's traces, does $\mathcal{S}$ satisfy ψ? We consider two kinds of properties: safety properties and possibility properties. Both can be modeled using *observers*, which are deterministic IOSTS with a set of recognizing locations.

3. Since we have assumed that the guards are in a theory where existential quantifiers can be eliminated, satisfiability is decidable in that theory as well.

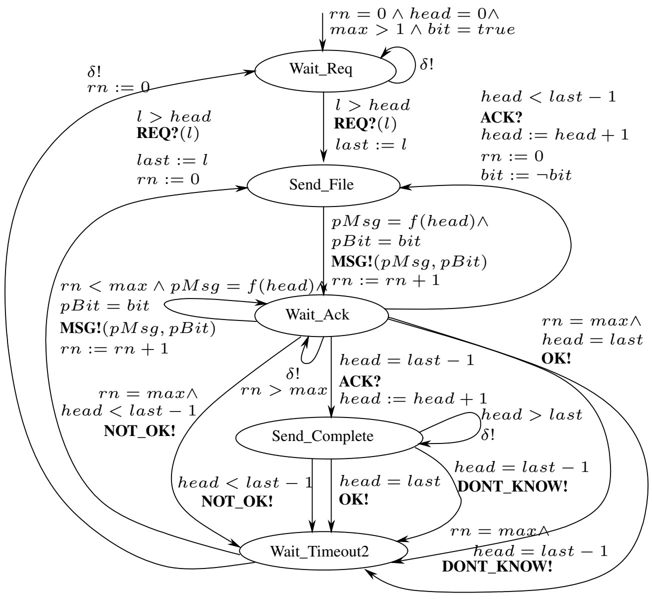


Figure 2.3. *Determinized BRP*

DEFINITION 2.10.– *An observer is a deterministic IOSTS ω together with a set of distinguished locations $F \subseteq L_\omega$ with no outgoing transitions, i.e., no location in F is the origin of any symbolic transition of ω. An observer (ω, F) is compatible with an IOSTS M if ω is compatible with M. The set of observers compatible with M is denoted by $\Omega(M)$.*

In section 2.2.2, we defined the set of traces recognized by a set of locations of an IOSTS. For an observer (ω, F) this gives:

$$Traces(\omega, F) = \{\sigma \in \Lambda^*_\omega \mid \exists s_o \in S^0_\omega,\ \exists l \in F,\ \exists \nu \in \mathcal{V},\ s_o \xrightarrow{\sigma}_\omega \langle l, \nu \rangle\}. \qquad (2.2)$$

We will distinguish between *positive* observers, which express possibility properties, and *negative* observers, which express safety properties. Whether an observer is positive or negative is purely a matter of interpretation, although we will by convention denote the set of recognizing locations of positive observers by *Satisfy*, and those of negative observers by *Violate*. We now define what it means for an observer to be satisfied by an IOSTS.

DEFINITION 2.11.– *Let M be an IOSTS and $(\omega, Satisfy_\omega) \in \Omega(M)$ a positive observer compatible with M. Then, M satisfies $(\omega, Satisfy_\omega)$, denoted by $M \models^+ (\omega, Satisfy_\omega)$, if $Traces(M) \cap Traces(\omega, Satisfy_\omega) \neq \emptyset$.*

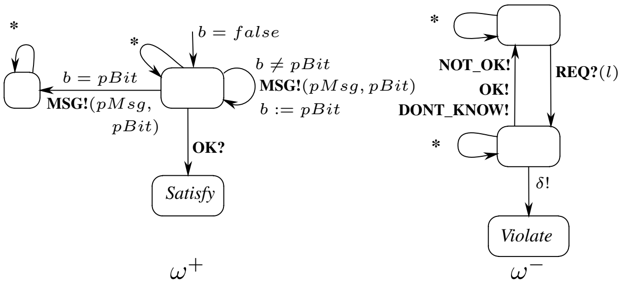


Figure 2.4. *Properties: possibility (left), safety (right)*

DEFINITION 2.12.– *Let M be an IOSTS and $(\omega, Violate_\omega) \in \Omega(M)$ a negative observer compatible with M. Then, M satisfies $(\omega, Violate_\omega)$, denoted by $M \models^- (\omega, Violate_\omega)$, if $Traces(M) \cap Traces(\omega, Violate_\omega) = \emptyset$.*

Hence, satisfying a negative observer (a safety property) amounts to not satisfying, i.e., to violating a positive observer (a possibility property), and vice versa.

2.4.1.1. *Verifying safety properties*

Consider an IOSTS M and a negative observer $(\omega, Violate_\omega) \in \Omega(M)$. The safety property defined by the observer is satisfied by sequences in $(\Lambda^!_M \cup \Lambda^?_M)^* \setminus Traces(\omega, Violate_\omega)$ (and these sequences only). In particular, if M is the suspended IOSTS $\mathcal{S}^\delta$ of a given IOSTS $\mathcal{S}$, the property is satisfied by a subset of $(\Lambda^!_s \cup \{\delta\} \cup \Lambda^?_s)^*$.

EXAMPLE 2.4.– *The safety property, depicted in the right-hand side of Figure 2.4, expresses the fact that the BRP sender is not allowed to be blocked between the request* **REQ?** *and any of the confirmations* **OK!**, **NOT_OK!** *or* **DONT_KNOW!**. *If the special output $\delta!$ denoting blocking/quiescence is observed between request and confirmation, the Violate location is reached, which expresses violation of the safety property. All other valued actions (that do explicitly appear in the automata representing the properties) are implicitly ignored.*

Let L be the set of locations of the IOSTS M. Then, $Traces(M) = Traces(M, L)$, and by Lemma 2.1, $Traces(M \| \omega, L \times Violate_\omega) = Traces(M) \cap Traces(\omega, Violate_\omega)$.

Thus, verifying $M \models^- (\omega, Violate_\omega)$ amounts to verifying whether $Traces(M \| \omega, L \times Violate_\omega)$ is empty, which can be done by establishing that the set of locations $L \times Violate_\omega$ is unreachable from the initial states of $M \| \omega$, or, equivalently, the initial states of $M \| \omega$ are not co-reachable for $L \times Violate_\omega$.

Reachability is probably the most studied verification problem and many different approaches for solving it have been proposed. Among the automatic approaches, we consider *model checking* [CLA 99], which, in general, explores a finite subset of the set of reachable states, and *abstract interpretation* [COU 77], which symbolically explores a super-set of the set of reachable states. Thus, model checking can prove reachability, but in general cannot disprove it; and abstract interpretation can disprove reachability, but in general cannot prove it. Hence, we use abstract interpretation to try to prove that a safety property is satisfied, and model checking to try to prove that it is violated.

For this, our STG (Symbolic Test Generation) tool [CLA 02] was connected with the abstract interpretation tool NBac [JEA 03]. To prove a safety property using a negative observer $(\omega, \mathit{Violate}_\omega)$, STG computes the product $\omega \| M$; then, NBac automatically computes an over-approximation of the set of states that are both reachable from the initial states and co-reachable for the locations $L \times \mathit{Violate}_\omega$. If the result is empty, the safety property holds, otherwise, no conclusions can be drawn.

For example, this approach makes it possible to establish that the IOSTS $Sender^\delta$ depicted in Figure 2.2 satisfies the observer ω_1 depicted in the right-hand side of Figure 2.4. Intuitively, this is because the guards of transitions labeled by $\delta!$ in locations *Wait_Ack* and *Send_Complete* are always *false* in the over-approximation of the set of reachable states computed by NBac. On the other hand, *violations* of safety properties may be detected by model checking but neither satisfaction nor violation can be established in general because of undecidability issues.

2.4.1.2. *Verifying possibility properties*

In contrast to safety properties, possibility properties express that "something good can happen". We express possibility properties using positive observers $(\omega, \mathit{Satisfy}_\omega)$; the property is satisfied if some location in the set $\mathit{Satisfy}_\omega$ is reached. If the observer is compatible with IOSTS M then the property is satisfied by M if at least one trace of M is in $\mathit{Traces}(\omega, \mathit{Satisfy}_\omega)$.

EXAMPLE 2.5.– *The possibility property on the left-hand side of Figure 2.4 describes an ideal scenario in which the sender transmits each message exactly once, without retransmissions. The scenario ends in the Satisfy location after the* **OK!** *confirmation. Retransmissions lead to the location represented at the extreme left of the figure, from which the Satisfy location is unreachable. The observer uses a Boolean variable b in order to distinguish new messages from retransmissions. This property is satisfied by the IOSTS $Sender^\delta$ depicted in Figure 2.2: unsurprisingly, the sender of the BRP contains a scenario in which no retransmission is performed.*

2.4.1.3. *Combining observers*

The parallel product of two observers (ω_1, F_1) and (ω_2, F_2) can also be interpreted as an observer by choosing the set of recognizing locations. These combinations will be encountered in section 2.5. A natural choice for this set is $F_1 \times F_2$, although other choices are possible. If both observers are negative, i.e., they both describe safety properties, then reaching $F_1 \times F_2$ in the observer $(\omega_1 \| \omega_2, F_1 \times F_2)$ expresses violation of both properties. If both observers are positive (for possibility properties), then reaching $F_1 \times F_2$ denotes the satisfaction of both. If the recognizing locations are, for example, $F_1 \times (L_{\omega_2} \setminus F_2)$, then the corresponding observer denotes violation of the first safety property and not of the second one, or, for possibility properties, satisfaction of the first property but not of the second one.

It is also possible to give an interpretation of the parallel product of a positive and a negative observer. For example, if (ω_1, F_1) is positive and (ω_2, F_2) is negative, reaching $F_1 \times F_2$ in $\omega_1 \| \omega_2$ means that the safety property has been violated and the possibility property has been satisfied, that is, the two properties are in conflict because the "bad thing" prohibited by the safety property coincides, on some traces, with the "good thing" required by the possibility property.

2.4.2. *Conformance testing*

A *conformance relation* formalizes the set of implementations that behave consistently with a specification. An implementation $\mathcal{I}$ is not a formal object (it is a physical system) but, in order to reason about conformance, it is necessary to assume that the semantics of $\mathcal{I}$ can be modeled by a formal object. We assume here that it is modeled by an IOLTS (see Definition 2.2). The notions of trace and quiescence are defined for IOLTS just as for IOSTS. The implementation is assumed to be *input-complete*, i.e., all its inputs are enabled in all states, and that it has the same interface (input and output actions with their signatures) as the specification. These assumptions are called the *test hypothesis* in conformance testing. The standard *ioco* relation defined by Tretmans [TRE 99] can be rephrased in the following way.

DEFINITION 2.13.– *An implementation $\mathcal{I}$ ioco-conforms to a specification $\mathcal{S}$, denoted by $\mathcal{I}$ ioco $\mathcal{S}$, if Traces$(\mathcal{S}^\delta) \cdot (\Lambda^! \cup \{\delta\}) \cap Traces(\mathcal{I}^\delta) \subseteq$ Traces$(\mathcal{S}^\delta)$.*

Intuitively, an implementation $\mathcal{I}$ *ioco*-conforms to its specification $\mathcal{S}$, if, after each trace of the suspension IOSTS $\mathcal{S}^\delta$, the implementation only exhibits outputs and quiescences allowed by $\mathcal{S}^\delta$. In this framework, the specification may be *partial* with respect to inputs, i.e., after an input that is not described by the specification, the implementation can have any behavior, without violating conformance to the specification. This corresponds to the idea that a specification models a given set of services that must be provided by a system; a particular implementation of the system may implement more services than specified, but these additional features should not influence its conformance.

2.5. Test generation

This section describes how to generate tests for checking conformance to a given specification, as well as other safety or possibility properties. The test cases attempt to detect violations or satisfactions of the properties by an implementation of the system, and violations of the conformance between the implementation and the specification. In addition, if the verification steps described in the previous section did not completely succeed, test execution may also detect violation or satisfaction of properties by the *specification*. Hence, the testing step completes verification. We show that the test cases generated by our method always return correct verdicts. In this sense, the test generation method itself is correct.

We first define the *output-completion* $\Sigma^!(M)$ of a determinisitic IOSTS M. We then show that the output-completion of the IOSTS $\det(\mathcal{S}^\delta)$, where $\det()$ is the determinization operation[4], is a *canonical tester* [BRI 88] for $\mathcal{S}$ and the *ioco* relation defined in section 2.4.2. A canonical tester for a specification with respect to a given relation makes it possible, in principle, to detect every implementation that disagrees with the specification according to the relation. This derives from the fact, stated in Lemma 2.3 below, that *ioco*-conformance to a specification $\mathcal{S}$ is equivalent to satisfying (a safety property described by) an observer obtained from $\Sigma^!(\det(\mathcal{S}^\delta))$. By composing this observer of non-conformance with other observers for safety or possibility properties, we obtain test cases for checking the conformance to $\mathcal{S}$ and the satisfaction/violation of the properties.

DEFINITION 2.14.– *Given a deterministic IOSTS* $M = \langle D, \Theta, L, l^0, \Sigma, \mathcal{T}\rangle$*, the* output completion *of* M *is the IOSTS denoted by* $\Sigma^!(M) = \langle D, \Theta, L \cup \{\mathit{Fail}\}, l^0, \Sigma, \mathcal{T} \cup \bigcup_{l \in L, a \in \Sigma^!} \langle l, a, \bigwedge_{t = \langle l, a, G_t, A_t, l'\rangle \in \mathcal{T}} \neg G_t, (x := x)_{x \in V}, \mathit{Fail}\rangle\rangle$*, where* $\mathit{Fail} \notin L$.

Interpretation: $\Sigma^!(M)$ is obtained from M by adding a new location *Fail* $\notin L$, and for each $l \in L$ and $a \in \Sigma^!$, a transition with origin l, destination *Fail*, action a, identity assignments and guard $\bigwedge_{t = \langle l, a, G_t, A_t, l'_t\rangle \in \mathcal{T}} \neg G_t$. Thus, any output not firable in M becomes firable in $\Sigma^!(M)$ and leads to the new (deadlock) location *Fail*. The output-completion of an IOSTS M can be seen as a negative observer by choosing $\{\mathit{Fail}\}$ as the set of recognizing locations. The following lemma says that conformance to a specification $\mathcal{S}$ is a safety property, namely, the property represented by the negative observer $(\Sigma^!(\det(\mathcal{S}^\delta)), \{\mathit{Fail}\})$.

LEMMA 2.3.– $\mathcal{I}$ *ioco* $\mathcal{S}$ *iff* $\mathcal{I}^\delta \models^- (\Sigma^!(\det(\mathcal{S}^\delta)), \{\mathit{Fail}\})$.

The lemma also says that the observer $(\Sigma^!(\det(\mathcal{S}^\delta)), \{\mathit{Fail}\})$ is a canonical tester for *ioco*-conformance to $\mathcal{S}$. Indeed, $\mathcal{I}^\delta \models^- (\Sigma^!(\det(\mathcal{S}^\delta)), \{\mathit{Fail}\})$ basically says that the execution of $\Sigma^!(\det(\mathcal{S}^\delta))$ on the implementation $\mathcal{I}$ never reaches the *Fail* location, i.e., it never leads to a "Fail" verdict; the fact that this is equivalent to $\mathcal{I}$ *ioco* $\mathcal{S}$ (as stated by Lemma 2.3) is the property characterizing a canonical tester [BRI 88]. In the rest of the chapter we denote the canonical tester of a specification $\mathcal{S}$ by *canon*($\mathcal{S}$).

4. Defined in [RUS 06, JÉR 06].

With respect to the IOSTS depicted in Figure 2.3, the canonical tester $canon(Sender)$ has one more location *Fail*, and implicit transitions from each location to it, as defined by the output-completion operation.

A canonical tester is, in principle, enough to detect all implementations that do not conform to a given specification. However, our goal is to detect, in addition to such non-conformances, the violations/satisfactions of other (additional) safety/possibility properties coming from, e.g., the system's requirements. The observers expressing such properties can also serve as mechanisms for test selection. Using Lemma 2.1, the product between an observer and the canonical tester defines a subset of "interesting" traces among all the traces generated by the canonical tester.

We also consider a safety property represented by its observer $(\omega^-, \mathit{Violate}_{\omega^-})$ and a possibility property represented by an observer $(\omega^+, \mathit{Violate}_{\omega^+})$. The product of the three observers can be seen as a test case that refines the canonical tester, in the sense that violations/satisfactions of the safety/possibility properties can also be detected.

This information is obtained by running $\mathit{test}(\mathcal{S}, \omega+, \omega^-) = \omega^+ \| \mathit{canon}(\mathcal{S}) \| \omega^-$ parallel with implementation $\mathcal{I}$. When *Satisfy*, *Fail* or *Violate* locations (or combinations of them) are reached, a certain verdict is emitted. The verdicts are summarized in Table 2.1, and their meaning is explained below. In the table, an empty box denotes a location other that *Satisfy*, *Fail* or *Violate*.

	ω^+	$canon(\mathcal{S})$	ω^-	Verdict
(1)	*Satisfy*			**Satisfy**
(2)			*Violate*	**Violate**
(3)		*Fail*		**Fail**
(4)	*Satisfy*	*Fail*		**SatisfyFail**
(5)		*Fail*	*Violate*	**ViolateFail**
(6)	*Satisfy*		*Violate*	**SatisfyViolate**
(7)	*Satisfy*	*Fail*	*Violate*	**SatisfyViolateFail**

Table 2.1. *Verdicts*

1) $\mathit{Satisfy} = \mathit{Satisfy}_{\omega^+} \times (L_{canon(\mathcal{S})} \setminus \{\mathit{Fail}\}) \times (L_{\omega^-} \setminus \mathit{Violate}_{\omega^-})$
Both implementation $\mathcal{I}$ and specification $\mathcal{S}$ satisfy the possibility property. No non-conformances between $\mathcal{I}$ and $\mathcal{S}$ were detected. The corresponding verdict, here called **Satisfy**, corresponds to the verdict usually called **Pass** in conformance testing.

Satisfy: both implementation and specification satisfy the possibility property

2) $Violate = (L_{\omega^+} \setminus Satisfy_{\omega^+}) \times (L_{canon(\mathcal{S})} \setminus \{Fail\}) \times \{Violate_{\omega^-}\}$
If test execution reaches the *Violate* location, then both implementation and specification violate the safety property. This is a situation that may only occur if verification of the property on the specification did not succeed, and shows that testing can sometimes complement verification.

Violate: both implementation and specification violate the safety property

3) $Fail = (L_{\omega^+} \setminus Satisfy_{\omega^+}) \times \{Fail\} \times (L_{\omega^-} \setminus Violate_{\omega^-})$
This is the standard **Fail** verdict in conformance testing (non-conformance detected).

Fail: the implementation does not conform to the specification

4) $SatisfyFail = \{Satisfy_{\omega^+}\} \times \{Fail\} \times (L_{\omega^-} \setminus Violate_{\omega^-})$
In this case, the implementation satisfies the possibility property, but the specification did not show evidence that it satisfies the property. There is a non-conformance between implementation and specification.

SatisfyFail: the implementation does not conform to the specification but it satisfies the possibility property

5) $ViolateFail = (L_{\omega^+} \setminus Satisfy_{\omega^+}) \times \{Fail\} \times \{Violate_{\omega^-}\}$
In this case the implementation violates both the safety property and conformance to the specification.

ViolateFail: the implementation violates the safety property and does not conform to the specification

6) $\mathit{SatisfyViolate} = \{\mathit{Satisfy}_{\omega^+}\} \times (L_{\mathit{canon}(\mathcal{S})} \setminus \mathit{Fail}_{\mathit{canon}(\mathcal{S})}) \times \{\mathit{Violate}_{\omega^-}\}$
Both implementation and specification satisfy the possibility property and violate the safety property. This indicates that the two properties are mutually inconsistent (the "bad thing" that the former prohibits the "good thing" that the latter requires).

SatisfyViolate: Implementation and specification satisfy the possibility property, and violate the safety property

7) $\mathit{SatisfyViolateFail} = \{\mathit{Satisfy}_{\omega^+}\} \times \{\mathit{Fail}\} \times \{\mathit{Violate}_{\omega^-}\}$
Reaching these locations indicates inconsistencies between the two properties, as well as non-conformance between implementation and specification, and violation of the safety property by the implementation.

SatisfyViolateFail : the implementation does not conform to the specification, it violates the safety property and satisfies the reachability property

2.6. Test selection

A test case such as $\mathit{test}(\mathcal{S}, \omega^+, \omega^-)$ with so many verdicts is mostly interesting from a theoretical point of view, as all information about non-conformances and satisfaction/violation of several properties are embodied in it. In practice, it is expected that more focused test cases will be preferred, which target one, or a few, properties at a time. This can be performed by the *test selection* operation, described below and illustrated in the BRP case study.

Assume that we have built the test case $\mathit{test}(\mathcal{S}, \omega^+, \omega^-)$ as in the previous section (including the *mirror* operation, i.e., the transformation of inputs into outputs and vice versa, as the test case plays the role of environment for the implementation), and we have decided on a subset of the verdicts that we specifically want to target. These could be any of the verdicts of the test case, and are encoded in a subset L_{target} of its verdict locations. For example, if we want to target the satisfaction of the possibility property described by the observer ω^+, the target locations are $L_{target} = \mathit{Satisfy} \cup \mathit{SatisfyFail} \cup \mathit{SatisfyViolateFail}$.

For a state s of an IOSTS and a location l of the IOSTS, we say that s is *co-reachable* for the location l if there exists a valuation v of the variables and a trace σ such that $s \xrightarrow{\sigma} \langle l, \nu \rangle$, and denote by $\mathit{coreach}(l)$ the set of states that are co-reachable for the location l. For a set of locations $L' \subseteq L$ of the IOSTS, $coreach(L') \triangleq \bigcup_{l' \in L'} coreach(l')$. Then, the test selection process consists (ideally) of selecting, from a given test case, the subset of states that are co-reachable for L_{target}.

It should be quite clear that an exact computation of this set of states is generally impossible. However, there exist techniques that make it possible to compute over-approximations. We use one such technique based on abstract interpretation and implemented in the NBac tool [JEA 03]. The tool computes, for each location l, a *symbolic co-reachable state*, which over-approximates the states with location l that are co-reachable for L'.

DEFINITION 2.15.– *For a location l and a set of locations L' of an IOSTS $\mathcal{S}$, we say $\langle l, \varphi_{l \to L'} \rangle$ is a* symbolic co-reachable state *if $\varphi_{l \to L'}$ is a formula such that we have the inclusion $\{\langle l, \nu \rangle \mid \nu \models \varphi_{l \to L'}\} \supseteq \{\langle l, \nu \rangle \mid \nu \in \mathcal{V}\} \cap coreach(L')$.*

The following algorithm uses this information to perform test selection.

DEFINITION 2.16.–

1) The test case test$(\mathcal{S}, \omega^+, \omega^-)$ is built as described in section 2.5, including the mirror operation. Let L be its set of locations, $\mathcal{T}$ its set of transitions, and $\Sigma = \Sigma^! \cup \Sigma^?$ its alphabet, where $\Sigma^! = \Sigma^?_S$ and $\Sigma^? = \Sigma^!_S \cup \{\delta\}$. Let also Inconc $\notin L$ be a new location.

2) For each location $l \in L$, a symbolic co-reachable state $\langle l, \varphi_{l \to L_{target}} \rangle$ is computed.

3) Next, for each location $l \in L$ of the IOSTS, and each transition $t \in \mathcal{T}$ of the IOSTS with origin l, guard G, and label a,

- if $a \in \Sigma^!$ and if $G \wedge \varphi_{l \to L_{target}}$ is unsatisfiable, then t is eliminated from $\mathcal{T}$, otherwise, the guard of t becomes $G \wedge \varphi_{l \to L_{target}}$;

- if $a \in \Sigma^?$, then the guard of t becomes $G \wedge \varphi_{l \to L_{target}}$ and a new transition is added to $\mathcal{T}$, with origin l, destination Inconc, action a, guard $G \wedge \neg\varphi_{l \to L_{target}}$ and identity assignments.

The test selection operation consists of detecting transitions from states that are not co-reachable for the target set of locations L_{target}. This is done by performing a coreachability analysis to these locations using the NBac tool [JEA 03]. If a "useless" transition is labeled by an *output*, then it may be removed from the test case: a test case controls its outputs, hence it may decide not to perform an output if it "knows" that the target locations is unreachable. On the other hand, *inputs* cannot be prevented from occurring, hence the transitions labeled by *inputs*, by which the locations in L_{target} cannot be reached, are reoriented to a new location, called *Inconc*. This location also corresponds to an "Inconclusive" verdict, where the current test cannot reach the chosen target, and therefore the current test execution can be stopped.

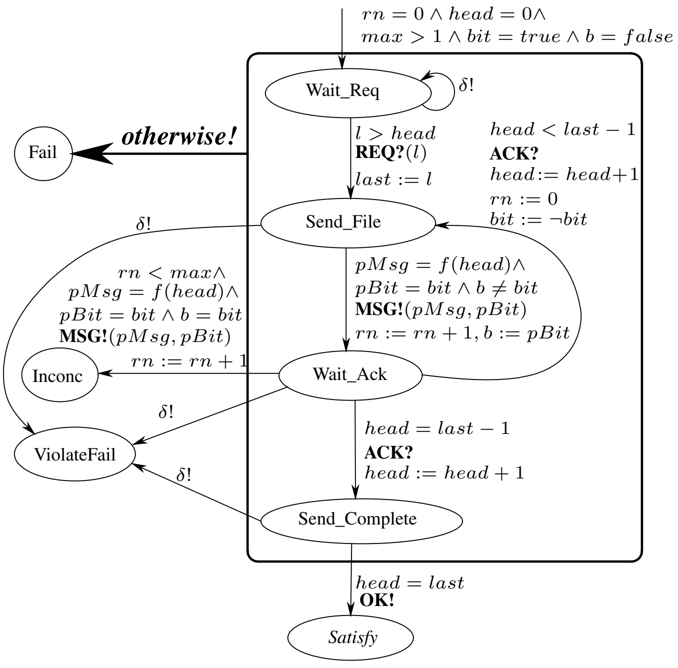


Figure 2.5. *Test case obtained by selection*

In this way, all traces from the original test case leading to the chosen target L_{target}, are preserved, and the correctness of the other verdicts is preserved as well.

The test case obtained by specializing the canonical tester to the properties depicted in Figure 2.4, followed by selection targeting the *possibility* property, is depicted in Figure 2.5. Several locations are grouped into "macro-states" *à la* statecharts.

2.7. Conclusion and related work

A system may be viewed at several levels of abstraction: high-level *properties*, operational *specification*, and black-box *implementation*. In our framework, properties and specifications are described using IOSTS, which are extended automata that operate on symbolic variables and communicate with the environment through input and output actions carrying parameters. IOSTS are given formal semantics in terms of IOLTS. The implementation is a black-box, but it is assumed that its semantics can be described by an unknown IOLTS. This makes it possible to formally link the implementation and the specification by a conformance relation. A satisfaction relation links them both to higher-level properties: safety properties, which express that something bad never happens, and possibility properties, which express that something good can happen.

A validation methodology is proposed for checking these relations, i.e., for checking the consistency between the different views of the system: first, the properties are tentatively verified on the specification using automatic approximate analysis techniques, which are sound but, because of undecidability problems, are inherently incomplete. Then, test cases are automatically generated from the specification and the properties, and are executed on the implementation of the system. If the verification step was successful, that is, it has proved or disproved each property on the specification, then test execution may detect violation or satisfaction of the properties by the implementation and the violation of the conformance relation between implementation and specification. On the other hand, if the verification did not make it possible to prove or disprove some properties, the test execution may additionally detect violation or satisfaction of the properties by the specification. In this sense, the testing step completes the verification step. The approach is illustrated on a bounded retransmission protocol [HEL 94].

Related work. The approach described in [AMM 01] considers a deterministic finite-state specification S and an invariant P assumed to hold on S. Then, mutants S' of S are built using standard mutation operators, and a combined machine is generated, which extends sequences of S with sequences of S'. Next, a model checker is used to generate sequences that violate P, which prove that S' is a mutant of S violating P. Finally, the obtained sequences are interpreted as test cases to be executed on the implementation. In contrast, our approach is able to deal with certain classes of non-deterministic infinite-state specifications, and we perform the tests on black-box implementations, whereas [AMM 01] requires a mechanism for observing internal details (such as values of variables) of the implementation.

The approach described in [GAR 99] starts with a formal specification S and a temporal-logic property P assumed to hold on S, and uses the ability of model checkers to construct counter-examples for $\neg P$ on S. These counter-examples can be interpreted as *witnesses* (and eventually transformed into test cases) for P on S. Other authors have investigated this approach, e.g., [HAM 04], who propose efficient test generation techniques in this framework.

[BLO 04, HON 02] extend this idea by formalizing standard coverage criteria (all-definitions, all-uses, etc.) using observers (or in temporal logic). Again, test cases are generated by model checking the observers (or the temporal-logic formulae) on the specification. The approaches described in these papers are also restricted to deterministic finite-state systems, and do not apply to actual black-box implementations, as they require the ability to observe the values of variables in the implementation.

In [FER 03] an approach for generating tests from finite-state specifications (which can be non-deterministic) and from observers describing linear-time temporal logic properties is described. The generated test cases do not check for conformance to the specification, but only compare the black-box implementation to the properties.

The specification is only used as a guideline for test selection. In contrast, we deal with infinite-state specifications, and generate tests for checking the implementation against the specification and the properties.

[VRI 01] describes a general framework for test generation and execution of test cases from non-deterministic specifications. The testing mechanism additionally uses *observation objectives*, which are sets of traces of observable actions. The verdicts obtained by test execution describe relations between the black-box implementation and the specification, as well as the satisfaction or dissatisfaction of the objectives by the implementation. The authors instantiate their general approach in the TorX tool [BEL 96], where test purposes are finite automata and test generation and execution are performed on-the-fly. The work presented here can be seen as another instantiation of the same general approach, which differs from the instantiation given in [VRI 01] with respect to expressiveness of observers (infinite-state extended automata rather than finite) and with respect to the test generation mechanism (off-line, symbolic rather than on-the-fly, enumerative). Off-line test selection allows, in principle, for better test selection than a purely on-the-fly approach, and symbolic test generation allows, in principle, a better handling of the state-space explosion problem.

[RUS 04] presents an approach for combining model checking of safety properties and conformance testing for finite-state systems. This paper can be seen as the first step towards the approach presented here, which deals with infinite-state systems. In the finite-state settings of [RUS 04] verification is decidable, which influences the whole approach: for example the generated test cases do not need to take into account the possibility that the properties might be violated by the specification (such violations are always detected by the model-checking step, in which case the test generation step is simply canceled).

A different approach for combining model checking and black-box testing is black-box checking [PEL 01]. Under some assumptions from the implementation (the implementation is deterministic; an upper bound n on its number of states is known), the black-box checking approach constructs a complete test suite of exponential size in n for checking properties expressed by Büchi automata.

Our approach can also be related to the combination of verification, testing and monitoring proposed in [HAV 02]. In their approach, monitoring is passive (pure observation), whereas ours is reactive and adaptive, guided by the choice of inputs to deliver to the system as pre-computed in a test case.

In [BOZ 00], the authors use a rudimentary combination of positive or negative observers for enumerative test generation for the SSCOP protocol, using the TGV and ObjectGéode (Telelogic) tools. An observer describes an abstract view of the actions of the protocol, which over-approximates its behaviors. Test generation uses additional test purposes, which can be classified as positive observers. During test generation, violation of the negative observer by the specification can be observed. However, the generated test cases are not meant to observe violations of the negative observer by the implementation.

Finally, in [RUS 00, JEA 05] we propose a symbolic algorithm for selecting test cases from a specification be means of test purposes, which are positive observers (possibility properties). The difference with this chapter lies mainly in methodology. Test purposes in [RUS 00, JEA 05] are essentially a pragmatic means for test selection and are not necessarily related to properties of the specification. In contrast, test selection in this chapter uses safety or possibility properties, which are actual properties that the specification should satisfy; and the testing step is formally integrated with the verification of the properties on the specification.

2.8. Bibliography

[AMM 01] AMMANN P., DING W., XU D., "Using a model checker to test safety properties", *International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society, 2001.

[BEL 96] BELINFANTE A., FEENSTRA J., DE VRIES R., TRETMANS J., GOGA N., FEIJS L., MAUW S., "Formal test automation: a simple experiment", *International Workshop on the Testing of Communicating Systems (IWTCS'99)*, p. 179–196, 1996.

[BLO 04] BLOM J., HESSEL A., JONSSON B., PETTERSSON P., "Specifying and generating test cases using observer automata", GRABOWSKI J., NIELSEN B., Eds., in *Proc. of Formal Approaches to Software Testing*, vol. 3395 of *LNCS*, Springer, p. 137–152, 2004.

[BOZ 00] BOZGA M., FERNANDEZ J.-C., GHIRVU L., JARD C., JÉRON T., KERBRAT A., MOREL P., MOUNIER L., "Verification and test generation for the SSCOP protocol", *Journal of Science of Computer Programming, special issue on Formal Methods in Industry*, vol. 36, num. 1, p. 27–52, Elsevier Science B. V., January 2000.

[BRI 88] BRINSKMA E., "A theory for the derivation of tests", *Protocol Specification, Testing and Verification (PSTV'88)*, p. 63–74, 1988.

[BRI 90] BRINSKMA E., ALDEREN A., LANGERAK R., VAN DE LAAGEMAT J., TRETMANS J., "A formal approach to conformance testing", *Protocol Specification, Testing and Verification (PSTV'90)*, p. 349–363, 1990.

[CLA 99] CLARKE E. M., GRUMBERG O., PELED D. A., *Model Checking*, MIT Press, 1999.

[CLA 02] CLARKE D., JÉRON T., RUSU V., ZINOVIEVA E., "STG: a symbolic test generation tool", *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, vol. 2280 of *LNCS*, Springer, p. 470–475, 2002.

[CON 07] CONSTANT C., JÉRON T., MARCHAND H., RUSU V., "Integrating formal verification and conformance testing for reactive systems", *IEEE Transactions on Software Engineering*, vol. 33, num. 8, p. 558–574, August 2007.

[COU 77] COUSOT P., COUSOT R., “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”, *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Los Angeles, California, p. 238–252, 1977.

[FER 03] FERNANDEZ J., MOUNIER L., PACHON C., “Property-oriented test generation”, *Formal Aspects of Software Testing Workshop*, vol. 2931 of *LNCS*, Springer, 2003.

[GAR 99] GARGANTINI A., HEITMEYER C., “Using model checking to generate tests from requirements specifications”, *ESEC/SIGSOFT FSE*, p. 146–162, 1999.

[HAM 04] HAMON G., DEMOURA L., RUSHBY J., “Generating efficient test sets with a model checker”, *2nd International Conference on Software Engineering and Formal Methods*, Beijing, China, IEEE Computer Society, p. 261–270, September 2004.

[HAV 02] HAVELUND K., ROSU G., “Synthesizing monitors for safety properties”, *Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’02)*, vol. 2280 of *LNCS*, Grenoble, France, Springer, p. 342–356, 2002.

[HEL 94] HELMINK L., SELLINK M. P. A., VAANDRAGER F., “Proof-checking a data link protocol”, *Types for Proofs and Programs (TYPES’94)*, vol. 806 of *LNCS*, Springer, p. 127–165, 1994.

[HON 02] HONG H., LEE I., SOKOLSKY O., URAL H., “A temporal logic based theory of test coverage and generation”, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS’02)*, vol. 2280 of *LNCS*, Grenoble, France, Springer, p. 327–341, April 2002.

[ISO 92] ISO/IEC 9646, “Conformance testing methodology and framework”, 1992.

[JAR 04] JARD C., JÉRON T., “TGV: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems”, *Software Tools for Technology Transfer (STTT)*, vol. 6, Springer, October 2004.

[JEA 03] JEANNET B., “Dynamic partitioning in linear relation analysis”, *Formal Methods in System Design*, vol. 23, num. 1, p. 5–37, Kluwer, 2003.

[JEA 05] JEANNET B., JÉRON T., RUSU V., ZINOVIEVA E., “Symbolic test selection based on approximate analysis”, *11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’05)*, vol. 3440 of *LNCS*, Edinburgh, UK, Springer, April 2005.

[JÉR 02] JÉRON T., “TGV: théorie, principes et algorithmes”, *Techniques et Sciences Informatiques, numéro spécial Test de Logiciels*, vol. 21, 2002, in French.

[JÉR 06] JÉRON T., MARCHAND H., RUSU V., “Symbolic determinisation for extended automata”, *4th IFIP Conference on Theoretical Computer Science (TCS’06)*, Springer, 2006.

[LYN 99] LYNCH N., TUTTLE M., “Introduction to IO automata”, *CWI Quarterly*, vol. 3, num. 2, 1999.

[PEL 01] PELED D., VARDI M., YANNAKAKIS M., “Black-box checking”, *Journal of Automata, Languages and Combinatorics*, vol. 7, num. 2, p. 225–246, 2001.

[RUS 00] RUSU V., DU BOUSQUET L., JÉRON T., "An approach to symbolic test generation", *International Conference on Integrating Formal Methods (IFM'00)*, vol. 1945 of *LNCS*, Springer, p. 338–357, 2000.

[RUS 04] RUSU V., MARCHAND H., TSCHAEN V., JÉRON T., JEANNET B., "From safety verifcation to safety testing", *International Conference on Testing of Communicating Systems (TestCom04)*, vol. 2978 of *LNCS*, Springer, 2004.

[RUS 05] RUSU V., MARCHAND H., JÉRON T., "Automatic verification and conformance testing for validating safety properties of reactive systems", FITZGERALD J., TARLECKI A., HAYES I., Eds., *Formal Methods 2005 (FM05)*, vol. 3582 of *LNCS*, Springer, July 2005.

[RUS 06] RUSU V., Formal verification and conformance testing for reactive systems, Authorization to supervise research, University of Rennes 1, December 2006.

[TRE 96] TRETMANS J., "Test generation with inputs, outputs and repetitive quiescence", *Software—Concepts and Tools*, vol. 17, num. 3, p. 103–120, Springer Verlag, 1996, Also: Technical Report No. 96-26, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[TRE 99] TRETMANS J., "Testing concurrent systems: a formal approach", *CONCUR'99*, vol. 1664 of *LNCS*, Springer, p. 46–65, 1999.

[VRI 01] DE VRIES R., TRETMANS J., "Towards formal test purposes", *Formal Approaches to Testing of Software (FATES'01)*, p. 61–76, 2001.

Chapter 3

An Introduction to Model Checking

3.1. Introduction

In formal logic, *model checking* designates the problem of determining whether a formula φ evaluates to true or false in an interpretation $\mathcal{K}$, written $\mathcal{K} \models \varphi$. This problem finds applications in computer science: for example, $\mathcal{K}$ might represent a knowledge base and φ could be a query of which we wish to determine if it is implied by the knowledge in the base. We are then interested in finding efficient algorithms for determining whether $\mathcal{K} \models \varphi$ holds. In this chapter, we are interested in applications where $\mathcal{K}$ represents a system and φ a formula that represents a correctness property of this system. Typically, the systems we are interested in are *reactive*, that is, they interact repeatedly with their environment. They are often more concerned with control than with data and are usually composed of several components operating in parallel. Starting from a simple lift control application, we present basic ideas and concepts of verification algorithms in this context. The first publications about model checking appeared in 1981 by Clarke and Emerson [CLA 81] and by Queille and Sifakis [QUE 81]. Since then much progress has been made, and model checking has left the academic domain to enter mainstream development, notably of embedded systems and of communication protocols. Advances in the theory and application of model checking are reported in several important international conferences (including CAV, CHARME, and TACAS).

The inputs of the model checker are a description of the system to be analyzed and the property to verify. The tool either confirms that the property is true in the model or informs the user that it does not hold. In that case, the model checker will also provide a counter-example: a run of the system that violates the property. This answer helps

Chapter written by Stephan MERZ.

find the reason for the failure and has significantly contributed to the success of model checking in practice. Unfortunately, in practice model checking does not always yield such clear-cut results because the resource requirements (in terms of execution time and memory needed) can prohibit verifying more than an approximate model of the system. The positive outcome of model checking then no longer guarantees the correctness of the system and, reciprocally, an error found by the model checker may be due to an inaccurate abstraction of the system. Model checking is therefore not a substitute for standard procedures to ensure system quality, but it is an additional technique that can help uncover design problems at early stages of system development.

This chapter is intended as an introduction to the fundamental concepts and techniques of algorithmic verification. It reflects a necessarily subjective reading of the (abundant) studies. We try to give many references to original work so that the chapter can be read as an annotated bibliography. More extensive presentations of the subject can be found in books and more detailed articles, including [BÉR 01, CLA 99, CLA 00, HUT 04]. The structure of the chapter is as follows: section 3.2 presents an example of verifying a lift controller using the model checker SPIN. Basic definitions of transition systems, as well as an algorithm for verifying system invariants, are given in section 3.3. Section 3.4 is devoted to introducing temporal logics and ω-automata, which serve as the bases for the model checking algorithms presented in section 3.5. Finally, some topics of research are presented in section 3.6, which concludes the chapter. We restrict ourselves here to the case of discrete, untimed systems. Model checking techniques for timed systems are presented in detail in Chapter 4.

3.2. Example: control of an elevator

We will consider a simple model for the control of an elevator serving the floors of a building. This model serves merely to demonstrate the basic ideas, it is not intended as a realistic model of elevator control. It is generally advisable to start model checking for rough abstractions of the intended system in order to manage the complexity of the model and to get the first results quickly. Additional features required for a more realistic model can then be added one after another.

The model of Figure 3.1 is written in the language PROMELA [HOL 03]. In this language, a system description is given by several parallel processes that execute asynchronously. Our model consists of a process `lift` that represents the lift (controller) and as many additional processes `button` as there are floors in the building: that number is specified by the constant `FLOORS`. All processes are started by the process `init`. The externally observable interface of the elevator control is made up of the global variables `floor` whose value gives the floor where the lift is currently situated, and `request`, a Boolean array that indicates the requests to be served by the lift.

Each process `button(i)` represents the behavior of passengers who may call the lift to serve floor `i`. The process contains a non-terminating loop (`do` ... `od`) in the body of which the lift can be requested at that floor provided the lift is currently located at a different floor. In this model we do not distinguish between calls from within the lift cabin and from the floors.

The process `lift` declares the local Boolean variables `moving` and `going_up` that indicate if the lift is currently moving and, if so, in which direction. The integer variable `j` is used as an auxiliary scratch variable. The body of that process again consists of an infinite loop that models the elevator control. There are four branches (introduced by `::`) according to the different states the lift can be in. If it is currently moving, it will arrive at the next-higher or next-lower floor; it will stop there in case there is a pending request and then reset the request. If the lift is stopped at a floor, the controller checks first if there is another request in the current direction or, if that is not the case, in the opposite direction. In the case of a pending request, the lift will start moving in the required direction, otherwise it will remain stopped at the current floor.

The two final branches of the loop contain assertions that are used to verify elementary correctness properties of the model. The first assertion states that the lift never moves out of the range of legal floor values – this is not completely obvious from the way the model is written. The second assertion verifies that the lift does not pass a floor for which there is a pending request. Assertions are a particular kind of *safety properties*, which state that "nothing bad ever happens".

Beyond these assertions, an important correctness property of the elevator is that every request will eventually be served. This is an example of a *liveness property* which state that "something good happens eventually". For the first floor, this property is expressed in temporal logic (see section 3.4) by the formula

$$\mathrm{G}(\texttt{request[1]} \Rightarrow \mathrm{F}(\texttt{floor} = 1)). \tag{3.1}$$

The model checker SPIN can be used to verify assertions as well as temporal logic formulae over PROMELA models. For the verification of the liveness property (3.1), we have to specify the option *with weak fairness* in order to ensure that the process `lift` will not remain inactive forever. Figure 3.2 shows the size of the models (in terms of numbers of states and transitions), as well as the memory consumption (in MB) and the time (in seconds) required by SPIN to verify the indicated properties over the model of Figure 3.1. We can observe that the overall cost of verification increases by about an order of magnitude per additional floor, and that the verification of Formula (3.1) is significantly more expensive than just assertion checking.

3.3. Transition systems and invariant checking

The semantic framework for system models used for model checking is provided by the concepts of *transition systems* and *Kripke structures*. We now formally define these notions and present an algorithm for verifying invariant properties.

```
#define FLOORS  4       /* number of floors */
int floor = 0;          /* current floor */
bool request[FLOORS];   /* outstanding requests */

proctype lift() {
  bool moving = false; bool going_up = true; int j;

  do
  :: moving && going_up -> atomic { floor++;
       if :: request[floor] -> moving = false; request[floor] = false
          :: else -> skip
       fi     }
  :: moving && !going_up -> atomic { floor--;
       if :: request[floor] -> moving = false; request[floor] = false
          :: else -> skip
       fi     }
  :: !moving && going_up -> atomic { j = floor+1;
       do :: j == FLOORS -> going_up = false; break
          :: j < FLOORS && request[j] -> moving = true; break
          :: else -> j++
       od     }
  :: !moving && !going_up -> atomic { j = floor-1;
       do :: j < 0 -> going_up = true; break
          :: j >= 0 && request[j] -> moving = true; break
          :: else -> j--
       od     }
  :: assert (0 <= floor && floor < FLOORS)
  :: assert (!moving || !request[floor])
  od
}

proctype button(int myfloor) {
  do :: atomic { myfloor != floor -> request[myfloor] = true }
     :: true -> skip
  od
}

init {
  int i=0;
  run lift();
  do :: i < FLOORS -> run button(i); i++ od
}
```

Figure 3.1. *Model of an elevator in* PROMELA

	Assertions				Formula (3.1)			
FLOORS	States	Transitions	Time	Memory	States	Transitions	Time	Memory
4	6,151	35,297	0.38	2.9	8,769	164,763	1.10	6.1
5	35,721	239,788	1.11	8.1	52,052	1.38e06	9.59	9.4
6	194,556	1.48e06	9.79	21.1	288,395	1.01e07	76.64	58.5

Figure 3.2. *Model size and resource consumption by* SPIN

3.3.1. *Transition systems and their runs*

Transition systems describe the states, the initial states and the possible state transitions of systems. They provide a general framework for describing the (operational) semantics of reactive systems, independently of concrete formalisms used for their specification.

DEFINITION 3.1.– *A* labeled transition system $\mathcal{T} = (Q, I, E, \delta)$ *is given by:*

– *a set Q of* states*;*

– *a subset $I \subseteq Q$ of* initial states*;*

– *a set E of (action)* labels*;*

– *and a transition relation $\delta \subseteq Q \times E \times Q$.*

We require that δ is a total relation: for every $q \in Q$ there exist $e \in E$ and $q' \in Q$ such that $(q, e, q') \in \delta$. An action (label) $e \in E$ is called enabled *at a state $q \in Q$ if $(q, e, q') \in \delta$ holds for some $q' \in Q$.*

A run *of $\mathcal{T}$ is an ω-sequence $\rho = q_0 \overset{e_0}{\Rightarrow} q_1 \overset{e_1}{\Rightarrow} \cdots$ of states $q_i \in Q$ and labels $e_i \in E$ such that $q_0 \in I$ is an initial state and $(q_i, e_i, q_{i+1}) \in \delta$ is a transition for all $i \in \mathbb{N}$. A state $q \in Q$ is called* reachable *in $\mathcal{T}$ if there exists a run $q_0 \overset{e_0}{\Rightarrow} q_1 \overset{e_1}{\Rightarrow} \cdots$ of $\mathcal{T}$ such that $q_n = q$ for some $n \in \mathbb{N}$.*

Definition 3.1 is generic in the sets Q and E: it does not specify the structure of the sets of states and labels of a transition system and can be specialized for different settings. Often, states will be given as variable assignments, and labels represent individual system actions. The latter are particularly useful for specifying fairness constraints. Another example can be found in the definition of a timed transition system in section 4.2. A transition system is *finite* if its set of states is finite. Throughout this chapter, we will consider verification techniques for finite transition systems.

We have assumed that the transition relation δ is total in order to simplify some of the technical development. In particular, we need only consider infinite system executions rather than distinguish between finite and infinite ones. This hypothesis is easily satisfied by assuming a "stuttering" action $\tau \in E$ with $(q, \tau, q') \in \delta$ if and only if $q = q'$, for all $q, q' \in Q$. In this case, the deadlock states of a transition system are those that only enable the τ transition.

We emphasize that transition systems are a semantic concept for the description of system behavior. In practice, verification models are usually described in modeling languages, including (pseudo) programming languages such as PROMELA, process algebras or Petri nets (see Chapter 1). In general, the size of the transition system corresponding to a system description in some such language will be exponential in the size of its description. Different model checkers are optimized for certain classes of systems such as shared-variable or message passing programs.

Verification algorithms determine whether a transition system satisfies a given property. As we have seen in the elevator example of section 3.2, properties are built from elementary propositions that can be true or false in a system state. The following definition formalizes this idea with the concept of a Kripke structure that extends a transition system with an interpretation of atomic propositions in states.

DEFINITION 3.2.– *Let $\mathcal{V}$ be a set. A Kripke structure $\mathcal{K} = (Q, I, E, \delta, \lambda)$ extends a transition system by a mapping $\lambda : Q \to 2^{\mathcal{V}}$ that associates with every state $q \in Q$ the set of propositions true at state q. The runs of a Kripke structure are just the runs of its underlying transition system.*

The labeling λ of the states of a Kripke structure with sets of atomic propositions allows us to evaluate formulae of propositional logic built from the propositions in $\mathcal{V}$. We write $q \models P$ if state q satisfies propositional formula P. Let us note in passing that in principle, propositional logic is expressive enough for describing state properties of finite transition systems.

3.3.2. *Verification of invariants*

A *system invariant* is a state property P such that $q \models P$ holds for all reachable system states q. Invariants are elementary safety properties. In the particular case of an *inductive invariant*, P is assumed to hold true for every initial state and to be preserved by all transitions, i.e. $q \models P$ and $(q, E, q') \in \delta$ implies that $q \models P$. Of course, every inductive invariant is a system invariant, but the converse is not necessarily true. In particular, a system invariant need not be preserved by transitions from non-reachable states. Inductive invariants are the basis for deductive system verification, but less important for model checking.

Verifying that a finite Kripke structure $\mathcal{K}$ satisfies an invariant property P is a conceptually simple problem, and it illustrates well the basic ideas of algorithms for model checking. We can simply enumerate the reachable system states and verify that P is satisfied by every one of them. Termination is guaranteed by the finiteness of $\mathcal{K}$. A basic algorithm for invariant checking is presented in Figure 3.3: the variable `seen` contains the set of states that have already been visited, whereas `todo` represents a set of states that need to be explored. For each state s in `todo`, the algorithm checks whether the predicate holds at s and otherwise aborts the search, returning false. It then adds to `todo` all successors s' of s (with respect to arbitrary labels) that have not yet been seen. The algorithm returns `true` when there are no more states to explore.

In an implementation of this algorithm, the set `seen` will be represented by a hash table so that membership of an element in that set can be decided in quasi-constant time. The set `todo` will typically be represented by a stack or a queue, corresponding to depth-first or breadth-first search. If `todo` is a stack, it is easy to produce a counter-example in case the invariant does not hold: the algorithm can be organized such that

```
Boolean verify_inv(KripkeStructure ks, Predicate inv) {
  Set seen = new Set();
  Set todo = new Set();
  foreach (State i in ks.getInitials()) {
    if (!seen.contains(i)) {
      todo.add(i);
      while (!todo.isEmpty()) {
        State s = todo.getElement();
        todo.remove(s); seen.add(s);
        if (!s.satisfies(inv)) { return false; }
        foreach (State s' in ks.successors(s)) {
          if (!seen.contains(s')) {
            todo.add(s')
  } } } } }
  return true;
}
```

Figure 3.3. *Checking invariants by state enumeration*

when a state s violating the predicate is found, the stack contains a path from an initial state to s, which can be displayed to the user. If `todo` is a queue, we are certain to find invariant violations at a minimum depth in the state-space graph, and this makes it easier for the user to understand why the invariant fails to hold. However, the generation of counter-examples in a queue-based implementation requires an auxiliary data structure for retrieving the predecessor of a state.

SPIN uses the algorithm of Figure 3.3 for the verification of assertions. The user can choose whether depth-first or breadth-first search should be used. There are specialized tools for invariant checking such as Murϕ [DIL 92], and the main challenge in implementing such a tool is to be able to search large state spaces, beyond 10^6–10^7 states. For such system sizes, the set `seen` can no longer be stored in the main memory. Although the disk can be used in principle, suitable access strategies must be found to simulate random access, and verification will be slowed down considerably. Combinations of the three following principles are useful for the analysis of large systems in practice (see also section 3.6 for an additional discussion of these techniques):

– *Compression techniques* rely on compact representations of data structures such as states and sets of states in memory. For example, the implementation may decide to store state signatures (hash codes) instead of proper states in the set `seen`. A collision between the signatures of two distinct states could then lead to cutting off parts of the search space, hence possibly missing invariant violations. However, any error reported by the algorithm would still correspond to a valid counter-example. By estimating the probability of collisions and using different hash functions during several runs over the same model, we can estimate the coverage of verification and improve the reliability of the result.

– *Reduction-based techniques* attempt to determine a subset of the runs whose exploration guarantees the correctness of the system invariant over the overall system. In particular, independence of different transitions enabled in a given state or symmetry relations among data or system parameters can significantly reduce the number of runs that need to be explored by the algorithm.

– Finally, *abstraction* can help to construct a significantly smaller model that can be verified exhaustively and whose correctness guarantees the correctness of the original model. Whereas abstractions are usually constructed during system modeling in an ad-hoc manner, this relation can be formalized and the construction of abstractions can be done automatically. In general, failure of verification over an abstract model does not imply that the invariant does not hold over the original model because the abstraction could have identified reachable and unreachable states of the original model. However, a spurious counter-example produced over an abstract model often helps to suggest an improvement of the abstraction.

3.4. Temporal logic

Given a Kripke structure $\mathcal{K}$, we are interested in their properties, such as:

– Does (the reachable part of) $\mathcal{K}$ contain "bad" states, such as deadlock states where only the τ action is enabled, or states that do not satisfy an invariant?

– Are there executions of $\mathcal{K}$ such that, after some time, a "good" state is never reached or a certain action never executed? Such executions correspond to *livelocks* where some process does not make progress yet the entire system is not blocked.

– Can the system be re-initialized? In other words, is it always possible to reach an initial system state?

Temporal logic provides a language in which such properties can be formulated. Different temporal logics can be distinguished according to their syntactic features, or to the semantic structures over which their formulae are evaluated. Linear-time temporal logic and branching-time temporal logic are the two main kinds of temporal logic, and they will be introduced below. We will then present some principles of the theory of ω-automata and its connection with linear-time temporal logic. This correspondence will be useful for the presentation of a model checking algorithm in section 3.5.

3.4.1. *Linear-time temporal logic*

Linear-time temporal logic extends classical logic by temporal modalities to refer to different (future or past) time points. Its formulae are interpreted over infinite sequences of states, such as the runs of Kripke structures from which the labels have been omitted. We will consider here a propositional version PTL of this logic. As before, we assume given a set $\mathcal{V}$ of atomic propositions. The interpretation of PTL is based on a function $\lambda : Q \to 2^{\mathcal{V}}$ that evaluates the atomic propositions over states, just as in the definition 3.2 of a Kripke structure. For a sequence $\sigma = q_0 q_1 \cdots$ of states, we denote by σ_i the state q_i, and by $\sigma|_i$ the suffix $q_i q_{i+1} \cdots$ of σ.

DEFINITION 3.3.– Formulae of the logic PTL and their semantics over sequences $\sigma = q_0 q_1 \cdots$ of states (with respect to a function $\lambda : Q \to 2^{\mathcal{V}}$) are inductively defined as follows:

– an atomic proposition $v \in \mathcal{V}$ is a formula and $\sigma \models v$ iff $v \in \lambda(\sigma_0)$,

– propositional combinations of formulae (by means of the operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$) are formulae and their semantics are the usual ones,

– if φ is a formula then so is $\mathrm{X}\,\varphi$ ("next φ") and $\sigma \models \mathrm{X}\,\varphi$ if and only if $\sigma|_1 \models \varphi$,

– if φ and ψ are formulae then $\varphi \;\mathrm{U}\; \psi$ ("φ until ψ") is a formula and $\sigma \models \varphi \;\mathrm{U}\; \psi$ if and only if there exists some $k \in \mathbb{N}$ such that $\sigma|_k \models \psi$ and $\sigma|_i \models \varphi$ for all i with $0 \leqslant i < k$.

The set $\mathit{Mod}(\varphi)$ of the *models* of a formula φ of PTL is the set of those state sequences σ such that $\sigma \models \varphi$.

Formula φ is *valid* if $\sigma \models \varphi$ holds for all σ. It is *satisfiable* if $\sigma \models \varphi$ holds for some σ. Formula φ is *valid in a Kripke structure* $\mathcal{K}$, written $\mathcal{K} \models \varphi$, if $\sigma \models \varphi$ holds for all runs σ of $\mathcal{K}$.

The formula $\varphi \;\mathrm{U}\; \psi$ requires that ψ eventually becomes true and that φ holds at least until that happens. Other useful formulae are defined as abbreviations. Thus, $\mathrm{F}\,\varphi$ ("eventually φ") is defined as $\mathrm{true} \;\mathrm{U}\; \varphi$, and holds true of σ if φ is true of some suffix of σ. The dual formula $\mathrm{G}\,\varphi$ ("always φ") is defined as $\neg\,\mathrm{F}\,\neg\varphi$ and requires that φ holds true of all suffixes of σ. Finally, the formula $\varphi \;\mathrm{W}\; \psi$ ("φ unless ψ") abbreviates $(\varphi \;\mathrm{U}\; \psi) \vee \mathrm{G}\,\varphi$. It states that φ stays true while ψ is false: if ψ remains false forever then φ must hold true of all suffixes.

The PTL formula $\mathrm{G}\,\mathrm{F}\,\varphi$ asserts that for every suffix $\sigma|_i$ there exists a suffix $\sigma|_j$, where $j \geqslant i$, that satisfies φ. In other words, φ has to be true infinitely often. Dually, the formula $\mathrm{F}\,\mathrm{G}\,\varphi$ asserts that φ will eventually stay true.

The notions of (general) validity and satisfiability of PTL are standard. More important for the purposes of model checking is the notion of *system validity*: a property expressed by a PTL formula holds for a system if it is true for all of its runs. For example, the following formulae express typical correctness properties of a system for managing a resource shared between two processes. In writing these properties, we assume that the propositions req_i and own_i (for $i = 1, 2$) represent the states in which process i has requested (respectively, obtained) the resource.

G $\neg(\mathit{own}_1 \wedge \mathit{own}_2)$. This formula describes mutual exclusion for access to the resource: at no moment do both processes own the resource. More generally, formulae of the form $\mathrm{G}\,P$, for a non-temporal formula P, express invariants.

$\mathbf{G}(\mathit{req}_1 \Rightarrow \mathbf{F}\, \mathit{own}_1)$. Every request of access to the resource by process 1 will eventually be honored in the sense that process 1 will be granted access to the resource. Formulae of the form $\mathrm{G}(P \Rightarrow \mathrm{F}\, Q)$, for non-temporal formulae P and Q, are often called *response properties* [MAN 90]. We have encountered a formula of this form in the lift example; see Formula (3.1).

$\mathbf{G}\,\mathbf{F}(\mathit{req}_1 \wedge \neg(\mathit{own}_1 \vee \mathit{own}_2)) \Rightarrow \mathbf{G}\,\mathbf{F}\, \mathit{own}_1$. This formula is weaker than the preceding one as it requires process 1 to obtain the resource infinitely often provided that it is requested infinitely often at a time when the resource is free. For example, the previous property is impossible to satisfy (together with the basic requirement of mutual exclusion) if the second process never releases the resource after it obtained it, whereas the present property states no obligation in such a case. Formulae of the shape $\mathrm{G}\,\mathrm{F}\, P \Rightarrow \mathrm{G}\,\mathrm{F}\, Q$ can also be used to express assumptions of (strong) fairness.

$\mathbf{G}(\mathit{req}_1 \wedge \mathit{req}_2 \Rightarrow (\neg \mathit{owns}_2\ \mathbf{W}\ (\mathit{owns}_2\ \mathbf{W}\ (\neg \mathit{owns}_2\ \mathbf{W}\ \mathit{owns}_1))))$. When both processes request the shared resource, process 2 will obtain the resource at most once before process 2 gets access. This property, called *one-bounded overtaking*, is an example of a precedence property, as it expresses requirements on the relative ordering of events. Intuitively, the formula asserts the existence of four (possibly empty or infinite) intervals during which the different propositions hold.

EXTENSIONS OF PTL.– The logic PTL is generally recognized as a useful language for formulating correctness properties of systems. Nevertheless, its expressive power is limited, and several authors have proposed extensions of PTL by additional operators. In particular, the modalities of PTL are all directed towards the future, and we can obviously define symmetric operators such as $\mathrm{P}\, \varphi$, which asserts that φ was true at some preceding instant of time. Kamp [KAM 68] has shown that this extension does not increase the expressive power of the logic (see also [GAB 94] for generalizations of this result) as far as system validity is concerned: every formula containing past time modalities can be rewritten into a corresponding "future-only" formula that describes the same property. For example, the formula $\mathrm{G}(\mathit{own}_1 \Rightarrow \mathrm{P}\, \mathit{req}_1)$ which asserts that the resource is obtained only in response to a preceding request is equivalent with respect to system validity to the formula $\neg \mathit{own}_1\ \mathrm{W}\ \mathit{req}_1$. This observation has been used to justify the elimination of past time operators from model checking tools. Nevertheless, past time modalities can increase the readability of formulae, and they can be (exponentially) more succinct [LAR 02].

PTL can also be extended by operators corresponding to regular expressions [WOL 83] or, equivalently, by operators defined as smallest or largest fixed points or by quantification over atomic propositions. We will observe a close relationship between PTL and (ω-)regular languages in sections 3.4.3 and 3.4.4, which explains that "regular" operators can be added to PTL without significant complications in model checking algorithms. The language PSL [PSL 04] (*property specification language*), which was recognized as an IEEE standard in 2005, is based on an extension of PTL by past time and regular modalities.

3.4.2. *Branching-time temporal logic*

Whereas formulae of linear-time temporal logic are evaluated over the runs of a system, branching-time temporal logic makes assertions about the system as a whole. In particular, besides expressing properties that should be true for all system executions, it is also possible to assert the existence of runs satisfying certain conditions. For example, the property of reinitializability states that for every reachable state there exists some path leading back to an initial state. The following definition introduces the branching-time logic CTL (*computation tree logic*), which has been very popular for model checking.

DEFINITION 3.4.– *The formulae of the logic* CTL *and their semantics, with respect to a state q of a Kripke structure $\mathcal{K}$, are defined as follows:*

– an atomic proposition $v \in \mathcal{V}$ is a formula and $\mathcal{K}, q \models v$ if $v \in \lambda(q)$,

– propositional combinations of formulae (by means of the operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$) are formulae and their semantics are the usual ones,

– if φ is a formula, then so is $\mathrm{EX}\,\varphi$ *and $\mathcal{K}, q \models \mathrm{EX}\,\varphi$ if and only if there exist e and q' with $(q, e, q') \in \delta$ and $\mathcal{K}, q' \models \varphi$,*

– if φ is a formula, then so is $\mathrm{EG}\,\varphi$ *and $\mathcal{K}, q \models \mathrm{EG}\,\varphi$ if and only if there exists some path $q = q_0 \overset{e_0}{\Rightarrow} q_1 \cdots$ such that $\mathcal{K}, q_i \models \varphi$ for all $i \in \mathbb{N}$,*

– if φ and ψ are formulae, then φ EU *ψ is a formula and $\mathcal{K}, q \models \varphi$* EU *$\psi$ if and only if there exists some path $q = q_0 \overset{e_0}{\Rightarrow} q_1 \cdots$ and some $k \in \mathbb{N}$ such that $\mathcal{K}, q_k \models \psi$ and $\mathcal{K}, q_i \models \varphi$ for all i with $0 \leqslant i < k$.*

The satisfaction set *$[\![\varphi]\!]_{\mathcal{K}}$ of a formula φ in a Kripke structure $\mathcal{K}$ is the set of all states $q \in Q$ such that $\mathcal{K}, q \models \varphi$. Formula φ is* valid in a Kripke structure *$\mathcal{K}$, written $\mathcal{K} \models \varphi$, if $\mathcal{K}, q \models \varphi$ for all initial states $q \in I$, that is, if $I \subseteq [\![\varphi]\!]_{\mathcal{K}}$.*

Formula φ is valid *(*satisfiable*) if it is valid in every (some) Kripke structure $\mathcal{K}$.*

The modal operators of CTL combine temporal references with quantification over paths. Further connectives can be defined as abbreviations. Thus, $\mathrm{EF}\,\varphi$ abbreviates true EU φ and asserts that φ will become true along some path starting at the state of evaluation. The formulae $\mathrm{AX}\,\varphi$ and $\mathrm{AG}\,\varphi$ are defined as $\neg\,\mathrm{EX}\,\neg\varphi$ and $\neg\,\mathrm{EF}\,\neg\varphi$, and state that φ holds at all immediate successors, respectively at all states reachable from the state of evaluation. Similar definitions can be given to introduce the operators AF, AU, and AW. Operators of the shape A _ express *universal properties* that have to hold along all possible paths starting at the current state, whereas the operators E _ express *existential properties*. In particular, AG P asserts that the non-temporal formula P is

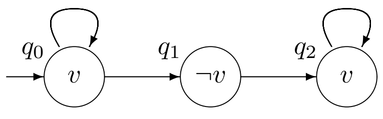


Figure 3.4. *A Kripke structure satisfying* $\mathrm{F\,G}\,v$ *but not* $\mathrm{AF\,AG}\,v$

a system invariant. The formula $\mathrm{AG}(req_1 \Rightarrow \mathrm{EF}\,own_1)$ requires that every request for the resource by process 1 may be followed by getting access to the resource, although there may also be some executions along which the process is never granted access. Similarly, system validity of the formula $\mathrm{AG\,EF}\,init$ (for a suitable proposition $init$) means that the system can be reinitialized from every reachable state.

Although they are interpreted over structures of different shape, the expressiveness of PTL and CTL can be compared based on the notion of system validity: a PTL formula φ and a CTL formula ψ correspond to each other if they are valid in the same Kripke structures. Obviously, PTL cannot define existential properties such as $\mathrm{AG\,EF}\,init$. Perhaps more surprisingly, there also exist PTL formulae for which there is no corresponding CTL formula, and therefore the expressive power of PTL and CTL is incomparable [LAM 80]. Thus, reactivity properties $\mathrm{G\,F}\,P \Rightarrow \mathrm{G\,F}\,Q$ do not have a counterpart in CTL. A simpler example is provided by the Kripke structure $\mathcal{K}$ shown in Figure 3.4 (transition labels have been omitted). It is easy to see that $\mathcal{K} \models \mathrm{F\,G}\,v$: that property is true for the run of $\mathcal{K}$ that loops at q_0, but also for the executions that eventually move to state q_2. On the other hand, the CTL formula[1] $\mathrm{AF\,AG}\,v$ is not valid in the Kripke structure $\mathcal{K}$. To see this, it is enough to observe that the formula $\mathrm{AG}\,v$ is never satisfied along the run of $\mathcal{K}$ that loops at q_0 because from q_0 we can always move to state q_1, which does not satisfy v.

OTHER BRANCHING-TIME LOGICS.– This relative lack of expressiveness of CTL is due to the strict alternation between path quantifiers (A, E) and temporal operators (X, G, F, U, W). The logic CTL* relaxes this constraint and (strictly) subsumes the logics PTL and CTL. For example, the CTL* formula $\mathrm{A\,F\,G}\,v$ corresponds to the PTL formula $\mathrm{F\,G}\,v$. See [EME 90, EME 86a, VAR 01] for more in-depth comparisons between linear-time and branching-time temporal logics.

The *modal μ-calculus* [KOZ 83, STI 01] is based on defining temporal connectives by their characteristic recursive equivalences. For example, let us consider

$$\mathrm{EG}\,\varphi \;\Leftrightarrow\; \varphi \wedge \mathrm{EX\,EG}\,\varphi \quad \text{and} \quad \mathrm{EG}_2\,\varphi \;\Leftrightarrow\; \varphi \wedge \mathrm{EX\,EX\,EG}_2\,\varphi.$$

1. Obviously, this observation does not prove that there is no other CTL formula that corresponds to $\mathrm{F\,G}\,v$. The proof of this assertion requires a more detailed argument [LAM 80].

The left-hand equivalence is valid for the connective EG of CTL, the one on the right-hand side characterizes an operator EG_2 that requires the existence of a path such that φ is true at all states with an even distance from the original point of evaluation. (Formula φ can be true or false at the other states.) It can be shown that such an operator is not definable in CTL or CTL*. In the μ-calculus it can be defined by the formula $\nu X : \varphi \wedge \mathrm{EX}\,\mathrm{EX}\,X$.

The temporal logic ATL (*alternating time temporal logic*) [ALU 97] refines the quantification over the paths in a Kripke structure by referring to the labels of transitions, interpreted as identifying the processes of a system. For example, ATL formulae can be written to express that the controller by itself can guarantee the mutual exclusion of access to the resource, or that the controller and the first process can conspire to exclude the second process from accessing it. The logic ATL is therefore particularly useful for the specification and analysis of open systems formed as the composition of independent components.

3.4.3. *ω-automata*

The algorithm for the verification of invariants presented in section 3.3.2 is conceptually simple, but it is not immediately clear how to generalize it for the verification of arbitrary temporal properties. Transition systems, even if they are finite, usually generate an infinite set of runs, each of which is an infinite sequence of states. Certain verification algorithms are grounded in a close correspondence between temporal logics and finite automata operating on infinite objects (sequences or trees). In contrast to the more "declarative" presentation of properties by formulae, automata provide a more "operational" description, and are amenable to decision procedures. We are now going to study the principles of this theory, which goes back to the work of Büchi [BÜC 62], Muller [MUL 63], and Rabin [RAB 69], but limit the discussion mainly to Büchi automata. Much more detailed information can be found in the excellent articles by Thomas [THO 97] or Vardi *et al.* [VAR 95, KUP 94]. As before, we assume given a set $\mathcal{V}$ of atomic propositions, which defines the alphabet $2^{\mathcal{V}}$ of our automata.

DEFINITION 3.5.– *A* Büchi automaton $\mathcal{B} = (L, L_0, \delta, F)$ *is given by a finite set L of locations, a set $L_0 \subseteq L$ of initial locations, a relation $\delta \subseteq L \times 2^{\mathcal{V}} \times L$ and a set $F \subseteq L$ of accepting locations.*

A run *of $\mathcal{B}$ over a sequence $\sigma = s_0 s_1 \cdots$ where $s_i \subseteq 2^{\mathcal{V}}$ is a sequence $\rho = l_0 l_1 \cdots$ of locations $l_i \in L$ such that $l_0 \in L_0$ is an initial location and $(l_i, s_i, l_{i+1}) \in \delta$ holds for all $i \in \mathbb{N}$. The run ρ is* accepting *if it contains an infinite number of locations $l_k \in F$.*

The language $\mathcal{L}(\mathcal{B})$ *of $\mathcal{B}$ is the set of sequences σ for which there exists an accepting run of $\mathcal{B}$.*

Observe that the structure of a Büchi automaton is just that of a non-deterministic finite automaton (NFA). The only difference is in the definition of the acceptance

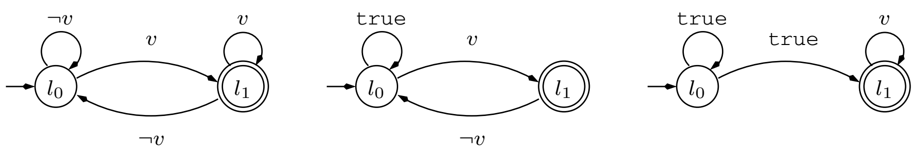


Figure 3.5. *Three Büchi automata*

condition which requires that the run passes infinitely often through an accepting location. A ω-language L (i.e., a set of ω-sequences σ) is called *ω-regular* if it is generated by a Büchi automaton, i.e. if $L = \mathcal{L}(\mathcal{B})$ for some Büchi automaton $\mathcal{B}$.

Büchi automata, as we have defined them, operate on ω-sequences of subsets of $\mathcal{V}$, and this is in close correspondence with the interpretation of (linear-time) temporal logic over Kripke structures. Indeed, any run $q_0 \overset{e_0}{\Rightarrow} q_1 \overset{e_1}{\Rightarrow} \cdots$ of a Kripke structure $\mathcal{K}$ can be identified with the corresponding sequence $\lambda(q_0)\lambda(q_1)\cdots$ where λ is the propositional valuation of states of $\mathcal{K}$. In this sense, PTL formulae and Büchi automata operate over the same class of structures.

Figure 3.5 shows three Büchi automata, each of which contains two locations l_0 and l_1, of which l_0 is initial and l_1 is accepting. The transitions are labeled with propositional formulae in an obvious manner: for example, the transition label $\neg v$ represents those subsets of $\mathcal{V}$ that do not contain v. The left-hand automaton is deterministic because for any propositional interpretation $s \subseteq \mathcal{V}$, the successor of both locations is uniquely determined. It accepts precisely those sequences where v is true infinitely often and, intuitively, corresponds to the PTL formula $\mathrm{G}\,\mathrm{F}\,v$. The middle automaton is non-deterministic because there is a choice between staying at l_0 or moving to l_1 when reading an interpretation satisfying v while in location l_0. Any sequence accepted by this automaton must contain an infinite number of interpretations satisfying v, followed by an interpretation satisfying $\neg v$. The language of this automaton is therefore characterized by the PTL formula[2] $\mathrm{G}\,\mathrm{F}(v \wedge \mathrm{X}\,\neg v)$. It is not difficult to find a deterministic Büchi automaton that accepts the same language. The right-hand automaton is also non-deterministic, and any sequence accepted by it must terminate in a sequence of states satisfying v. This language is also described by the PTL formula $\mathrm{F}\,\mathrm{G}\,v$. It can be shown [THO 97] that there is no deterministic Büchi automaton defining this language. Intuitively, the reason is that an infinite "prophecy" is required when choosing to move to location l_1.

In particular, and unlike standard non-deterministic finite automata over finite words, non-deterministic Büchi automata are strictly more expressive than deterministic ones. Apart from this difference, the theory of ω-regular languages closely parallels

2. Let us note that an equivalent formula is $\mathrm{G}\,\mathrm{F}\,v \wedge \mathrm{G}\,\mathrm{F}\,\neg v$.

that of ordinary regular languages. Specifically, we will make use of the decidability of the emptiness problem.

THEOREM 3.1.– *For a Büchi automaton $\mathcal{B} = (L, L_0, \delta, F)$, the emptiness problem $\mathcal{L}(\mathcal{B}) = \emptyset$ can be decided in linear-time in the size of the automaton.*

Proof. Since L is a finite set, $\mathcal{L}(\mathcal{B}) = \emptyset$ if and only if there exists locations $l_0 \in L_0$ and $l_f \in F$ such that l_f is reachable from l_0 and (non-trivially) from itself, i.e. there exist finite words x and $y \neq \varepsilon$ over $2^{\mathcal{V}}$ such that $l_0 \overset{x}{\Longrightarrow} l_f$ and $l_f \overset{y}{\Longrightarrow} l_f$. The existence of such a cycle in the graph of $\mathcal{B}$ can be decided in linear-time, for example by using the algorithm of Tarjan and Paige [TAR 72] that enumerates the strongly connected components of $\mathcal{B}$ and checking that some SCC contains an accepting location. □

It follows from the proof of Theorem 3.1 that every non-empty ω-regular language contains a word of the form xy^ω where x and y are finite words and y^ω denotes infinite repetition of the word y.

Further important results about ω-regular languages show closure under Boolean operation and projection. Closure under union and projection are easy to prove, using essentially the same automaton constructions as for the corresponding results for NFA over finite words. Closure under intersection is essentially proved by constructing the product automata for the two original languages, although some care must be taken to define the acceptance condition. However, proving closure under complementation is difficult. The standard proof known from NFA first constructs a deterministic finite automaton, and this fails for Büchi automata because they cannot be determinized. The original proof by Büchi was non-constructive and combinatorial in nature, and a series of papers over the following 25 years established explicit constructions, culminating in optimal constructions of complexity $O(2^{n \log n})$ by Safra [SAF 88] and by Kupferman and Vardi [KUP 97b].

OTHER TYPES OF ω-AUTOMATA.– There exist many other types of non-deterministic ω-automata that differ essentially in the definition of the acceptance condition. In particular, generalized Büchi automata are of the shape $\mathcal{B} = (L, L_0, \delta, \mathcal{F})$, with an acceptance condition $\mathcal{F} = \{F_1, \ldots, F_n\}$ of sets F_i of locations. A run is accepted if every set F_i is visited infinitely often. Using a counter modulo n, it is not hard to simulate a generalized Büchi automaton by a standard one [VAR 94]. For Muller automata, the acceptance condition is also a set $\mathcal{F} \subseteq 2^L$, and a run is accepted if the set of all locations that appear infinitely often is an element of $\mathcal{F}$. Muller automata again define the class of ω-regular languages. Rabin and Streett automata are special cases of Muller automata. Beyond independent interest in these classes of automata, they are used in Safra's complementation proof. The more recent complementation by Kupferman and Vardi extends rather smoothly to different kinds of non-deterministic automata [KUP 05].

Alternating automata [MUL 88, VAR 95, KUP 97b] differ from Büchi automata in that several locations can be simultaneously active during a run. The transition relation of an alternating automaton can be specified using propositional formulae built from the propositions in $\mathcal{V}$ and the locations, where the latter are restricted to occur positively. For example,

$$\delta(q_1) = \big(v \wedge w \wedge q_1 \wedge (q_3 \vee q_4)\big) \ \vee \ \big(\neg w \wedge (q_1 \vee q_2)\big),$$

specifies that if the location q_1 is currently active and the current interpretation satisfies $v \wedge w$, then q_1 and q_3 or q_1 and q_4 will be activated after the transition. If the current interpretation satisfies $\neg w$, then q_1 or q_2 will be active. Otherwise, the automaton blocks. Alternating automata thus combine the non-determinism of Büchi automata and parallelism, and their runs are infinite trees labeled with locations (or directed acyclic graphs, *dags*, if multiple copies of the same location are merged). With suitable acceptance conditions, these automata again define ω-regular languages but they can be exponentially more succinct than Büchi automata. On the other hand, the emptiness problem becomes of exponential complexity, and this trade-off can be useful in certain model checking applications [HAM 05]. Alternating automata are closely related to certain logical games, which have also received much attention during recent years.

3.4.4. *Automata and PTL*

We have already pointed out some (informal) correspondence between automata and PTL formulae, and indeed formula φ can be considered as defining the ω-language $\mathit{Mod}(\varphi)$. It is therefore quite natural to compare the expressive power of formulae and automata, and we will now sketch the construction of a generalized Büchi automaton $\mathcal{B}_\varphi$ that specifically accepts the models of the PTL formula φ.

The construction avoids the difficult complementation of Büchi automata by using a "global" algorithm that considers all subformulae of φ simultaneously. More precisely, let $\mathcal{C}(\varphi)$ denote the set of subformulae ψ of φ and their complements $\overline{\psi}$, identifying $\neg\neg\psi$ and ψ. It is easy to see that the size of set $\mathcal{C}(\varphi)$ is linear in the length of formula φ.

The locations of $\mathcal{B}_\varphi$ correspond to subsets of $\mathcal{C}(\varphi)$, with the intuitive meaning that whenever $\rho = l_0 l_1 \cdots$ is an accepting run of $\mathcal{B}_\varphi$ over σ then $\sigma|_i$ satisfies all formulae in l_i, for all $i \in \mathbb{N}$. Formally, the set L of locations of $\mathcal{B}_\varphi$ consists of all sets $L \subseteq \mathcal{C}(\varphi)$ that satisfy the following "healthiness conditions":

– for all formulae $\psi \in \mathcal{C}(\varphi)$, either $\psi \in L$ or $\overline{\psi} \in L$ but not both;

– if $\psi_1 \vee \psi_2 \in \mathcal{C}(\varphi)$, then $\psi_1 \vee \psi_2 \in l$ if and only if $\psi_1 \in l$ or $\psi_2 \in l$;

– similar conditions hold for the other propositional connectives;

– if $\psi_1 \ \mathrm{U} \ \psi_2 \in l$, then $\psi_1 \in l$ or $\psi_2 \in l$;

– if $\neg(\psi_1 \text{ U } \psi_2) \in l$, then $\overline{\psi_2} \in l$.

The initial locations of $\mathcal{B}_\varphi$ are those locations containing the formula φ. The transition relation δ of $\mathcal{B}_\varphi$ consists of the triples (l, s, l') that satisfy the following conditions:

– $s = l \cap \mathcal{V}$: the state s must satisfy precisely those atomic propositions "promised" by the source location l;

– if $\text{X}\,\psi \in l$, then $\psi \in l'$ and if $\neg\,\text{X}\,\psi \in l$, then $\overline{\psi} \in l'$;

– if $\psi_1 \text{ U } \psi_2 \in l$ and $\overline{\psi_2} \in l$, then $\psi_1 \text{ U } \psi_2 \in l'$;

– if $\neg(\psi_1 \text{ U } \psi_2) \in l$ and $\psi_1 \in l$, then $\neg(\psi_1 \text{ U } \psi_2) \in l'$.

The last two conditions can be explained by the "recursion law" for the U operator:

$$\psi_1 \text{ U } \psi_2 \;\Leftrightarrow\; \psi_2 \vee (\psi_1 \wedge \text{X}(\psi_1 \text{ U } \psi_2)). \tag{3.2}$$

The conditions above define the initial locations and the transition relation, it remains to define the acceptance condition of the generalized Büchi automaton $\mathcal{B}_\varphi$. The intuitive idea is to make sure that whenever a location that contains formula $\psi_1 \text{ U } \psi_2$ appears in an accepting run of $\mathcal{B}_\varphi$ it will be followed by a location containing ψ_2. Let therefore $\psi_1^1 \text{ U } \psi_2^1, \ldots, \psi_1^k \text{ U } \psi_2^k$ be all formulae of this shape in $\mathcal{C}(\varphi)$. The acceptance condition $\mathcal{F} = \{F_1, \ldots, F_k\}$ of automaton $\mathcal{B}_\varphi$ contains a set of locations F_i for each formula $\psi_1^i \text{ U } \psi_2^i$, where $l \in F_i$ if and only if $\psi_2^i \in l$ or $\psi_1^i \text{ U } \psi_2^i \notin l$.

For example, the automaton shown in Figure 3.6 results from an application of the above construction to the formula $\varphi \equiv (p \text{ U } q) \vee (\neg p \text{ U } q)$. We have omitted the transition labels, which are just given by the sets of atomic propositions that appear in the source states. The acceptance condition consists of sets $F_1 = \{l_1, l_3, l_4, l_5, l_6\}$ and $F_2 = \{l_1, l_2, l_3, l_5, l_6\}$, corresponding to the two subformulae $p \text{ U } q$ and $\neg p \text{ U } q$. For example, set F_1 serves to exclude runs that terminate in the loop at location l_2 because these runs "promise" $p \text{ U } q$ without ever satisfying q.

The above construction of the automaton $\mathcal{B}_\varphi$ is similar to that of a tableau for the logic PTL [WOL 83]. The following correctness theorem is due to Vardi *et al.*; see for example [VAR 94, GER 95].

PROPOSITION 3.1.– *For any* PTL *formula φ of length n there is a Büchi automaton $\mathcal{B}_\varphi$ with $2^{O(n)}$ locations such that $\mathcal{L}(\mathcal{B}_\varphi) = Mod(\varphi)$.*

Together, Theorem 3.1 and Proposition 3.1 imply that the satisfiability and validity problems of the logic PTL are decidable in exponential time: formula φ is satisfiable if and only if $\mathcal{L}(\mathcal{B}_\varphi) \neq \emptyset$, and it is valid if and only if $\mathcal{L}(\mathcal{B}_{\neg\varphi}) = \emptyset$. On the other hand, as Sistla and Clarke [SIS 87] have shown that these problems are PSPACE-complete, we can therefore not hope for a significantly more efficient algorithm in the general case.

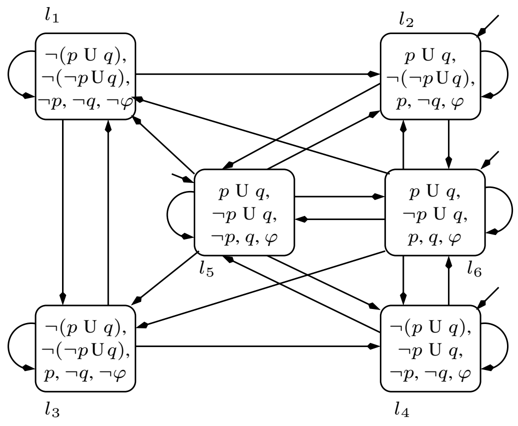

l_1
$\neg(p \text{ U } q)$, $\neg(\neg p \text{U} q)$, $\neg p, \neg q, \neg\varphi$
l_2
$p \text{ U } q$, $\neg(\neg p \text{U} q)$, $p, \neg q, \varphi$
$p \text{ U } q$, $\neg p \text{ U } q$, $\neg p, q, \varphi$
l_5
$p \text{ U } q$, $\neg p \text{ U } q$, p, q, φ
l_6
$\neg(p \text{ U } q)$, $\neg(\neg p \text{U} q)$, $p, \neg q, \neg\varphi$
l_3
$\neg(p \text{ U } q)$, $\neg p \text{ U } q$, $\neg p, \neg q, \varphi$
l_4

Figure 3.6. *Büchi automaton $\mathcal{B}_\varphi$ for $\varphi \equiv (p \text{ U } q) \vee (\neg p \text{ U } q)$*

However, the simple construction of $\mathcal{B}_\varphi$ described above systematically generates an automaton of exponential size. Constructions used in practice [DAN 99, GAS 01] attempt to avoid this blowup whenever possible. Let us finally note that the construction of an alternating automaton for a PTL formula is of linear complexity [VAR 95].

Proposition 3.1 naturally leads to the reciprocal question whether every ω-regular language can be defined by a PTL formula. The answer was already given by Kamp [KAM 68] in 1968: he showed that PTL corresponds to the monadic first-order theory of linear orders. This logical language only contains unary predicate (and no function) symbols, the equality predicate $=$, the relation $<$, interpreted over the natural numbers; see for example [GAB 94, THO 97]. However, Büchi [BÜC 62] proved that the class of ω-regular languages correspond to the analogous fragment of second-order logic, and this language is strictly more expressive. For example, the linear-time counterpart G_2 to the connective EG_2 considered at the end of section 3.4.2 where $\text{G}_2\ \varphi$ is true for σ if φ holds for all suffixes $\sigma|_{2n}$ with even offset is definable by a Büchi automaton but not by a PTL formula [WOL 83]. Starting from this observation, extensions of PTL by "grammar operators" that directly correspond to ω-regular expressions [WOL 83], by fixed-point definitions similar to the modal μ-calculus [STI 01], or by quantification over atomic propositions, have been proposed.

Automata corresponding to branching-time temporal logics can also be defined, and they operate over infinite trees [KUP 94, THO 97]. We do not present the details here because they are not necessary for the model checking algorithms for branching-time logics that we will describe in section 3.5.2.

3.5. Model checking algorithms

Given a Kripke structure $\mathcal{K}$ and a formula φ of temporal logic, the model checking problem is to determine whether φ is valid in $\mathcal{K}$, written $\mathcal{K} \models \varphi$ (see Definitions 3.3

and 3.4). Beyond a yes/no answer to this question, model checking tools generally attempt to explain their verdict. For example, a PTL model checker will produce a counter-example, i.e. a run of $\mathcal{K}$ that does not satisfy φ if $\mathcal{K} \not\models \varphi$. The model checking problems for the logics that we have considered so far are decidable when $\mathcal{K}$ is a finite-state system, and we will explain in this section the principles of model checking algorithms for PTL and for CTL.

Observe that the model checking problem has two parameters, $\mathcal{K}$ and φ. We can therefore imagine two basic strategies for its solution: *global* algorithms recurse on the syntax of formula φ and evaluate its subformulae at the states of $\mathcal{K}$ in order to determine the satisfaction of φ. *Local* algorithms recurse on the structure of $\mathcal{K}$: they explore the parts of $\mathcal{K}$ that contribute to evaluating φ, much like how the algorithm of section 3.3.2 uses graph search to determine satisfaction of a proposed invariant. Global algorithms are traditionally used for CTL model checking, while PTL model checking is mostly based on local algorithms because system validity for PTL does not decompose along the formula structure. For example, $\mathcal{K} \models \varphi \vee \psi$ does not require either $\mathcal{K} \models \varphi$ or $\mathcal{K} \models \psi$.

3.5.1. *Local* PTL *model checking*

The translation from PTL formulae to Büchi automata introduced in section 3.4.4 provides the basis for a PTL model checking algorithm, refining the procedure for satisfiability checking described there. Indeed, $\mathcal{K} \not\models \varphi$ if and only if there exists a run of $\mathcal{K}$ that does not satisfy φ. If we consider $\mathcal{K}$ as a Büchi automaton (with trivial acceptance condition) that defines the set $\mathcal{L}(\mathcal{K})$ of runs of $\mathcal{K}$, we arrive at the following chain of equivalences:

$$\mathcal{K} \not\models \varphi \quad \Longleftrightarrow \quad \mathcal{L}(\mathcal{K}) \cap Mod(\neg\varphi) \neq \emptyset \quad \Longleftrightarrow \quad \mathcal{L}(\mathcal{K}) \cap \mathcal{L}(\mathcal{B}_{\neg\varphi}) \neq \emptyset,$$

where the second equivalence is justified by Proposition 3.1. The last problem in this chain can be solved by determining language emptiness for the product of $\mathcal{K}$ and $\mathcal{B}_{\neg\varphi}$.

More formally, assume that $\mathcal{K} = (Q, I, E, \delta_{\mathcal{K}}, \lambda)$ is a Kripke structure and that $\mathcal{B}_{\neg\varphi} = (L, L_0, \delta_{\mathcal{B}}, F)$ is the Büchi automaton[3] corresponding to the negation of φ. The model checking algorithm operates on pairs (q, l) of states q of $\mathcal{K}$ and locations l of $\mathcal{B}_{\neg\varphi}$. The initial pairs consist of initial states and locations of the two components. The successors of a pair (q, l) are all pairs (q', l') such that

– q' is a successor state of q in $\mathcal{K}$, i.e. $(q, e, q') \in \delta_{\mathcal{K}}$ for some $e \in E$, and

– l' is a possible successor of l under the interpretation of the atomic propositions determined by q, i.e. $(l, \lambda(q), l') \in \delta_{\mathcal{B}}$.

3. Here we assume that $\mathcal{B}_{\neg\varphi}$ is an ordinary (not generalized) Büchi automata. As noted in section 3.4.3, the translation from generalized Büchi automata to ordinary Büchi automata is of polynomial complexity.

```
void mc_ptl(Kripke ks, Buchi aut) {
   void dfs(Boolean search_cycle) {
     Pair p = stack.top();
     foreach (Pair q in p.successors()) {
        if (search_cycle && (q == seed))
        { report acceptance cycle and exit }
        if (!seen.contains(q, search_cycle)) {
           stack.push(q); seen.add(q, search_cycle);
           dfs(search_cycle);
           if (!search_cycle && q.isAccepting()) {
              seed = q; dfs(true);
     }  }  }
     stack.pop();
   }
   // initialization
   Stack stack = new Stack(); Set seen = new Set(); seed = null;
   foreach (Pair p in makeInitialPairs(ks, aut)) {
      stack.push(p); seen.add(p, false);
      dfs(false);
}  }
```

Figure 3.7. *On-the-fly* PTL *model checking algorithm*

The two automata therefore take simultaneous transitions. The acceptance condition of the product is determined by that of the Büchi automaton: a pair (q, l) is accepting if $l \in F$ is an accepting state of $\mathcal{B}_{\neg\varphi}$.

The Büchi automaton $\mathcal{A}$ thus defined recognizes the language $\mathcal{L}(\mathcal{K}) \cap \mathcal{L}(\mathcal{B}_{\neg\varphi})$, and in particular $\mathcal{K} \models \varphi$ if and only if $\mathcal{L}(\mathcal{A}) = \emptyset$. By Theorem 3.1, this condition can be verified in linear-time in the size of $\mathcal{A}$ by searching for an accepting cycle. Because the size of $\mathcal{A}$ is proportional to the product of the sizes of $\mathcal{K}$ and of $\mathcal{B}_{\neg\varphi}$, the explicit construction of $\mathcal{A}$ is prohibitive in practice. Courcoubetis *et al.* [COU 92] invented an algorithm that constructs $\mathcal{A}$ "on-the-fly", that is, during the search for an acceptance cycle. This algorithm appears in Figure 3.7. The current state of the search is represented by a stack of pairs. The algorithm combines two depth-first search algorithms: starting from an initial pair, the procedure `dfs` explores a path of the product automaton. When this search completes, it backtracks to the last accepting pair q and then initializes a second search (indicated by the parameter `search_cycle` set to true) of a path that leads back to q. Courcoubetis *et al.* proved that this algorithm will report an acceptance cycle if the product automaton contains one, although it may not find all existing cycles (even if the search were to continue after the first reported cycle).

The algorithm avoids the construction of the product automaton: the stack only contains the prefix of the currently explored path, and the set `seen` stores the pairs that have already been visited during the search. A possible optimization would store only a subset of these pairs by trading some redundant search for less memory consumption. Just as for the algorithm for invariant checking of Figure 3.3, when the procedure `dfs` finds an acceptance cycle, it is represented in the search stack and can be displayed as a counter-example.

The complexity of the algorithm in Figure 3.7 is still linear in the size of the product of $\mathcal{K}$ and $\mathcal{B}_{\neg\varphi}$; by Theorem 3.1 it is therefore linear in the size of $\mathcal{K}$ and exponential in the size of φ. In practice, the linear factor is often more problematic because correctness properties tend to be short formulae.

For systems where the size of the product automaton exceeds a few million pairs, the set `seen` no longer fits into the main memory. In order to reduce memory consumption, it is possible to store signatures of states as computed by a hash function, rather than the states themselves [HOL 98]. When doing so, different pairs may generate the same signature, leading to a premature termination of the algorithm. This risk can be reduced by repeating the verification, using different hash functions.

Different algorithms for the model checking of PTL formulae, still based on ω-automata, have been proposed by Couvreur [COU 99] and by Geldenhuys and Valmari [GEL 04]; Schwoon and Esparza [SCH 05] discuss the pros and cons of these algorithms in more detail. The on-the-fly algorithm based on alternating automata [HAM 05] is intended for the verification of large formulae.

3.5.2. *Global CTL model checking*

Global model checking algorithms recurse on the syntax of the formula to be verified. We consider here an algorithm for the branching-time logic CTL that calculates the satisfaction sets $[\![\psi]\!]_{\mathcal{K}}$ (see Definition 3.4) for the subformulae of the formula φ of interest. Recall that in CTL, φ is system valid if $I \subseteq [\![\varphi]\!]_{\mathcal{K}}$, where I is the set of initial states of $\mathcal{K}$. The first clauses for the recursive definition of $[\![\varphi]\!]_{\mathcal{K}}$ are quite obvious:

$$\begin{array}{rcl}
[\![v]\!]_{\mathcal{K}} & = & \{q \in Q : v \in \lambda(q)\} \quad \text{for } v \in \mathcal{V} \\
[\![\neg\psi]\!]_{\mathcal{K}} & = & Q \setminus [\![\psi]\!]_{\mathcal{K}} \\
[\![\psi_1 \vee \psi_2]\!]_{\mathcal{K}} & = & [\![\psi_1]\!]_{\mathcal{K}} \cup [\![\psi_2]\!]_{\mathcal{K}} \\
[\![\psi_1 \wedge \psi_2]\!]_{\mathcal{K}} & = & [\![\psi_1]\!]_{\mathcal{K}} \cap [\![\psi_2]\!]_{\mathcal{K}} \\
[\![\mathrm{EX}\,\psi]\!]_{\mathcal{K}} & = & \delta^{-1}([\![\psi]\!]_{\mathcal{K}}) \\
 & = & \{q \in Q : \text{there are } e, q' \text{ such that } (q,e,q') \in \delta \text{ and } q' \in [\![\psi]\!]_{\mathcal{K}}\}
\end{array}$$

It remains to find appropriate definitions for the connectives EG and EU, and we will make use of their recursive characterizations:

$$\begin{array}{rcl}
\mathrm{EG}\,\psi & \Leftrightarrow & \psi \wedge \mathrm{EX}\,\mathrm{EG}\,\psi \\
\psi_1 \ \mathrm{EU}\ \psi_2 & \Leftrightarrow & \psi_2 \vee (\psi_1 \wedge \mathrm{EX}(\psi_1 \ \mathrm{EU}\ \psi_2)).
\end{array}$$

Using the above definitions of $[\![\_]\!]_{\mathcal{K}}$, these laws can be rewritten as

$$\begin{array}{rcl}
[\![\mathrm{EG}\,\psi]\!]_{\mathcal{K}} & = & [\![\psi]\!]_{\mathcal{K}} \cap \delta^{-1}([\![\mathrm{EG}\,\psi]\!]_{\mathcal{K}}) \\
[\![\psi_1 \ \mathrm{EU}\ \psi_2]\!]_{\mathcal{K}} & = & [\![\psi_2]\!]_{\mathcal{K}} \cup ([\![\psi_1]\!]_{\mathcal{K}} \cap \delta^{-1}([\![\psi_1 \ \mathrm{EU}\ \psi_2]\!]_{\mathcal{K}})),
\end{array}$$

yielding implicit characterizations of these sets. Indeed, $[\![\mathrm{EG}\,\psi]\!]_{\mathcal{K}}$ and $[\![\psi_1 \text{ EU } \psi_2]\!]_{\mathcal{K}}$ are respectively the greatest and the least fixed points of the following functions:

$$f_{\mathrm{EG}\,\psi} : \begin{cases} 2^Q \to 2^Q \\ S \mapsto [\![\psi]\!]_{\mathcal{K}} \cap \delta^{-1}(S) \end{cases} \qquad f_{\psi_1 \mathrm{EU} \psi_2} : \begin{cases} 2^Q \to 2^Q \\ S \mapsto [\![\psi_2]\!]_{\mathcal{K}} \cup ([\![\psi_1]\!]_{\mathcal{K}} \cap \delta^{-1}(S)). \end{cases}$$

Both functions $f_{\mathrm{EG}\,\psi}$ and $f_{\psi_1 \mathrm{EU} \psi_2}$ are monotonic and (by Tarski's fixed point theorem) therefore have least and greatest fixed points. Moreover, since Q and therefore 2^Q are finite sets, these functions are even continuous, and the fixed points can be effectively computed as the limits of the sequences

$$\begin{array}{l} Q \supseteq f_{\mathrm{EG}\,\psi}(Q) \supseteq f_{\mathrm{EG}\,\psi}(f_{\mathrm{EG}\,\psi}(Q)) \supseteq \cdots \\ \emptyset \subseteq f_{\psi_1 \mathrm{EU} \psi_2}(\emptyset) \subseteq f_{\psi_1 \mathrm{EU} \psi_2}(f_{\psi_1 \mathrm{EU} \psi_2}(\emptyset)) \subseteq \cdots \end{array}$$

Because at least one state is removed (respectively added) at each step as long as the fixed point has not been reached, the computation terminates after at most $|Q|$ iterations. The complexity of computing these functions at each iteration is linear in $|\delta|$, hence at worst quadratic in $|Q|$. The model checking algorithm requires computing the sets $[\![\psi]\!]_{\mathcal{K}}$ for all subformulae ψ of φ, whose number is linear in the length of φ. Thus, the overall complexity of this "naive"CTL model checking algorithm is linear in $|\varphi|$ and cubic in $|Q|$.

Clarke, Emerson and Sistla [CLA 86] have defined a more efficient algorithm whose complexity is linear in the product of the size of the formula and the size of $|\mathcal{K}|$ (and therefore at worst quadratic in $|Q|$). For the computation of $[\![\psi_1 \text{ EU } \psi_2]\!]_{\mathcal{K}}$, the idea is to perform a backward search starting from the states in $[\![\psi_2]\!]_{\mathcal{K}}$. For $[\![\mathrm{EG}\,\psi]\!]_{\mathcal{K}}$, the graph of $\mathcal{K}$ is first restricted to those states satisfying ψ, and the algorithm enumerates the strongly connected components (SCCs) of this subgraph. The set $[\![\mathrm{EG}\,\psi]\!]_{\mathcal{K}}$ consists of all states from where such an SCC is reachable. These states are easily enumerated using a breadth-first search algorithm.

We have observed in section 3.4.2 that fairness conditions cannot be expressed in CTL. Verification of CTL properties under fairness hypotheses therefore requires adapting the model checking algorithm. (For PTL this is not necessary because we can verify formulae *fair* $\Rightarrow \varphi$ where *fair* is a PTL encoding of the fairness conditions.) McMillan [MCM 93] proposed to characterize "fair" states of a Kripke structure using CTL formulae. A run σ of $\mathcal{K}$ is fair if it contains infinitely many states satisfying these formulae. For example, weak fairness with respect to a certain action can be expressed in this scheme by identifying the states where the action is either disabled or has just been taken. We can then define variants EG_f and EU_f of the operators EG and EU whose semantics differ by asserting the existence of a *fair* path (with respect to all fairness constraints) satisfying the temporal conditions, rather than the existence of an arbitrary path. It is easy to see that the formula $\psi_1 \ \mathrm{EU}_\mathrm{f} \ \psi_2$ is equivalent to the formula $\psi_1 \text{ EU } (\psi_2 \wedge \mathrm{EG}_\mathrm{f}\, \mathrm{true})$, and it is therefore enough to find a model checking algorithm for formulae of the form $\mathrm{EG_f}\,\psi$. For the algorithm of Clarke, Emerson and Sistla, it is enough to consider the SCCs of the subgraph of $\mathcal{K}$ containing the states satisfying ψ that contain, for each fairness constraint, some state satisfying that constraint. This information can be obtained while the SCCs are searched, and the complexity of the algorithm remains linear in the size of the model and the formula.

Global model checking algorithms can also be defined for other branching-time logics, and in particular for the propositional μ-calculus. In that case, the complexity is of the order $|\varphi| \cdot |\mathcal{K}|^{qd(\varphi)}$, where $qd(\varphi)$ denotes the maximum nesting depth of fixed point operators in φ. Emerson and Lei [EME 86b] have observed that the computation of nested fixed points of the same type (least or greatest fixed point) can be carried out simultaneously. In this way, they obtained an algorithm of complexity $|\varphi| \cdot |\mathcal{K}|^{ad(\varphi)}$ where $ad(\varphi)$ denotes the maximum alternation depth of fixed point operators. In particular, the alternation-free fragment of the μ-calculus has an algorithm of the same complexity as the logic CTL but offers a strictly greater expressiveness [CLE 93, EME 93].

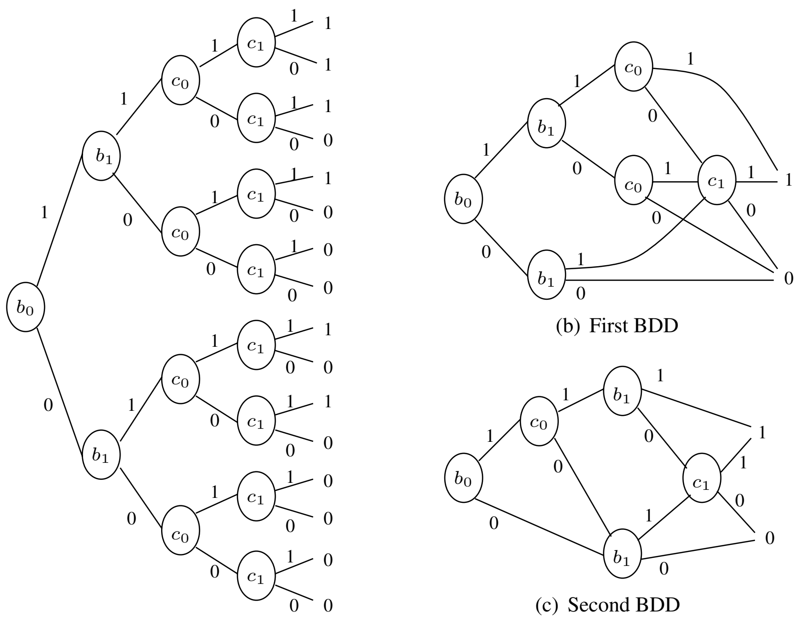
3.5.3. *Symbolic model checking algorithms*

Model checking algorithms manipulate sets of states. Efficient data structures to represent such sets are therefore critical for obtaining practically useful implementations of these algorithms. Explicit enumeration of states is limited to a few million states. A popular alternative is to represent sets symbolically, since these representations are more sensitive to the structure of the set than its size, and they can help verify much larger systems. In particular, the use of binary decision diagrams (BDDs, more precisely reduced ordered BDDs) has been a technological breakthrough for the implementation of model checking algorithms. Main advantages of BDDs are, first, that they provide canonical representations of sets, and therefore set equality can be decided by simple pointer comparison. Second, Boolean operations can be performed in polynomial time.

The basic idea is to represent a set S by its characteristic predicate χ_S, for which $x \in S$ if and only if $\chi_S(x)$ is true. Without loss of generality, we assume that states of finite transition systems are represented by a finite set $\{b_1, \dots, b_n\}$ of Boolean variables. The transition relation can then be represented as a predicate over the set of variables $\{b_1, \dots, b_n, b'_1, \dots, b'_n\}$ such that variables b_i represent the source state of the transition, and variables b'_i the target state.

A BDD can be understood as an efficient representation of a binary decision tree. For example, Figure 3.8(a) shows a decision tree that determines whether the addition of two two-bit numbers b_1b_0 and c_1c_0 will produce a carry bit. This decision tree represents the set

$$\{(11, 11), (11, 01), (11, 10), (01, 11), (10, 11), (10, 10)\}$$


(a) Decision tree

(b) First BDD

(c) Second BDD

Figure 3.8. *Decision tree and two BDDs*

of pairs of two-digit numbers; these are the labels of all paths leading to the result 1. (Note that the variables appear in the same order b_0, b_1, c_0, c_1 along any branch in the tree.) The same set is represented by the propositional formula

$$(b_1 \wedge c_1) \vee (b_0 \wedge c_0 \wedge (b_1 \vee c_1))$$

if we identify the values 0 and 1 with the truth values "false" and "true".

The tree in Figure 3.8(a) contains many redundancies. For example, the values of c_0 and c_1 are of no importance if b_0 and b_1 are both 0. A more compact representation eliminates these redundancies. First, isomorphic subtrees can be identified, and this results in a dag representation. Second, nodes whose children along both branches are identical can be eliminated. If we carry out these steps for our example, we obtain the BDD that appears in Figure 3.8(b). This representation is more compact because nodes are shared whenever possible. An implementation of a BDD package ensures maximum sharing by remembering each allocated BDD so that it can be reused instead of reallocated when it is needed a second time.

Every Boolean function is represented by a unique BDD with respect to a fixed variable order. However, the choice of this order can significantly influence the size of the BDD representing a given function. For example, the BDD shown in Figure 3.8(c) again represents our carry function, but with respect to the variable order b_0, c_0, b_1, c_1. When computing the carry for a growing number of bits, this second BDD grows linearly whereas the BDD for the original ordering grows exponentially. Choosing an optimal variable order for a given Boolean function is an NP-complete problem [BRY 92]. Standard implementations of BDD libraries use heuristics to determine the variable order [BER 95, FEL 93], but manual tuning is often necessary. In general, it is advisable that variables that are strongly inter-dependent are close to each other in the variable order [END 93, FUJ 93]. Unfortunately, there exist functions for which the size of BDD representations grow exponentially independently of the chosen variable order. Examples are multiplication of bit vectors or representations of FIFO queues.

Given two BDDs f and g (for a fixed variable order), their Boolean combinations can be computed recursively:

– if f or g are terminal BDD nodes (0 or 1) then the result is easily determined by the operation;

– otherwise, let b be the smallest variable according to the variable order at the roots of BDDs f and g, and let h_0 and h_1 be, recursively, the results of the operation applied to the sub-BDDs of f and g corresponding to b being 0 and 1. If $h_0 = h_1$, then the result does not depend on the value of b, so we return h_0, otherwise the result is the BDD with root b and successors h_0 and h_1.

The number of sub-problems to be computed is bounded by the number of pairs of nodes in the BDDs f and g. Assuming that the results of recursive computations are stored in a hash table with (nearly) constant access time, the cost of the operation is proportional to the product of the sizes of f and g.

Another useful BDD operation is *projection*, that is, existential quantification over a Boolean variable. Observing that

$$(\exists b : f) \;=\; f|_{b=0} \vee f|_{b=1},$$

the BDD representing this formula can be computed by disjunction and substitution of constants for variables. Quantification over several variables can be done simultaneously. Although the worst-case complexity of projection is exponential, this is rarely observed in practice.

A symbolic CTL model checking algorithm is easily obtained by implementing the naive recursive computation of the satisfaction sets $[\![\psi]\!]_{\mathcal{K}}$ mentioned in section 3.5.2 based on a BDD representation. For this purpose, we assume that the set of initial states of $\mathcal{K}$ and the transition relation are also represented in BDD form. The computation of pre-images $\delta^{-1}(s)$ translates into evaluating the expression

$$\exists \vec{b}' : \delta \wedge S'$$

where $\vec{b}'$ is the set of primed variables representing the states of $\mathcal{K}$, and where S' is a copy of the BDD for S where each variable b_i has been replaced by its primed version b_i'. The BDD machinery provides the necessary primitives for carrying out all required computations. In particular, termination of the fixed point computations is easily detected using pointer comparison.

It is interesting to compare the complexity of the BDD-based algorithm with that of the explicit-state algorithm. The representation of the sets manipulated by the algorithm can be exponentially more succinct. However, the number of iterations remains bounded by the size of the model and can therefore be exponential in the size of the BDD. Moreover, computing pre-images is based on quantification and therefore has exponential worst-case complexity. Fortunately, in practice the necessary fixed point iterations tend to converge quickly, particularly for the verification of electronic circuits with short data paths. Indeed, symbolic model checking has permitted the verification of systems with 10^{100} states or more [CLA 93b]. The crucial point is usually to find a variable order that allows the model checker to represent the transition relation.

The symbolic algorithm described here can be used for the verification of PTL formulae, using a symbolic representation of the Büchi automaton, and therefore of the product automaton. This technique is described in detail by Clarke *et al.* [CLA 97] and by Schneider [SCH 99].

BOUNDED MODEL CHECKING.– Although BDD representations have long dominated as the data structure used for representing Boolean functions in the model checking area, other representations can be useful. In particular, SAT algorithms for deciding the satisfiability of formulae of (non-temporal) propositional logic have made significant progress since the late 1990s [MIT 05]. These algorithms usually assume their input to be given as a list of clauses, and are not based on canonical representations of Boolean functions, virtually ruling out the calculation of fixed points. Instead, bounded model checking algorithms attempt to find an execution of finite length that violates the property of interest. The maximum size of a potential counter-example can (at least in theory) be determined from the size of the model and the formula to be verified. For example, an invariant holds for a Kripke structure if it is true for all prefixes of runs whose length is at most that of the longest loop in the graph of $\mathcal{K}$ (called the diameter of $\mathcal{K}$).

Given a bound $k \in \mathbb{N}$, the existence of a finite run of size $\leqslant k$ and leading to a state that does not verify an invariant is easily coded as a satisfiability problem in propositional logic by providing k copies of the state variables and relating subsequent copies by a (non-temporal) formula encoding transition relation. If finite loops back into the execution prefix are also considered, this idea can be generalized for full temporal logic [BIE 03]. In practice, bounded model checking is very useful to quickly find relatively small counter-examples. It is less appropriate for full-fledged verification because the worst-case bound on the maximum path lengths that need to be considered is again exponential in the size of the formula.

3.6. Some research topics

At the end of this introductory discussion of model checking concepts and techniques, we list references to various current topics of research, without trying to be exhaustive concerning either the list of topics or the references. The general objective is to make model checking and verification techniques applicable to more classes of systems and to larger systems.

REDUCTION TECHNIQUES.– In order to ensure that a system satisfies a given property, it is often enough to explore a representative subset of system runs instead of all runs. Because model checking is usually applied to reactive systems with several parallel components, reduction techniques geared towards concurrent systems are particularly useful. Two actions e_1 and e_2 are called *independent* if at any state where e_1 and e_2 are both possible, the execution of e_1 leaves e_2 executable and vice versa, and if the combined effects of executing the two action sequences $e_1; e_2$ and $e_2; e_1$ are the same. For example, the actions that represent two different processes sending messages along two different channels are independent. If the property of interest is not sensitive to the order of execution of the two actions (for example, if the property does not mention the communication channels), the execution of action e_2 can be delayed. Depending on the degree of independence and concurrency of a system, the systematic application of this idea can lead to substantial reductions in the number of transitions to explore. Nevertheless, we should take care that delayed actions will be considered eventually, and in particular before closing a loop in the reduced state space. Different authors have proposed reduction algorithms along these lines [ESP 94, GOD 94, HOL 94, PEL 96, VAL 90]. The key to making this idea work in practice is to find a good compromise between the cost of determining that actions are independent, say, by static analysis of the model, and the savings obtained during verification.

A different form of reduction makes use of *symmetries* in the state spaces of systems, which are also a source of redundancy during verification. For example, communication protocols usually do not inspect the data values that are transmitted over channels, and we can identify two states if they only differ in the values that channels contain. Similarly, systems often consist of several copies of identical processes whose precise identity is not relevant for system execution. These forms of symmetries induce an equivalence relation on the state space of the system, and it is enough to verify the quotient of the original system state space with respect to this equivalence relation, as long as the property to be verified is also symmetric [CLA 93a, IP 93, STA 91].

COMPOSITIONAL VERIFICATION.– In order to alleviate the effects of combinatorial explosion, it can be useful to preserve the process structure of a system instead of representing it as a “flat” transition system. In particular, specifications are written for each system component whose combination entails the overall correctness. In order to establish overall correctness we have to:

– ensure that each component satisfies its specification, and

– verify that the correctness of each component implies system correctness.

In general, an individual component cannot be expected to operate correctly in an arbitrary environment. Hypotheses about the functioning of the component’s environment are therefore explicitly identified in the component specification, and must be verified for the considered system. This becomes non-trivial in the case of mutual assumptions between components. Important approaches to compositional verification are presented in [ROE 98, ROE 01]. In the context of algorithmic verification, compositional verification is often referred to as the *module checking* problem, see [GRU 94, KUP 97a, MCM 97], among other works. An interesting extension of this problem is to synthesize components from temporal logic specifications [KUP 00, PIT 06, PNU 89] rather than verify the correctness of models *a posteriori*.

ABSTRACTION AND SOFTWARE MODEL CHECKING.– The models used for verification are generally abstract representations of real systems. Instead of manually providing these abstractions, tools can be built that attempt to compute sound abstractions from a more detailed description of a system. The construction of such abstractions has been widely studied, for some foundational articles see [CLA 94, DAM 94, LOI 95]. An intuitive and elegant presentation of abstractions consists of defining the state space of the abstract model by a set of predicates over the concrete state space. This representation, known as *predicate abstraction*, was originally considered for combinations of deductive and algorithmic verification tools, see for example [BJO 00, CAN 01, GRA 97, KES 99]. More recently, several tools combine this format with techniques of abstract interpretation, with the objective of model checking program code. The main challenge here is to obtain a meaningful finite-state abstraction of (usually infinite-state) software. A useful strategy is to start from the control-flow graph and first compute an over-approximation of the state space by abstracting the data values manipulated by the program. This abstraction serves as input for a conventional finite-state model checker, which will probably return a counter-example. Analyzing this abstract run over the concrete program, the verification tool determines if the counter-example represents a run of the concrete system (in which case it is reported to the user) or whether it is due to an imprecise abstraction. In the latter case, it will construct a more detailed abstract model, and the procedure is repeated, with the goal of eventually reaching a definitive verdict. This technique is known as *counter-example guided abstraction refinement* (CEGAR); see for example [BAL 02, HEN 04, SUW 05]. It is characterized by combining different approaches to verification, including abstract interpretation, automatic theorem proving and model checking.

INFINITE-STATE SYSTEMS.– In this chapter we have discussed model checking algorithms for finite-state systems. The extension of these techniques to infinite-state systems (directly or in combination with abstraction techniques) is an important research topic that has been studied at many different angles. Of course, real-time and hybrid systems are examples of infinite-state systems, and they are the main topic of this book. Probabilistic systems have also become of more and more interest, and again we refer to Chapters 8 and 9. More generally, verification problems can be tractable for system classes that generate sufficiently regular state spaces, such as certain properties of pushdown systems. See [ESP 01] for a classification of infinite-state systems and an annotated bibliography about this field.

3.7. Bibliography

[ALU 97] ALUR R., HENZINGER T. A., KUPFERMAN O., "Alternating-time Temporal Logic", *38th IEEE Symposium on Foundations of Computer Science*, IEEE Press, p. 100–109, 1997.

[BAL 02] BALL T., RAJAMANI S. K., "The SLAM Project: Debugging System Software via Static Analysis", *Principles of Programming Languages (POPL 2002)*, p. 1–3, 2002.

[BER 95] BERN J., MEINEL C., SLOBODOVÁ A., "Global rebuilding of BDDs – avoiding the memory requirement maxima", WOLPER P., Ed., *7th Intl. Conf. Computer Aided Verification (CAV'95)*, vol. 939 of *Lect. Notes in Comp. Sci.*, Springer, p. 4–15, 1995.

[BÉR 01] BÉRARD B., BIDOIT M., FINKEL A., LAROUSSINIE F., PETIT A., PETRUCCI L., SCHNOEBELEN PH., *Systems and Software Verification. Model-Checking Techniques and Tools*, Springer, 2001, Version française: Vérification de logiciels : techniques et outils du model-checking. Vuibert, 1999.

[BIE 03] BIERE A., CIMATTI A., CLARKE E., STRICHMAN O., ZHU Y., "Bounded Model Checking", *Highly Dependable Software*, vol. 58 of *Advances in Computers*, Academic Press, 2003.

[BJO 00] BJORNER N., BROWNE A., COLON M., FINKBEINER B., MANNA Z., SIPMA H., URIBE T., "Verifying Temporal Properties of Reactive Systems: A STeP Tutorial", *Formal Methods in System Design*, vol. 16, p. 227–270, 2000.

[BRY 92] BRYANT R. E., "Symbolic Boolean manipulations with ordered binary decision diagrams", *ACM Computing Surveys*, vol. 24, num. 3, p. 293–317, 1992.

[BÜC 62] BÜCHI J. R., "On a Decision Method in Restricted Second-Order Arithmetics", *Intl. Cong. Logic, Method and Philosophy of Science*, Stanford Univ. Press, p. 1–12, 1962.

[CAN 01] CANSELL D., MÉRY D., MERZ S., "Diagram Refinements for the Design of Reactive Systems", *Universal Computer Science*, vol. 7, num. 2, p. 159–174, 2001.

[CLA 81] CLARKE E. M., EMERSON E. A., "Synthesis of synchronization skeletons for branching-time temporal logic", *Workshop on Logic of Programs*, vol. 131 of *Lect. Notes in Comp. Sci.*, Yorktown Heights, N.Y., Springer, 1981.

[CLA 86] CLARKE E., EMERSON E., SISTLA A., "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Trans. Prog. Lang. and Systems*, vol. 8, num. 2, p. 244–263, 1986.

[CLA 93a] CLARKE E. M., ENDERS R., FILKORN T., JHA S., "Exploiting symmetry in temporal logic model checking", *Formal Methods in System Design*, vol. 9, p. 77–104, 1993.

[CLA 93b] CLARKE E., GRUMBERG O., HIRAISHI H., JHA S., LONG D., MCMILLAN K., NESS L., "Verification of the Futurebus+ Cache Coherence Protocol", AGNEW D., CLAESEN L., CAMPOSANO R., Eds., *IFIP Conf. Computer Hardware Description Lang. and Applications*, Ottawa, Canada, Elsevier, p. 5–20, 1993.

[CLA 94] CLARKE E. M., GRUMBERG O., LONG D. E., "Model Checking and Abstraction", *ACM Trans. Prog. Lang. and Systems*, vol. 16, num. 5, p. 1512–1542, 1994.

[CLA 97] CLARKE E. M., GRUMBERG O., HAMAGUCHI K., "Another look at LTL model checking", *Formal Methods in System Design*, vol. 10, p. 47–71, 1997.

[CLA 99] CLARKE E. M., GRUMBERG O., PELED D., *Model Checking*, MIT Press, Cambridge, Mass., 1999.

[CLA 00] CLARKE E. M., SCHLINGLOFF H., "Model Checking", ROBINSON A., VORONKOV A., Eds., *Handbook of Automated Deduction*, p. 1367–1522, Elsevier, 2000.

[CLE 93] CLEAVELAND R., STEFFEN B., "A linear-time model-checking algorithm for the alternation-free modal μ-calculus", *Formal Methods in System Design*, vol. 2, p. 121–147, 1993.

[COU 92] COURCOUBETIS C., VARDI M., WOLPER P., YANNAKAKIS M., "Memory-efficient algorithms for the verification of temporal properties", *Formal methods in system design*, vol. 1, p. 275–288, 1992.

[COU 99] COUVREUR J.-M., "On-the-Fly Verification of Linear Temporal Logic", WING J., WOODCOCK J., DAVIES J., Eds., *Formal Methods (FM'99)*, vol. 1708 of *Lect. Notes in Comp. Sci.*, Toulouse, France, Springer, p. 253–271, 1999.

[DAM 94] DAMS D., GRUMBERG O., GERTH R., "Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*", OLDEROG E.-R., Ed., *Prog. Concepts, Methods, and Calculi*, Amsterdam, Elsevier, p. 561–581, 1994.

[DAN 99] DANIELE M., GIUNCHIGLIA F., VARDI M., "Improved Automata Generation for Linear Temporal Logic", *Computer Aided Verification (CAV'99)*, vol. 1633 of *Lect. Notes in Comp. Sci.*, Trento, Italy, Springer, p. 249–260, 1999.

[DIL 92] DILL D. L., DREXLER A. J., HU A. J., YANG C. H., "Protocol Verification as a Hardware Design Aid", *Intl. Conf. Computer Design: VLSI in Computers and Processors*, IEEE Comp. Soc., p. 522–525, 1992.

[EME 86a] EMERSON E. A., HALPERN J. Y., ""Sometimes" and "not never" revisited: on branching-time vs. linear-time", *Journal of the ACM*, vol. 33, p. 151–178, 1986.

[EME 86b] EMERSON E. A., LEI C. L., "Efficient model checking in fragments of the propositional μ-calculus", *1st Symp. Logic in Comp. Sci.*, Boston, Mass., IEEE Press, 1986.

[EME 90] EMERSON E. A., "Temporal and modal logic", VAN LEEUWEN J., Ed., *Handbook of Theoretical Computer Science*, vol. B, p. 997–1071, Elsevier, 1990.

[EME 93] EMERSON E. A., JUTLA C. S., SISTLA A. P., "On model checking for fragments of μ-calculus", COURCOUBETIS C., Ed., *Computer-Aided Verification (CAV'93)*, vol. 697 of *Lect. Notes in Comp. Sci.*, Springer, 1993.

[END 93] ENDERS R., FILKORN T., TAUBNER D., "Generating BDDs for symbolic model checking", *Distributed Computing*, vol. 6, p. 155–164, 1993.

[ESP 94] ESPARZA J., "Model checking using net unfoldings", *Science of Computer Programming*, vol. 23, p. 151–195, 1994.

[ESP 01] ESPARZA J., "Verification of Systems with an Infinite State Space: an annotated bibliography", CASSEZ F., JARD C., ROZOY B., RYAN M. D., Eds., *Modeling and Verification of Parallel Processes*, vol. 2067 of *Lect. Notes in Comp. Sci.*, p. 183–186, Springer, 2001.

[FEL 93] FELT E., YORK G., BRAYTON R., VINCENTELLI A. S., "Dynamic variable reordering for BDD minimization", *Eur. Design Automation Conf.*, p. 130–135, 1993.

[FUJ 93] FUJI H., OOMOTO G., HORI C., "Interleaving based variable ordering methods for binary decision diagrams", *Intl. Conf. Computer Aided Design*, IEEE Press, 1993.

[GAB 94] GABBAY D., HODKINSON I., REYNOLDS M., *Temporal Logic: Mathematical Foundations and Computational Aspects*, vol. 1, Clarendon Press, Oxford, UK, 1994.

[GAS 01] GASTIN P., ODDOUX D., "Fast LTL to Büchi Automata Translation", BERRY G., COMON H., FINKEL A., Eds., *13th Intl. Conf. Computer Aided Verification (CAV'01)*, num. 2102 Lect. Notes in Comp. Sci., Paris, France, Springer, p. 53–65, 2001.

[GEL 04] GELDENHUYS J., VALMARI A., "Tarjan's algorithm makes LTL verification more efficient", JENSEN K., PODELSKI A., Eds., *10th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, vol. 2988 of *Lect. Notes in Comp. Sci.*, Barcelona, Spain, Springer, p. 205–219, 2004.

[GER 95] GERTH R., PELED D., VARDI M., WOLPER P., "Simple on-the-fly automatic verification of linear temporal logic", *Protocol Specification, Testing, and Verification*, Varsovie, Poland, Chapman & Hall, p. 3–18, 1995.

[GOD 94] GODEFROID P., WOLPER P., "A Partial Approach to Model Checking", *Information and Computation*, vol. 110, num. 2, p. 305–326, 1994.

[GRA 97] GRAF S., SAIDI H., "Construction of abstract state graphs with PVS", GRUMBERG O., Ed., *Computer Aided Verification: 9th International Conference*, vol. 1254 of *Lect. Notes in Comp. Sci.*, Springer Verlag, p. 72–83, 1997.

[GRU 94] GRUMBERG O., LONG D. E., "Model checking and modular verification", *ACM Trans. Prog. Lang. and Systems*, vol. 16, num. 3, p. 843–871, 1994.

[HAM 05] HAMMER M., KNAPP A., MERZ S., "Truly On-The-Fly LTL Model Checking", HALBWACHS N., ZUCK L., Eds., *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, vol. 3440 of *Lect. Notes in Comp. Sci.*, Edinburgh, UK, Springer, p. 191–205, 2005.

[HEN 04] HENZINGER T. A., JHALA R., MAJUMDAR R., MCMILLAN K., "Abstractions from Proofs", *31st Symp. Princ. of Prog. Lang. (POPL'04)*, ACM Press, p. 232–244, 2004.

[HOL 94] HOLZMANN G., PELED D., "An Improvement in Formal Verification", *IFIP Conf. Formal Description Techniques*, Berne, Switzerland, Chapman & Hall, p. 197–214, 1994.

[HOL 98] HOLZMANN G., "An analysis of bitstate hashing", *Formal Methods in System Design*, vol. 13, num. 3, p. 289–307, 1998.

[HOL 03] HOLZMANN G., *The SPIN Model Checker*, Addison-Wesley, 2003.

[HUT 04] HUTH M., RYAN M. D., *Logic in Computer Science*, Cambridge University Press, Cambridge, UK, 2nd edition, 2004.

[IP 93] IP C. N., DILL D., "Better verification through symmetry", *11th Intl. Symp. Comp. Hardware Description Languages and their Applications*, North Holland, p. 87–100, 1993.

[KAM 68] KAMP H. W., Tense Logic and the Theory of Linear Order, PhD thesis, Univ. of California at Los Angeles, 1968.

[KES 99] KESTEN Y., PNUELI A., "Verifying Liveness by Augmented Abstraction", *Computer Science Logic (CSL'99)*, Lect. Notes in Comp. Sci., Madrid, Spain, Springer, 1999.

[KOZ 83] KOZEN D., "Results on the propositional mu-calculus", *Theor. Comp. Sci.*, vol. 27, p. 333–354, 1983.

[KUP 94] KUPFERMAN O., VARDI M., WOLPER P., "An Automata-Theoretic Approach to Branching-Time Model Checking", *6th Intl. Conf. Computer-Aided Verification (CAV'94)*, Lect. Notes in Comp. Sci., Springer, 1994.

[KUP 97a] KUPFERMAN O., VARDI M., "Module Checking Revisited", *9th Intl. Conf. Computer Aided Verification (CAV'97)*, vol. 1254 of *Lect. Notes in Comp. Sci.*, Springer, p. 36–47, 1997.

[KUP 97b] KUPFERMAN O., VARDI M. Y., "Weak alternating automata are not so weak", *5th Israeli Symp. Theory of Computing and Systems*, IEEE Press, p. 147–158, 1997.

[KUP 00] KUPFERMAN O., MADHUSUDAN P., THIAGARAJAN P., VARDI M., "Open Systems in Reactive Environments: Control and Synthesis", PALAMIDESSI C., Ed., *Proc. 11th Int. Conf. on Concurrency Theory*, vol. 1877 of *Lecture Notes in Computer Science*, Springer Verlag, p. 92–107, 2000.

[KUP 05] KUPFERMAN O., VARDI M., "Complementation Constructions for Nondeterministic Automata on Infinite Words", HALBWACHS N., ZUCK L., Eds., *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, vol. 3440 of *Lect. Notes in Comp. Sci.*, Edinburgh, UK, Springer, p. 206–221, 2005.

[LAM 80] LAMPORT L., "'Sometime' is sometimes 'not never'", *7th Symp. Princ. of Prog. Lang. (POPL'80)*, p. 174–185, 1980.

[LAR 02] LAROUSSINIE F., MARKEY N., SCHNOEBELEN PH., "Temporal Logic with Forgettable Past", *17th IEEE Symp. Logic in Computer Science (LICS'02)*, Copenhagen, Denmark, IEEE Press, p. 383–392, 2002.

[LOI 95] LOISEAUX C., GRAF S., SIFAKIS J., BOUAJJANI A., BENSALEM S., "Property preserving abstractions for the verification of concurrent systems", *Formal Methods in System Design*, vol. 6, p. 11–44, 1995.

[MAN 90] MANNA Z., PNUELI A., "A Hierarchy of Temporal Properties", *9th ACM Symp. Princ. Distributed Computing*, ACM, p. 377–408, 1990.

[MCM 93] MCMILLAN K., *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[MCM 97] MCMILLAN K. L., "A compositional rule for hardware design refinement", GRUMBERG O., Ed., *9th Intl. Conf. Computer Aided Verification (CAV'97)*, vol. 1254 of *Lect. Notes in Comp. Sci.*, Haifa, Israel, Springer, p. 24–35, 1997.

[MIT 05] MITCHELL D. G., "A SAT Solver Primer", *EATCS Bulletin*, vol. 85, p. 112–133, 2005.

[MUL 63] MULLER D. E., "Infinite Sequences and Finite Machines", *Switching Circuit Theory and Logical Design*, New York, IEEE Press, p. 3–16, 1963.

[MUL 88] MULLER D., SAOUDI A., SCHUPP P., "Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time", *3rd IEEE Symp. Logic in Computer Science*, IEEE Press, p. 422–427, 1988.

[PEL 96] PELED D., "Combining Partial Order Reductions with On-the-Fly Model-Checking", *Formal Methods in System Design*, vol. 8, num. 1, p. 39–64, 1996.

[PIT 06] PITERMAN N., PNUELI A., SA'AR Y., "Synthesis of Reactive Designs", EMERSON E. A., NAMJOSHI K. S., Eds., *Verification, Model Checking, and Abstract Interpretation (VMCAI 2006)*, vol. 3855 of *Lect. Notes in Comp. Sci.*, Charleston, SC, Springer, p. 364–380, 2006.

[PNU 89] PNUELI A., ROSNER R., "On the Synthesis of a Reactive Module.", *Principles of Programming Languages (POPL 1989)*, ACM Press, p. 179–190, 1989.

[PSL 04] PSL CONSORT., Property specification language (version 1.1), Technical report, Accellera, June 2004.

[QUE 81] QUEILLE J., SIFAKIS J., "Specification and verification of concurrent systems in Cesar", *5th Intl. Symp. Programming*, vol. 137 of *Lect. Notes in Comp. Sci.*, Springer, p. 337–351, 1981.

[RAB 69] RABIN M. O., "Decidability of Second-Order Theories and Automata on Infinite Trees", *Trans. American Math. Soc.*, vol. 141, p. 1–35, 1969.

[ROE 98] DE ROEVER W.-P., LANGMAACK H., PNUELI A., Eds., *Compositionality: The Significant Difference*, vol. 1536 of *Lect. Notes in Comp. Sci.*, Springer Verlag, 1998.

[ROE 01] DE ROEVER W.-P., DE BOER F., HANNEMANN U., HOOMAN J., LAKHNECH Y., POEL M., ZWIERS J., *Concurrency verification: introduction to compositional and non-compositional methods*, Cambridge University Press, 2001.

[SAF 88] SAFRA S., "On The Complexity of ω-Automata", *29th IEEE Symp. Found. Comp. Sci.*, IEEE Press, p. 319–327, 1988.

[SCH 99] SCHNEIDER K., "Yet Another Look at LTL Model Checking", *IFIP Conf. Correct Hardware Design and Verification Methods (CHARME'99)*, *Lect. Notes in Comp. Sci.*, Bad Herrenalb, Germany, Springer, 1999.

[SCH 05] SCHWOON S., ESPARZA J., “A note on on-the-fly verification algorithms”, HALBWACHS N., ZUCK L., Eds., *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, vol. 3440 of *Lect. Notes in Comp. Sci.*, Edinburgh, UK, Springer, p. 174–190, 2005.

[SIS 87] SISTLA A., VARDI M., WOLPER P., “The complementation problem for Büchi automata with applications to temporal logic”, *Theor. Comp. Sci.*, vol. 49, p. 217–237, 1987.

[STA 91] STARKE P. H., “Reachability analysis of Petri nets using symmetries”, *Syst. Anal. Model. Simul.*, vol. 8, p. 293–303, 1991.

[STI 01] STIRLING C., *Modal and Temporal Properties of Processes*, Springer, Berlin, 2001.

[SUW 05] SUWIMONTEERABUTH D., SCHWOON S., ESPARZA J., “jMoped: A Java bytecode checker based on Moped”, HALBWACHS N., ZUCK L., Eds., *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, vol. 3440 of *Lect. Notes in Comp. Sci.*, Edinburgh, UK, Springer, p. 541–545, 2005.

[TAR 72] TARJAN R. E., “Depth first search and linear graph algorithms”, *SIAM Journal of Computing*, vol. 1, p. 146–160, 1972.

[THO 97] THOMAS W., “Languages, Automata, and Logic”, ROZENBERG G., SALOMAA A., Eds., *Handbook of Formal Language Theory*, vol. III, p. 389–455, Springer Verlag, New York, 1997.

[VAL 90] VALMARI A., “A stubborn attack on state explosion”, *2nd Intl. Conf. Computer Aided Verification*, vol. 531 of *Lect. Notes in Comp. Sci.*, Rutgers, N.J., Springer, p. 156–165, 1990.

[VAR 94] VARDI M., WOLPER P., “Reasoning about infinite computations”, *Information and Computation*, vol. 115, num. 1, p. 1–37, 1994.

[VAR 95] VARDI M. Y., “Alternating Automata and Program Verification”, *Computer Science Today*, vol. 1000 of *Lect. Notes in Comp. Sci.*, p. 471–485, Springer, 1995.

[VAR 01] VARDI M., “Branching vs. Linear Time—Final Showdown”, MARGARIA T., YI W., Eds., *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, vol. 2031 of *Lect. Notes in Comp. Sci.*, Genoa, Italy, Springer, p. 1–22, 2001.

[WOL 83] WOLPER P., “Temporal Logic Can Be More Expressive”, *Information and Control*, vol. 56, p. 72–93, 1983.

Chapter 4

Model Checking Timed Automata

4.1. Introduction

Currently, formal verification of reactive, critical or embedded systems is a crucial problem, and automatic verification, more specifically *model checking*, has been widely developed during the last 20 years (see [CLA 99, SCH 01] for surveys). In this approach, we build a formal model $\mathcal{M}$ (e.g. an automaton, Petri net, etc.) describing the behavior of the system under verification; the correctness property Φ is stated with a formal specification language (e.g. temporal logic), and then a model-checker is used to automatically decide whether $\mathcal{M}$ satisfies Φ or not.

Often, it is necessary to consider real-time aspects: quantitative information about time elapsing has to be handled explicitly. This can be the case when describing a particular behavior (for instance, a time-out) or stating a complex property (for example, "the alarm has to be activated *within at most 10 time units* after a problem has occurred"). In 1990, Alur and Dill have proposed *timed automata* as a model to represent the behavior of real-time systems [ALU 90, ALU 94a]. This formalism extends classical automata with a set of real-valued variables – called clocks – that increase synchronously with time, and associates guards (specifying when, i.e. for which values of the clocks, the transition can be performed) and update operations (to be applied when the transition is performed) with every transition. Thanks to these clocks, it becomes possible to express constraints over delays between two transitions.

Temporal logics have also been extended to deal with real-time constraints. For example, the modalities of the classical CTL logic (computation tree logic [CLA 81])

Chapter written by Patricia BOUYER and François LAROUSSINIE.

have been adapted to handle quantitative constraints over time elapsing [ALU 94c, ALU 93a, ACE 02].

Finally, model checking algorithms have been developed [ALU 93a, HEN 94, LAR 95b], and a lot of research has been done on the timed verification algorithmics: efficient data-structures, on-the-fly algorithms, compositional methods, etc. have been proposed. Timed model checkers have also been developed [YOV 97, LAR 97b] and are applied to industrial case studies [TRI 98, BEN 02]. Timed model checking is clearly an active research topic.

In this chapter, we present the classical timed automata model. We explain the main characteristics of this model, and describe the famous region graph technique that is a crucial construction to obtain the decidability of many verification problems in this framework. We also mention several possible extensions of timed automata and several interesting subclasses. Finally, we describe algorithmic aspects and the basic data-structure that is used to implement verification algorithms, and we present the Uppaal tool [LAR 97b].

4.2. Timed automata

Timed automata have been proposed by R. Alur and D. Dill in the 1990s [ALU 90, ALU 94a] as a model for real-time systems. A timed automaton is a classical finite automaton which can manipulate clocks, evolving continuously and synchronously with absolute time. Each transition of such an automaton is labeled by a constraint over clock values (also called guard), which indicates when the transition can be fired, and a set of clocks to be reset when the transition is fired. Each location is constrained by an invariant, which restricts the possible values of the clocks for being in the state, which can then enforce a transition to be taken. The time domain can be $\mathbb{N}$, the set of non-negative integers, or $\mathbf{Q}$, the set of non-negative rationals, or even $\mathbf{R}$, the set of non-negative real numbers. In this chapter, we choose $\mathbf{R}$ as the time domain, but most results are unchanged when considering $\mathbf{Q}$ or $\mathbb{N}$.

4.2.1. *Some notations*

Let X be a finite set of variables, called clocks, taking values in $\mathbf{R}$. A (clock) valuation v over X is a function $v : X \rightarrow \mathbf{R}$ which associates with every clock x its value $v(x) \in \mathbf{R}$. We denote by $\mathbf{R}^X$ the set of clock valuations over X. Given a real $d \in \mathbf{R}$, we write $v + d$ for the clock valuation associating with clock x the value $v(x) + d$, If r is a subset of X, $[r \leftarrow 0]v$ is the valuation v' such that $v'(x) = 0$ if $x \in r$, and $v'(x) = v(x)$ otherwise.

We write $\mathcal{C}(X)$ for the set of clock constraints over X, i.e., the set of Boolean combinations of atomic constraints of the form $x \bowtie c$ with $x \in X$, $\bowtie \in \{=, <, \leqslant, >, \geqslant\}$ and $c \in \mathbb{N}$. We note by $\mathcal{C}_{\leqslant}(X)$ the restriction of $\mathcal{C}(X)$ to positive Boolean combinations only containing constraints of the form $x \leqslant c$ or $x < c$. We interpret clock constraints over clock valuations: a valuation v satisfies the atomic constraint $x \bowtie c$ whenever $v(x) \bowtie c$; the extension to general constraints is then immediate and natural. When a valuation v satisfies a constraint g, we write $v \models g$.

4.2.2. *Timed automata, syntax and semantics*

The formal definition of a timed automaton is as follows.

DEFINITION 4.1.– *A* timed automaton $\mathcal{A}$ *is a tuple* $(L, \ell_0, X, \mathsf{Inv}, T, \Sigma)$ *where:*

– *L is a finite set of control states, also called locations,*

– *$\ell_0 \in L$ is the initial location,*

– *X is a finite set of clocks,*

– *$T \subseteq L \times \mathcal{C}(X) \times \Sigma \times 2^X \times L$ is a finite set of transitions: $e = \langle \ell, g, a, r, \ell' \rangle \in T$ represents a transition from ℓ to ℓ', g is the guard of e, r is the set of clocks that is reset by e, and a is the action of e. We also write $\ell \xrightarrow{g,a,r} \ell'$ for e,*

– *$\mathsf{Inv} : L \to \mathcal{C}_{\leqslant}(X)$ associates with each location an invariant,*

– *Σ is an alphabet of actions.*

An example of timed automaton is given in Figure 4.2.

A state, or configuration, of a timed automaton is a pair $(\ell, v) \in L \times \mathbf{R}^X$ where ℓ is the current location and v is the clock valuation. The semantics of a timed automaton is given as a timed transition system with action transitions (labeled with elements of Σ) and delay transitions (labeled with real numbers representing the delay). This is stated more precisely as follows.

DEFINITION 4.2.–*A* timed transition system *(TTS) is a tuple $\mathcal{S} = (S, s_0, \to, \Sigma)$ where S is a (possibly infinite) set of states, $s_0 \in S$ is the initial state and $\to \subseteq S \times (\Sigma \cup \mathbf{R}) \times S$ is the transition relation. Moreover, the relation $\to$ satisfies the three following conditions: (1) if $s \xrightarrow{0} s'$, then $s = s'$, (2) if $s \xrightarrow{d} s'$ and $s' \xrightarrow{d'} s''$ with $d, d' \in \mathbf{R}$, then $s \xrightarrow{d+d'} s''$, and (3) if $s \xrightarrow{d} s'$ with $d \in \mathbf{R}$, then for all $0 \leqslant d' \leqslant d$, there exists $s'' \in S$ such that $s \xrightarrow{d'} s''$ and $s'' \xrightarrow{d-d'} s'$.*

The three conditions mentioned above are classical in the framework of timed systems, see e.g. [YI 90], they simply express that the time is continuous and deterministic.

Classically, an execution in a TTS is a sequence of consecutive transitions. A state $s \in S$ is said to be reachable in $\mathcal{S}$ if there exists an execution from s_0 to s.

DEFINITION 4.3.– *Let $\mathcal{A} = (L, \ell_0, X, \mathsf{Inv}, T, \Sigma)$ be a timed automaton. The semantics of $\mathcal{A}$ is defined as the TTS $\mathcal{S}_{\mathcal{A}} = (S, s_0, \to, \Sigma)$ where:*

– $S = L \times \mathbf{R}^X$,

– $s_0 = (\ell_0, v_0)$ *with* $v_0(x) = 0$ *for every* $x \in X$,

– *the transition relation* $\rightarrow$ *is composed of:*

- *action transitions:* $(\ell, v) \xrightarrow{a} (\ell', v')$ *if and only if there exists* $\ell \xrightarrow{g,a,r} \ell' \in T$ *such that* $v \models g$, $v' = [r \leftarrow 0]v$ *and* $v' \models \mathsf{Inv}(\ell')$.

- *delay transitions: if* $d \in \mathbf{R}$, $(\ell, v) \xrightarrow{d} (\ell, v{+}d)$ *if and only if* $v{+}d \models \mathsf{Inv}(\ell)$[1].

Informally, the system starts from the initial configuration (location ℓ_0 and all clocks set to zero), and then alternatively takes action transitions if the current clock valuations satisfies the guard (this move is instantaneous and some clocks are then set to zero), and delay transitions which increase all clocks by the same amount of time (clocks are synchronous) while respecting the invariant associated with the current location.

A possible execution of the timed automaton of Figure 4.2 is: $(\ell_0, (0, 0)) \xrightarrow{2.67} (\ell_0, (2.67, 2.67)) \xrightarrow{a} (\ell_1, (2.67, 0)) \xrightarrow{1} (\ell_1, (3.67, 1)) \xrightarrow{b} (\ell_2, (3.67, 1)) \cdots$ where the pair $(3.67, 1)$ represents the valuation v such that $v(x) = 3.67$ and $v(y) = 1$.

An execution in a timed automaton can also be seen as a timed word, i.e., a sequence of pairs (action, date). We can then write: $(\ell_0, v_0, t_0) \xrightarrow{a_1} (\ell_1, v_1, t_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (\ell_n, v_n, t_n)$ with $t_i \in \mathbf{R}$, $t_0 = 0$ and $t_{i+1} \geqslant t_i$ for every i. Date t_i corresponds to the time point at which action a_i has been performed. The step $(\ell_i, v_i, t_i) \xrightarrow{a_{i+1}} (\ell_{i+1}, v_{i+1}, t_{i+1})$ corresponds to a delay $t_{i+1} - t_i$ followed by the firing of a transition labeled by a_{i+1}, the valuation v_{i+1} is then obtained from $v_i + (t_{i+1} - t_i)$ by resetting to zero some of the clocks (depending on the transition which has been fired). The associated timed word is then $(a_1, t_1)(a_2, t_2) \cdots$ For instance, the timed word associated with the above-mentioned execution is $(a, 2.67)(b, 3.67) \cdots$.

4.2.3. *Parallel composition*

It is possible to define the parallel composition of timed automata (or of TTSs). For instance, we can define an n-ary synchronization relation with renaming. If this feature is essential for modeling systems, it does not add expressivity power from a theoretical point-of-view: indeed, it is always possible to construct a product automaton having the same behavior as the parallel composition (it is even strongly bisimilar; see section 4.4).

1. Which, given the form of the invariants, is $v + d' \models \mathsf{Inv}(q)$ for every $0 \leqslant d' \leqslant d$.

4.3. Decision procedure for checking reachability

In this section, we describe a construction initially proposed in [ALU 90, ALU 94a] to decide the reachability of a control state in a timed automaton. This construction relies on an abstraction of the behaviors of the timed automaton, so that checking whether a location is reachable in the initial timed automaton is equivalent to checking whether a state (or set of states) is reachable in a finite automaton.

For this aim, an equivalence relation of finite index is defined over the set of configurations of the timed automaton: two configurations s and s' are *equivalent* when any action transition "a" (resp. any delay transition d) enabled from s can be simulated from s' by an action transition "a" (resp. a delay transition d') such that the two resulting configurations are equivalent (and vice versa for s'). Note that precise delays are not respected and the equivalence will only correspond to a *time-abstract bisimulation*. For timed automata, such an equivalence relation (with finite index) always exists, and it is defined as follows. Two configurations (ℓ, v) and (ℓ', v') are equivalent if $\ell = \ell'$ and if $v \equiv_M v'$ (where M is the maximal constant appearing in the automaton). The relation $v \equiv_M v'$ holds whenever for each clock $x \in X$,

1) $v(x) > M \Leftrightarrow v'(x) > M$,

2) if $v(x) \leqslant M$, then $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$, and $\Big(\{v(x)\} = 0 \Leftrightarrow \{v'(x)\} = 0\Big)$[2],

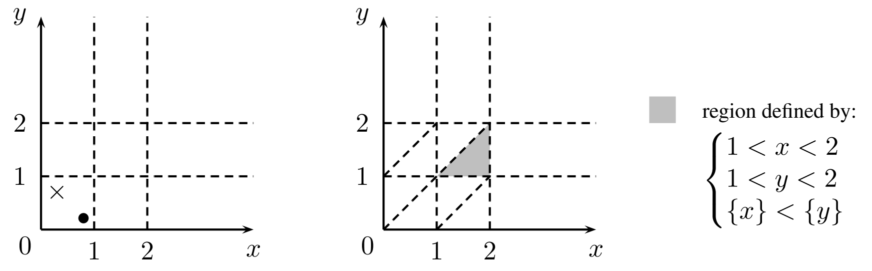
and for each pair of clocks (x, y),

3) if $v(x) \leqslant M$ and $v(y) \leqslant M$, then $\{v(x)\} \leqslant \{v(y)\} \Leftrightarrow \{v'(x)\} \leqslant \{v'(y)\}$.

Intuitively, the two first conditions express that two equivalent valuations satisfy exactly the same clock constraints of the timed automaton. The last condition ensures that from two equivalent configurations, letting time elapse will lead to the same integral values for the clocks, in the very same order. The equivalence $\equiv_M$ is called the *region equivalence*, and an equivalence class is then called a *region*.

We illustrate this construction in Figure 4.1 in the case of two clocks x and y, and the maximal constant is supposed to be 2. The partition depicted in Figure 4.1(a) respects all constraints defined with integral constants smaller than or equal to 2, but the two valuations $\bullet$ and $\times$ are not equivalent due to time elapsing (item 3 above): indeed, if we let some time elapse from the valuation $\bullet$, we will first satisfy the constraint $x = 1$ and then $y = 1$, while it will be the converse for the valuation $\times$. Thus, the possible behaviors from $\bullet$ and $\times$ are different. Condition 3 refines the partition of Figure 4.1(a) by adding diagonal lines (that somehow represent time elapsing), and the resulting partition is given on Figure 4.1(b) and is a time-abstract bisimulation.

2. $\lfloor \alpha \rfloor$ represents the integral part of α whereas $\{\alpha\}$ represents its fractional part.



y
2
1
0
1
2
x
y
2
1
0
1
2
x
region defined by:
$\begin{cases}1 < x < 2\\ 1 < y < 2\\ \{x\} < \{y\}\end{cases}$

(a) Partition respecting 1) and 2) (b) Partition respecting 1), 2) and 3)

Figure 4.1. *Region partitioning for two clocks and maximal constant* 2

From the initial timed automaton and this equivalence relation, we build a finite automaton as follows: the states of the automaton are pairs (ℓ, R) where ℓ is a location of the timed automaton and R a region; the transitions are $(\ell, R) \xrightarrow{a} (\ell', R')$ if there exists a transition $\ell \xrightarrow{g,a,r} \ell'$ in $\mathcal{A}$, a valuation $v \in R$, and $t \geqslant 0$ such that $v + t \models \mathsf{Inv}(\ell)$, $v + t \models g$, $[r \leftarrow 0](v + t) \models \mathsf{Inv}(\ell')$ and $[r \leftarrow 0](v + t) \in R'$.

The resulting finite automaton $\mathcal{R}_{\mathcal{A}}$ is called the *region automaton* associated with the initial timed automaton. The fundamental property of this finite automaton is that it recognizes exactly the set of words $a_1 a_2 \cdots$ such that there exists a timed word $(a_1, t_1)(a_2, t_2) \cdots$ recognized by the initial timed automaton. Hence, given a timed automaton $\mathcal{A}$ and its region automaton $\mathcal{R}_{\mathcal{A}}$, we can reduce the emptiness check for the timed language accepted by $\mathcal{A}$ (or equivalently the reachability checking of a location of $\mathcal{A}$) to a reachability problem in $\mathcal{R}_{\mathcal{A}}$. This produces an algorithm to solve these two problems.

THEOREM 4.1.– *[ALU 94a] Checking the reachability of a location in a timed automaton is a PSPACE-complete problem.*

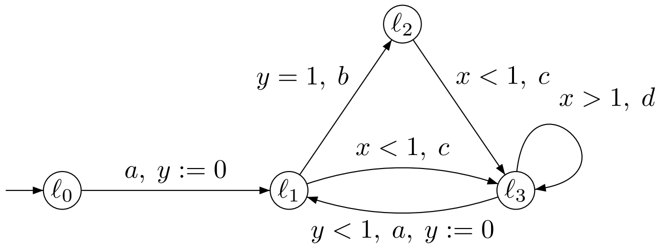


Figure 4.2. *Timed automaton* $\mathcal{A}$

We illustrate the construction of the region automaton on the timed automaton depicted in Figure 4.2 and taken from [ALU 94a]. The corresponding region automaton is depicted in Figure 4.3. In this example, the location ℓ_3 of $\mathcal{A}$ is reachable if and only if one of the states (ℓ_3, R) with a region R is reachable in the finite automaton given in Figure 4.3. In this last automaton, the path $(\ell_0, x{=}y{=}0) \xrightarrow{a} (\ell_1, 0{=}y{<}x{<}1) \xrightarrow{c} (\ell_3, 0{<}y{<}x{<}1)$ leads to location ℓ_3, which implies that, in the timed automaton $\mathcal{A}$, there is an execution $(\ell_0, v_0) \xrightarrow{t_1} (\ell_0, v_0 + t_1) \xrightarrow{a} (\ell_1, v_1) \xrightarrow{t_2} (\ell_1, v_1 + t_2) \xrightarrow{c} (\ell_3, v_2)$ leading to ℓ_3 (for some real numbers t_1 and t_2).

The complexity of the reachability problem, mentioned in the previous theorem, has been originally stated in the papers [ALU 90, ALU 94a]:

– the PSPACE-hardness comes from the fact that we can encode the behavior of a linearly space bounded Turing machine on a given input. Indeed, it is possible to construct a timed automaton in which clock values encode the content of the Turing machine tape along the execution. Notice that such an encoding can be done using only three clocks [COU 92];

– the PSPACE membership is obtained by applying a non-deterministic algorithm which stores the current abstract state of the automaton (location+region) and guesses the next abstract state, until reaching a goal location (or aborting the computation when a counter becomes greater than the size of the region automaton, which is exponential).

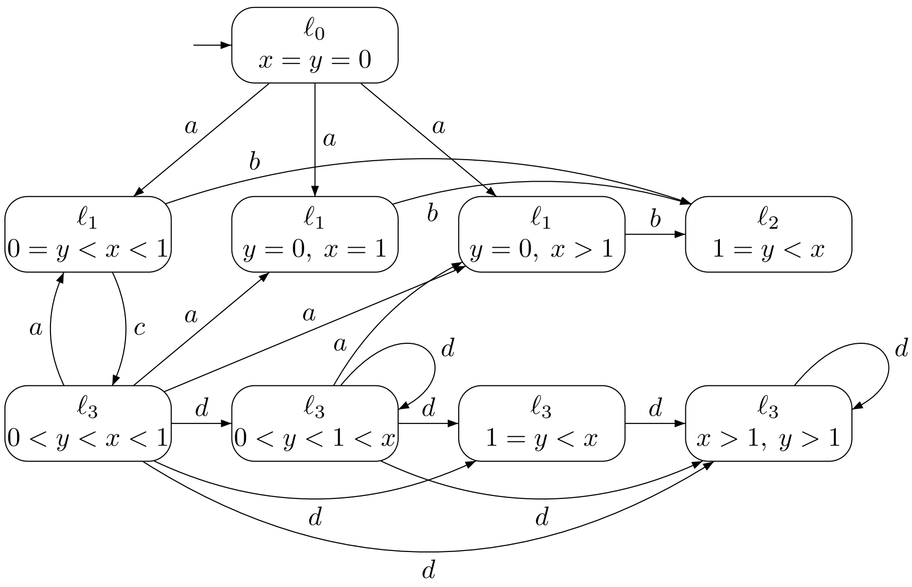


Figure 4.3. *Region automaton associated with* $\mathcal{A}$

4.4. Other verification problems

Reachability is a key problem in verification. Correctness can often be stated as a reachability question: "Is the `BAD` state reachable from the initial configuration?", or "Is it true that any reachable state is either `BLUE` or `RED`?". Nevertheless it is sometimes useful to consider more complex properties on the behavior of the system, and we hence need formal specification languages to state properties. For example, assume we want to express the following timed property:

"The alarm is activated within at most 10 time units after a problem occurs." (4.1)

There are several ways to express such properties over timed systems, and we briefly mention the most classical ones.

4.4.1. *Timed languages*

As mentioned earlier, we can associate a *timed word* with an execution of a timed automaton. It is also possible to consider different acceptance conditions in timed automata (final states, Büchi or Muller conditions, etc.). In this setting, the behavior of a timed automaton $\mathcal{A}$ is seen as a *timed language* $\mathcal{L}(\mathcal{A})$ containing the timed words read over all accepting executions. Now given a property Φ described as a timed language $\mathcal{L}_\Phi$ (for example, the set of words "$(\texttt{request}, t_1), (\texttt{service}, t_2)$" with $t_1 \leqslant t_2 \leqslant t_1 + 10$), the verification problem "does $\mathcal{A}$ satisfy the property Φ?" can be reduced to an inclusion checking over timed languages: is $\mathcal{L}(\mathcal{A})$ included in $\mathcal{L}_\Phi$? In the untimed case, this classical problem is solved by considering the complement of $\mathcal{L}_\Phi$ (i.e. $\mathcal{L}_{\neg\Phi}$) and checking the emptiness of $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}_{\neg\Phi}$. Unfortunately, this approach cannot be used in the timed case because the language $\mathcal{L}_{\neg\Phi}$ is not always expressible with a timed automaton: most timed languages families (of finite words, or infinite words with Büchi or Muller conditions) are not closed under complementation. Indeed the inclusion problem is in general undecidable. Thus, this approach is only possible for restricted classes (e.g. deterministic Muller timed automata).

4.4.2. *Branching-time timed logics*

Temporal logic is a very convenient formalism to specify properties over reactive systems [PNU 77]. They make it possible to express properties over the ordering of events of a system. We can distinguish branching-time temporal logic and linear-time temporal logic: in the former case, formulae are interpreted over states having several possible successors (we can quantify existentially or universally over the different possible futures of a given state). In the latter case, a system is viewed as a set of runs, and formulae express properties over these runs: in such a structure, a state is always considered as *a state along a given run* and then it has a unique successor. These formalisms differ from an expressiveness point of view, and the model checking algorithms are also very different.

The most popular (untimed) branching-time temporal logic is CTL (computation tree logic) [CLA 81]. It contains the modalities "always" (AG), "potentially" (EF), "exists-until" ($\mathsf{E\_U\_}$), and "for-all-until" ($\mathsf{A\_U\_}$). For example, the formula $\mathsf{E}\phi\mathsf{U}\psi$ holds in a state s if and only if there exists a path from s where ψ is true at some position s', and ϕ is true at any position between s and s'. See [EME 91] for a precise introduction to temporal logics for the specification and verification of reactive systems.

There exist several timed extensions of temporal logics. First we can add subscripts with timing constraints to the classical Until operator. Such a constraint is of the form "$\bowtie c$" with $\bowtie\ \in \{=, <, \leqslant, >, \geqslant\}$ and $c \in \mathbb{N}$. For example, the formula $\mathsf{E}\phi\mathsf{U}_{<c}\psi$ holds in a state s if and only if there exists a run ρ leading to a state s' such that (1) s' satisfies ψ, (2) the duration of ρ is less than c, and (3) any state lying between s and s' along the run ρ satisfies ϕ.

The logic TCTL (for timed CTL) is defined with these extended modalities: it contains the atomic propositions, the Boolean operators and the modalities $\mathsf{E\_U}_{\bowtie c}\mathsf{\_}$ and $\mathsf{A\_U}_{\bowtie c}\mathsf{\_}$. Thus, Property 4.1 can then be expressed as follows:

$$\mathsf{AG}\Big(\texttt{problem} \Rightarrow \mathsf{AF}_{\leqslant 10}\,\texttt{alarm}\Big)$$

where $\mathsf{AF}_{\leqslant 10}\phi$ is an abbreviation for $\mathsf{A}\ \texttt{true}\ \mathsf{U}_{\leqslant 10}\phi$: along every path, there is a position before 10 time units in which ϕ holds.

There is another way to add timing constraints in CTL. The idea is to consider a new set of clocks – the formula clocks – and to add atomic constraints "$x \bowtie c$" in the logic and a new operator ($\underline{\texttt{in}}$) to reset a given clock to zero [ALU 94c]. This extension is called TCTL_c. The previous property can be expressed as follows with TCTL_c:

$$\mathsf{AG}\Big(\texttt{problem} \Rightarrow \Big(x\ \underline{\texttt{in}}\ \mathsf{AF}\left(\texttt{alarm} \wedge (x \leqslant 10)\right)\Big)\Big)$$

where x is a formula clock which is reset to zero when the proposition $\texttt{problem}$ is true, and it is used to ensure that the time elapsed between the problem and the activation of the alarm is less than 10 time units.

This extension with explicit formula clocks allows us to express easily every modality of TCTL. For example, we have the following equivalence when φ and ψ are TCTL formulae:

$$\mathsf{E}\left(\varphi\mathsf{U}_{\bowtie c}\,\psi\right) \equiv x\ \underline{\texttt{in}}\ \mathsf{E}\left(\varphi\mathsf{U}(\psi \wedge x \bowtie c)\right)$$

TCTL_c makes it possible to express very precise properties over timed systems. It has been shown recently that it is indeed more powerful than TCTL in the dense time framework [BOU 05b]. For example, the following TCTL_c formula cannot be stated with TCTL:

$$x \ \underline{\mathtt{in}}\ \mathsf{EF}\Big(P_1 \wedge x < 1 \wedge \mathsf{EG}(x < 1 \Rightarrow \neg P_2)\Big)$$

This formula expresses that it is possible to reach a state s' satisfying P_1 in t time units with $t < 1$ and from then it is possible to avoid P_2 during (at least) $1 - t$ time units.

These specification languages are very convenient to express properties over a timed system. Moreover, verification problems are still decidable.

THEOREM 4.2.– *[ALU 93a] The* TCTL *and* TCTL_c *model checking problems for timed automata are PSPACE-complete.*

The algorithms use the same techniques as for the reachability problem: given a timed automaton $\mathcal{A}$ and a TCTL formula Φ, it is possible to define a region automaton $\mathcal{A}'$ (over the automata clocks and the formula clocks) and a CTL formula Φ' such that $\mathcal{A} \models \Phi$ if and only if $\mathcal{A}' \models \Phi'$. Verifying TCTL formulae over parallel compositions of timed automata can be done with the Kronos tool [YOV 97].

4.4.3. *Linear-time timed logics*

Linear-time temporal logics (as LTL [PNU 81]) can also be extended with timing constraints in the same manner as for the branching-time case. For example, formula $\mathsf{G}(\mathtt{problem} \Rightarrow \mathsf{F}_{\leqslant 10}\mathtt{alarm})$ expresses Property 4.1. The only difference is that such formulae are interpreted over the runs of a timed automaton. By convention we write $\mathcal{A} \models \Phi$ to specify that *every* run of the timed automaton $\mathcal{A}$ satisfies Φ.

In this framework we can mention MTL [KOY 90, ALU 93b] which contains modalities U_I where I is an interval of the form $(l; u)$, $[l; u]$, ... with $l, u \in \mathbb{N} \cup \{\infty\}$. This interval provides the timing constraint in a natural way: formula $P_1 \mathsf{U}_{[3;\infty]} P_2$ is equivalent to $P_1 \mathsf{U}_{\geqslant 3} P_2$, etc. We denote MITL [ALU 96] the fragment of MTL where singular intervals $[c; c]$ are not allowed (and then the modality $\mathsf{U}_{=c}$ is forbidden).

Model checking MTL is undecidable [ALU 96, OUA 06]. Note that it is also possible to consider a different semantic where atomic propositions are interpreted as punctual events occurring at some date: in this case, model checking MTL becomes decidable over finite runs [OUA 05].

The model checking problem for MITL is easier: it is EXPSPACE-complete (with the standard semantics) and even becomes PSPACE-complete as soon as we only consider the modalities $\mathsf{U}_{<c}$ and $\mathsf{U}_{>c}$ (i.e. $\mathsf{U}_{[0;c)}$ and $\mathsf{U}_{(c;\infty)}$) [ALU 96]. Other tractable fragments of MTL have been recently investigated [BOU 07b].

4.4.4. *Timed modal logics*

It is also possible to consider timed extensions of modal logics (see for example the Hennessy-Milner logic [HEN 85]). In this case, we use modalities $\langle a \rangle$ and $[a]$ to deal with the label of transitions. For example, $\langle a \rangle\ \phi$ states that there exists an a-transition leading to a state verifying ϕ, and $[a]\ \phi$ expresses that *every* state reachable via an a-transition satisfies ϕ. These modalities only deal with states reachable in one step. However, it is possible to use fixpoint to express properties over unbounded behaviors [LAR 90, STI 01]. This kind of formalism can also be extended with formula clocks, atomic constraints "$x \bowtie c$" and reset operator $\underline{\texttt{in}}$ as in TCTL_c [LAR 95a]. These logics make it possible to express very precise and subtle properties (e.g. the timed bisimilarity). Model checking these timed modal logics is usually EXPTIME-complete [ACE 02].

4.4.5. *Testing automata*

It is sometimes easy to describe a property to be checked with a timed testing automaton $\mathcal{T}_\phi$. The idea is then to synchronize $\mathcal{T}_\phi$ with the system under verification, and the property checking reduces to some reachability problem of a bad state (when the system does not satisfy the property) or a good state (when the system meets the property) [ACE 98]. The relationship between this approach and the timed modal logics is studied in [ACE 03].

4.4.6. *Behavioral equivalences*

As for the untimed systems, it is also possible to compare timed systems with respect to several behavioral equivalences. For example, we can consider the timed bisimulation: two states s and s' are said to be strongly timed bisimilar when any transition from s can be simulated from s' by a transition with the same label (the same action or the same amount of time) and the successor states have to also be strongly timed bisimilar. This equivalence is very strong: two systems that are strongly bisimilar cannot be distinguished by any temporal or modal logics mentioned previously, they satisfy exactly the same formulae. Deciding whether two timed automata are strongly timed bisimilar is an EXPTIME-complete problem [LAR 00].

Other equivalences can be considered, for example the time-abstract bisimulation (mentioned in section 4.3 concerning the region graph technique) only requires that a delay transition is simulated by another delay transition (however, possibly with another amount of time).

4.5. Some extensions of timed automata

To help model real systems, it might be useful to manipulate a high-level description language. Hence, several extensions of timed automata havebeen considered in other works, and we will present some of them in this section. For each of these extensions, we will be interested in: 1) its decidability, 2) its expressiveness with respect to the original model, 3) its conciseness with respect to the original model. The first item is crucial if we aim at using this extension for modeling real systems. It is also important to have models which can express many systems (item 2) ensures that we can model many systems, whereas item 3) characterizes how easy it is to model systems: the smaller a model is, the more readable it is.

4.5.1. *Diagonal clock constraints*

In the timed automata model we have presented, clock constraints that can be used on transitions are rather simple and can only compare the value of a clock with a constant. In [ALU 90, ALU 94a], another type of constraint was mentioned, the diagonal constraint, which allow tests of the form $x - y \bowtie c$ where x and y are clocks, $\bowtie$ is a comparison operator and c is an integer. The extended timed automata model using this kind of constraint satisfies the following properties:

– checking reachability properties is also a PSPACE-complete problem [ALU 94a];

– diagonal constraints do not add expressiveness to the original model [BÉR 98];

– diagonal constraints give conciseness to the model [BOU 05a].

The decidability of the reachability problem was already proved in the original papers [ALU 90, ALU 94a], and also relies on the construction of a region equivalence, which refines the one presented in section 4.3, and the complexity remains the same. The second property (which concerns the expressiveness) is well-known, and it has been proved in [BÉR 98]: it consists of removing one-by-one diagonal constraints, and of building an equivalent timed automaton without diagonal constraints (the equivalence is the strong timed bisimulation). The construction which removes one diagonal constraint is illustrated on Figure 4.4 (here, we remove the constraint $x - y \leqslant c$ where c is a non-negative integer). The key idea of the construction is that the truth value of the diagonal constraint $x - y \leqslant c$ remains unchanged when time elapses, and can only change when one of the two clocks x or y is reset to zero. Thus, we make two copies of the original automaton: in one of them the constraint $x - y \leqslant c$ will be satisfied, whereas in the other one, $x - y > c$ will hold. When x or y is reset to zero, we move from one copy to the other one, depending on the values of the clocks. For instance, if we reset clock y, we move to the copy where $x - y \leqslant c$ holds if $x \leqslant c$, and we move to the copy where $x - y > c$ holds if $x > c$. This construction doubles the size of the automaton, inducing an exponential blowup (in the number of diagonal constraints) for removing all diagonal constraints. This exponential blowup is unavoidable in general, as timed automata with diagonal constraints are exponentially more concise than classical timed automata [BOU 05a], which means that systems can be modeled using exponentially more succinct automata if diagonal constraints are used.

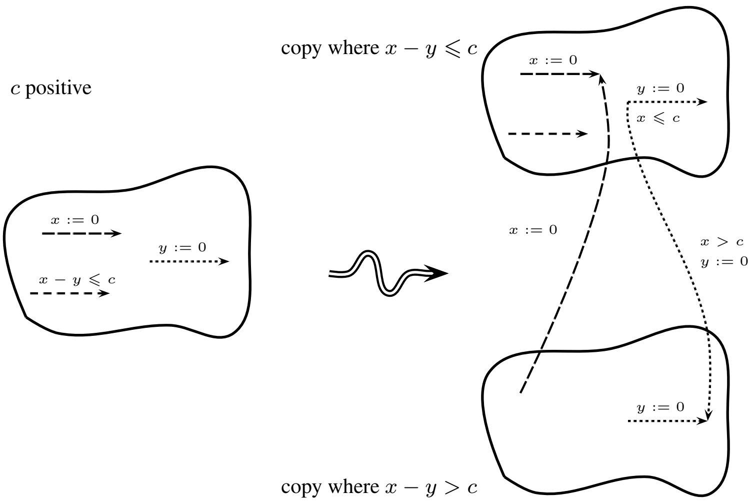


Figure 4.4. *Diagonal constraints are removed one-by-one*

4.5.2. *Additive clock constraints*

Other types of constraints can be added to the model of timed automata. We consider here the so-called additive clock constraints, i.e., constraints of the form $x + y \bowtie c$ where x and y are clocks, $\bowtie$ is a comparison operator and c is a positive integer. This extension has been studied in [BÉR 00], and it makes it possible to recognize timed languages which are not recognized by any classical timed automaton. The automaton in Figure 4.5 recognizes such a timed language: actions a are done at time points $\frac{1}{2}$, $\frac{3}{4}$, $\frac{7}{8}$, $\frac{15}{16}$, etc.

$$L^{+} = \{(a^n, t_1 \cdots t_n) \mid n \geqslant 1 \text{ and } t_i = 1 - \tfrac{1}{2^i}\}$$

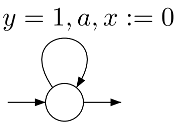


Figure 4.5. *A timed language not recognized by any classical timed automaton*

This model of timed automata with additive clock constraints satisfies the following properties [BÉR 00]:

– checking reachability properties in this extended model is decidable when restricting to automata with two clocks;

– checking reachability properties in this extended model is undecidable for automata with four clocks or more.

The decidability of the model with no more than two clocks also relies on the construction of a region equivalence, the set of regions being a refinement of the classical set of regions (see Figure 4.6). For models with four clocks or more, the model becomes undecidable. The proof is rather involved but also interesting, and uses the small automaton of Figure 4.5 several times.

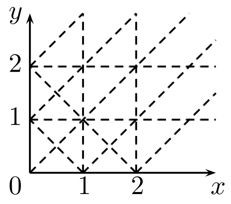


Figure 4.6. *Region equivalence for timed automata with additive clock constraints and two clocks*

Note that it is not known whether the reachability problem is decidable or not for timed automata with additive clock constraints and three clocks. However, it has been proved that a simple extension of the classical construction based on an equivalence of finite index cannot be used [ROB 04].

4.5.3. *Internal actions*

In finite automata, it is well-known that internal actions (also called ε-transitions in this context) can be removed and hence do not increase the expressiveness of finite automata (see for instance [HOP 79]). The timed automata framework is, maybe surprisingly, much different: though internal actions do not change anything to the decidability of reachability properties (the construction of the region automaton can be done similarly), they add expressive power to the model [BÉR 98]. The automaton depicted in Figure 4.7 recognizes a timed language that cannot be recognized by a classical timed automaton. This language is the set of timed words over a single letter a, where every a is done at an integral even date: at every two time units, either the transition labeled by a is taken, or the transition labeled by ε is taken.

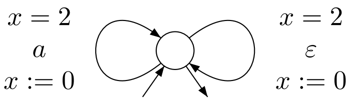


Figure 4.7. *A timed language not recognized by any classical timed automaton*

4.5.4. *Updates of clocks*

In the original model, there is only a single operation that can modify the value of the clocks (apart from time elapsing), which is the reset to zero. It is hence natural to consider more complicated operations on clocks. An update is an operation of the form $x :\bowtie c$ or $x :\bowtie y + c$, where x and y are clocks, $\bowtie$ is a comparison operator, and c is a constant. For instance, the update $x :\leqslant c$ means that we assign non-deterministically a value smaller than (or equal to) c to the clock x; the update $x := y - 1$ means that we assign to clock x the value of y decremented by 1. Classical reset to zero hence corresponds to updates of the form $x := 0$. Timed automata that use these updates, called updatable timed automata, have been studied in [BOU 04b]. It is fairly straightforward to check that the general model is undecidable as all these updates are rather powerful (it is possible to increment, decrement, test to zero). However, several subclasses have been proved decidable, and we summarize some of the most noticeable results of [BOU 04b]. The reachability problem is:

– decidable for timed automata with updates of the form $x := c$;

– decidable for timed automata with self-incrementation[3] but without diagonal constraints;

– undecidable for timed automata with self-incrementation and diagonal constraints;

– undecidable for timed automata with self-decrementation[4].

Once more, decidability results are consequences of a region automaton construction. The refinement of the region equivalence is illustrated in Figure 4.8 for timed automata with clock constraints $\{x - y < 1,\ y > 1\}$ and updates $\{x := 0,\ y := 0,\ y := 1\}$. The classical set of regions would be the set of regions depicted with dashed lines, but it is not correct in that wider framework (the image of the gray region by update $y := 1$ overlaps two regions and does not satisfy a time-abstract bisimulation property); it is thus necessary to refine and add the dotted line to distinguish $x = 2$.

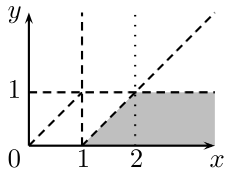


Figure 4.8. *A set of regions for automata using constraints* $\{x - y < 1, y > 1\}$ *and updates* $\{x := 0, y := 0, y := 1\}$

3. I.e., with updates of the form $x := x + 1$.

4. I.e., with updates of the form $x := x - 1$.

The reachability problem is undecidable for timed automata using self-decrementation. Indeed, we can easily encode the behavior of a two-counter machine with these automata: the value of counter C is stored in clock x_c, incrementing this counter is encoded by letting one time unit elapse, and then by decrementing the clock associated with the second counter by 1; decrementing this counter is directly encoded using the update $x_c := x_c - 1$.

Note that classes of updatable timed automata that have been proved decidable can be transformed into equivalent classical timed automata with internal actions [BOU 04b][5]. On the other hand, these updatable timed automata are exponentially more concise than classical timed automata [BOU 05a].

Finally, it is worth noticing that updates of clocks are a kind of macros which are very useful to model real systems. For instance, we can mention the modelization scheduling problems which naturally uses updates [FER 02].

4.5.5. *Linear hybrid automata*

Linear hybrid automata extend timed automata in several directions:

– general linear constraints can be used, for instance constraints of the form

$$3x_1 + 4x_2 - 2x_3 < 56;$$

– very rich updates can be used, for instance, affine functions on variables;

– derivatives of variables can change from one location to the other: instead of having only clocks (whose derivative is always 1), we can use dynamical variables.

In fact, even just one of these extensions lead to the undecidability of all verification questions! We have already mentioned that for additive clock constraint, and updates of clocks. It is also the case for variables with several possible slopes in the model: the reachability problem is undecidable for timed automata in which a single variable can have two different slopes. We refer to [HEN 98] or more recently to [RAS 05] for surveys on these questions, where some of the decidable classes of linear hybrid automata are described.

Note that looking for decidable subclasses of hybrid automata is an important research topic (for instance, rectangular initialized hybrid automata are decidable [HEN 98], and o-minimal hybrid automata are decidable [LAF 00]). It is also important to find either semi-algorithms,[6] or approximation and optimization algorithms

5. The equivalence relation is then the timed language equivalence.

6. That is, computation procedures that may not terminate. A semi-algorithm can then answer either “The property is satisfied”, “The property is not satisfied”, or “I don’t know”.

for undecidable classes of hybrid automata. Indeed, undecidable classes can have a great interest in practice, like the class of p-automata [BÉR 99] for the description of telecommunication protocols, or stopwatch automata (i.e., timed automata in which clocks can be stopped for a while) for scheduling problems with pre-emption.

Most of these methods rely on the manipulation of linear constraints to represent a set of states of the system [ALU 95], and algorithms that use polyhedra libraries (as the Parma Polyhedra Library[7]). One of the most prominent tools for linear hybrid systems is HyTech, which allows both the step-by-step computation of successors (or predecessors) of sets of states, and fix-point computations (although without a guarantee that the computation will terminate). See [HEN 97] for more details and examples. The tool HyTech can be downloaded at `http://www-cad.eecs.berkeley.edu/~tah/HyTech/`.

4.6. Subclasses of timed automata

As explained above, timed automata are a very expressive formalism and almost every extension leads to undecidability of verification problems. Instead of extending the expressiveness of timed automata, it is also possible to consider restricted versions in order to obtain new properties, for instance efficient algorithms. In this section, we present some of the classical restricted classes: the event-recording automata, the one-clock timed automata and the timed extensions of classical Kripke structures.

4.6.1. *Event-recording automata*

In this subclass, the set of automata clocks is $X_\Sigma = \{x_a \mid a \in \Sigma\}$: every action has a corresponding clock and every clock is associated with an action. The definition of the event-recording automata (see [ALU 94b]) also requires that clock x_a is reset to zero when an a-transition is performed. Then, given a configuration (q, v), the value $v(x_a)$ is the time elapsed since the last occurrence of an a action[8].

The event-recording automata (ER-TA) have important properties. First, non-determinism can be removed: any ER-TA can be transformed into a deterministic ER-TA. Secondly, the timed languages associated with this class are closed under complement. However, note that from the complexity point of view, there is no change: emptiness checking remains PSPACE-complete (the same holds for TCTL model checking).

7. See `http://www.cs.unipr.it/ppl/`.

8. A variant – the *event-predicting* automata – consists of storing in $v(x_a)$ the amount of time *before* the next action a (in this case, clocks are initialized with a negative value).

4.6.2. *One-clock timed automata*

In [COU 92], it is shown that reachability of a control location in timed automata with three clocks is PSPACE-hard. This has prompted the study of model checking for timed automata with one or two clocks.

In [LAR 04], the following results have been proved for the one-clock timed automata (1C-TA for short):

– reachability of a control location in 1C-TA is NLOGSPACE-complete (i.e. the same complexity class as reachability in standard graphs);

– there exist polynomial time algorithms for model checking $\mathsf{TCTL}_{\leqslant,\geqslant}$ over 1C-TA. ($\mathsf{TCTL}_{\leqslant,\geqslant}$ is the fragment of TCTL where timing constraints "$= c$" are forbidden);

– model checking TCTL over 1C-TA is PSPACE-complete.

The main result is that model checking 1C-TA can be done efficiently if the specification is stated with $\mathsf{TCTL}_{\leqslant,\geqslant}$. Note that the complexity blow-up induced by the *punctuality* (the constraint "$= c$") occurs in other cases (for instance, in the case of linear-time timed logics [ALU 96]).

The model checking algorithm for $\mathsf{TCTL}_{\leqslant,\geqslant}$ over 1C-TA works as follows. Given a 1C-TA $\mathcal{A}$ and a $\mathsf{TCTL}_{\leqslant,\geqslant}$ formula Φ, we compute, for any state q and any subformula ψ, the set of valuations v such that $(q, v) \models \psi$. As $\mathcal{A}$ has only one clock, such a valuation v is a unique value, and the sets of valuations $\mathsf{Sat}[q, \psi]$ can be represented as a union of intervals $\bigcup_i \langle \alpha_i, \beta_i \rangle$, with $\langle \in \{[, (\}, \rangle \in \{],)\}$ and $\alpha_i, \beta_i \in \mathbb{N} \cup \{\infty\}$. We can show that the number of intervals in $\mathsf{Sat}[\ell, \varphi]$ is bounded by $2 \cdot |\varphi| \cdot |\mathcal{A}|$.

For example, consider the subformula $\mathsf{E}\phi\mathsf{U}_{\leqslant c}\psi$ and assume that the sets $\mathsf{Sat}[q, \varphi]$ and $\mathsf{Sat}[q, \psi]$ have already been computed for any q. The aim is to compute the minimal duration – denoted $\delta^{\min}(q, v)$ – to reach from (q, v) a ψ state along a path satisfying φ. To compute the function $\delta^{\min}$, we first build a simplified region automaton (its size is polynomial in $|\mathcal{A}|$ and $|\Phi|$) whose states are pairs (q, γ), where γ is an interval of valuations such that the truth values of φ, ψ and any guard in $\mathcal{A}$ do not change along γ. This property entails that the function $\delta^{\min}$ has a special form over every γ: it is either decreasing with slope -1 (the shortest paths to reach ψ go through the rightmost position of γ), or constant (the shortest paths to reach ψ have to perform an action transition with a reset of the clock before any delay transition) or it combines the two previous cases, that is, it is first constant over an subinterval of γ and then decreasing. Thus, every function $\delta^{\min}_{|(q,\gamma)}$ can be defined by its value on the leftmost and rightmost positions of γ, and these value can be easily computed with a shortest path algorithm. It remains to use the threshold $\leqslant c$ to find the intervals where $\mathsf{E}\phi\mathsf{U}_{\leqslant c}\psi$ is true [LAR 04].

Note also that one-clock timed automata have other very interesting properties:

– the timed language inclusion is decidable for finite words for 1C-TA [OUA 04] (but remains undecidable for infinite words [ABD 05]);

– checking emptiness is decidable for one-clock *alternating* timed automata [LAS 05, OUA 05];

– model checking one-clock *probabilistic* timed automata can be done in polynomial time for $\mathsf{PTCTL}_{\leqslant,\geqslant}$ formulae, and almost sure reachability is P-complete [JUR 07];

– model checking WCTL is decidable (in PSPACE) for the *priced* timed automata with one clock [BOU 07a]: WCTL is a specification language (its syntax is the same as TCTL) to express quantitative properties over the cost of executions in priced timed automata (where a cost slope is associated with every location);

– computing optimal costs can be done for *priced* timed *game* with one clock [BOU 06].

Note that these properties no longer hold when the timed automata have two clocks (reachability is NP-hard for two-clocks timed automata [LAR 04], almost-sure reachability is EXPTIME-complete in two clocks probabilistic timed automata [JUR 07], etc.).

4.6.3. *Discrete-time models*

Instead of considering $\mathbf{R}$, it is possible to use a discrete-time domain. For example, we can consider the semantics of timed automata with integral clocks. This change does not modify the main complexity results of model checking: for example, reachability and the TCTL model checking remain PSPACE-complete. To obtain polynomial time algorithms, we need to consider simpler models.

In many works, classical Kripke structures have been used to model real-time systems with the hypothesis that every transition takes exactly one time unit. In this case, we can use TCTL formulae to specify quantitative properties (over the number of transitions) along the paths. With this simple approach, there exist polynomial time model checking algorithms for model checking TCTL [EME 92], and they can also be extended to models where the transitions take 0 or 1 time unit [LAR 03].

A natural extension consists of associating integral durations with the transitions of a Kripke structure. Several semantics can be defined for these systems. In this framework, the main interesting result is that model checking $\mathsf{TCTL}_{\leqslant,\geqslant}$ can be done in polynomial time (contrary to TCTL, whose model checking is either Δ_2^p-complete or PSPACE-complete depending on the semantics which is chosen) [LAR 06]. This polynomial-time algorithm has been implemented in the tool TSMV [MAR 04].

4.7. Algorithms for timed verification

In this section, we describe algorithms implemented in tools such as Uppaal or Kronos for verifying timed automata. Indeed, in practice the region automaton construction is not used in tools because the region partition is too refined and hence it is not efficient to manipulate the regions. Tools should use the symbolic representation called *zones*, and rely on on-the-fly algorithms.

There are mainly two families of (semi-)algorithms for analyzing reachability properties of systems. The first, called forward analysis, consists of computing iteratively the successors of the initial states and of checking that the state we want to reach is eventually computed or not. The second, called backward analysis, consists of computing iteratively the predecessors of the states we want to reach and of checking that an initial state is eventually computed or not. These methods are generic and are used in many contexts, for instance on the model of linear hybrid automata that we already mentioned in section 4.5.

Before presenting these analysis methods, we first present the most commonly used symbolic representation for the verification of timed systems.

4.7.1. *A symbolic representation for timed automata: the zones*

The set of configurations of a timed automaton is infinite. To verify this model, it is thus mandatory to be able to manipulate large sets of configurations and thus to have an efficient symbolic representation for these sets of states. The most commonly used is the zone representation: a zone is a set of valuations defined by a conjunction of atomic constraints of the form $x \bowtie c$ or $x - y \bowtie c$ where x and y are clocks, $\bowtie$ is a comparison operator, and c is a constant. Thus, in the forward and backward analysis algorithms, objects that are manipulated are pairs (ℓ, Z) where ℓ is a location and Z a zone.

Many operations can be performed using this representation:

– the future of a zone Z, defined by $\overrightarrow{Z} = \{v + t \mid v \in Z \text{ and } t \in \mathbb{T}\}$;

– the past of a zone Z, defined by $\overleftarrow{Z} = \{v - t \mid v \in Z \text{ and } t \in \mathbb{T}\}$;

– the intersection of Z and Z', defined by $Z \cap Z' = \{v \mid v \in Z \text{ and } v \in Z'\}$;

– the reset to zero $r \subseteq X$ of Z, defined by $[r \leftarrow 0]Z = \{[r \leftarrow 0]v \mid v \in Z\}$;

– the relaxation with respect to $r \subseteq X$ of Z, defined by $[r \leftarrow 0]^{-1}Z = \{v \mid [r \leftarrow 0]v \in Z\}$.

These operations, defined as first order formulae over zones, preserve zones (see the Fourier-Motzkin elimination principle [SCH 98]).

We now present the backward analysis algorithm as it is the simplest one. We will then turn to the forward analysis algorithm, which is indeed the most commonly used method, but also the most complicated one.

4.7.2. *Backward analysis in timed automata*

As already said, the backward analysis consists of computing step-by-step the predecessors of the final configurations, starting with the one step predecessors, then the two steps predecessors, etc. and of checking whether an initial state is eventually computed. If such an initial state is computed, it means that the goal location is reachable, and if such an initial state is not computed, it means that the goal location is not reachable. The principle of the backward analysis is illustrated in Figure 4.9.

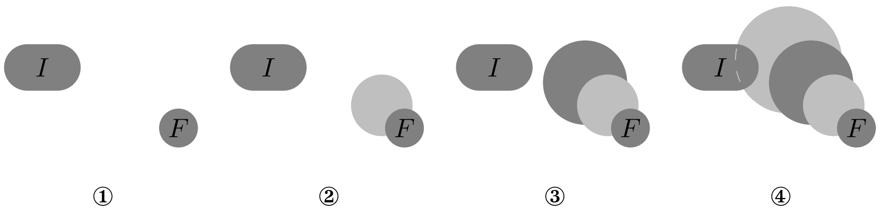


Figure 4.9. *Backward analysis: step-by-step, predecessors of goal states are computed*

One step of the backward analysis can easily be computed using zones. Indeed, if $t = \ell \xrightarrow{g,a,r} \ell'$ is a transition of the automaton and if Z' is a zone, the set of one-step predecessors of (ℓ', Z') when taking transition t is the set of configurations (ℓ, v) where v is in the zone $Z = \overleftarrow{g \cap [r \leftarrow 0]^{-1}(Z' \cap (r = 0))}$.

The characteristic of the backward analysis is that the iterative computation always terminates: indeed it can be proved that if Z' is a zone and if this zone is a union of regions (see section 4.3), then the zone Z' we have described before is a zone and also a union of regions! As there are finitely many regions, there are only finitely many pairs (ℓ, Z) which can be computed.

Though the backward analysis has some non-negligible qualities, in practice, it is not commonly implemented in tools, and forward analysis is preferred. There are numerous reasons for this implementation choice: forward analysis only visits reachable states, *i.e.*, states that are relevant in the system; furthermore, backward analysis is not appropriate for verifying systems defined with high-level data structures such as integral variables or C-like instructions, etc. For instance, the Uppaal tool (see section 4.8) only implements the forward analysis paradigm.

4.7.3. *Forward analysis of timed automata*

As previously stated, forward analysis consists of computing step-by-step the successors of the initial configurations, starting with the one step successors, then the two steps successors, etc. and of checking whether a goal location is eventually computed. If such a final location is computed, it means that the goal location is reachable, and if such an initial state is not computed, it means that the goal location is not reachable. The principle of the forward analysis is illustrated in Figure 4.10.

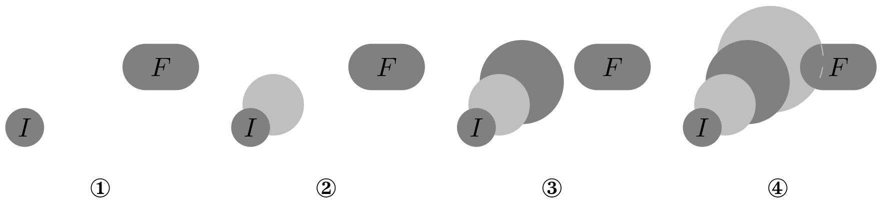


Figure 4.10. *Forward analysis: step-by-step, successors of initial states are computed*

One step of the forward analysis algorithm can be computed using zones. Indeed, if $t = \ell \xrightarrow{g,a,r} \ell'$ is a transition of the timed automaton and if Z is a zone, the set of successors in one step of (ℓ, Z) by taking transition t is the set of states (ℓ', v') where v' belongs to the zone $Z' = [r \leftarrow 0](g \cap \overrightarrow{Z})$.

Contrary to the backward computation, the iterative forward computation does not terminate in general. This is illustrated by the timed automaton of Figure 4.11. In this example, each iteration of the algorithm increases the value of the clock by 1, and the computation will thus never terminate.

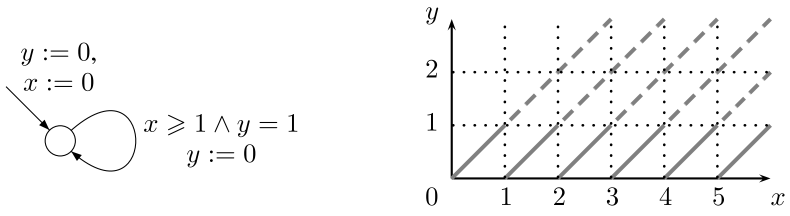


Figure 4.11. *The iterative forward computation may not terminate*

To overcome this termination problem, an abstraction operator is usually applied at each iteration of the computation. In other works, this abstraction operator is called normalization, or extrapolation; we use the latter formulation here. We assume that k is the largest constant appearing in the constraints of the timed automaton. If Z is a zone, the extrapolation of Z with respect to k is the smallest zone which contains Z and which is defined with constants between $-k$ and $+k$. The idea behind this operator is the following: the automaton cannot distinguish between clock values above k, and it may thus not be relevant to keep in the zone information above k. It is worth noticing first that applying this extrapolation at each iteration of the algorithm ensures termination of the computation as there are only finitely many zones defined with constants between $-k$ and $+k$. However, there is another problem: at each iteration, an over-approximation of the set of states which is actually reachable is computed. It may thus happen that a location is computed whereas it is not reachable.

In [BOU 04a], it is proven that this iterative and abstracted forward computation is correct for the class of timed automata without diagonal constraints, but incorrect for the class of timed automata with diagonal constraints.

4.7.4. *A data structure for timed systems: DBMs*

The DBM acronym stands for *difference bound matrix*. It is a rather classical data structure used for representing systems of difference constraints [COR 90], and they have in particular a great interest for the verification of timed systems because they can be used to represent zones. DBMs were first used to analyze time Petri nets [BER 83], and they are now intensively used to analyze timed automata [DIL 90].

If n is the number of clocks, a DBM M is an $(n+1)$-square matrix with integral coefficients[9]. If $M = (m_{i,j})_{0 \leqslant i,j \leqslant n}$, the coefficient $m_{i,j}$ represents the constraint $x_i - x_j \leqslant m_{i,j}$ where $\{x_i \mid 1 \leqslant i \leqslant n\}$ is the set of clocks and x_0 is a fictitious clock whose value is always 0. Thus, to represent a constraint $x_i \leqslant 6$, we will write $m_{i,0} = 6$ as this constraint is equivalent to $x_i - x_0 \leqslant 6$.

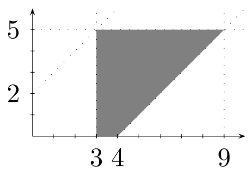


Figure 4.12. *Zone defined by the constraint* $(x_1 \geqslant 3) \wedge (x_2 \leqslant 5) \wedge (x_1 - x_2 \leqslant 4)$

The following DBM represents the set of valuations defined by the constraint $(x_1 \geqslant 3) \wedge (x_2 \leqslant 5) \wedge (x_1 - x_2 \leqslant 4)$, and is represented in Figure 4.12:

$$\begin{array}{cc} & \begin{array}{ccc} x_0 & x_1 & x_2 \end{array} \\ \begin{array}{c} x_0 \\ x_1 \\ x_2 \end{array} & \left(\begin{array}{ccc} +\infty & -3 & +\infty \\ +\infty & +\infty & 4 \\ 5 & +\infty & +\infty \end{array} \right) \end{array}.$$

9. In general, coefficients need to be pairs $(m, \prec)$ where m is an integer and $\prec$ is either $<$, or $\leqslant$, but here, to simplify the presentation, we forget about comparison operators in DBMs.

A coefficient $+\infty$ means that there is no constraint on the corresponding clock difference. The coefficient $m_{0,1} = -3$ represents the constraints $x_1 \geqslant 3$ because this constraint is equivalent to $x_0 - x_i \leqslant -3$.

Every DBM represents a zone, and every zone can be represented by a DBM. However, a zone can be represented by several DBMs (for instance, in the previous DBM, if we replace the coefficient $m_{1,0} = +\infty$ by $m_{1,0} = 9$, it will not change the zone which is represented). There exists a normal form for DBMs, which can be computed using the Floyd-Warshall shortest paths algorithm [COR 90]: the DBM which is obtained stores the strongest constraints which define the corresponding zone. For the previous example, the normal form DBM is:

$$\begin{pmatrix} 0 & -3 & 0 \\ 9 & 0 & 4 \\ 5 & 2 & 0 \end{pmatrix}.$$

All operations that are needed for both the backward and the forward analysis iterative computations can be done using DBMs (see [CLA 99, BOU 04a] for a detailed description of operations using DBMs).

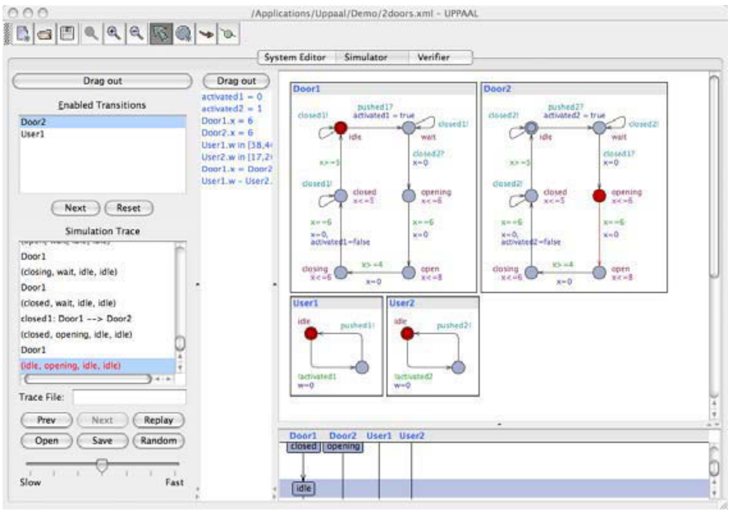
DBMs are basic data structures for manipulating sets of configurations of timed automata, but several improvements can be made, for instance, a minimization of DBMs [LAR 97a] or the use of CDDs (*clock difference diagrams*) [LAR 99] or more recently of federations [DAV 06], which makes it possible to represent and manipulate more compactly the unions of DBMs.

4.8. The model-checking tool Uppaal

Uppaal is a model-checking tool for verifying timed systems. It has been jointly developed by Uppsala University (Sweden) and Aalborg University (Denmark) [LAR 97b]. (This tool can be downloaded at `http://www.uppaal.com/`.) The model that can be verified by Uppaal is a variant of the classical timed automata model. This model is syntactically very rich as we can explicitly add urgency in the model (for instance, we can enforce a transition to be taken immediately, without any delay), we can enforce atomicity of several transitions (a sequence of transitions must then be taken instantaneously), we can add C-like instructions, etc. All these features do not add expressivity to the model, but they make the modeling phase easier, as we can build rather concise and readable models.

Properties that can be verified using the Uppaal tool are reachability properties, safety properties, and response properties. A tutorial for this tool is available online [BEH 04].

Apart from the modeling GUI and the verification module, Uppaal has a simulation module in which it is possible to "play" with the model and hence have a first



Door2
User1

activated1 = 0
activated2 = 1
Door1.x = 6
Door2.x = 6
User1.w in [38,4
User2.w in [17,2
Door1.x = Door2
User1.w - User2.

Door1
(closing, wait, idle, idle)
Door1
(closed, wait, idle, idle)
closed1: Door1 --> Door2
(closed, opening, idle, idle)
Door1
(idle, opening, idle, idle)

Figure 4.13. *The* Uppaal *tool*

check that the model does what it is expected to do. A screenshot of the tool is given in Figure 4.13.

The Uppaal tool has been developed for more than ten years, and it has been successfully used to verify industrial systems. For instance, we can mention audio protocols like [BEN 02], or the Bang & Olufsen protocol whose analysis has located a known bug, and for which a validated correction has been provided [HAV 97].

The current version of Uppaal is 4.0 and the new features are described in [BEH 06].

4.9. Bibliography

[ABD 05] ABDULLA P. A., DENEUX J., OUAKNINE J., WORRELL J., "Decidability and complexity results for timed automata via channel machines", *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP'05)*, vol. 3580 of *Lecture Notes in Computer Science*, Springer, p. 1089–1101, 2005.

[ACE 98] ACETO L., BURGUEÑO A., LARSEN K. G., "Model-checking via reachability testing for timed automata", *Proc. 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, vol. 1384 of *Lecture Notes in Computer Science*, Springer, p. 263–280, 1998.

[ACE 02] ACETO L., LAROUSSINIE F., "Is your model-checker on time? On the complexity of model-checking for timed modal logics", *Journal of Logic and Algebraic Programming*, vol. 52–53, p. 7–51, 2002.

[ACE 03] ACETO L., BOUYER P., BURGUEÑO A., LARSEN K. G., "The power of reachability testing for timed automata", *Theoretical Computer Science*, vol. 300, num. 1–3, p. 411–475, 2003.

[ALU 90] ALUR R., DILL D., "Automata for modeling real-time systems", *Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, vol. 443 of *Lecture Notes in Computer Science*, Springer, p. 322–335, 1990.

[ALU 93a] ALUR R., COURCOUBETIS C., DILL D., "Model-checking in dense real-time", *Information and Computation*, vol. 104, num. 1, p. 2–34, 1993.

[ALU 93b] ALUR R., HENZINGER TH. A., "Real-time logics: complexity and expressiveness", *Information and Computation*, vol. 104, num. 1, p. 35–77, 1993.

[ALU 94a] ALUR R., DILL D., "A theory of timed automata", *Theoretical Computer Science*, vol. 126, num. 2, p. 183–235, 1994.

[ALU 94b] ALUR R., FIX L., HENZINGER TH. A., "A determinizable class of timed automata", *Proc. 6th International Conference on Computer Aided Verification (CAV'94)*, vol. 818 of *Lecture Notes in Computer Science*, Springer, p. 1–13, 1994.

[ALU 94c] ALUR R., HENZINGER TH. A., "A really temporal logic", *Journal of the ACM*, vol. 41, num. 1, p. 181–204, 1994.

[ALU 95] ALUR R., COURCOUBETIS C., HALBWACHS N., HENZINGER TH. A., HO P.-H., NICOLLIN X., OLIVERO A., SIFAKIS J., YOVINE S., "The algorithmic analysis of hybrid systems", *Theoretical Computer Science*, vol. 138, num. 1, p. 3–34, 1995.

[ALU 96] ALUR R., FEDER T., HENZINGER TH. A., "The benefits of relaxing punctuality", *Journal of the ACM*, vol. 43, num. 1, p. 116–146, 1996.

[BEH 04] BEHRMANN G., DAVID A., LARSEN K. G., "A tutorial on Uppaal", *Proc. 4th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Real-Time (SFM-04:RT)*, vol. 3185 of *Lecture Notes in Computer Science*, Springer, p. 200–236, 2004.

[BEH 06] BEHRMANN G., DAVID A., LARSEN K. G., HÅKANSSON J., PETTERSSON P., YI W., HENDRIKS M., "Uppaal 4.0", *Proc. 3rd International Conference on the Quantitative Evaluation of Systems (QEST'06)*, IEEE Computer Society Press, p. 125–126, 2006.

[BEN 02] BENGTSSON J., GRIFFIOEN W. D., KRISTOFFERSEN K. J., LARSEN K. G., LARSSON F., PETTERSSON P., YI W., "Automated verification of an audio-control protocol using Uppaal", *Journal of Logic and Algebraic Programming*, vol. 52–53, p. 163–181, 2002.

[BER 83] BERTHOMIEU B., MENASCHE M., "An enumerative approach for analyzing time Petri nets", *Proc. IFIP 9th World Computer Congress*, vol. 83 of *Information Processing*, North-Holland/ IFIP, p. 41–46, 1983.

[BÉR 98] BÉRARD B., DIEKERT V., GASTIN P., PETIT A., "Characterization of the expressive power of silent transitions in timed automata", *Fundamenta Informaticae*, vol. 36, num. 2–3, p. 145–182, 1998.

[BÉR 99] BÉRARD B., FRIBOURG L., "Automated verification of a parametric real-time program: the ABR Conformance Protocol", *Proc. 11th International Conference on Computer Aided Verification (CAV'99)*, vol. 1633 of *Lecture Notes in Computer Science*, Springer, p. 96–107, 1999.

[BÉR 00] BÉRARD B., DUFOURD C., "Timed automata and additive clock constraints", *Information Processing Letters*, vol. 75, num. 1–2, p. 1–7, 2000.

[BOU 04a] BOUYER P., "Forward Analysis of Updatable Timed Automata", *Formal Methods in System Design*, vol. 24, num. 3, p. 281–320, 2004.

[BOU 04b] BOUYER P., DUFOURD C., FLEURY E., PETIT A., "Updatable timed automata", *Theoretical Computer Science*, vol. 321, num. 2–3, p. 291–345, 2004.

[BOU 05a] BOUYER P., CHEVALIER F., "On conciseness of extensions of timed automata", *Journal of Automata, Languages and Combinatorics*, vol. 10, num. 4, p. 393–405, 2005.

[BOU 05b] BOUYER P., CHEVALIER F., MARKEY N., "On the expressiveness of TPTL and MTL", *Proc. 25th Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS'05)*, vol. 3821 of *Lecture Notes in Computer Science*, Springer, p. 432–443, 2005.

[BOU 06] BOUYER P., LARSEN K. G., MARKEY N., RASMUSSEN J. I., "Almost optimal strategies in one-clock priced timed automata", *Proc. 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS'06)*, vol. 4337 of *Lecture Notes in Computer Science*, Springer, p. 345–356, 2006.

[BOU 07a] BOUYER P., LARSEN K. G., MARKEY N., "Model-checking one-clock priced timed automata", *Proc. 10th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'07)*, vol. 4423 of *Lecture Notes in Computer Science*, Springer, p. 108–122, 2007.

[BOU 07b] BOUYER P., MARKEY N., OUAKNINE J., WORRELL J., "The cost of punctuality", *Proc. 21st Annual Symposium on Logic in Computer Science (LICS'07)*, IEEE Computer Society Press, p. 109–118, 2007.

[CLA 81] CLARKE E. M., EMERSON E. A., "Design and synthesis of synchronous skeletons using branching-time temporal logic", *Proc. 3rd Workshop on Logics of Programs (LOP'81)*, vol. 131 of *Lecture Notes in Computer Science*, Springer Verlag, p. 52–71, 1981.

[CLA 99] CLARKE E., GRUMBERG O., PELED D., *Model checking*, The MIT Press, Cambridge, Massachusetts, 1999.

[COR 90] CORMEN TH. H., LEISERSON C. E., RIVEST R. L., *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1990.

[COU 92] COURCOUBETIS C., YANNAKAKIS M., "Minimum and maximum delay problems in real-time systems", *Formal Methods in System Design*, vol. 1, num. 4, p. 385–415, 1992.

[DAV 06] DAVID A., "Merging DBMs efficiently", *Proc. 17th Nordic Workshop on Programming Theory*, DIKU, University of Copenhagen, p. 54–56, 2006.

[DIL 90] DILL D., "Timing assumptions and verification of finite-state concurrent systems", *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems (1989)*, vol. 407 of *Lecture Notes in Computer Science*, Springer, p. 197–212, 1990.

[EME 91] EMERSON E. A., *Temporal and Modal Logic*, vol. B (Formal Models and Semantics) of *Handbook of Theoretical Computer Science*, p. 995–1072, MIT Press Cambridge, 1991.

[EME 92] EMERSON E. A., MOK A. K., SISTLA A. P., SRINIVASAN J., "Quantitative temporal reasoning", *Real-Time Systems*, vol. 4, num. 4, p. 331–352, 1992.

[FER 02] FERSMAN E., PETTERSON P., YI W., "Timed automata with asynchronous processes: schedulability and decidability", *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, vol. 2280 of *Lecture Notes in Computer Science*, Springer, p. 67–82, 2002.

[HAV 97] HAVELUND K., SKOU A., LARSEN K. G., LUND K., "Formal modeling and analysis of an audio/video protocol: an industrial case study using Uppaal", *Proc. 18th IEEE Real-Time Systems Symposium (RTSS'97)*, IEEE Computer Society Press, p. 2–13, 1997.

[HEN 85] HENNESSY M., MILNER R., "Algebraic laws for nondeterminism and concurrency", *Journal of the ACM*, vol. 32, num. 1, p. 137–161, 1985.

[HEN 94] HENZINGER TH. A., NICOLLIN X., SIFAKIS J., YOVINE S., "Symbolic model-checking for real-time systems", *Information and Computation*, vol. 111, num. 2, p. 193–244, 1994.

[HEN 97] HENZINGER TH. A., HO P.-H., WONG-TOI H., "HyTech: A model-checker for hybrid systems", *Journal on Software Tools for Technology Transfer*, vol. 1, num. 1–2, p. 110–122, 1997.

[HEN 98] HENZINGER TH. A., KOPKE P. W., PURI A., VARAIYA P., "What's decidable about hybrid automata?", *Journal of Computer and System Sciences*, vol. 57, num. 1, p. 94–124, 1998.

[HOP 79] HOPCROFT J. E., ULLMAN J. D., *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

[JUR 07] JURDZIŃSKI M., LAROUSSINIE F., SPROSTON J., "Model checking probabilistic timed automata with one or two clocks", *Proc. of the 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'07)*, vol. 4424 of *Lecture Notes in Computer Science*, Springer, p. 170–184, 2007.

[KOY 90] KOYMANS R., "Specifying real-time properties with metric temporal logic", *Real-Time Systems*, vol. 2, num. 4, p. 255–299, 1990.

[LAF 00] LAFFERRIERE G., PAPPAS G. J., SASTRY S., "O-minimal hybrid systems", *Mathematics of Control, Signals, and Systems*, vol. 13, num. 1, p. 1–21, 2000.

[LAR 90] LARSEN K. G., "Proof systems for satisfiability in Hennessy-Milner logic with recursion", *Theoretical Computer Science*, vol. 72, num. 2–3, p. 265–288, 1990.

[LAR 95a] LAROUSSINIE F., LARSEN K. G., WEISE C., "From timed automata to logic – and back", *Proc. 20th International Symposium on Mathematical Foundations of Computer Science (MFCS'95)*, vol. 969 of *Lecture Notes in Computer Science*, Springer, p. 529–539, 1995.

[LAR 95b] LARSEN K. G., PETTERSSON P., YI W., "Model-checking for real-time systems", *Proc. 10th International Conference on Fundamentals of Computation Theory (FCT'95)*, vol. 965 of *Lecture Notes in Computer Science*, Springer, p. 62–88, 1995.

[LAR 97a] LARSEN K. G., LARSSON F., PETTERSSON P., YI W., "Efficient verification of real-time systems: compact data structure and state-space reduction", *Proc. 18th IEEE Real-Time Systems Symposium (RTSS'97)*, IEEE Computer Society Press, p. 14–24, 1997.

[LAR 97b] LARSEN K. G., PETTERSSON P., YI W., "UPPAAL in a nutshell", *Journal of Software Tools for Technology Transfer*, vol. 1, num. 1–2, p. 134–152, 1997.

[LAR 99] LARSEN K. G., PEARSON J., WEISE C., YI W., "Clock difference diagrams", *Nordic Journal of Computing*, vol. 6, num. 3, p. 271–298, 1999.

[LAR 00] LAROUSSINIE F., SCHNOEBELEN PH., "The state-explosion problem from trace to bisimulation equivalence", *Proc. 3rd International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'00)*, vol. 1784 of *Lecture Notes in Computer Science*, Springer, p. 192–207, 2000.

[LAR 03] LAROUSSINIE F., SCHNOEBELEN PH., TURUANI M., "On the expressivity and complexity of quantitative branching-time temporal logics", *Theoretical Computer Science*, vol. 297, num. 1, p. 297–315, 2003.

[LAR 04] LAROUSSINIE F., MARKEY N., SCHNOEBELEN PH., "Model checking timed automata with one or two clocks", *Proc. 15th International Conference on Concurrency Theory (CONCUR'04)*, vol. 3170 of *Lecture Notes in Computer Science*, Springer, p. 387–401, 2004.

[LAR 06] LAROUSSINIE F., MARKEY N., SCHNOEBELEN PH., "Efficient timed model checking for discrete-time systems", *Theoretical Computer Science*, vol. 353, num. 1–3, p. 249–271, 2006.

[LAS 05] LASOTA S., WALUKIEWICZ I., "Alternating timed automata", *Proc. 8th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'05)*, vol. 3441 of *Lecture Notes in Computer Science*, Springer, p. 250–265, 2005.

[MAR 04] MARKEY N., SCHNOEBELEN PH., "Symbolic model checking of simply-timed systems", *Proc. Joint Conference on Formal Modeling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant System (FORMATS+FTRTFT'04)*, vol. 3253 of *Lecture Notes in Computer Science*, Springer, p. 102–117, 2004.

[OUA 04] OUAKNINE J., WORRELL J., "On the language inclusion problem for timed automata: closing a decidability gap", *Proc. 19th Annual Symposium on Logic in Computer Science (LICS'04)*, IEEE Computer Society Press, p. 54–63, 2004.

[OUA 05] OUAKNINE J., WORRELL J., "On the decidability of metric temporal logic", *Proc. 19th Annual Symposium on Logic in Computer Science (LICS'05)*, IEEE Computer Society Press, p. 188–197, 2005.

[OUA 06] OUAKNINE J., WORRELL J., "On metric temporal logic and faulty turing machines", *Proc. 9th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'06)*, vol. 3921 of *Lecture Notes in Computer Science*, Springer, p. 217–230, 2006.

[PNU 77] PNUELI A., “The temporal logic of programs”, *Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS’77)*, IEEE Computer Society Press, p. 46–57, 1977.

[PNU 81] PNUELI A., “The temporal semantics of concurrent programs”, *Theoretical Computer Science*, vol. 13, num. 1, p. 45–60, 1981.

[RAS 05] RASKIN J.-F., “An introduction to hybrid automata”, chapter in *Handbook of Networked and Embedded Control Systems*, p. 491–518, Springer, 2005.

[ROB 04] ROBIN A., “Aux frontières de la décidabilité...”, Master’s thesis, DEA Algorithmique, Paris, 2004.

[SCH 98] SCHRIJVER A., *Theory of Linear and Integer Programming*, Interscience Series in Discrete Mathematics and Optimization, Wiley, 1998.

[SCH 01] SCHNOEBELEN P., BÉRARD B., BIDOIT M., LAROUSSINIE F., PETIT A., *Systems and Software Verification – Model Checking Techniques and Tools*, Springer, 2001.

[STI 01] STIRLING C., *Modal and Temporal Properties of Processes*, Texts in Computer Science, Springer, 2001.

[TRI 98] TRIPAKIS S., YOVINE S., “Verification of the fast reservation protocol with delayed transmission using the tool Kronos”, *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS’98)*, IEEE Computer Society Press, p. 165–170, 1998.

[YI 90] YI W., “Real-time behavior of asynchronous agents”, *Proc. 1st International Conference on Theory of Concurrency (CONCUR’90)*, vol. 458 of *Lecture Notes in Computer Science*, Springer, p. 502–520, 1990.

[YOV 97] YOVINE S., “Kronos: A verification tool for real-time systems”, *Journal of Software Tools for Technology Transfer*, vol. 1, num. 1–2, p. 123–133, 1997.

Chapter 5

Specification and Analysis of Asynchronous Systems using CADP

5.1. Introduction

Industrial systems that involve asynchronous parallelism, such as communication protocols, embedded systems and multiprocessor hardware architectures, exhibit complex behaviors. Furthermore, systems of this kind are often critical, since their malfunctioning may entail the loss of human lives or important damages. In order to detect possible errors as early as possible in the development cycle of these systems, the usage of formal specification and verification methods, assisted by suitable software tools, becomes mandatory.

The model-based verification method offers a good cost-performance trade-off, which explains its successful application in industry. This method consists of building, from a formal description of the system, a semantic model (state space) on which the correctness properties, expressed using a suitable formalism (automata, temporal logics), are verified automatically by means of specialized algorithms. Although limited to finite-state systems, model-based verification makes it possible to detect errors in complex systems rapidly and economically, being particularly useful during the first phases of the design process, when errors are likely to occur more frequently.

Complex industrial systems often contain asynchronous parts, consisting of several entities physically distributed that communicate and synchronize by exchanging messages, as well as "hard" real-time parts, governed by strong temporal constraints

Chapter written by Radu MATEESCU.

(e.g., delays). In this chapter, we focus on the asynchronous aspects only; the modeling and analysis of hard real-time aspects can be carried out using specific techniques and tools (see, e.g., Chapters 1, 6 and 10).

The CADP toolbox [GAR 02b] for the engineering of asynchronous systems offers a large spectrum of functionalities, which assist the design process effectively: specification, simulation, rapid prototyping, verification and test generation. The underlying tools have been designed following a modular architecture, centered around the generic OPEN/CÆSAR [GAR 98] environment for on-the-fly state space exploration, which ensures language-independence and favors the reuse of components due to well-defined software interfaces. Although several functionalities for performance evaluation have been recently added to CADP [GAR 02a], we only illustrate here functional verification, which takes into account the logical ordering of events during the execution of the system.

We demonstrate the specification and analysis methodology promoted by CADP by considering the example of an industrial critical system dedicated to the drilling of metallic products. More precisely, we detail the design of the software controller in charge of driving the various physical devices that compose the drilling unit. This system has been studied as a common example for comparing the suitability of several specification languages and the power of their associated verification tools [BOS 02, BOR 05].

This chapter is structured as follows. Section 5.2 briefly describes the LOTOS language and several verification tools of CADP used for this case study. Next, section 5.3 details the specification in LOTOS of the drilling unit and section 5.4 presents its functional verification by means of bisimulations and temporal logics. Finally, section 5.5 summarizes the chapter and gives some current research directions in the field of model-based verification techniques.

5.2. The CADP toolbox

CADP[1] (*construction and analysis of distributed processes*) [GAR 02b] is a very efficient toolbox for the specification and verification of parallel asynchronous systems. These systems consist of several entities (processes or agents) that run in parallel, and synchronize and communicate by message-passing. To model the execution of these systems, CADP uses the interleaving semantics, which relies upon the fact that each action (or event) is atomic and only one action can be observed at a given moment. Examples of asynchronous systems are: telecommunication protocols, operating systems, distributed databases, multiprocessor architectures, embedded softwares, etc.

1. See `http://www.inrialpes.fr/vasy/cadp`.

5.2.1. *The LOTOS language*

CADP accepts several input specification formalisms, ranging from high-level languages, such as LOTOS, to lower-level languages, such as networks of communicating automata. LOTOS (*language of temporal ordering specification*) [ISO 89] is a formal description technique standardized by ISO. Although initially defined for the description of communication protocols according to the OSI model, the LOTOS language is revealed to be equally suitable for describing other classes of asynchronous systems, such as those aforementioned. LOTOS consists of two "orthogonal" parts:

– *A "data" part,* based on algebraic abstract data types, and more specifically on the ACTONE language [EHR 85]. This part makes it possible to describe the data structures handled by the system by means of sorts and algebraic operations, defined using equations and pattern matching.

– *A "control" part,* which combines the best primitives of the process algebras CCS [MIL 89] and CSP [BRO 84]. This part makes it possible to describe the parallel processes composing the system as terms constructed by applying algebraic operators (action prefix, choice, parallel composition with handshake synchronization, hiding, etc.).

CADP contains two compilers for LOTOS: CÆSAR.ADT [GAR 89] translates the data part of a LOTOS specification in C (the LOTOS sorts and operations are translated as C types and functions, respectively) and CÆSAR [GAR 90] translates the control part into a C program that can be embedded in a real system or used for simulation, verification or test generation.

5.2.2. *Labeled transition systems*

The labeled transition systems (LTSs) are the semantic model underlying the specification formalisms used by CADP. An LTS M is a quadruple (S, A, T, s_0) consisting of a set of states S, a set of actions (transition labels) A, a transition relation $T \subseteq S \times A \times S$ and an initial state $s_0 \in S$. A special invisible action $\tau \notin A$ makes it possible to model the internal (unobservable) activity of the system. A transition $(s_1, a, s_2) \in T$, also noted $s_1 \xrightarrow{a} s_2$, indicates that the system can move from state s_1 to state s_2 by performing action a. CADP provides two complementary representations for LTSs:

– *An explicit representation* as the list of transitions contained in the LTS, stored in a file of a compact binary format called BCG (*binary coded graphs*). The BCG environment offers a set of tools and libraries for the manipulation of BCG files (reading/writing, graphical visualization, minimization, conversion to other formats, etc.). This explicit representation of LTSs is suitable for global (or enumerative) verification, which proceeds by a backward exploration of the transition relation and therefore requires the prior construction of the entire LTS.

– *An implicit representation* as the successor function of the LTS, encoded as a C program conforming to the application programming interface defined by the OPEN/CÆSAR environment [GAR 98]. Along with the C types implementing the LTS states, actions and transitions, equipped with basic operations (comparison, hashing, enumeration of successor states, etc.), OPEN/CÆSAR offers also libraries dedicated to the on-the-fly exploration of LTSs (state tables, transition lists, stacks, etc.). This implicit representation of LTSs is suitable for local (or on-the-fly) verification, which proceeds by a forward exploration of the transition relation and therefore allows an incremental construction[2] of the LTS.

The on-the-fly verification is a simple manner of combating state explosion (prohibitive size of the LTS for systems containing many parallel processes and complex data types), making it possible to detect errors even when the complete construction of the LTS exceeds the available computing resources.

5.2.3. *Some verification tools*

CADP offers a large palette of tools dedicated to the analysis of LTSs, covering the whole spectrum of functionalities necessary to assist the design process: interactive and guided simulation, random execution, minimization modulo various equivalence relations, partial order reduction, equivalence checking, model checking and conformance test case generation. Here we briefly present some of the CADP tools that we used for this case study and whose functioning will be illustrated in the sequel:

– *BCG_MIN* performs the minimization of an LTS, represented as a BCG file, modulo various equivalence relations, such as strong bisimulation or branching bisimulation. BCG_MIN also handles probabilistic and stochastic LTSs [GAR 02a].

– *CÆSAR_SOLVE* [MAT 03a, MAT 06] is a generic software library of the OPEN/CÆSAR environment, dedicated to the on-the-fly verification of alternation-free Boolean equation systems (BESs). These BESs consist of several blocks of equations with Boolean variables on their left-hand sides and propositional formulae (disjunctions or conjunctions over Boolean variables) on their right-hand sides. Each equation block represents the minimal or the maximal fixed point of the functional taking as input the variables present on the left-hand sides of equations and returning the values of the formulae on the right-hand sides of equations.

  A Boolean variable defined by an equation depends upon the variables occurring on the right-hand side of that equation. A block depends upon another block if it defines a variable depending upon some variable of the other block; the

2. Of course, an explicit LTS already constructed can also be explored on-the-fly; within CADP, this is done by means of the BCG_OPEN tool, which implements a representation of BCG files compatible with the OPEN/CÆSAR interface.

alternation-free condition means the absence of cyclic dependencies between the blocks of a BES. This class of BESs benefits from resolution algorithms having a time and space complexity linear in the size of the BES (number of variables and Boolean operators), still being sufficiently general to represent several types of LTS analyses (equivalence checking, model checking, partial order reduction, etc.).

The on-the-fly resolution of a BES consists of computing the value of a Boolean variable of interest by incrementally exploring only the part of the BES necessary to determine the value of that variable. The underlying algorithms can be developed in a more intuitive way by representing BESs as *Boolean graphs* [AND 94] whose vertices and edges denote Boolean variables and dependencies between them, respectively. Reformulated in this context, resolution algorithms perform a forward exploration of the Boolean graph, starting at the variable of interest, intertwined with a backward propagation of stable variables (whose values have been determined) along dependencies.

CÆSAR_SOLVE currently provides four on-the-fly resolution algorithms having a linear complexity in the size of the BES. Algorithms A1 and A2, based respectively on depth-first and breadth-first traversals of the Boolean graph, can solve general BESs, without imposing constraints on the Boolean formulae in the right-hand sides of the equations. Algorithms A3 and A4, based on depth-first traversals, are optimized to solve acyclic and disjunctive/conjunctive BESs (frequently encountered in practice) with a lower memory consumption, by storing only Boolean variables and not the dependencies between them. All these algorithms also produce *diagnostics*, i.e., portions of the Boolean graph illustrating the result of the resolution, following the approach proposed in [MAT 00]. Due to the breadth-first traversal, algorithm A2 exhibits small-depth diagnostics, which are easier to interpret.

CÆSAR_SOLVE defines an implicit representation of Boolean graphs by means of an application programming interface in C similar to the interface for LTSs defined by OPEN/CÆSAR: the C type encoding the Boolean variables is equipped with primitives making it possible to explore the Boolean graph (comparison and hashing of variables, enumeration of successor variables, etc.). Diagnostics of resolutions are also produced as Boolean subgraphs represented implicitly as their successor function. CÆSAR_SOLVE is currently used within CADP as a computing engine for several on-the-fly verification tools, two of them being presented below.

– *BISIMULATOR* [MAT 03a, MAT 06] is an equivalence checker that compares on-the-fly two LTSs with respect to an equivalence or preorder relation. The first LTS, represented implicitly as an OPEN/CÆSAR program, denotes the behavior of a system (protocol), whereas the second LTS, represented explicitly as a BCG file, denotes the external behavior (service) expected for the system. BISIMULATOR implements seven equivalence relations between LTSs: four bisimulations (strong, branching, observational and $\tau^*.a$) and three simulation equivalences (safety, trace and weak trace), being one of the richest on-the-fly equivalence checkers currently available. For each relation, the tool is able to determine both the equivalence of LTSs and the inclusion of one LTS into the other modulo the corresponding preorder.

The method used by BISIMULATOR consists of reformulating the verification problem as the resolution of a BES containing a single block of maximal fixed point equations, directly derived from the mathematical definition of the equivalence relation. The tool is structured in two independent parts: a front-end in charge of translating the comparison modulo an equivalence in terms of a BES, and of interpreting the diagnostic of the resolution in terms of the two LTSs being compared; and a back-end (CÆSAR_SOLVE library) responsible for computing the variable of interest, which represents the fact that the initial states of the two LTSs are equivalent or related modulo the preorder considered.

This modular architecture facilitates the addition of new equivalence relations (each relation is implemented as a separate module containing the BES translation and the diagnostic interpretation) and does not penalize performance, BISIMULATOR competing favorably with other implementations of algorithms dedicated to on-the-fly equivalence checking [BER 05]. The diagnostics (counter-examples) issued by the tool when the LTSs are not equivalent (or not included) are acyclic graphs containing all the sequences that, simultaneously executed in the two LTSs, lead to non-equivalent states.

BISIMULATOR employs all the resolution algorithms provided by CÆSAR_SOLVE: A1 and A2 can be applied to all equivalences (A2, being based on a breadth-first traversal, has the practical advantage of exhibiting small-depth counter-examples); A3, optimized in memory for solving acyclic BESs, serves to verify the inclusion of execution sequences or trees in an LTS; and A4, optimized in memory for solving conjunctive BESs, is useful when one LTS is deterministic (for strong bisimulation) and does not contain invisible transitions (for weak equivalences).

– *EVALUATOR 3.5* [MAT 03a, MAT 06] is a model checker that evaluates on-the-fly a temporal logic formula on an LTS. The logic accepted as input is the regular alternation-free μ-calculus [MAT 03b], which consists of Boolean operators, possibility and necessity modalities containing regular expressions over action sequences (similar to those of PDL [FIS 79]) and fixed point operators of modal μ-calculus [KOZ 83]. The alternation-free condition, meaning the absence of mutual recursion between minimal and maximal fixed point operators, leads to verification algorithms having a linear complexity with respect to the size of the formula (number of operators) and the LTS (number of states and transitions). The regular alternation-free μ-calculus enables a concise and intuitive description of classical properties over LTSs (safety, liveness, as well as certain forms of fairness). The model checker also makes it possible to define reusable libraries containing derived temporal operators, such as those of ACTL (*action-based CTL*) [NIC 90] and the generic property patterns proposed in [DWY 99].

The method used by EVALUATOR 3.5 consists of reformulating the verification problem as the resolution of a BES containing an equation block for each temporal operator contained in the formula. The BES is obtained after several transformation phases applied to the formula (translation in positive normal form, elimination of derived operators, translation to modal equation systems, elimination of regular expressions contained in modalities and simplification).

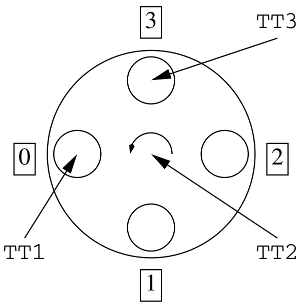
The tool is structured in two independent parts: a front-end in charge of translating the evaluation of the formula in terms of a BES and of interpreting the diagnostic of the resolution in terms of the LTS being analyzed; and a back-end (CÆSAR_SOLVE library) responsible for computing the variable of interest, which represents the fact that the initial state of the LTS satisfies the formula. The diagnostics (examples and counter-examples) issued by the tool are subgraphs of the LTS illustrating the truth value of the formula on the initial state of the LTS.

EVALUATOR 3.5 employs all the resolution algorithms provided by CÆSAR_SOLVE: A1 and A2 can be applied to all formulae of regular alternation-free μ-calculus (A2, being based on a breadth-first traversal, has the practical advantage of exhibiting small-depth diagnostics); A3, optimized in memory for solving acyclic BESs, serves to verify any μ-calculus formula on LTSs representing execution or simulation scenarios; and A4, optimized in memory for solving disjunctive/conjunctive BESs, is applied for evaluating formulae of ACTL and PDL, frequently encountered in practice.

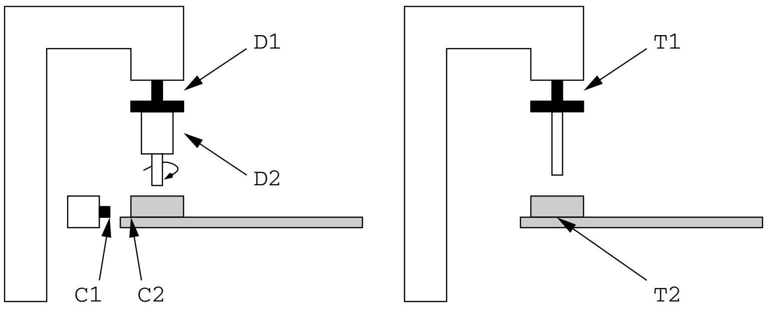
5.3. Specification of a drilling unit

We develop in this section a LOTOS specification of a drilling unit for metallic products. This example of an industrial critical system [BOS 02, BOR 05] has served as support for experimenting the modeling capabilities of various description languages (χ [SCH 03], Promela [HOL 03], timed automata [BEH 04], μCRL [GRO 97]) and the functionalities of their associated verification tools (SPIN [HOL 03], UPPAAL [BEH 04], CADP). The LOTOS specification presented below was derived from the χ description initially proposed in [BOS 02], with some details (e.g., the presence of the `TT3` sensor) inspired from the more elaborated description given in [BOR 05]. The drilling unit, illustrated in Figure 5.1, consists of a turning table, a drill equipped with a clamp and a tester.

The turning table (a) transports the metallic products to the drill and the tester. It is circular and has four slots, each one possibly containing at most one product. Each slot can be in one of four positions: entry position (0), drilling position (1), testing position (2) and exit position (3). Three sensors `TT1`, `TT2` and `TT3` attached to the



(a) Turning table



(b) Drill (c) Tester

Figure 5.1. *Drilling unit for metallic products*

table indicate if a product is present in the slot at position 0, if the table just completed a 90° counter-clockwise rotation, and if the slot at position 3 is empty, respectively.

The drill (b), located at position 1, is equipped with a clamp making it possible to lock the product during the drilling operation. Two sensors `D1` and `D2` attached to the drill detect whether it is in the upper or lower position, respectively. Two other sensors `C1` and `C2` attached to the clamp indicate whether it is released or blocked, respectively.

The tester (c), located at position 2, serves to detect whether a product was correctly drilled or not. It is equipped with two sensors `T1` and `T2`, which detect if the tester is in the upper or lower position, respectively. If the tester is in the lower position, this means that either the product present in the slot at position 2 was correctly drilled, or there is no product in this slot.

Each physical device (turning table, clamp, drill, tester) is equipped with a local controller in charge of driving the device. Local controllers receive signals from the sensors and send commands to the actuators attached to physical devices. Table 5.1 indicates the communication channels (called *gates* in LOTOS) used by local controllers. The turning table is controlled through the gate `TurnOn`, which commands a 90° counter-clockwise rotation. Thus, the products are transported from the entry position to the drilling position, then to the testing position, and finally to the exit position. The clamp, the drill and the tester are controlled through gates that model *switching mode* commands, so-called because they have two different effects, which change at each new invocation. For example, the command `COnOff` triggers either the blocking of the clamp if it is released, or the release of the clamp if it is blocked.

Gates for sending commands to actuators		
Device	Gate	Function
Table	`TurnOn`	Starts a 90° rotation
Clamp	`COnOff`	Blocks or releases the clamp
Drill	`DOnOff`	Starts or stops the engine of the drill
	`DUpDown`	Starts the ascending or descending movement
Tester	`TUpDown`	Starts the ascending or descending movement

Gates for receiving signals from sensors		
Device	Gate	Function
Table	`TT1`	Product present at position 0
	`TT2`	Rotation of 90° completed
	`TT3`	Product absent at position 3
Clamp	`C1`	Clamp released
	`C2`	Clamp blocked
Drill	`D1`	Drill in upper position
	`D2`	Drill in lower position
Tester	`T1`	Tester in upper position
	`T2`	Tester in lower position

Table 5.1. *Dialog between local controllers and physical devices*

The global functioning of the drilling unit is handled by a main controller, which is responsible for coordinating the activity of the various devices and for interacting with the environment. The main controller communicates with the local controllers associated with the physical devices through the gates `CMD` (emission of commands) and `INF` (reception of information) and with the environment through the gate `REQ` (emission of requests). Table 5.2 indicates the various signals (denoted by values of enumerated type `Sig`) sent on these gates. The specification assumes that the environment reacts correctly to the requests issued by the main controller: in particular, the products are put at position 0 and received at position 3 after every `Add` and `Remove` signal, respectively.

Gate	Signal	Meaning
CMD	Turn	Rotation of 90° of the table
	Drill	Drilling of the product at position 1
	Lock	Blocking of the clamp
	Unlock	Release of the clamp
	Test	Test of the product at position 2
INF	Turned	Rotation of 90° of the table completed
	Present	Product present at position 0
	Drilled	Drilling of the product at position 1 completed
	Locked	Clamp blocked
	Unlocked	Clamp released
	Tested	Product at position 2 tested
	Absent	Product absent at position 3
REQ	Add	Input of a product at position 0
	Remove	Output of a product at position 3

Table 5.2. *Dialog of the main controller with the local controllers and the environment*

5.3.1. *Architecture*

The architecture of the system specified in LOTOS is illustrated in Figure 5.2. The boxes represent the various parallel processes and the arrows indicate the communication gates. Each physical device and its associated local controller are modeled as couples of processes: `TT` and `TTC` (turning table), `D` and `DC` (drill), `C` and `CC` (clamp), `T` and `TC` (tester). The main controller is modeled by the process `MC`.

The process `TT` communicates with the environment through the gates `ADD` and `REM`, which model the input of a product in the slot at position 0 and the output of the product from the slot at position 3, respectively.

The LOTOS description of the architecture is illustrated below. Each element is represented by a process call parametrized by communication gates and possibly by data values. The concurrent execution of processes is described using the parallel composition operators "`|||`" and "`|[···]|`" of LOTOS, which denote parallel execution without synchronization and with synchronization on a set of gates, respectively. For instance, the parallel processes `TT` and `TTC` synchronize on the gates `TT1`, `TT2`, `TT3` and `TurnOn`, but they execute asynchronously with respect to the other processes `D`, `DC`, etc. The gates used for the dialog between physical devices and their local controllers are hidden (i.e., renamed into the invisible action, noted "`i`" in LOTOS) using the "*hide*" operator.

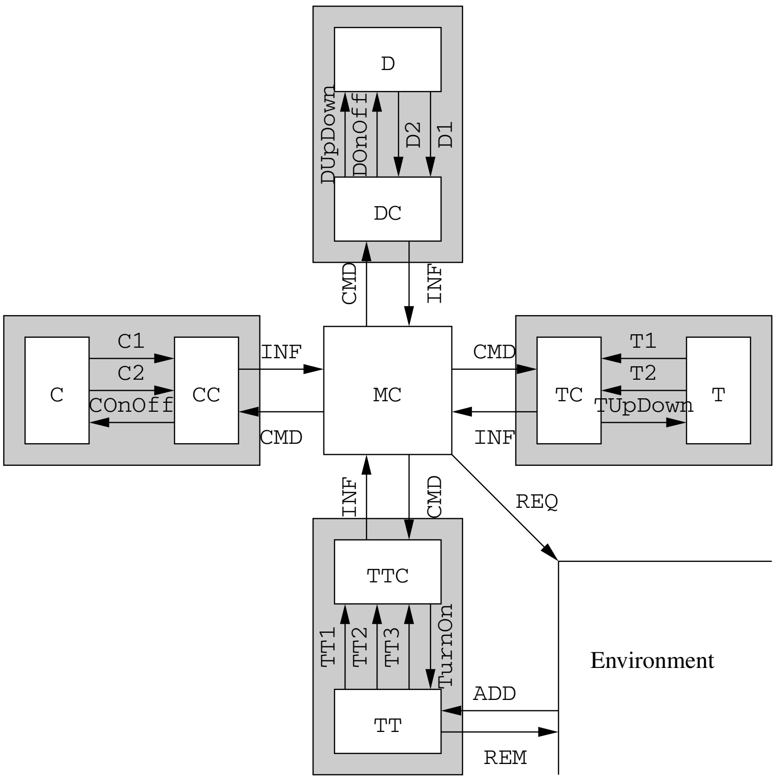


Figure 5.2. *Architecture of the drilling unit specified in LOTOS*

The processes `TT`, `D`, `C` and `T` representing the physical devices have data parameters (whose meaning will be defined in the following sections) recording the current state of these processes: when the system starts its execution, the values of these parameters indicate that all slots of the turning table are empty, the drill is stopped and in the upper position, the clamp is released and the tester is in the upper position. The process `MC` modeling the main controller also has data parameters reflecting the current state of the system.

```
(   (   hide TT1, TT2, TT3, TurnOn, D1, D2, DUpDown, DOnOff,
             C1, C2, COnOff, T1, T2, TUpDown in
        (   (
              TT [TT1, TT2, TT3, TurnOn, ADD, REM]
                 (false, false, false, false)
              |[TT1, TT2, TT3, TurnOn]|
              TTC [TT1, TT2, TT3, TurnOn, INF, CMD]
            )
```

```
                |||
                (
                   D [D1, D2, DUpDown, DOnOff] (false, true)
                   |[D1, D2, DUpDown, DOnOff]|
                   DC [D1, D2, DUpDown, DOnOff, INF, CMD]
                )
                |||
                (
                   C [C1, C2, COnOff] (false)
                   |[C1, C2, COnOff]|
                   CC [C1, C2, COnOff, INF, CMD]
                )
                |||
                (
                   T [T1, T2, TUpDown] (true)
                   |[T1, T2, TUpDown]|
                   TC [T1, T2, TUpDown, INF, CMD]
                )
             )
          )
          |[INF, CMD]|
          MC [REQ, INF, CMD] (false, false, false, false, false)
       )
       |[REQ, ADD, REM]|
       Env [REQ, ADD, REM, ERR]
```

5.3.2. *Physical devices and local controllers*

We give below the LOTOS specification of the various physical devices and of the local controllers that drive their functioning. For simplicity, the processes corresponding to the local controllers are illustrated graphically in Figure 5.3 (the initial states are marked with bold circles).

5.3.2.1. *Turning table*

The process `TT` has four Boolean parameters `p0`, `p1`, `p2` and `p3`, which are set to `true` if a product is present in the slot at the corresponding position of the turning table and are set to `false` otherwise. At any time, `TT` can perform one of the following behaviors: it receives a rotation command from its local controller, it carries out the rotation (this is not explicitly modeled here[3]), and then it sends back to the controller the corresponding response; if the slot at position 0 of the table is empty, it can input

3. In a system specification that takes into account the real-time aspects, the actions `TurnOn` and `TT2` (as well as the commands sent to the actuators and the responses of the sensors attached to the other physical devices) would be separated by a delay.

a product from the environment and signal to its controller that the slot at position 0 became occupied; if the slot at position 3 of the table is occupied, it can output the corresponding product to the environment. After performing one of these behaviors, TT updates its parameters and continues its execution cyclically.

```
process TT [TT1, TT2, TT3, TurnOn, ADD, REM] (p0, p1, p2, p3:Bool) :
           noexit :=
   TurnOn;
      TT2;
         TT [TT1, TT2, TT3, TurnOn, ADD, REM] (p3, p0, p1, p2)
   []
   [not (p0)] -> ADD;
      TT1;
         TT [TT1, TT2, TT3, TurnOn, ADD, REM] (true, p1, p2, p3)
   []
   [p3] -> REM;
      TT3;
         TT [TT1, TT2, TT3, TurnOn, ADD, REM] (p0, p1, p2, false)
endproc
```

The process TTC (see Figure 5.3) executes the following session cyclically: when the process TT informs it about the presence of a product in the slot at position 0 of the table, it passes this information to the main controller. When it receives a rotation command from the main controller, it transmits this command to the process TT, waits for its response, and then it passes this response to the main controller. Finally, when the process TT informs it about the presence of a product in the slot at position 3 of the table, it transmits this information to the main controller.

5.3.2.2. *Clamp*

The process C has one Boolean parameter locked, which is set to true when the clamp is blocked and to false otherwise. C executes the following behavior cyclically: it receives a block or release command from its local controller, executes it (this is not explicitly modeled here) and, according to its current state, sends the appropriate response to the controller.

```
process C [C1, C2, COnOff] (locked:Bool) : noexit :=
   COnOff;
   (  [locked] -> C1;
         C [C1, C2, COnOff] (not (locked))
      []
      [not (locked)] -> C2;
         C [C1, C2, COnOff] (not (locked))
   )
endproc
```

The process CC (see Figure 5.3) executes the following session cyclically: when it receives a block command from the main controller, it commands the blocking of the clamp, waits for the response from the process C, and then transmits this response to the main controller. Then, it waits for a release command from the main controller, it commands the release of the clamp, waits for the response from the process C, and propagates it to the main controller.

5.3.2.3. *Drill*

Process D has two Boolean parameters on and up, which are set to true if the drill is started and in the upper position, and are set to false otherwise. At any time, D can perform one of the following behaviors: it receives from its local controller a start or stop command; if the drill is started, it receives a command for moving up or down, executes it (this is not explicitly modeled here), and sends back the appropriate response to its controller. After performing one of these behaviors, D updates its parameters and continues its execution cyclically.

```
process D [D1, D2, DUpDown, DOnOff] (on, up:Bool) : noexit :=
   DOnOff;
      D [D1, D2, DUpDown, DOnOff] (not (on), up)
   []
   [on] -> (
      DUpDown;
      (  [up] -> D2;
            D [D1, D2, DUpDown, DOnOff] (on, not (up))
         []
         [not (up)] -> D1;
            D [D1, D2, DUpDown, DOnOff] (on, not (up))
      )
   )
endproc
```

The process DC (see Figure 5.3) executes the following session cyclically: when it receives a drill command from the main controller, it commands the start of the drill engine, then the descending movement of the drill, and waits for the response of the process D. Then, it commands the ascending movement of the drill, waits for the response from D, then commands the engine to stop. Finally, it sends to the main controller the response meaning that the drilling operation has been accomplished.

5.3.2.4. *Tester*

The process T has a Boolean parameter up, which is set to true if the tester is in the upper position and is set to false otherwise. T performs the following behavior cyclically: it receives a command for moving up or down from its local controller; it executes the command (this is not explicitly modeled here) and, according to its current state, sends an appropriate response to its controller. The fact that the product located in the slot at position 2 of the table was correctly drilled or not is modeled as a non-deterministic choice consisting of sending the response to the controller (product correctly drilled) or sending no response (product incorrectly drilled).

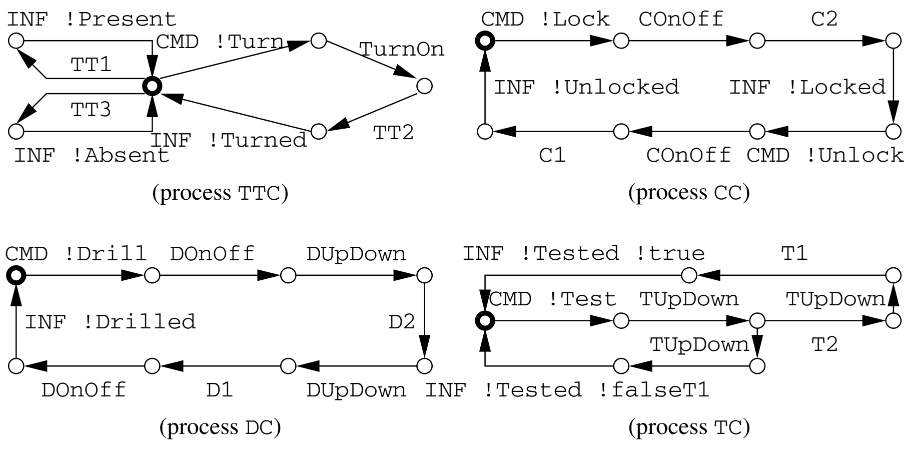


Figure 5.3. *LTSs modeling the behavior of local controllers*

```
process T [T1, T2, TUpDown] (up:Bool) : noexit :=
   TUpDown;
   (  [up] -> (
         T2;                                   (* Correct drilling *)
            T [T1, T2, TUpDown] (not (up))
         []
         T [T1, T2, TUpDown] (not (up))        (* Incorrect drilling *)
      )
      []
      [not (up)] -> T1;
         T [T1, T2, TUpDown] (not (up))
   )
endproc
```

The process TC (see Figure 5.3) executes the following session cyclically: when it receives a test command from the main controller, it commands the descending movement of the tester, then it waits for the response from the process T (an absence of response indicates a product incorrectly drilled). Afterwards, it sends an ascending movement command to the tester, then it waits for the response from the process T. Finally, it sends an appropriate response to the main controller, indicating by means of a Boolean value true or false that the product in the slot at position 2 of the table was correctly drilled or not.

5.3.3. *Main controller – sequential version*

We describe below the first version of the main controller, in which the various phases of dialog with the local controllers are performed sequentially, one after the other. The process MC has five Boolean parameters: p0, p1, p2 and p3 indicate the presence of products in the slots at the corresponding positions of the turning table;

`tr` is set to `true` if a product correctly drilled occupies the slot at position 3 and is set to `false` otherwise. `MC` performs the following session cyclically, consisting of five control phases executed one after the other:

1) if the slot at position 0 of the table is empty, it sends a product input request to the environment (which is supposed to react immediately by supplying a product), then it waits for the corresponding response from the controller `TTC`;

2) if a product is present in the slot at position 1, it sends a clamp block command to the controller `CC` and waits for its response. Then it sends a drill command to the controller `DC` and waits for its response, and finally it sends a clamp release command and waits for the response of the controller `CC`;

3) if a product is present in the slot at position 2, it sends a test command to the controller `TC` and then it waits for the corresponding response;

4) if a product is present in the slot at position 3, it sends a product output request to the environment (which is supposed to react immediately by removing the product), then it waits for the corresponding response from the controller `TTC`;

5) it sends a table rotation command to the controller `TTC`, then it waits for the response before updating its parameters and restarting its cyclic execution.

The chaining of the five control phases is modeled by means of the LOTOS sequential composition operators: "*exit*" and "*exit* $(V_1, \ldots, V_n)$" denote the termination of a behavior without returning any result and by returning values $V_1, \ldots, V_n$, respectively; "B_1 >> B_2" expresses the execution of behavior B_2 after the execution of B_1 has successfully terminated by an "*exit*"; and "B_1 >> **accept** $x_1{:}T_1, \ldots, x_n{:}T_n$ **in** B_2" denotes the execution of B_2 after B_1 successfully terminated its execution by performing an "*exit* $(V_1, \ldots, V_n)$" of which the resulting values are assigned to the variables $x_1, \ldots, x_n$, subsequently used by B_2.

```
process MC [REQ, INF, CMD] (p0, p1, p2, p3, tr:Bool) : noexit :=
   (  [not (p0)] -> REQ !Add; INF !Present;
         exit (true)
      []
      [p0] -> exit (p0)
   )
   >>
   accept new_p0:Bool in
   (  (  [p1] -> CMD !Lock; INF !Locked;
            CMD !Drill; INF !Drilled;
               CMD !Unlock; INF !Unlocked;
                  exit
         []
         [not (p1)] -> exit
      )
      >>
      (  [p2] -> CMD !Test; INF !Tested ?r:Bool;
            exit (r)
         []
```

```
            [not (p2)] -> exit (tr)
        )
        >>
        accept new_tr:Bool in
        (   (   [p3] -> REQ !Remove !tr; INF !Absent;
                    exit (false)
                []
                [not (p3)] -> exit (p3)
            )
            >>
            accept new_p3:Bool in
                CMD !Turn; INF !Turned;
                    MC [REQ, INF, CMD] (new_p3, new_p0, p1, p2, new_tr)
        )
    )
endproc
```

5.3.4. *Main controller – parallel version*

We describe below a second version of the main controller, in which the phases of dialog with the local controllers are performed in parallel. The behavior of this version is more complex (because of the interleavings of the various control phases), but also more efficient than the sequential version, the complete processing of a product being faster (this can be confirmed by evaluating the throughput of the turning table, following the approach proposed in [GAR 02a]). The new process `MC` is obtained by composing the first four control phases (associated with the positions of the turning table) described in the previous section, by using the asynchronous parallel composition operator "`|||`" of LOTOS.

```
process MC [REQ, INF, CMD] (p0, p1, p2, p3, tr:Bool) : noexit :=
    (   (   [not (p0)] -> REQ !Add; INF !Present;
                exit (true, p1, p2, any Bool, any Bool)
            []
            [p0] -> exit (p0, p1, p2, any Bool, any Bool)
        )
        |||
        (   [p1] -> CMD !Lock; INF !Locked;
                CMD !Drill; INF !Drilled;
                    CMD !Unlock; INF !Unlocked;
                        exit (any Bool, p1, p2, any Bool, any Bool)
            []
            [not (p1)] -> exit (any Bool, p1, p2, any Bool, any Bool)
        )
        |||
        (   [p2] ->  CMD !Test; INF !Tested ?r:Bool;
                exit (any Bool, p1, p2, any Bool, r)
            []
            [not (p2)] -> exit (any Bool, p1, p2, any Bool, tr)
        )
        |||
         (  [p3] -> REQ !Remove !tr; INF !Absent;
               exit (any Bool, p1, p2, false, any Bool)
            []
            [not (p3)] -> exit (any Bool, p1, p2, p3, any Bool)
         )
      )
      >> accept new_p0, new_p1, new_p2, new_p3, new_tr:Bool in
         CMD !Turn; INF !Turned;
            MC [REQ, INF, CMD] (new_p3, new_p0, new_p1, new_p2, new_tr)
endproc
```

The execution of each phase changes the state of the turning table, which is recorded by the Boolean parameters `p0`, `p1`, `p2`, `p3` and `tr` of the main controller. These changes are modeled using the "*exit*" operator, which makes it possible to indicate, for each parameter, either its new value obtained after executing the corresponding processing phase, or the fact that the parameter is modified by another phase (pattern "*any*"). The occurrences of the "*exit*" operator at the end of the four parallel phases must be compatible, i.e., each parameter must have the same value or be filtered by "*any*" in each of the four "*exit*" statements (for instance, parameter `p0` is set to `true` or left unchanged in the first phase and filtered by "*any*" in the other phases). This is necessary in order to ensure the correct synchronization of the four parallel phases upon their termination (*join*), as specified by the semantics of the "`|||`" operator.

Once the four parallel phases have terminated, the main controller commands the rotation of the table, waits for the corresponding response from the local controller, then updates its parameters (with the values catched using the ">> **accept** ... **in**" operator) and restarts its cyclic execution.

5.3.5. *Environment*

The last element that we must specify in order to obtain a complete description of the system is the environment, which handles the input and output requests of the main controller by inserting and removing products into and from the slots at positions 0 and 3 of the turning table, respectively.

```
process Env [REQ, ADD, REM, ERR] : noexit :=
    REQ !Add;
       ADD;
           Env [REQ, ADD, REM, ERR]
    []
    REQ !Remove ?r:Bool;
    (  [r] -> REM;
          Env [REQ, ADD, REM, ERR]
       []
       [not (r)] -> ERR; REM;
          Env [REQ, ADD, REM, ERR]
    )
endproc
```

When a product is removed, the main controller also indicates to the environment whether the product has been correctly drilled or not, this latter case being signaled by the environment through the gate `ERR`.

5.4. Analysis of the functioning of the drilling unit

Once the drilling unit has been specified in LOTOS, we can analyze its behavior by using the verification tools of CADP. We first study the coherence between the two versions of the system, equipped with the sequential and the parallel main controller, respectively. Then, we identify a set of correctness properties of the drilling unit, express them in temporal logic and verify them on the two versions of the system.

5.4.1. *Equivalence checking*

We begin by constructing, using the two LOTOS compilers CÆSAR and CÆSAR.ADT, the LTS model M_{seq} of the drilling unit equipped with the sequential main controller, in order to estimate its size and possibly to attempt its visual inspection. This LTS, minimized modulo branching bisimulation with the BCG_MIN tool and displayed graphically using the BCG_EDIT tool, is illustrated in Figure 5.4. It has 69 states and 72 transitions, and thus an average branching factor (number of transitions going out of a state) of 1.04, which reflects the sequential nature of this version of the system.

Starting at the initial state (numbered 0) of the LTS, we observe a sequence of actions modeling the insertion of products in the slots of the turning table (which was initially empty) until the permanent functioning regime is reached (all the slots of the table are occupied). This sequence is followed by two different execution branches, corresponding to the fact that the product present at position 2 was correctly drilled (branch on the left) or not (branch on the right). These two branches denote a similar behavior, except for the presence of an `ERR` action on the branch on the right, indicating the output of an incorrectly drilled product to the environment.

We then build the LTS model M_{par} of the drilling unit equipped with the parallel main controller. This model is much larger than the model corresponding to the sequential controller: it has $24,346$ states and $85,013$ transitions[4], its average branching factor being 3.49. This increase in size is caused by the interleaving of the four processing phases (corresponding to the positions of products on the turning table), which previously were chained sequentially. The size of the new LTS makes its visual inspection impractical; therefore, the use of verification tools becomes mandatory in order to assess the proper functioning of the system.

4. A reduction of M_{par} by τ-confluence [PAC 03] using the REDUCTOR tool of CADP would reduce its size to $5,373$ states and $17,711$ transitions.

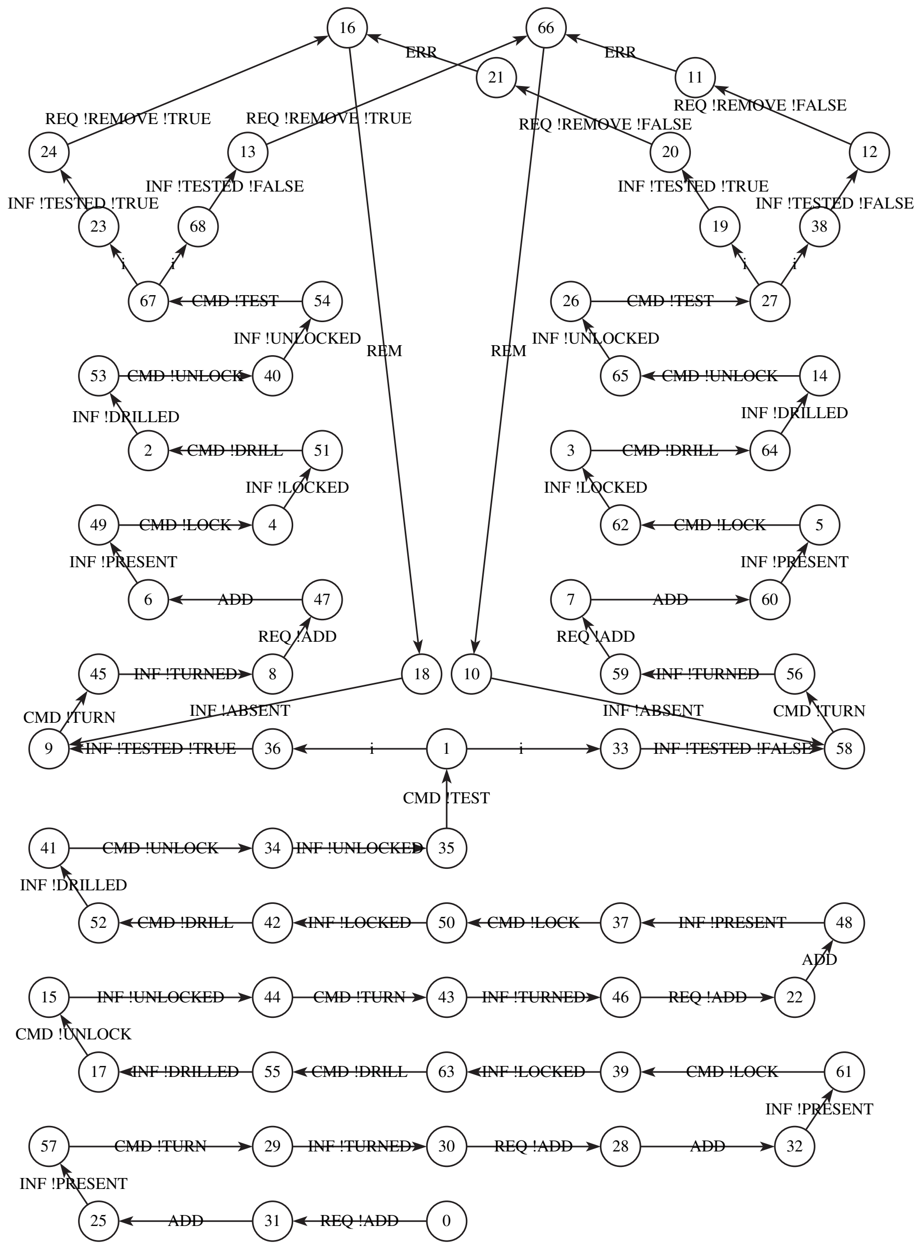


Figure 5.4. *LTS of the drilling unit equipped with the sequential controller*

The first verification consists of ensuring that the behaviors of the two versions of the system modeled by M_{seq} and M_{par} are coherent. Intuitively, since the sequential chaining of the processing phases is a particular case of their parallel interleaving, all behaviors of the system equipped with the sequential controller should be "simulated" by the system equipped with the parallel controller. This is indeed the case: using BISIMULATOR, we can check that M_{seq} is included in M_{par} modulo the preorder of branching bisimulation. In other words, if we abstract away from the actions modeling the dialog with the sensors and the actuators (see section 5.3.1), the execution trees contained in M_{seq} are also contained in M_{par}. This verification can be done by traversing M_{par} on-the-fly; the underlying BES explored by BISIMULATOR has 515 Boolean variables and 934 operators.

On the other hand, the two LTSs M_{seq} and M_{par} are equivalent modulo none of the seven equivalence relations implemented by BISIMULATOR. Indeed, the following execution sequence (restricted to visible actions only) is present in M_{par} but not in M_{seq}:

$$s_0 \xrightarrow{\texttt{REQ !ADD}} s_1 \xrightarrow{\texttt{ADD}} s_2 \xrightarrow{\texttt{INF !PRESENT}} s_3 \xrightarrow{\texttt{CMD !TURN}} s_4 \xrightarrow{\texttt{INF !TURNED}} s_5 \xrightarrow{\texttt{CMD !LOCK}} s_6$$

This sequence, obtained automatically as a counter-example by trying to check the weak trace equivalence of M_{seq} and M_{par}, shows that after the input of the first product and the first rotation of the table, the sequential controller does not begin the drilling of the product (currently located at position 1) by commanding the block of the clamp, but restarts its cyclic functioning by handling the insertion of a new product in the slot at position 0 (currently empty). On the other hand, the parallel controller is able to command the drilling before the insertion, because it enables the concurrent execution of the four processing phases.

5.4.2. *Model checking*

The equivalence checking provided some indication about the coherence of the two versions of the system; however, it does not guarantee their correct functioning. In this section, we identify several temporal properties characterizing the correct ordering of actions during the execution of the system, and we express them in regular alternation-free μ-calculus [MAT 03b], the temporal logic accepted as input by the EVALUATOR 3.5 model checker. We consider two kinds of classical properties (illustrated graphically in Figure 5.5):

– *Safety properties* intuitively specify that "something bad never happens" during the execution of the system. They can be expressed in regular μ-calculus by the "$[R]$ false" formula, where R is a regular expression (defined over the alphabet of predicates on LTS actions) characterizing the undesirable action sequences that violate the safety properties. The necessity modality above states that all execution sequences going out of the current state and satisfying R must lead to states satisfying false; since there are no such states, the corresponding sequences do not exist either.

– *Liveness properties* intuitively specify that "something good eventually happens" during the execution of the system. Most of the liveness properties that we will use here can be expressed in regular μ-calculus by (variants of) the "$[R]$ $\mathsf{inev}(A, B, P)$" formula, where R is a regular expressions over action sequences, A and B are action predicates, and P denotes a state formula. The formula above states that all execution sequences going out of the current state and satisfying R must lead to states from which all sequences are made of (zero or more) actions satisfying A, followed by an action satisfying B and leading to a state satisfying P. In other words, after every sequence satisfying R, it is inevitable to arrive (after some actions satisfying A, followed by an action satisfying B) at a state satisfying P.

Table 5.3 shows seven safety properties of the drilling unit, together with their definitions in regular alternation-free μ-calculus. EVALUATOR 3.5 makes it possible to express basic action predicates by using character strings surrounded by double quotes " " (denoting a single action) or by using regular expressions over character strings (denoting a set of actions). Properties P_1–P_5 characterize the order of the processing phases performed on a product by the drilling unit, induced by the counter-clockwise rotation (insertion, locking, drilling, unlocking, testing and removal). Property P_6 expresses the safety of the drilling unit with respect to the testing of the products. Property P_7 expresses a constraint over the order of drilling and testing execution when the corresponding positions are both occupied.

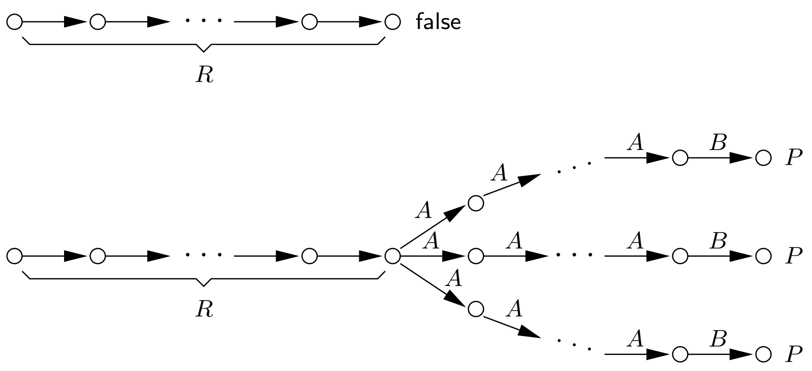


Figure 5.5. *Illustration of the "*$[R]$ false*" and "*$[R]$ $\mathsf{inev}(A, B, P)$*" operators*

No.	Formula	Description
P_1	[true* . “INF !PRESENT” . (not “INF !TURNED”)* . “INF !TURNED” . (not “INF !LOCKED”)* . “CMD !DRILL”] false	After the input of a product and a rotation of the table, the main controller cannot command a drilling before the clamp has been blocked.
P_2	[true* . “INF !DRILLED” . (not ‘INF !UNLOCKED.*’)* . “CMD !TURN”] false	After the drilling of a product, the main controller cannot command a rotation before the clamp has been released.
P_3	[true* . “INF !UNLOCKED” . (not “INF !TURNED”)* . “INF !TURNED” . (not ‘INF !TESTED.*’)* . “CMD !TURN”] false	After the release of the clamp on a product and a rotation of the table, the main controller cannot command another rotation before the product has been tested.
P_4	[true* . ‘INF !TESTED.*’ . (not “INF !TURNED”)* . “INF !TURNED” . (not “INF !ABSENT”)* . “CMD !TURN”] false	After the test of a product and a rotation of the table, the main controller cannot command another rotation before the product has been removed.
P_5	[true* . “INF !ABSENT” . (not “INF !TURNED”)*. “INF !TURNED″ . (not “INF !PRESENT”)*. “CMD !TURN”] false	After a product is removed and a rotation of the table, the main controller cannot command another rotation before a new product has been supplied.
P_6	[true* . “INF !TESTED !TRUE” . (not “INF !TURNED”)* . “INF !TURNED” . (not “INF !TURNED”)* . “ERR”] false	Every time the tester detects a correctly drilled product, no error will be signaled during the next processing cycle.
P_7	[true* . “INF !PRESENT” . true* . “INF !PRESENT” . true* . “INF !PRESENT” . true* . “CMD !TEST” . (not “INF !TURNED”)* . “CMD !DRILL”] false	After the testing and drilling positions of the table have been occupied, the main controller cannot command a test before commanding a drill.

Table 5.3. *Safety properties of the drilling unit*

Due to EVALUATOR 3.5, we can check that all properties P_1–P_6 are satisfied by the LTSs M_{seq} and M_{par} corresponding to the two versions of the main controller. The size of the underlying BESs explored by EVALUATOR 3.5 varies from 321 variables and 336 operators (for property P_2 on M_{seq}) to $48,712$ variables and $171,645$ operators (for property P_4 on M_{par}). However, property P_7 is satisfied by M_{par} but not by M_{seq}: by running EVALUATOR 3.5 with the breadth-first strategy, we obtain a minimal counter-example sequence consisting of 23 visible actions. This sequence, absent from M_{seq} but present in M_{par}, models three insertions of products and two rotations of the table (which ensures that the slots at the drilling and testing positions become occupied), a clamp blocking command (which prepares the drilling) and a testing command followed by a drilling command. Indeed, the sequential controller handles the products present on the table in a precise order – the testing of the product at position 2 being carried out after the drilling of the product present at position 1 – whether the parallel controller is able to handle the various processing phases simultaneously.

Although the safety properties described above forbid the malfunctionings of the drilling unit, they do not guarantee the completion of the various processing phases on the products: thus, a drilling unit whose turning table does not move satisfies all these safety properties. In order to ensure the start and the progress of the processing phases, the system must also satisfy certain liveness properties. Table 5.4 shows seven liveness properties of the drilling unit, together with their corresponding temporal formulae. Property P_8 describes the starting sequence of the system, during which the products are inserted in all the slots of the turning table, which reaches the permanent functioning regime. Properties P_9–P_{12} are the counterparts of properties P_1–P_4: they characterize the progress of each processing phase and also the rotation of the table. Property P_{13} indicates the responses to the commands and requests of the main controller, which are sent by the system or by the environment during each processing cycle. Finally, property P_{14} specifies the correct reaction of the system when the drilling of some products fail.

Due to EVALUATOR 3.5, we can verify that all properties P_8–P_{14} are satisfied by the LTSs M_{seq} and M_{par} corresponding to the two versions of the main controller. The size of the underlying BESs varies from 650 variables and 947 operators (for property P_8 on M_{seq}) to $275,277$ variables and $606,747$ operators (for property P_{13} on M_{par}).

5.5. Conclusion and future work

By means of the drilling unit example detailed in this chapter, we tried to illustrate the specification and verification methods offered by the CADP toolbox for analyzing the functional properties of industrial critical systems that involve asynchronous parallelism. The LOTOS language appears to be suitable for describing in a succinct

No.	Formula	Description
P_8	inev (not “CMD !TURN”, “REQ !ADD”, inev (not “CMD !TURN”, “CMD !TURN”, inev (not “CMD !TURN”, “REQ !ADD”, inev (not “CMD !TURN”, “CMD !TURN”, inev (not “CMD !TURN”, “REQ !ADD”, inev (not “CMD !TURN”, “CMD !TURN”, inev (not “CMD !TURN”, “REQ !ADD”, true)))))))	Initially, the main controller eventually commands the insertion of products in all the slots of the turning table.
P_9	[true* . “INF !PRESENT”] inev (not “CMD !TURN”, “CMD !TURN”, inev (not “CMD !TURN”, “CMD !LOCK”, inev (not “CMD !TURN”, “CMD !DRILL”, inev (not “CMD !TURN”, “CMD !UNLOCK”, true))))	Each product inserted will be drilled after the next rotation of the table.
P_{10}	[true* . “INF !UNLOCKED”] inev (not “CMD !TURN”, “CMD !TURN”, inev (not “CMD !TURN”, “CMD !TEST”, true))	Each product drilled will be tested after the next rotation of the table.
P_{11}	[true* . ‘INF !TESTED.*’] inev (not “CMD !TURN”, “CMD !TURN”, inev (not “CMD !TURN”, ‘REQ !REMOVE.*’, true))	Each product tested will be removed after the next rotation of the table.
P_{12}	[true* . “INF !ABSENT”] inev (not “CMD !TURN”, “CMD !TURN”, inev (not “CMD !TURN”, “REQ !ADD”, true))	Each product removal will be followed by the insertion of a new product after the next rotation of the table.
P_{13}	[true*] ([“REQ !ADD”] inev (not “INF !TURNED”, “INF !PRESENT”, true) and [“CMD !LOCK”] inev (not “INF !TURNED”, “INF !LOCKED”, true) and [“CMD !DRILL”] inev (not “INF !TURNED”, “INF !DRILLED”, true) and [“CMD !UNLOCK”] inev (not “INF !TURNED”, “INF !UNLOCKED”, true) and [“CMD !TEST”] inev (not “INF !TURNED”, ‘INF !TESTED.*’, true) and [‘REQ !REMOVE.*’] inev (not “INF !TURNED”, “INF !ABSENT”, true) and [“CMD !TURN”] inev (not “INF !TURNED”, “INF !TURNED”, true))	Each command (respectively request) sent by the main controller to the physical devices (respectively to the environment) will be eventually followed by its acknowledgment before the next rotation of the table.
P_{14}	[true* . “INF !TESTED !FALSE”] inev (not “CMD !TURN”, “CMD !TURN”, inev (not “CMD !TURN”, “ERR”, true))	Every time the tester detects an incorrectly drilled product, an error will be eventually signaled during the next processing cycle.

Table 5.4. *Liveness properties of the drilling unit*

and abstract manner the functioning of the controllers in charge of driving the physical devices. At the present time, CADP has been used for analyzing 94 industrial case studies[5] and the various generic components of the BCG and OPEN/CÆSAR environments have enabled the development of 29 derived tools[6]. The feedback received from these applications led to the development of new tools as well as to the improvement of existing tools as regards to performance and user-friendliness.

We presented here only a few basic tools of CADP, which implement classical verification methods (equivalence checking and model checking) on LTSs. CADP also offers sophisticated analysis functionalities in order to deal with large-scale systems: compositional verification using the EXP.OPEN 2.0 tool [LAN 05], distributed verification on clusters using the DISTRIBUTOR and BCG_MERGE tools [GAR 01b, GAR 06] and distributed BES resolution algorithms [JOU 04, JOU 05], partial order reduction [PAC 03, MAT 05]. These functionalities are orthogonal and operate on-the-fly, being based upon the implicit representation of LTSs defined by OPEN/CÆSAR: consequently, they can be combined in order to cumulate their benefits and scale up the analysis capabilities to large systems. Moreover, CADP offers the SVL language [GAR 01a] and its associated compiler, which enable a succinct and elegant description of complex verification scenarios, involving hundreds of invocations of the verification tools.

The research and development activities around CADP are currently pursued along several directions. The rise of massively parallel computing architectures such as clusters and grids require the design of specific distributed verification algorithms as well as the definition of LTS representations adequate for the distribution [GAR 01b]. CADP can also play the role of analysis engine for other languages, namely those dedicated to the description of asynchronous hardware [SAL 05]. Finally, the application of the verification techniques promoted by CADP in other domains, such as bioinformatics, appears particularly promising [BAT 04].

5.6. Bibliography

[AND 94] ANDERSEN H. R., "Model Checking and Boolean Graphs", *Theoretical Computer Science*, vol. 126, num. 1, p. 3–30, April 1994.

[BAT 04] BATT G., BERGAMINI D., DE JONG H., GARAVEL H., MATEESCU R., "Model Checking Genetic Regulatory Networks using GNA and CADP", GRAF S., MOUNIER L., Eds., *Proc. of the 11th Int. SPIN Workshop on Model Checking of Software SPIN'2004 (Barcelona, Spain)*, vol. 2989 of *Lecture Notes in Computer Science*, Springer Verlag, p. 156–161, April 2004.

5. See the online catalog `http://www.inrialpes.fr/vasy/cadp/case-studies`.

6. See the online catalog `http://www.inrialpes.fr/vasy/cadp/software`.

[BEH 04] BEHRMANN G., DAVID A., LARSEN K. G., “A Tutorial on Uppaal”, BERNARDO M., CORRADINI F., Eds., *Proc. of the 4th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems SFM-RT’04 (Bertinoro, Italy)*, vol. 3185 of *Lecture Notes in Computer Science*, Springer Verlag, p. 200–236, September 2004.

[BER 05] BERGAMINI D., DESCOUBES N., JOUBERT C., MATEESCU R., “BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking”, HALBWACHS N., ZUCK L., Eds., *Proc. of the 11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS’2005 (Edinburgh, UK)*, vol. 3440 of *Lecture Notes in Computer Science*, Springer Verlag, p. 581–585, April 2005.

[BOR 05] BORTNIK E., TRCKA N., WIJS A. J., LUTTIK S. P., VAN DE MORTEL-FRONCZAK J. M., BAETEN J. C. M., FOKKINK W. J., ROODA J. E., “Analyzing a χ Model of a Turntable System using Spin, CADP and UPPAAL”, *Journal of Logic and Algebraic Programming*, vol. 65, num. 2, p. 51–104, November 2005.

[BOS 02] BOS V., KLEIJN J., “Formal Specification and Analysis of Industrial Systems”, PhD Thesis, Technical University of Eindhoven, March 2002.

[BRO 84] BROOKES S. D., HOARE C. A. R., ROSCOE A. W., “A Theory of Communicating Sequential Processes”, *Journal of the ACM*, vol. 31, num. 3, p. 560–599, July 1984.

[DWY 99] DWYER M. B., AVRUNIN G. S., CORBETT J. C., “Patterns in Property Specifications for Finite-State Verification”, BOEHM B., GARLAN D., KRAMER J., Eds., *Proc. of the 21st Int. Conference on Software Engineering ICSE’99 (Los Angeles, CA, USA)*, ACM, p. 411–420, May 1999.

[EHR 85] EHRIG H., MAHR B., *Fundamentals of Algebraic Specification 1 – Equations and Initial Semantics*, vol. 6 of *EATCS Monographs on Theoretical Computer Science*, Springer Verlag, 1985.

[FIS 79] FISCHER M. J., LADNER R. E., “Propositional Dynamic Logic of Regular Programs”, *Journal of Computer and System Sciences*, vol. 18, num. 2, p. 194–211, 1979.

[GAR 89] GARAVEL H., “Compilation of LOTOS Abstract Data Types”, VUONG S. T., Ed., *Proc. of the 2nd Int. Conference on Formal Description Techniques FORTE’89 (Vancouver, Canada)*, North Holland, p. 147–162, December 1989.

[GAR 90] GARAVEL H., SIFAKIS J., “Compilation and Verification of LOTOS Specifications”, LOGRIPPO L., PROBERT R. L., URAL H., Eds., *Proc. of the 10th Int. Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, IFIP, North Holland, p. 379–394, June 1990.

[GAR 98] GARAVEL H., “OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing”, STEFFEN B., Ed., *Proc. of the First Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS’98 (Lisbon, Portugal)*, vol. 1384 of *Lecture Notes in Computer Science*, Springer Verlag, p. 68–84, March 1998.

[GAR 01a] GARAVEL H., LANG F., “SVL: a Scripting Language for Compositional Verification”, KIM M., CHIN B., KANG S., LEE D., Eds., *Proc. of the 21st IFIP WG 6.1 Int. Conference on Formal Techniques for Networked and Distributed Systems FORTE’2001 (Cheju Island, Korea)*, IFIP, Kluwer Academic Publishers, p. 377–392, August 2001.

[GAR 01b] GARAVEL H., MATEESCU R., SMARANDACHE I., "Parallel State Space Construction for Model-Checking", DWYER M. B., Ed., *Proc. of the 8th Int. SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada)*, vol. 2057 of *Lecture Notes in Computer Science*, Springer Verlag, p. 217–234, May 2001.

[GAR 02a] GARAVEL H., HERMANNS H., "On Combining Functional Verification and Performance Evaluation using CADP", ERIKSSON L.-H., LINDSAY P. A., Eds., *Proc. of the 11th Int. Symposium of Formal Methods Europe FME'2002 (Copenhagen, Denmark)*, vol. 2391 of *Lecture Notes in Computer Science*, Springer Verlag, p. 410–429, July 2002.

[GAR 02b] GARAVEL H., LANG F., MATEESCU R., "An Overview of CADP 2001", *European Association for Software Science and Technology (EASST) Newsletter*, vol. 4, p. 13–24, August 2002, also available as INRIA Technical Report RT-0254.

[GAR 06] GARAVEL H., MATEESCU R., BERGAMINI D., CURIC A., DESCOUBES N., JOUBERT C., SMARANDACHE-STURM I., STRAGIER G., "DISTRIBUTOR and BCG_MERGE: Tools for Distributed Explicit State Space Generation", HERMANNS H., PALBERG J., Eds., *Proc. of the 12th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2006 (Vienna, Austria)*, Lecture Notes in Computer Science, Springer Verlag, March 2006.

[GRO 97] GROOTE J. F., "The Syntax and Semantics of Timed μCRL", Technical Report num. SEN-R9709, CWI, Amsterdam, 1997.

[HOL 03] HOLZMANN G., *The SPIN Model Checker – Primer and Reference Manual*, Addison-Wesley Professional, 2003.

[ISO 89] ISO/IEC, LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, Int. Standard num. 8807, Int. Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva, September 1989.

[JOU 04] JOUBERT C., MATEESCU R., "Distributed On-the-Fly Equivalence Checking", BRIM L., LEUCKER M., Eds., *Proc. of the 3rd Int. Workshop on Parallel and Distributed Methods in Verification PDMC'2004 (London, UK)*, vol. 128 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2004.

[JOU 05] JOUBERT C., MATEESCU R., "Distributed Local Resolution of Boolean Equation Systems", TIRADO F., PRIETO M., Eds., *Proc. of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing PDP'2005 (Lugano, Switzerland)*, IEEE Computer Society, p. 264–271, February 2005.

[KOZ 83] KOZEN D., "Results on the Propositional μ-calculus", *Theoretical Computer Science*, vol. 27, p. 333–354, 1983.

[LAN 05] LANG F., "EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-Fly Verification Methods", VAN DE POL J., ROMIJN J., SMITH G., Eds., *Proc. of the 5th Int. Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, vol. 3771 of *Lecture Notes in Computer Science*, Springer Verlag, p. 70–88, November 2005.

[MAT 00] MATEESCU R., "Efficient Diagnostic Generation for Boolean Equation Systems", GRAF S., SCHWARTZBACH M., Eds., *Proc. of 6th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2000 (Berlin, Germany)*, vol. 1785 of *Lecture Notes in Computer Science*, Springer Verlag, p. 251–265, March 2000.

[MAT 03a] MATEESCU R., "A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems", GARAVEL H., HATCLIFF J., Eds., *Proc. of the 9th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003 (Warsaw, Poland)*, vol. 2619 of *Lecture Notes in Computer Science*, Springer Verlag, p. 81–96, April 2003.

[MAT 03b] MATEESCU R., SIGHIREANU M., "Efficient On-the-Fly Model-Checking for Regular Alternation-Free μ-Calculus", *Science of Computer Programming*, vol. 46, num. 3, p. 255–281, March 2003.

[MAT 05] MATEESCU R., "On-the-Fly State Space Reductions for Weak Equivalences", MARGARIA T., MASSINK M., Eds., *Proc. of the 10th Int. Workshop on Formal Methods for Industrial Critical Systems FMICS'05 (Lisbon, Portugal)*, ERCIM, ACM Computer Society Press, p. 80–89, September 2005.

[MAT 06] MATEESCU R., "CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems", *Springer Int. Journal on Software Tools for Technology Transfer (STTT)*, vol. 8, num. 1, p. 37–56, February 2006.

[MIL 89] MILNER R., *Communication and Concurrency*, Prentice-Hall, 1989.

[NIC 90] NICOLA R. D., VAANDRAGER F. W., "Action versus State Based Logics for Transition Systems", GUESSARIAN I., Ed., *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science (La Roche Posay, France)*, vol. 469 of *Lecture Notes in Computer Science*, Springer Verlag, p. 407–419, April 1990.

[PAC 03] PACE G., LANG F., MATEESCU R., "Calculating τ-Confluence Compositionally", WARREN A. HUNT J., SOMENZI F., Eds., *Proc. of the 15th Int. Conference on Computer Aided Verification CAV'2003 (Boulder, Colorado, USA)*, vol. 2725 of *Lecture Notes in Computer Science*, Springer Verlag, p. 446–459, July 2003.

[SAL 05] SALAÜN G., SERWE W., "Translating Hardware Process Algebras into Standard Process Algebras – Illustration with CHP and LOTOS", VAN DE POL J., ROMIJN J., SMITH G., Eds., *Proc. of the 5th Int. Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, vol. 3771 of *Lecture Notes in Computer Science*, Springer Verlag, p. 287–306, November 2005.

[SCH 03] SCHIFFELERS R., VAN BEEK D., MAN K., RENIERS M., ROODA J., "Formal Semantics of Hybrid χ", LARSEN K. G., NIEBERT P., Eds., *Proc. of the 1st Int. Workshop on Formal Modeling and Analysis of Timed Systems FORMATS'03 (Marseille, France)*, vol. 2791 of *Lecture Notes in Computer Science*, Springer Verlag, p. 151–165, September 2003.

This page intentionally left blank

Chapter 6

Synchronous Program Verification with Lustre/Lesar

The synchronous approach was proposed in the middle of the 1980s with the aim to reconcile concurrent programming with determinism. The Lustre language belongs to this approach. It proposes a data-flow programming style, close to classical models such as block diagrams or sequential circuits. The semantics of the language is formally defined, and thus formal verification of program functionality is possible.

During the 1990s, proof methods based on model exploration (model checking) have been applied with success in domains such as protocol or circuit verification. Model checking techniques were presented in Chapter 3. These methods have been adapted for the validation of programs written in the Lustre language, and a specific model checker (Lesar) has been developed. The first part of this chapter is dedicated to the presentation of the language and the problems raised by its formal verification: expression of properties and assumptions, conservative abstraction of infinite systems, etc. The second part is more general and technical. It details some exploration methods for finite state models. These methods are mainly inspired by previous works on the verification of sequential circuits (symbolic model checking).

6.1. Synchronous approach

6.1.1. *Reactive systems*

The targeted domain is safety critical reactive systems (embedded systems, control/command). The abstract behavior of a typical reactive system is represented in Figure 6.1:

Chapter written by Pascal RAYMOND.

– it reacts to the inputs provided by its environment by producing outputs;

– it is itself designed as a hierarchical composition of reactive (sub)systems running in parallel.

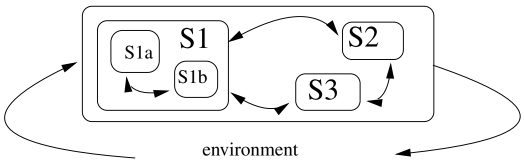


Figure 6.1. *A typical reactive system*

6.1.2. *The synchronous approach*

A classical approach for implementing a reactive system on a centralized architecture consists of transforming the description-level parallelism into execution-level parallelism:

– atomic processes (or threads) are attached to each sub-system,

– scheduling and communications are carried out by the execution platform (in general a real-time oriented operating system).

The main drawbacks of this approach are:

– the overhead due to the executive platform,

– the difficulty in analyzing the global behavior of the whole implementation (determinism problem).

The synchronous approach for programming and implementing reactive systems has been proposed to reconcile parallel design with determinism:

– At the design level, the programmer works in an "idealized" world, where both computations and communications are performed with zero-delay (simultaneity between inputs and outputs). He/she can then concentrate only on the *functionality* of the system.

– At the implementation level, the language and its compiler ensure that the execution time is bounded; the validity of the synchronous hypothesis can then be validated for a particular hardware platform.

6.1.3. *Synchronous languages*

Several languages, based on the synchronous approach, have been proposed:

1) Lustre is a data-flow language; it has been transferred to industry within the tool-suite SCADE[1] tool-suite;

1. `http://www.esterel-technologies.com/products/overview.html`.

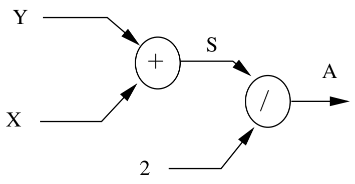
2) Signal [LEG 91] is also a data-flow language, which moreover provides a sophisticated notion of clock;

3) Esterel [BER 92] is an imperative language providing "classical" control structures (sequence, loop, concurrency, watch-dog, etc.). The graphical version is based on hierarchical concurrent automata (SynchCharts [AND 96]).

6.2. The Lustre language

6.2.1. *Principles*

The language [HAL 91] adopts the classical data-flow approach, where a system is viewed as a network of operators connected by wires carrying flows of values (Figure 6.2). Roughly speaking, Lustre is then a textual formalism for describing such diagrams.



```
node Average(X,Y:int)
returns(A:int);
var S:int;
let
   A = S/2;
   S = (X+Y);
tel
```

Figure 6.2. *A data-flow diagram and the corresponding Lustre program*

Besides its data-flow style, the main characteristics of the language are the following:

1) declarative language: the body of a program is a set of equations, whose order is meaningless. This is the substitution principle: if `id = expr`, then any occurrence of `id` can be replaced by `expr` without modifying the semantics of the program. For instance, in Figure 6.2, the intermediate variable `S` can be avoided, so the program is equivalent to the single equation `A = (X+Y)/2`;

2) synchronous semantics: the data-flow interpretation is defined according to an implicit discrete-time clock. For instance, `A` denotes an infinite sequence of integer values $\mathtt{A}_0, \mathtt{A}_1, \mathtt{A}_2, \ldots$. Data operators are operating point-wise on sequences; for instance, `A = (X+Y)/2` means that $\forall t \in \mathbb{N}$, $\mathtt{A}_t = (\mathtt{X}_t + \mathtt{Y}_t)/2$;

3) temporal operators: in order to describe complex dynamic behaviors, the language provides the `pre` operator (for previous), which shifts flows forward one instant: $\forall t \geqslant 1$ $(\mathtt{pre\ X})_t = \mathtt{X}_{t-1}$, and $(\mathtt{pre\ X})_0 = \bot$. The `pre` operator comes with an initialization operator in order to avoid undefined values: $(\mathtt{X\ ->\ Y})_0 = \mathtt{X}_0$ and $\forall t \geqslant 1$ $(\mathtt{X\ ->\ Y})_t = \mathtt{Y}_t$. For instance, `N = 0 -> pre N + 1` defines the sequence of positive integers $(0, 1, 2, 3, \ldots)$;

4) dedicated language: the language is specifically designed to program reactive kernels, not for programming complex data transformation. Basic data types are Boolean (`bool`), integers (`int`) and floating-point values (`real`). Predefined operators are all the classical logical and arithmetics operators. More complex data types, together with the corresponding operators, are kept abstracted in Lustre, and must be implemented in a host language (typically C)[2];

5) modularity: once defined, a program (called a `node`) can be reused in other programs, just like a predefined operator.

Note that the language, reduced to its Boolean subset, is equivalent to the classical formalism of sequential circuits: (`and`, `or`, `not`) are the logical gates, and the construct `false -> pre` is similar to the notion of register.

6.2.2. *Example: the beacon counter*

Before presenting the example, we first define an intermediate node which is particularly useful as a basic brick for building control systems: it is a simple flip/flop system, initially in the state `orig`, and commanded by the buttons `on` and `off`:

```
node switch(orig,on,off: bool) returns (s: bool);
let
   s = orig -> pre(if s then not off else on);
tel
```

A train runs on a railway where beacons are deployed regularly. Inside the train, a beacon counter receives a signal each time the train crosses a `beacon`, and also a signal `second` coming from a real time clock.

Knowing that the cruising speed should be *one beacon per second*, the program must check whether the train is `early`, `on time` or `late`. An hysteresis mechanism prevents oscillation artefacts (dissymmetry between state change conditions).

The corresponding Lustre program is shown in Figure 6.3.

6.3. Program verification

The methods presented here are indeed not specific to the Lustre language: they can be applied to any formalism relying on a discrete-time semantics. This is in particular the case for the other synchronous languages (Esterel, Signal, etc.), and more generally for any formalism similar to sequential circuits.

2. This remark mainly concerns the basic Lustre language; recent versions, in particular the one from the SCADE tool, are much richer.

```
node counter(sec,bea: bool) returns (ontime,late,early: bool);
var diff: int;
let
   diff = (0 -> pre diff) + (if bea then 1 else 0) +
         (if sec then -1 else 0);
   early = switch(false, ontime and (diff > 3), diff <= 1);
   late = switch(false, ontime and (diff < -3), diff >= -1);
   ontime = not (early or late);
tel
```

Figure 6.3. *The beacon counter*

6.3.1. *Notion of temporal property*

Our goal is functional verification: does the program compute the "right" outputs? In other terms, the goal is to check whether the program satisfies the expected properties. Since we consider dynamic systems, the expected properties are not simply relations between instantaneous inputs and outputs, but rather between inputs and outputs sequences, hence the term *temporal properties*.

6.3.2. *Safety and liveness*

There exists a whole theory on the classification of temporal properties (and the corresponding logics [MAN 90]). For our purposes, we simply need to recall the "intuitive" distinction between:

– *safety* properties, expressing that something (bad) never happens;

– *liveness* properties, expressing that something (good) may or must happens.

6.3.3. *Beacon counter properties*

Here are some properties that we may expect from the beacon counter system:

– it is never both early and late;

– it never switches directly from late to early (and conversely);

– it is never late for a single instant;

– if the train stops, it eventually becomes late.

The first three properties are *safeties*, while the last one is clearly a typical *liveness*.

6.3.4. *State machine*

The abstract behavior of a Lustre program (more generally, of a synchronous program) is the one of a deterministic *state machine* (Figure 6.4):

– the initial state of the memory M is known;

– the values of the outputs, together with those of the next state, are functionally defined according to the current values of the inputs and the memory: $f : I \times M \rightarrow O$ is the output function, $g : I \times M \rightarrow M$ is the transition function.

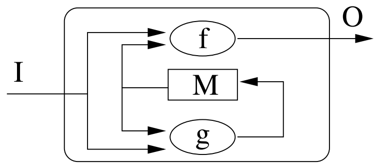


Figure 6.4. *State machine*

6.3.5. *Explicit automata*

A state machine is also an implicit automata: it is equivalent to a state/transition system (automaton) where each state is a particular configuration of the memory.

Figure 6.5 shows a small part of the beacon counter automaton. States are representing the values of the memory: the value of `pre diff` is the integer inside the state; the values of the three Boolean memories (`pre(ontime)`, `pre(early)` and `pre(late)`) are implicitly represented by the position of the state:

– in the top states, `pre(ontime)` is true and the others are false;

– in the bottom-right states, `(pre early)` is true, and the others are false;

– in the bottom-left states, `(pre late)` is true, and the others are false.

Moreover, transitions are guarded by conditions on inputs; for the sake of simplicity, self-loops and guards are not represented on Figure 6.5:

– "`bea and not sec`" guards all the transitions going from left to right;

– "`not bea and sec`" guards all the transitions going from right to left;

– "`bea = sec`" guards all the self-loops.

6.3.6. *Principles of model checking*

The explicit automata is a model which synthesizes all the possible behaviors of the program. The exploration of this model makes it possible to study the functional properties of the program. However, in general, the automaton is infinite, or at least enormous, and an exhaustive exploration is impossible. The idea is then to consider a finite (and not too big) approximation of the automaton.

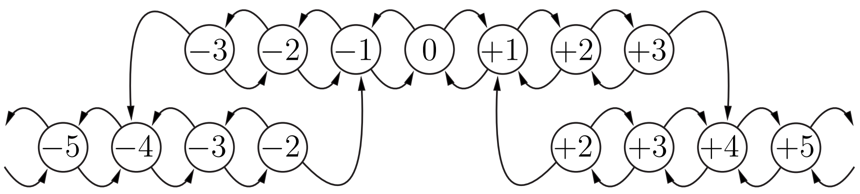
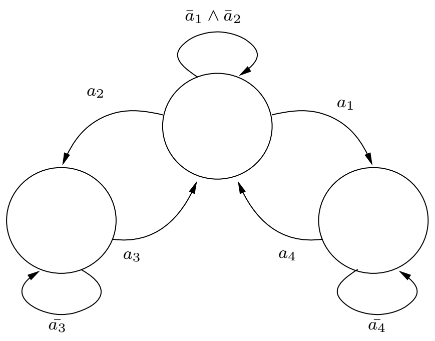


Figure 6.5. *Explicit states of the beacon counter*

Such an approximation is obtained by forgetting information: it is an *abstraction* of the program. This abstraction must be conservative for some class of properties, otherwise it would be useless.

6.3.7. *Example of abstraction*

The Boolean abstraction consists of considering only the logical part of the system. For Lustre programs, this means that numerical variables and operators are abstracted away. In the example, it consists of ignoring the internal counter `diff`, and introducing free logical variables at the "frontier" between Boolean and numerical values (Figure 6.6).



– a_1 replaces `diff > 3`,
– a_2 replaces `diff < -3`,
– a_3 replaces `diff >= -1`,
– a_4 replaces `diff <= 1`,

Figure 6.6. *A Boolean abstraction of the beacon counter*

This abstraction does not represent exactly the behaviors of the program, but an over-approximation: some behaviors are possible on the abstraction, while they were impossible on the concrete program.

Nevertheless, some properties are still satisfied:

– impossible to be both early and late (safety);

– impossible to pass directly from late to early (safety);

– if the train stops, it eventually becomes late (liveness).

Other properties are lost:

– impossible to remain late for a single instant (safety).

Additionally, the converse of this property, not satisfied by the program, has been introduced by the abstraction:

– it is possible to remain late for a single instant (liveness).

It is then very important to precisely state which conclusions we can take from the study of the abstraction.

6.3.8. *Conservative abstraction and safety*

The Boolean abstraction is a particular case of over-approximation: it may introduce new behaviors, but does not remove behaviors from the concrete program. As a conclusion, if a behavior does not exist on the abstraction, it certainly does not exist on the real program.

It is then easy to conclude that any over-approximation is conservative for the class of safety properties: safety properties may be kept or lost, but cannot be introduced.

Over-approximation is in fact also conservative for certain subclasses of liveness properties; this is the case for properties stating that an event eventually happens. This kind of property is often called “unbounded liveness”, since the event may occur anytime in the future.

On the contrary, over-approximation is clearly not conservative for liveness properties stating that some behavior is possible.

6.4. Expressing properties

6.4.1. *Model checking: general scheme*

Model checking is an important domain with important studies (see Chapter 3). The application domains are (historically) communication protocols and logical circuits. In the case of non-deterministic, deadlocking systems, reasoning in terms of behavior sets is not sufficient: it is necessary to reason in terms of computation trees. This case, related to computation tree logic (CTL), is not presented here. We only consider here the case where the semantics can be expressed in terms of behavior sets (related to linear temporal logic, LTL).

The classical model checking principle is shown on Figure 6.7:

– A model of the system is provided, which comes from a high level specification (in the case of protocols), or which has been automatically obtained from a concrete implementation (in the case of logical circuits).

– The expected property is expressed in a particular logic (typically LTL).

– The LTL formula is translated into an operational form, typically an automaton with some accepting criterion. In the case of unbounded liveness, a criterion for accepting infinite, non-monotonic sequences is necessary (Büchi automata).

– Model checking itself consists of exploring the product of the two automata to verify that any sequence that can be generated by the model is accepted by the formula.

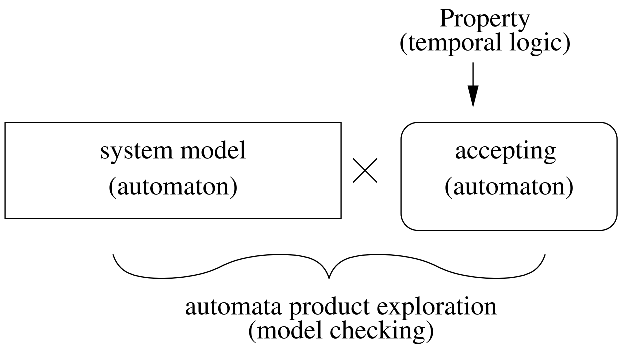


Figure 6.7. *Classical model checking scheme*

6.4.2. *Model checking synchronous program*

The classical scheme is adapted and simplified for the case of synchronous programs. First, liveness properties are not treated at all, for both theoretical, pragmatic and practical reasons:

– as explained previously, abstractions are required that do not conserve the whole class of liveness properties: the sub-class expressing potentiality cannot be model checked at all;

– abstraction is conservative for unbounded liveness (expressing eventuality). However, from a pragmatic point of view, such a liveness may seem too “vague”. Moreover, it is usually possible to replace it by a more precise property: for instance, “an alarm is eventually raised in case of a problem”, can be replaced by “an alarm is raised within 20 second in case of a problem”, which is a proper safety property;

– from a practical point of view, handling liveness properties requires sophisticated formalism and models (temporal logics, Büchi automata). On the other hand, safety properties (that are nothing more than classical program invariants) can be *programmed*, as explained in the next section.

For all these reasons, we restrict ourselves to the verification of safety properties.

6.4.3. *Observers*

Since only safety properties are considered, it is not necessary to use (complex) temporal logics for expressing properties: we can express properties by programming a suitable *observer*. Intuitively, an observer is a synchronous program taking as inputs the variables of the system to check, and producing a Boolean output which remains true as long as the expected property is satisfied. If we can establish that this output remains true whatever the input sequence, then the underlying safety property is proved. One advantage of using observers is that they can be programmed in the same language than the system to check, and thus it is not necessary to learn new formalisms. This pragmatic argument is very important for the diffusion of formal methods in industry. Note that it is completely equivalent to observe failures: in this case, the observer outputs an alarm when the property is violated, and the model checking consists of verifying that the alarm is never emitted, whatever the input sequence is.

6.4.4. *Examples*

Here are some safety properties expressed in Lustre. By convention, the observer result is called `ok`:

1) impossible to be early and late:

```
ok = not (early and late).
```

2) impossible to pass directly form late to early:

```
ok = true -> not ((pre late) and early).
```

3) impossible to remain late for a single instant:

```
ok = not Pre(Pre(late)) and Pre(late) => late
```

where `Pre(x) = false -> pre x`.

6.4.5. *Hypothesis*

In general, a program is not supposed to work properly if the environment does not satisfy some properties. Moreover, it is natural to breakdown a specification according to the classical cause/consequence duality.

For instance, the specification “if the train keeps the cruising speed, it remains on time” can be split according to the cause/consequence scheme:

– the property is simply `ok = ontime`;

– the hypothesis (keeps the cruising speed) is a little bit more complicated. A naive version could be that beacons and seconds are exactly synchronous `(sec = bea)`; a more sophisticated (and realistic) version is that beacons and seconds are alternating `(alternate(sec,bea)`, where `alternate` is defined as follows:

```
node alternate(x,y: bool) returns (ok: bool);
var forbid_x, forbid_y : bool;
let
   forbid_x = switch(false, x, y);
   forbid_y = switch(false, y, x);
   ok = not(x and forbid_x or y and forbid_y);
tel
```

Note that one practical advantage of using observers is the ability of defining libraries of generic observers that can themselves be formally validated.

6.4.6. *Model checking of synchronous programs*

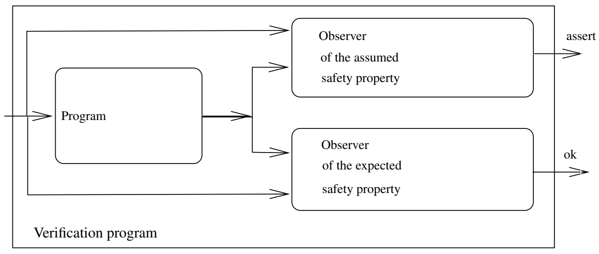


Figure 6.8. *Verification program*

Figure 6.8 shows the principle of the verification with observers. This principle is used in several verification tools: the Lustre model checker (Lesar), the Esterel model checker, and also in the SCADE integrated verification tool.

The user may provide a verification program made of:

– the program under verification itself;

– an observer of the expected property;

– an observer of the assumed property (hypothesis); in Lesar, such assumptions are given through the special construct `assert`.

Given this verification program, the role of the model checker consists of:

– automatically extracting a finite abstraction (typically a Boolean one, as explained before);

– exploring this abstraction in order to establish that, whatever the input sequence, if `assert` remains true, then `ok` remains true as well.

In terms of temporal logics, the actual property to check is:

```
(always assert) ⇒ (always ok)
```

This kind of property is not a proper safety, since the negation of the premise is actually a liveness. However, this very special case (invariant implies invariant) does not raise theoretical problems: finite abstraction is also conservative for this kind of property.

The problem is technical, since checking such a property is much more costly than checking a simple invariant. Intuitively, a first computation is necessary to remove from the model all paths that eventually lead to the violation of the assumption. Then, a second exploration is necessary to check whether `ok` remains true on this restricted model.

This is why we make another (conservative) abstraction. We use a Boolean variable `so_far_assert`, which is true if and only if the assertion has not been violated so far:

```
so_far_assert = assert and (true -> pre so_far_assert);
```

The exact property is then approximated by:

$$\text{always}(\texttt{so\_far\_assert} \Rightarrow \texttt{ok})$$

6.5. Algorithms

6.5.1. *Boolean automaton*

The Boolean abstraction results in a purely logical system. Such a system is an implicit *Boolean automaton* characterized by:

– a set of free variables (inputs) V;

– a set of state variables (registers) S, and, for each state variable $k = 1 \cdots |S|$, the corresponding transition function $g_k : \mathbb{B}^{|S|} \times \mathbb{B}^{|V|} \rightarrow \mathbb{B}$;

– a set of initial states, characterized by a condition $Init : \mathbb{B}^{|S|} \rightarrow \mathbb{B}$ (in general, there is a unique initial state, but it has no importance here);

– the assumption $H : \mathbb{B}^{|S|} \times \mathbb{B}^{|V|} \rightarrow \mathbb{B}$;

– and the property $\Phi : \mathbb{B}^{|S|} \times \mathbb{B}^{|V|} \rightarrow \mathbb{B}$.

6.5.2. *Explicit automaton*

The Boolean automaton "encodes" an explicit state/transition system:

– the set of states is $Q = \mathbb{B}^{|S|}$ (all the possible values of the memory);

– initial states are the subset characterized by $Init \subseteq Q$ (we safely mix up sets and characteristic functions);

– the transition relation $R \subseteq Q \times \mathbb{B}^{|V|} \times Q$ is defined by:

$$(q, v, q') \in R \Leftrightarrow q'_k = g_k(q, v) \quad k = 1 \cdots |S|.$$

We note $q \xrightarrow{v} q'$ for $(q, v, q') \in R$. In our case, R is in fact a function: q' is unique for any pair (q, v). This is why we call "transitions" the pairs from $T = Q \times \mathbb{B}^{|V|}$.

– $H \subseteq T$ is the subset of transitions satisfying the assumption;

– $\Phi \subseteq T$ is the subset of transitions satisfying the (expected) property.

6.5.3. *The "pre" and "post" functions*

In order to simplify the problem, we first formalize the problem in terms of states, by defining the *post* functions (states that can be reached from somewhere by satisfying the assumption), and the *pre* functions (states that can lead to somewhere by satisfying the assumption).

For any state $q \in Q$, and any set $X \subseteq Q$ we define:

– $\mathit{post}_H(q) = \{q' \mid \exists v \; q \xrightarrow{v} q' \wedge H(q, v)\}$;

– $\mathit{POST}_H(X) = \bigcup_{q \in X} \mathit{post}_H(q)$;

– $\mathit{pre}_H(q) = \{q' \mid \exists v \; q' \xrightarrow{v} q \wedge H(q', v)\}$;

– $\mathit{PRE}_H(X) = \bigcup_{q \in X} \mathit{pre}_H(q)$.

6.5.4. *Outstanding states*

Starting from initial states Init, we define the set of accessible states as the following fix-point:

$$\text{Acc} \;=\; \mu X \cdot (X = \text{Init} \cup \mathit{POST}_H(X)).$$

that is, the smallest set containing initial states and kept invariant by the function POST_H.

Immediate error states are those such that at least one of the outgoing transitions satisfies the assumption but violates the property:

$$\text{Err} \;=\; \{q \mid \exists v \; H(q, v) \wedge \neg\Phi(q, v)\},$$

By extension, we define the set of "bad" states as those that may lead to an error state:

$$\text{Bad} \;=\; \mu X \cdot (X = \text{Err} \cup \mathit{PRE}_H(X)),$$

that is, the smallest set containing (immediate) error states and kept invariant by the function PRE_H.

In the following, we note $\mathrm{Acc}_0 = \mathrm{Init}$ and $\mathrm{Bad}_0 = \mathrm{Err}$.

6.5.5. *Principles of the exploration*

The goal of the proof is to establish (if possible) that $\mathrm{Acc} \cap \mathrm{Bad} = \emptyset$, that is, no accessible state is also a bad state.

Indeed, it is useless to compute the two fix-points for that, and we may try to establish indiscriminately:

– that $\mathrm{Acc} \cap \mathrm{Bad}_0 = \emptyset$ (*forward* method);

– that $\mathrm{Bad} \cap \mathrm{Acc}_0 = \emptyset$ (*backward* method).

In the case of deterministic systems, the two methods are not symmetric: the transition relation is a function (forward method), but its reverse relation has no special property in general (backward method).

6.6. Enumerative algorithm

The forward enumerative algorithm (Figure 6.9) is a typical example of fix-point calculus.

Starting from the initial state(s), reachable states are explored one by one, checking each of them for the validity of Φ. If the exploration converges without errors, the property is proved.

```
Known := Init;    Explored := $\emptyset$
while it exists $q$ in $Known \setminus Explored$ {
    for all $q'$ in $post(q)$ {
        if $q' \in$ Err then Output(failure)
        move $q'$ to Known
    }
    move $q$ to Explored
}
Output(success)
```

Figure 6.9. *Enumerative algorithm*

Figure 6.10 illustrates, in the left-hand side, an exploration that has succeeded: the exploration has converged without reaching any state in Err. In the right-hand side, a state from Err is reached during the exploration and the proof fails.

In a concrete implementation, a lot of work is necessary to make the algorithm efficient (or at least to limit its inefficiency!).

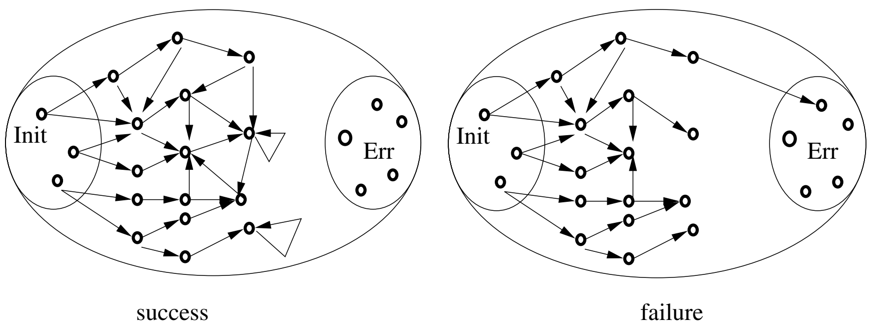


Figure 6.10. *Forward enumerative exploration*

As a matter of fact, the theoretical complexity is incredibly huge: the number of reachable states is potentially exponential ($2^{|S|}$), and, for each state, an exponential number of transitions must be explored ($2^{|V|}$). Enumerative methods are then limited to relatively small systems (typically a few tens of variables).

By exchanging roles of Init and Err, and replacing *POST* by *PRE*, we trivially obtain a backward enumerative algorithm. However, such an algorithm is never used because it is likely to be even more inefficient. The problem is related to determinism: forward enumeration mainly consists of applying functions, while enumerating backward consists of solving relations.

6.7. Symbolic methods and binary decision diagrams

Symbolic model checking methods [BUR 90, COU 89b, COU 90, TOU 90] have been proposed during the 1990s. They are mainly due to:

– the idea to overpass the limits due to state explosion by adopting implicit representations rather than explicit ones;

– the (re)discovery of *binary decision diagrams* (BDD), making it possible to (often consisely) represent logical predicates, and operating on them ([AKE 78, BRY 86]).

6.7.1. *Notations*

Thereafter, f and g denote predicates (i.e. Boolean functions) and:

– 0 denotes the always-false predicate, 1 the always-true predicate;

– we note either $f \cdot g$ or $f \wedge g$ for the conjunction (logical and);

– we note either $f + g$ or $f \vee g$ for the disjunction (inclusive or);

– $f \oplus g$ represents the difference (exclusive or);

– $\neg(f)$ denotes the negation, which is also denoted by $\bar{x}$ when the predicate is reduced to a single variable x.

A tuple (of variables, of functions, etc.) is represented either

– as an explicit sequence $\vec{x} = [x_1, \ldots, x_n]$;

– or as a recursive list $\vec{x} = x_1 :: \vec{x}'$, when the goal is simply to isolate the first element x_1 from the others $\vec{x}'$. The empty tuple/list is denoted by [].

6.7.2. *Handling predicates*

Any set of states or transitions can be represented implicitly by a predicate: its characteristic function.

For instance, if x, y, z are state variables, the predicate $(x \oplus y).\bar{z}$ encodes all the states where z is false and where either x or y (but not both) is true. If these three variables are the only ones, this set contains 2 elements from $2^3 = 8$ possible states ($(0,1,0)$ and $(0,0,1)$). However, supposing that there exist n other variables, whose values are not relevant, then the same predicate encodes a set with $2 * 2^n$ elements. As a consequence, we can consider that the predicate is a concise symbolic representation of the underlying set.

Operations on sets can obviously be defined in terms of predicates: union is isomorphic to disjunction, intersection to conjunction and complement to negation. It is then possible to define exploration algorithms that directly handle sets (of states, of transitions). In order to implement such algorithms, it is necessary to define some appropriate data-structure for predicates.

In particular, such a data structure must make it possible to check the emptiness of a set, or, equivalently, the equality of two sets. In terms of logic, it is equivalent to check the satisfiability of a predicate, which is nothing but the archetype of the NP-complete decision problem. In other terms, no algorithm is known that solves the problem with a sub-exponential cost.

As a consequence we can already conclude that symbolic methods will be limited to a class of "simple systems". However, this class is likely to be much larger than the one that can be handled with enumerative methods: symbolic methods are potentially exponential, while enumerative methods are always exponential.

6.7.3. *Representation of the predicates*

6.7.3.1. *Shannon's decomposition*

Any Boolean function $f(x, y, \ldots, z)$ can be decomposed according to the value of x into a pair of functions, f_x and $f_{\bar{x}}$, not depending on x:

$$f(x, y, \ldots, z) = x \cdot f_x(y, \ldots, z) + \bar{x} \cdot f_{\bar{x}}(y, \ldots, z)$$

$$\text{with: } f_x(y, \ldots, z) = f(1, y, \ldots, z)$$

$$\text{and: } f_{\bar{x}}(y, \ldots, z) = f(0, y, \ldots, z)$$

For instance, for $f(x, y, z) = x \cdot y + (y \oplus z)$, we get:

$$\begin{aligned} f(x, y, z) &= x \cdot f(1, y, z) + \bar{x} \cdot f(0, y, z) \\ &= x \cdot (1 \cdot y + (y \oplus z)) + \bar{x} \cdot (0 \cdot y + (y \oplus z)) \\ &= x \cdot (y + (y \oplus z)) + \bar{x} \cdot (y \oplus z) \end{aligned}$$

This decomposition can be represented by a binary tree whose root is labeled by the variable, and whose right branch leads to f_x (called the high branch), while its left branch leads to $f_{\bar{x}}$ (called the low branch):

$$\begin{tikzcd} & x \arrow[dl, dash] \arrow[dr, dash] & \\ (y \oplus z) & & y + (y \oplus z) \end{tikzcd}$$

By recursively applying the decomposition for all the variables, we obtain a complete binary tree whose leaves are not depending on any variable, that is, the constants 1 (always true) or 0 (always false). Shannon's tree of f is represented in Figure 6.11.

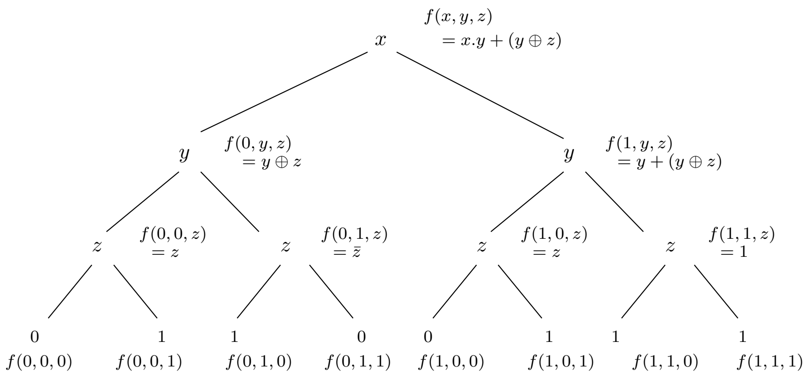


Figure 6.11. *Shannon's tree of* $f(x, y, z) = x \cdot y + (y \oplus z)$

For a fixed variable order, the complete binary tree is a canonical representation of the function, which means that any function semantically equivalent to f gives the same tree. For instance, $g(x, y, z) = y \cdot (x + \bar{z}) + \bar{y} \cdot z$ is syntactically different from f, but leads to the same Shannon tree when decomposed according to the same variable order: it is actually the same function.

6.7.3.2. *Binary decision diagrams*

Some nodes in Shannon's tree are useless: this is the case for any binary node whose high and low branches are leading to isomorphic sub-trees. Such a node can be eliminated by directly connecting the sub-tree to its father node. By eliminating all useless nodes, we obtain the “reduced Shannon’s tree”, which is still a canonical representation of the corresponding function.

Finally, isomorphic sub-trees can be shared in order to obtain a directed acyclic graph (DAG), as shown in Figure 6.12. This graph is called the “binary decision diagram” of the function, for the considered variable order.

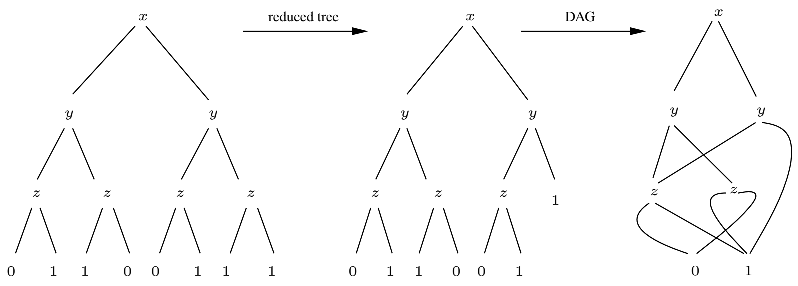


Figure 6.12. *From Shannon’s tree to BDD*

6.7.4. *Typical interface of a BDD library*

Figure 6.12 illustrates the principle of BDD, but this is not the way BDDs are actually built: it is not necessary to build an exponential binary tree as an intermediate structure for a graph that we expect to be concise! In practice, BDD operations are available through an abstract interface allowing the user to manipulate predicates without considering the details of the implementation (in particular the variable order). A typical interface provides:

– the equivalence (which is actually the equality);

– constant functions (leaves 0 and 1);

– identity, for all x;

– all the classical logical operators (and, or, not, etc.);

– quantifiers, which, in the Boolean case, can be defined in terms of logical operations, for instance: $\exists x : f(x, y, \ldots, z) = f(1, y, \ldots, z) + f(0, y, \ldots, z)$.

6.7.5. *Implementation of BDDs*

Internally, a BDD library handles a set of variables $\mathcal{V}$, totally ordered by a relation $\leqslant$. We classically denote $<$ (respectively $\geqslant$) the induced strict relation (respectively the reverse relation). The set of BDDs over $(\mathcal{V}, <)$ is denoted $\mathcal{B}$. In order to define it formally, the structure $(\mathcal{V}, <)$ is extended with a special value ∞ such that $\forall x \in \mathcal{V} : x < \infty$. $\mathcal{B}$ is defined together with the function $range : \mathcal{B} \rightarrow \mathcal{V} \cup \{\infty\}$ which, intuitively, gives the least significant variable appearing in a BDD, or ∞ in the case of a leaf.

The set of BDDs contains:

– the leaves 0 and 1, with $range(0) = range(1) = \infty$;

– the binary graphs $\gamma = \begin{array}{c} x \\ /\ \backslash \\ \alpha \quad \beta \end{array}$, such that $x < range(\alpha)$ and $x < range(\beta)$, with $range(\gamma) = x$.

The internal procedure builds, if necessary, a binary node, given one variable and two existing BDDs. A call to this procedure is represented by $\begin{array}{c} x \\ /\!\!/\ \backslash\!\!\backslash \\ \alpha \quad \beta \end{array}$, in order to distinguish it from actual nodes that will be represented with simple lines:

– $\begin{array}{c} x \\ /\!\!/\ \backslash\!\!\backslash \\ \alpha \quad \beta \end{array}$ undefined if $x \not< range(\alpha)$ or $x \not< range(\beta)$;

– $\begin{array}{c} x \\ /\!\!/\ \backslash\!\!\backslash \\ \alpha \quad \alpha \end{array} = \alpha$ (never build useless node);

– otherwise, $\begin{array}{c} x \\ /\!\!/\ \backslash\!\!\backslash \\ \alpha \quad \beta \end{array} = \begin{array}{c} x \\ /\ \backslash \\ \alpha \quad \beta \end{array}$.

6.7.6. *Operations on BDDs*

6.7.6.1. *Negation*

Building the negation is achieved by a simple recursive descent:

– case of leaves: $\neg 1 = 0$ et $\neg 0 = 1$;

– case of a binary node: $\neg \begin{array}{c} x \\ /\ \backslash \\ \alpha_0 \quad \alpha_1 \end{array} = \begin{array}{c} x \\ /\!\!/\ \backslash\!\!\backslash \\ \neg\alpha_0 \quad \neg\alpha_1 \end{array}$.

6.7.6.2. *Binary operators*

All classical operators distribute over Shannon's decomposition; let $\star$ be any operator (among $.$, $+$, $\oplus$, $\Leftrightarrow$):

$$\begin{array}{c} x \\ /\ \backslash \\ \alpha_0 \quad \alpha_1 \end{array} \star \begin{array}{c} x \\ /\ \backslash \\ \beta_0 \quad \beta_1 \end{array} = \begin{array}{c} x \\ /\!\!/\ \backslash\!\!\backslash \\ \alpha_0 \star \beta_0 \quad \alpha_1 \star \beta_1 \end{array}$$

If the arguments have different root variables, a "balance" should be performed in the recursive descent in order to meet the range constraints. If some variable $x < range(\beta)$ does not appear within a β, that means that β is (virtually) equivalent to $\begin{array}{c} x \\ /\!\!/\ \backslash\!\!\backslash \\ \beta \quad \beta \end{array}$.

Let $\alpha = \begin{array}{c} x \\ / \ \backslash \\ \alpha_0 \quad \alpha_1 \end{array}$ and $\beta = \begin{array}{c} y \\ / \ \backslash \\ \beta_0 \quad \beta_1 \end{array}$ be two BDDs with $x \neq y$:

– if $x < y$, $\alpha \star \beta = \begin{array}{c} x \\ / \ \backslash \\ \alpha_0 \quad \alpha_1 \end{array} \star \begin{array}{c} x \\ /\!\!/ \ \backslash\!\!\backslash \\ \beta \quad \beta \end{array} = \begin{array}{c} x \\ /\!\!/ \quad \backslash\!\!\backslash \\ \alpha_0 \star \beta \quad \alpha_1 \star \beta \end{array}$;

– if $y < x$, $\alpha \star \beta = \begin{array}{c} y \\ /\!\!/ \ \backslash\!\!\backslash \\ \alpha \quad \alpha \end{array} \star \begin{array}{c} y \\ / \ \backslash \\ \beta_0 \quad \beta_1 \end{array} = \begin{array}{c} y \\ /\!\!/ \quad \backslash\!\!\backslash \\ \alpha \star \beta_0 \quad \alpha \star \beta_1 \end{array}$.

These generic rules must be completed by terminal rules specific to each particular operator, for instance:

– $\alpha + 0 = 0 + \alpha = \alpha$ and $\alpha + 1 = 1 + \alpha = 1$;

– $\alpha \cdot 0 = 0 \cdot \alpha = 0$ and $\alpha \cdot 1 = 1 \cdot \alpha = \alpha$;

– $\alpha \oplus 0 = 0 \oplus \alpha = \alpha$ and $\alpha \oplus 1 = 1 \oplus \alpha = \neg\alpha$, etc.

Since the representation is canonical, it is also possible (and better) to exploit the known properties of the operators, for instance, for any BDD α:

– $\alpha + \alpha = \alpha$, $\alpha \cdot \alpha = \alpha$, $\alpha \oplus \alpha = 0$.

6.7.6.3. *Cofactors and quantifiers*

Instantiating a variable x within a BDD α with the value true or false is achieved by computing the cofactors *high*(x, α) or *low*(x, α):

– *high*(x, α) = *low*$(x, \alpha) = \alpha$ if $x < range(\alpha)$, i.e. x does not appear within α;

– *high* $\left(x, \begin{array}{c} x \\ / \ \backslash \\ \alpha_0 \quad \alpha_1 \end{array}\right) = \alpha_1$ and *low* $\left(x, \begin{array}{c} x \\ / \ \backslash \\ \alpha_0 \quad \alpha_1 \end{array}\right) = \alpha_0$;

– at last, if $y < x$, $c\left(x, \begin{array}{c} y \\ / \ \backslash \\ \alpha_0 \quad \alpha_1 \end{array}\right) = \begin{array}{c} y \\ /\!\!/ \quad \backslash\!\!\backslash \\ c(x, \alpha_0) \quad c(x, \alpha_1) \end{array}$, whatever the cofactor c (*high* or *low*).

Quantifiers can be defined in terms of cofactors:

– $(\exists x : \alpha)$ = *high*(x, α) + *low*(x, α);

– $(\forall x : \alpha)$ = *high*$(x, \alpha) \cdot$ *low*(x, α).

However, for the sake of efficiency, it is better to propose a more direct algorithm, based on a single recursive descent over α. We only give the algorithm for $\exists$ as the one for $\forall$ is similar:

– $(\exists x : \alpha) = \alpha$ if $x < range(\alpha)$;

– $\left(\exists x : \begin{array}{c} x \\ \diagup\ \diagdown \\ \alpha_0 \quad \alpha_1 \end{array}\right) = \alpha_0 + \alpha_1;$

– at last, if $y < x$ $\left(\exists x : \begin{array}{c} y \\ \diagup\ \diagdown \\ \alpha_0 \quad \alpha_1 \end{array}\right) \quad = \quad \begin{array}{c} y \\ \diagup\!\!\!\diagup \quad \diagdown\!\!\!\diagdown \\ (\exists x : \alpha_0)(\exists x\ \alpha_1) \end{array}.$

6.7.7. *Notes on complexity*

The rules presented in the previous sections correspond to high-level algorithms. An efficient implementation must use some memory cache mechanism in order to store (and retrieve) intermediate results.

If we suppose this memory cache perfect (no intermediate result is lost, and the overhead can be neglected), we can give a theoretical bound for the cost of operations on BDDs.

The complexity mainly depends on the size of the arguments, expressed in the number of BDD nodes (that is, taking into account the sharing of common subgraphs). The size is denoted $|\alpha|$.

The complexity of a binary operation between α and β is trivially bounded by $|\alpha| * |\beta|$: if the memory cache is perfect, each pair (node from α, node from β) is visited at most once.

For a quantification, for instance $\exists x : \alpha$, a rough approximation gives $|\alpha|^3$: for any occurrence of the variable x (roughly bounded by $|\alpha|$), a binary operation is performed between the corresponding branches (each of them also being roughly bounded by $|\alpha|$).

The bounds presented here are obviously very pessimistic; however, they are sufficient to state that the complexity of one operation is of polynomial magnitude.

This result holds for one single operation, and it is not meaningful for a typical use of the BDDs. As a matter of fact, the typical use of a BDD library consists of building the BDD of a predicate given as a classical Boolean expression. Let e be such an algebraic expression, and n the number of operators appearing in it (and, or, not, exists, etc.). The whole BDD construction cost is exponential in n in the worst case. The worst case is reached, for instance, with the following expression, with $n = 2k$:

$$(x_1 \oplus x_{k+1}) \cdot (x_2 \oplus x_{k+2}) \cdot \cdots \cdot (x_{k-1} \oplus x_{n-1}) \cdot (x_k \oplus x_n).$$

With the variable order $x_1 < x_2 < \cdots < x_n$, it is easily established that the size of the BDD is of order 2^n (exactly $3(2^k - 1)$).

This example also illustrates the dramatic influence of the variables ordering: the same expression with the ordering $x_1 < x_{k+1} < x_2 < x_{k+2} < \cdots < x_k < x_n$ gives a BDD whose size is linear in n (exactly $3(k-1)$).

For any expression, some optimal ordering exists, which gives a BDD of minimal size. However, finding such optimal ordering is in practice intractable because of the complexity.

Efficient BDDs implementations only try to avoid "clearly bad orderings": if the memory size grows too fast at run-time, it is probably because the current ordering is badly chosen. The BDD manager thus tries to modify the order in order to make the memory consuming decrease. This king of techniques is known as dynamic reordering ([RUD 93]).

6.7.8. ***Typed decision diagrams***

Efficient implementations are using a "trick" that makes it possible to share the graphs corresponding to the opposite functions f and $\neg f$.

Intuitively, only one of these functions is actually represented in the memory, while the other is defined, thanks to a single sign bit, to be its opposite. The first definition of this method is, as far as we know, due to J.P. Billon (*typed decision graph* or TDG [BIL 87, BIL 88]).

6.7.8.1. *Positive functions*

In order to guarantee canonicity, the set of logical functions $\mathcal{F}$ is split into:

– the set of positive functions, $\mathcal{F}^+$, which evaluate to true when all their arguments are true;

– the set of negative functions, $\mathcal{F}^-$, which evaluate to false when all their arguments are true.

For instance, the always-true function 1 is positive, and so are all the functions x, $x + y$, etc. On the contrary, 0, $\bar{x}$, $x \oplus y$, are examples of negative functions.

6.7.8.2. *TDG*

TDGs are a representation exploiting the partition between positive and negative functions. A TDG is a pair (a, α), where a is the sign $\in \{+, -\}$, and α is a binary decision graph corresponding to a positive function. We use the schematic notation $\overset{|+}{\alpha}$ or $\overset{|-}{\alpha}$.

Within a graph, edges are also labeled with a sign, according to the following rules:

– the unique leaf is the always-true function 1;

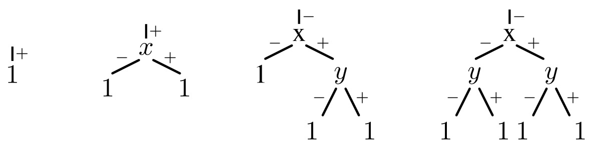


Figure 6.13. *From left to right, TDGs of* 1, x, $x \cdot \bar{y}$, $x \oplus y$

– binary nodes are either of the form $\underset{\alpha_0 \quad \alpha_1}{\overset{x}{{}^{+}\!/\ \backslash^{+}}}$, or $\underset{\alpha_0 \quad \alpha_1}{\overset{x}{{}^{-}\!/\ \backslash^{+}}}$, where α_1 and α_0 are themselves positive function graphs. Range rules, related to the variable ordering, are the same as those of classical BDDs.

Figure 6.13 shows some TDG examples for the ordering $x \prec y$ (sub-graph sharing is not represented for the sake of readability).

6.7.8.3. *TDG implementation*

Handling TDGs only requires a negligible overhead in terms of internal operations. All the rules defined for classical BDDs are still valid modulo a simple decoding/encoding of the sign bit:

– $\underset{1}{|^{+}}$ is interpreted as 1, $\underset{1}{|^{-}}$ as 0;

– $\underset{\alpha_0 \qquad \alpha_1}{\overset{|+}{{}^{+}\!\swarrow \overset{x}{}\searrow^{+}}}$ is equivalent to $\underset{\alpha_0 \quad \alpha_1}{\overset{x}{/\ \backslash}}$ and $\underset{\alpha_0 \qquad \alpha_1}{\overset{|+}{{}^{-}\!\swarrow \overset{x}{}\searrow^{+}}}$ to $\underset{\neg\alpha_0 \quad \alpha_1}{\overset{x}{/\ \backslash}}$;

– $\underset{\alpha_0 \qquad \alpha_1}{\overset{|-}{{}^{+}\!\swarrow \overset{x}{}\searrow^{+}}}$ is equivalent to $\underset{\neg\alpha_0 \quad \neg\alpha_1}{\overset{x}{/\ \backslash}}$ and $\underset{\alpha_0 \qquad \alpha_1}{\overset{|-}{{}^{-}\!\swarrow \overset{x}{}\searrow^{+}}}$ to $\underset{\alpha_0 \quad \neg\alpha_1}{\overset{x}{/\ \backslash}}$.

6.7.8.4. *Interest in TDGs*

The main result is that negation on TDGs has a constant (thus negligible) cost. In particular, we can exploit new properties in order to speed up computations, for instance: $\alpha + \neg\alpha = 1, \quad \alpha \cdot \neg\alpha = 0, \quad \alpha \oplus \neg\alpha = 1$.

For memory consuming concerns, we can establish that a TDG is always smaller than the equivalent BDD (for the same ordering), and that the gain is $1/2$ in the best case.

Finally, it appears that TDGs are strictly better than classical BDDs. As a matter of fact, even if it is not always explicit, all efficient implementations are using this representation (sometimes called signed BDDs). From the user point of view, this optimization is completely transparent. We can still reason in terms of Shannon's decomposition, except that we can exploit the fact that negation has a negligible cost.

6.7.9. *Care set and generalized cofactor*

6.7.9.1. *"Knowing that" operators*

When dealing with a logical function f, it is likely that the value of f has no importance outside some known subset of its argument. This *care set* can itself be described by its characteristic function g.

In order to optimize algorithms, we may want to exploit this information for simplifying the representation of function f. The idea is then to define some "knowing that" operator such that the result "h = f knowing that g" satisfies:

– h is equivalent to f when g is true, that is when $g \Rightarrow (h \Leftrightarrow f)$;

– h is "as simple as possible", in particular, if it appears that $g \Rightarrow f$, we expect to get $h = 1$, and, conversely, if $g \Rightarrow \neg f$, we expect to get $h = 0$.

Such an operator is obviously not unique; the first condition merely states that the expected result belongs to a logical interval: $(f \cdot g) \Rightarrow h \Rightarrow (f + \neg g)$. Finding within this interval the function whose BDD is minimal is far too expensive. We can merely propose operators based on heuristics, so that the complexity remains reasonable (similar to the one of classical binary operators).

6.7.9.2. *Generalized cofactor*

We present here a "knowing that" operator, denoted by $f \downarrow g$, and usually called the *constrain*, or "generalized cofactor":

– $\alpha \downarrow 0$ is undefined, $\quad \alpha \downarrow 1 = 1$,

– $\begin{array}{c} x \\ / \quad \backslash \\ \alpha_0 \quad \alpha_1 \end{array} \downarrow \begin{array}{c} x \\ / \quad \backslash \\ 0 \quad \beta \end{array} = \alpha_1 \downarrow \beta$ and $\begin{array}{c} x \\ / \quad \backslash \\ \alpha_0 \quad \alpha_1 \end{array} \downarrow \begin{array}{c} x \\ / \quad \backslash \\ \beta \quad 0 \end{array} = \alpha_0 \downarrow \beta$;

– otherwise $\begin{array}{c} x \\ / \quad \backslash \\ \alpha_0 \quad \alpha_1 \end{array} \downarrow \begin{array}{c} x \\ / \quad \backslash \\ \beta_0 \quad \beta_1 \end{array} = \begin{array}{c} x \\ /\!\!/ \quad \backslash\!\!\backslash \\ \alpha_0 \downarrow \beta_0 \quad \alpha_1 \downarrow \beta_1 \end{array}$;

– the other rules are the classical balancing rules (see section 6.7.6.2).

The name "generalized cofactor" comes from the fact that, for any function f and any non-zero function g we have: $f = g \cdot (f \downarrow g) + \neg g \cdot (f \downarrow \neg g)$. In particular, if g is a single variable x, we get the classical Shannon cofactors: $f = x \cdot (f \downarrow x) + \bar{x} \cdot (f \downarrow \bar{x})$.

6.7.9.3. *Restriction*

The generalized cofactor does not have particularly good properties in terms of minimization; it may appear that the result of $\alpha \downarrow \beta$ is bigger than the argument α. Moreover, in some cases, the result may contain variables that were not present in α (see Figure 6.14). In this example, the problem is due to the balancing rules that are introducing variable x in the result, while it was not present in the argument.

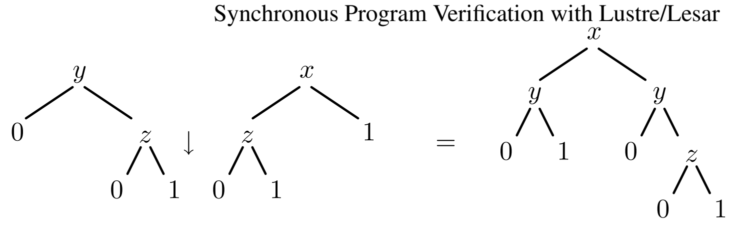


Figure 6.14. *With the ordering* $x \prec y \prec z$, $(y \cdot z) \downarrow (x + z) = y \cdot (\bar{x} + z)$

In order to avoid this problem, we define another operator, denoted by $\Downarrow$ and usually called the “restriction”. It only differs from the constraint in the way balancing rules are defined:

– if $x \prec y$ then $\quad \alpha \Downarrow \beta \;=\; \begin{tikzcd}[column sep=tiny] & x \arrow[dl, dash] \arrow[dr, dash] & \\ \alpha_0 \Downarrow \beta & & \alpha_1 \Downarrow \beta \end{tikzcd}$;

– if $y \prec x$ then $\quad \alpha \Downarrow \beta \;=\; \alpha \Downarrow (\exists y : \beta) \;=\; \alpha \Downarrow (\beta_0 + \beta_1)$.

The name restriction comes from the fact that the graph of the result $f \Downarrow g$ is always a sub-graph of that of f.

6.7.9.4. *Algebraic properties of the generalized cofactor*

Conversely to the restriction, the generalized cofactor does not satisfy simple properties in terms of minimization: $f \downarrow g$ is “often” simpler than f, but not always.

Nevertheless, the generalized cofactor satisfies interesting algebraic properties that restriction does not have:

– it left-distributes on negation (see demonstration in section 6.11):

$$(\neg\alpha) \downarrow \beta \;=\; \neg(\alpha \downarrow \beta) \tag{6.1}$$

– under existential quantification, it is equivalent to the conjunction (see demonstration in section 6.11):

$$(\exists\vec{x} : (\alpha \downarrow \beta)(\vec{x})) \;=\; (\exists\vec{x} : (\alpha \cdot \beta)(\vec{x})) \tag{6.2}$$

Being, in general, less costly than the conjunction, it is interesting to use the generalized cofactor whenever it is possible.

6.8. Forward symbolic exploration

The goal is still that of section 6.5: explore the whole state space of a finite automaton. However, the granularity has changed: instead of exploring each state one by one, we directly handle state subsets.

Operating on subsets is made possible by the use of BDDs. As explained before, BDDs make it possible to represent (in general concisely) the characteristic functions of the subsets.

6.8.1. *General scheme*

The algorithm is represented in Figure 6.15: at the end of the i^{th} iteration, the variable A holds the (characteristic function of) set A_i (states that can be reached from the initial ones in at most i transitions). If, during the iteration number k, A_k intersects Err, then the proof fails. Otherwise, the exploration goes on, slice by slice, and converges with the set Acc (states reachable in any number of transitions). Figure 6.16 illustrates the two cases: success (left) and failure (right).

```
$A$ := Init
repeat {
        if $A \cap$ Err $\neq \emptyset$ then Exit(failure)
        $A'$ := $A \cup POST_H(A)$
        if $A'$ = $A$ then Exit(success)
        else $A$ := $A'$
}
```

Figure 6.15. *Forward symbolic algorithm*

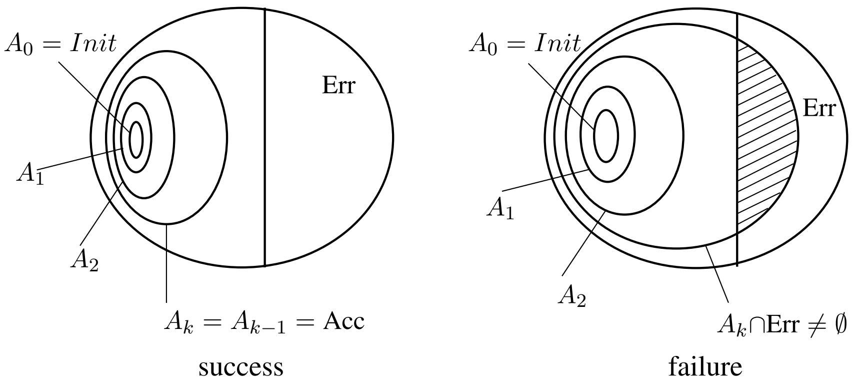


Figure 6.16. *Forward symbolic exploration*

6.8.2. *Detailed implementation*

Concretely, sets are identified by their characteristic functions, themselves implemented with BDDs: union is implemented by disjunction, intersection by conjunction, the empty set is implemented by the "always false" BDD, etc.

Complexity is mainly due to the computation of $\mathit{POST}_H(A)$, which takes as argument the BDD encoding the source states, and returns the BDD encoding the corresponding target states. The implementation of this function is presented in section 6.8.3, but it is already clear that its complexity mainly depends on the size of the argument (i.e. the number of nodes in the BDD encoding A).

From this point of view, the presented algorithm makes no optimization at all. States reachable in at most $n+1$ transitions (A_{n+1}) are computed as the *post* of all states reachable in at most n transitions:

$$A_{n+1} = A_n \cup \mathit{POST}_H(A_n)$$

In fact, only the computation of the states reachable in exactly $n+1$ is necessary: $\mathit{POST}_H(A_n \setminus A_{n-1})$. As a matter of fact, the other states ($\mathit{POST}_H(A_{n-1})$) already belong, by construction, to A_n.

However, even if $A_n \setminus A_{n-1}$ is a smaller set than A_n, it is not necessarily a good idea to compute $\mathit{POST}_H(A_n \setminus A_{n-1})$: one important characteristic of BDDs is that their size is not related to the number of elements they encode.

A good solution is then to use the restriction operator ($\Downarrow$, see section 6.7.9.1):

$$A_{n+1} = A_n \cup \mathit{POST}_H(A_n \Downarrow \neg A_{n-1})$$

This method guarantees that the argument of POST_H:

– contains at least $A_n \cap \neg A_{n-1}$;

– contains at most $A_n \cup A_{n-1} = A_n$;

– and its BDD is smaller than the ones of those two bounds.

Figure 6.17 shows the optimized version of the algorithm. At each iteration n, A holds the states reachable in at most n transitions and A_p the states reachable in at most $n-1$ transitions.

```
$A_p$ := $\emptyset$;   $A$ := Init
repeat {
        if $A \cap$ Err $\neq \emptyset$ then  Exit(failure)
        $A'$ := $A \cup \mathit{POST}_H(A \Downarrow A_p)$
        if $A'$ = $A$ then  Exit(success)
        $A_p$ := $A$;     $A$ := $A'$
}
```

Figure 6.17. *Optimized symbolic forward algorithm*

6.8.3. *Symbolic image computing*

The symbolic image computing, $\mathit{POST}_H(X)$, consists of building the characteristic function of the target states knowing the characteristic function of the source states. This computation is extremely costly; sophisticated, non-trivial algorithms are necessary to reach a (relative) efficiency. However, we present first a "naive" algorithm, in order to illustrate the feasibility of the computation. The necessary optimizations are presented in the second part.

The naive algorithm consists of strictly following the formal definition of *POST*. For that purpose, we build a "huge" formula involving:

– the source state variables $\vec{s} = (s_1, \ldots, s_n)$;

– the free variables $\vec{v} = (v_1, \ldots, v_m)$;

– the target state variables $\vec{s}\,' = (s'_1, \ldots, s'_n)$.

This formula is the conjunction of the following constraints:

– $\vec{s}$ is a source state, i.e. $X(\vec{s})$;

– the transition satisfies the hypothesis, i.e. $H(\vec{s}, \vec{v})$;

– each target state variable s'_i is the image of the corresponding transition function, i.e. $s'_i = g_i(\vec{s}, \vec{v})$ for all $i = 1 \cdots n$.

This formula is the *transition relation*, relating the source states from X with their target states.

Once this formula is built, it is necessary to quantify existentially both the inputs and the source state variables ($\vec{s}$ and $\vec{v}$), in order to obtain the expected formula, which only depends on the target state variables ($\vec{s}\,'$):

$$\mathit{POST}_H(X) \quad = \quad \exists \vec{s}, \vec{v} : \left(X(\vec{s}) \cdot H(\vec{s}, \vec{v}) \cdot \bigwedge_{i=1}^{n} (s'_i = g_i(\vec{s}, \vec{v})) \right)$$

This definition only uses "simple" Boolean operations, and thus it can be directly implemented with classical BDD operators.

6.8.4. *Optimized image computing*

6.8.4.1. *Principles*

The main default of the naive algorithm is that it builds a huge intermediate formula: state variables are appearing twice, in their source (s_i) and target (s'_i) versions.

In the worst case, the intermediate BDD has a size of order 2^{2n+m}, while the expected result is "only" of order 2^n. It may appear that the intermediate BDD can not be built, even if the result would have finally been of "reasonable" size.

The idea of optimized algorithm is mainly to avoid building the transition relation formula. As a matter of fact, the system is deterministic and thus, the transition relation is actually a function.

The algorithm we present here is due to Berthet, Madre and Coudert ([COU 89a, COU 89b, COU 90]).

6.8.4.2. *Universal image*

In order to simplify the notations, we denote $\vec{t} = (\vec{s}, \vec{v})$ the pairs of source state variables and input (free) variables. We also denote $Y = X \wedge H$ the conjunction of the source states and the hypothesis.

The expected result $\mathit{POST}_H(X)$ is the image of the states satisfying Y by the (vector of) transition functions $[g_1, \ldots, g_n]$, we note:

$$\mathit{POST}_H(X) = \mathcal{I}^Y_{[g_1,\ldots,g_n]} = \exists\vec{t}\colon Y(\vec{t}) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(\vec{t}).$$

The first step consists of integrating the source constraint Y within the transition functions. This integration is based on the algebraic properties of the generalized cofactor, from which the following theorem can be established (see demonstration in section 6.11):

$$\exists\vec{t}\colon Y(\vec{t}) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(\vec{t}) \Leftrightarrow \exists\vec{t}\colon \bigwedge_{i=1}^{n}(s'_i = (g_i \downarrow Y)(\vec{t})). \tag{6.3}$$

It follows that computing the image of set Y by a vector of transition functions is equivalent to computing the universal image of this vector of functions constrained by Y:

$$\mathcal{I}^Y_{[g_1,\ldots,g_n]} = \exists\vec{t}\colon \bigwedge_{i=1}^{n}(s'_i = (g_i \downarrow Y)(\vec{t})) = \mathcal{I}^1_{[g_1\downarrow Y,\ldots,g_n\downarrow Y]}.$$

From now on, when computing a universal image ($\mathcal{I}^1_{[f_1,\ldots,f_n]}$) we simply note $\mathcal{I}_{[f_1,\ldots,f_n]}$.

6.8.4.3. *Case of a single transition function*

In order to understand the algorithm, we first present how to compute the image of a single transition function.

The predicate $\mathcal{I}[f_n]$ depends on a single variable: the (target) state variable s'_n. It denotes the possible values of the result of f_n.

Shannon's decomposition, according to s'_n, gives:

$$\mathcal{I}_{[f_n]} = \exists \vec{t}\colon (s'_n = f_n(\vec{t})) = s'_n \cdot (\exists \vec{t}\colon\ f_n(\vec{t}))\ +\ \bar{s}'_n \cdot (\exists \vec{t}\colon\ \neg f_n(\vec{t})).$$

There are three possible cases:

1) f_n is identically true, thus $(\exists \vec{t}: f_n(\vec{t})) = 1$ and $(\exists \vec{t}: \neg f_n(\vec{t})) = 0$; it results that the target variable s'_n is eventually true:

$$\mathcal{I}_{[1]} = s'_n;$$

2) f_n is identically false, thus $(\exists \vec{t}: f_n(\vec{t})) = 0$ and $(\exists \vec{t}: \neg f_n(\vec{t})) = 1$; the target variable s'_n is eventually false:

$$\mathcal{I}_{[0]} = \bar{s}'_n;$$

3) otherwise, f_n can be either true or false depending on the quantified variables, thus $(\exists \vec{t}: f_n(\vec{t})) = (\exists \vec{t}: \neg f_n(\vec{t})) = 1$; the target variable s'_n can take any value:

$$\mathcal{I}_{[f_n]} = s'_n + \bar{s}'_n = 1\ .$$

6.8.4.4. *Shannon's decomposition of the image*

We consider now the case of several transition functions: $\mathcal{I}_{[f_k,\ldots,f_n]}$ with $k < n$. Shannon's decomposition according to the first target variable s'_k gives:

$$\mathcal{I}_{[f_k,f_{k+1},\ldots,f_n]} = \exists \vec{t}\colon \bigwedge_{i=k+1}^{n} s'_i = f_i(\vec{t}) = \exists \vec{t}\colon (s'_k = f_k(\vec{t})) \wedge \bigwedge_{i=k+1}^{n} s'_i = f_i(\vec{t})$$

$$= s'_k \cdot \left(\exists \vec{t}\ f_k(\vec{t}) \wedge \bigwedge_{i=k+1}^{n} s'_i = f_i(\vec{t})\right) + \bar{s}'_k \cdot \left(\exists \vec{t}\ \neg f_k(\vec{t}) \wedge \bigwedge_{i=k+1}^{n} s'_i = f_i(\vec{t})\right).$$

There are, once again, three cases:

1) f_k is identically true (and $\neg f_k$ is identically false):

$$\mathcal{I}_{[1,f_{k+1},\ldots,f_n]} = s'_k \cdot \left(\exists \vec{t}\ \bigwedge_{i=k+1}^{n} s'_i = f_i(\vec{t})\right) = s'_k \cdot \mathcal{I}_{[f_{k+1},\ldots,f_n]};$$

2) f_k is identically false (and $\neg f_k$ is identically true):

$$\mathcal{I}_{[0,f_{k+1},\ldots,f_n]} = \bar{s}'_k \cdot \left(\exists \vec{t}\ \bigwedge_{i=k+1}^{n} s'_i = f_i(\vec{t})\right) = \bar{s}'_k \cdot \mathcal{I}_{[f_{k+1},\ldots,f_n]};$$

3) otherwise, f_k can be either true or false, and we can recognize the definition of the image for the transitions vector $[f_{k+1}, \ldots, f_n]$ applied to f_k (right-hand side) and $\neg f_k$ (left-hand side):

$$\mathcal{I}_{[f_k, f_{k+1}, \ldots, f_n]} = s'_k \cdot \mathcal{I}^{f_k}_{[f_{k+1}, \ldots, f_n]} + \bar{s}'_k \cdot \mathcal{I}^{\neg f_k}_{[f_{k+1}, \ldots, f_n]}.$$

We use the properties of the $\downarrow$ operator in order to give an equivalent definition which uses only universal image computation. Moreover, we can note that the case of a single transition function fits the general case if we define that $\mathcal{I}_{[]} = 1$.

Finally, we obtain a purely recursive definition of the universal image computing. This definition leads to an algorithm that recursively builds the Shannon decomposition of the universal image, that is, its BDD. This algorithm (Figure 6.18) makes heavy use of the generalized cofactor.

- $\mathcal{I}_{[]} = 1$
- $\mathcal{I}_{[1, f_{k+1}, \ldots, f_n]} =$
- $\mathcal{I}_{[0, f_{k+1}, \ldots, f_n]} =$
- $\mathcal{I}_{[f_k, f_{k+1}, \ldots, f_n]} =$

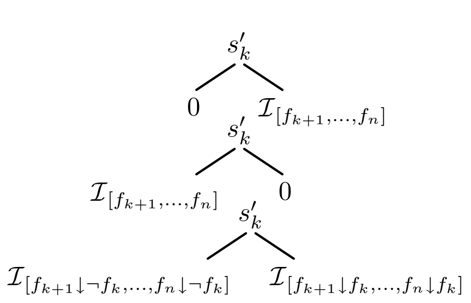

s'_k
0
$\mathcal{I}_{[f_{k+1}, \ldots, f_n]}$
s'_k
$\mathcal{I}_{[f_{k+1}, \ldots, f_n]}$
0
s'_k
$\mathcal{I}_{[f_{k+1}\downarrow\neg f_k, \ldots, f_n\downarrow\neg f_k]}$
$\mathcal{I}_{[f_{k+1}\downarrow f_k, \ldots, f_n\downarrow f_k]}$

Figure 6.18. *Recursive computation of the universal image*

6.9. Backward symbolic exploration

6.9.1. *General scheme*

The backward exploration algorithm is the dual of the forward algorithm (Figure 6.19): we simply have to swap the roles of Init and Err, and to replace the image computation by the "reverse-image" computation (*PRE$_H$* function).

When the algorithm succeeds, the variable B holds the set of states that can lead to an error, which is exactly the set "Bad" defined in section 6.5.4.

The complexity is mainly due to the reverse image computation (*PRE*), whose formal definition is:

$$\mathit{PRE}_H(X) = \exists \vec{v}, \vec{s}' : X(\vec{s}') \cdot H(\vec{s}, \vec{v}) \cdot \bigwedge_{i=1}^{n} s'_i = g_i(\vec{s}, \vec{v}).$$

```
$B$ := Err;  $B_p = \emptyset$
repeat {
        if $B \cap$ Init $\neq \emptyset$ then  Exit(failure)
        $B' := B \cup PRE_H(B \Downarrow \neg B_p)$
        if $B' = B$ then  Exit(success)
        else $B_p = B$;   $B := B'$
}
```

Figure 6.19. *Symbolic backward algorithm*

Just like for the *POST* computation, a naive implementation consists of applying this definition:

– build a (huge) BDD depending on source state variables ($\vec{s}$), inputs ($\vec{v}$) and target state variables ($\vec{s}\,'$);

– existentially quantify $\vec{s}\,'$ and $\vec{v}$ in order to obtain the expected BDD, depending only on $\vec{s}$.

6.9.2. *Reverse image computing*

Just like for the *POST*, it is possible to avoid the construction of the huge intermediate formula by using Shannon's decomposition of the result.

First, since the hypothesis H does not depend on the variables $\vec{s}\,'$, it is possible to keep it out of the corresponding quantifier:

$$PRE_H(X) = \exists \vec{v} : H(\vec{s}, \vec{v}) \cdot \mathcal{R}_{\vec{g}}(X) \ ,$$

where $\mathcal{R}_{\vec{g}}(X)$ is the reverse image of X for the transition functions $\vec{g}$, i.e. the pairs $(\vec{s}, \vec{v})$ leading to some state in X:

$$\mathcal{R}_{\vec{g}}(X) = \exists \vec{s}\,' : X(\vec{s}\,') \cdot \bigwedge_{i=1}^{n} s'_i = g_i(\vec{s}, \vec{v}).$$

By considering Shannon's decomposition of the argument X, we can easily show that:

$$\mathcal{R}_{g_1::\vec{g}\,'} \left(\begin{array}{ccc} & s'_1 & \\ & / \ \backslash & \\ X_0 & & X_1 \end{array} \right) = g_1 \cdot \mathcal{R}_{\vec{g}\,'}(X_1) + \neg g_1 \cdot \mathcal{R}_{\vec{g}\,'}(X_0).$$

The reverse image computation is then equivalent to a recursive composition of transition functions. In order to obtain the complete algorithm (Figure 6.20), we use the following properties:

– $\mathcal{R}_{\vec{g}}(0) = 0$ and $\mathcal{R}_{\vec{g}}(1) = 1$, since $\vec{g}$ is a total function;

– the computation can be optimized by the use of the restriction, which trivially satisfies $f \cdot g = f \cdot (g \Downarrow f)$ (see section 6.7.9.1).

– $\mathcal{R}^0_{\vec{g}} = 0$ and $\mathcal{R}^1_{\vec{g}} = 1$

$$- \mathcal{R}_{g_k::\vec{g}}\left(\begin{array}{c} s'_k \\ / \ \backslash \\ X_0 \quad X_1 \end{array}\right) = \begin{array}{c} g_k \\ / \qquad \backslash \\ \mathcal{R}_{\vec{g}\Downarrow\neg g_k}(X_0) \quad \mathcal{R}_{\vec{g}\Downarrow g_k}(X_1) \end{array}$$

Figure 6.20. *Recursive computation of the reverse-image*

6.9.3. *Comparing forward and backward methods*

It is quite difficult to compare the two methods since their efficiency dramatically depends on the "nature" of the explored model. We can simply state that, empirically, the forward symbolic method is "very often" more efficient. One possible explanation is that image computation is somehow intrinsically simpler than reverse-image computation. In reality, this explanation is not convincing, since both algorithms are clearly of the same order of complexity (over-exponential), and that the efficiency factor (if it even exists) is likely to be negligible.

In fact, experience tends to show that there exist (at least) two kinds of properties:

– Accessibility properties are those that strongly depend on the actual initial state(s), in the sense that slightly modifying the initial states may lead either to success or failure. For this kind of property, the forward method appears to be the more efficient.

– Inductive properties are those that do not (strongly) depend on the initial states. Intuitively, such a property Φ satisfies $\mathit{PRE}_H(\Phi) \Rightarrow \Phi$, or more generally $\bigwedge_{i=1}^{k} \mathit{PRE}^i_H(\Phi) \Rightarrow \Phi$, for some (small) constant k. In this case, the backward method is likely to converge much more quickly than the forward one.

Finally, the empirical "folk-theorem" stating that the forward method is better than the backward method seems to mainly reflect the fact that model checking is generally used to prove accessibility properties rather than inductive properties.

6.10. Conclusion and related works

Finite state systems verification is theoretically decidable, but practically limited by the combinational explosion of state space.

With symbolic methods based on BDDs, the intrinsic limitation due to the actual number of states can be overpassed: the complexity is no longer related to the number of variables, but rather to the relations between the variables. However, the use of BDDs remains extremely costly both in time and memory.

SAT decision methods (for satisfiability) consist of enumerating the solutions of a formula without storing them in memory. Compared to BDDs, such methods have an exponential cost in time, but only a polynomial cost in memory. Model checking algorithms can be adapted to use a decision engine based on SAT methods rather than BDDs. In this case, iterations are likely to be more efficient (at least in term of memory), but the counterpart is that checking the convergence becomes dramatically costly. This is why this kind of method is often used to check properties for bounded-time executions: *bounded model checking* [BIE 03].

6.11. Demonstrations

Lemma: cofactor main property

For a given order of the variables, and thus an operator $\downarrow$ perfectly defined, for any function $g : \mathbb{B}^n :\rightarrow \mathbb{B}$ not identically false, there exists a function $\pi_g : \mathbb{B}^n \rightarrow \mathbb{B}^n$ such that:

– for any $f : \mathbb{B}^n \rightarrow \mathbb{B}$ and any $\vec{x} \in \mathbb{B}^n$:

$$(f \downarrow g)(\vec{x}) = f(\pi_g(\vec{x})) \quad \text{i.e.} \quad f \downarrow g = f \circ \pi_g \tag{6.4}$$

– π_g sends any point from $\mathbb{B}^n$ into g:

$$\forall \vec{x} : g(\pi_g(\vec{x})) \quad \text{i.e.} \quad (\exists \vec{y} : \vec{x} = \pi_g(\vec{y})) \Leftrightarrow g(\vec{x}) \tag{6.5}$$

The recursive definition of π_g directly comes from $\downarrow$. Let $g = x \cdot g_1 + \bar{x} \cdot g_0$ be Shannon's decomposition of g according the least variable in the considered order:

– $\pi_g(\vec{x}) = \vec{x}$ if $g(\vec{x})$

– $\pi_g(0 :: \vec{x}) =$ if $(g_0 = 0)$ then $(1 :: \pi_{g_1}(\vec{x}))$ else $(0 :: \pi_{g_0}(\vec{x}))$

– $\pi_g(1 :: \vec{x}) =$ if $(g_1 = 0)$ then $(0 :: \pi_{g_0}(\vec{x}))$ else $(1 :: \pi_{g_1}(\vec{x}))$

Theorem 6.1

$((\neg f) \downarrow g)(\vec{x}) = (\neg f)(\pi_g(\vec{x}))$ [Lemma 6.4]

$= \neg f(\pi_g(\vec{x})) = \neg(f \downarrow g)(\vec{x})$

Theorem 6.2

$\exists \vec{x}\ (f \downarrow g)(\vec{x}) \Leftrightarrow \exists \vec{x}\ f(\pi_g(\vec{x}))$ [Lemma 6.4]

$\Leftrightarrow \exists \vec{x}\ \exists \vec{y}\ f(\vec{y}) \wedge (\vec{y} = \pi_g(\vec{x}))$

$$\Leftrightarrow \exists \vec{y}\, f(\vec{y}) \wedge (\exists \vec{x}\, (\vec{y} = \pi_g(\vec{x})))$$

$$\Leftrightarrow \exists \vec{y}\, f(\vec{y}) \wedge g(\vec{y}) \quad \text{[Lemma 6.5]}$$

$$\Leftrightarrow \exists \vec{y}\, (f \wedge g)(\vec{y})$$

Theorem 6.3

$$\exists \vec{t}\; Y(\vec{t}) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(\vec{t})$$

$$\Leftrightarrow \exists \vec{t}\; \bigwedge_{i=1}^{n} (Y(\vec{t}) \wedge (s'_i = g_i(\vec{t}))$$

$$\Leftrightarrow \exists \vec{t}\; \bigwedge_{i=1}^{n} ((s'_i \wedge g_i(\vec{t}) \wedge Y(\vec{t})) \vee (\neg s'_i \wedge \neg g_i(\vec{t}) \wedge Y(\vec{t})))$$

$$\Leftrightarrow \exists \vec{t}\; \bigwedge_{i=1}^{n} ((s'_i \wedge (g_i \wedge Y)(\vec{t})) \vee (\neg s'_i \wedge (\neg g_i \wedge Y)(\vec{t})))$$

$$\Leftrightarrow \exists \vec{t}\; \bigwedge_{i=1}^{n} ((s'_i \wedge (g_i \downarrow Y)(\vec{t})) \vee (\neg s'_i \wedge (\neg g_i \downarrow Y)(\vec{t}))) \quad \text{[Theorem 6.1]}$$

$$\Leftrightarrow \exists \vec{t}\; \bigwedge_{i=1}^{n} ((s'_i \wedge (g_i \downarrow Y)(\vec{t})) \vee (\neg s'_i \wedge \neg(g_i \downarrow Y)(\vec{t}))) \quad \text{[Theorem 6.2]}$$

$$\Leftrightarrow \exists \vec{t}\; \bigwedge_{i=1}^{n} (s'_i = (g_i \downarrow Y)(\vec{t}))$$

6.12. Bibliography

[AKE 78] AKERS S. B., "Binary decision diagrams", *IEEE Transactions on Computers*, vol. C-27, num. 6, 1978.

[AND 96] ANDRÉ C., "Representation and analysis of reactive behaviors: a synchronous approach", *IEEE-SMC'96, Computational Engineering in Systems Applications*, Lille, July 1996.

[BEN 91] BENVENISTE A., BERRY G., "The synchronous approach to reactive and real-time systems", *Proceedings of the IEEE*, vol. 79, num. 9, p. 1270–1282, September 1991.

[BER 92] BERRY G., GONTHIER G., "The ESTEREL synchronous programming language: design, semantics, implementation", *Science of Computer Programming*, vol. 19, num. 2, p. 87–152, 1992.

[BIE 03] BIERE A., CIMATTI A., CLARKE E., STRICHMAN O., ZHU Y., "Bounded model checking", *Advances in Computers*, vol. 58, Academic Press, 2003.

[BIL 87] BILLON J., "Perfect normal forms for discrete functions", Report num. 87019, BULL, March 1987.

[BIL 88] BILLON J., MADRE J., "Original concepts of PRIAM, an industrial tool for efficient formal verification of combinational circuits", MILNE G., Ed., *IFIP WG 10.2 Int Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, North Holland, July 1988.

[BRY 86] BRYANT R. E., "Graph-based algorithms for Boolean function manipulation", *IEEE Transactions on Computers*, vol. C-35, num. 8, p. 677–692, 1986.

[BUR 90] BURCH J., CLARKE E., MCMILLAN K., DILL D., HWANG J., "Symbolic model checking: 10^{20} states and beyond", *Fifth IEEE Symposium on Logic in Computer Science*, Philadelphia, p. 428–439, June 1990.

[COU 89a] COUDERT O., BERTHET C., MADRE J. C., "Verification of sequential machines using Boolean functional vectors", CLAESEN L. J. M., Ed., *Formal VLSI Correctness Verification*, North Holland, November 1989.

[COU 89b] COUDERT O., BERTHET C., MADRE J. C., "Verification of synchronous sequential machines based on symbolic execution", *International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, LNCS 407, Springer Verlag, 1989.

[COU 90] COUDERT O., MADRE J. C., BERTHET C., "Verifying temporal properties of sequential machines without building their state diagrams", KURSHAN R., Ed., *International Workshop on Computer Aided Verification*, Rutgers, New Jersey, June 1990.

[HAL 91] HALBWACHS N., CASPI P., RAYMOND P., PILAUD D., "The synchronous dataflow programming language LUSTRE", *Proceedings of the IEEE*, vol. 79, num. 9, p. 1305–1320, September 1991.

[HAL 93] HALBWACHS N., *Synchronous programming of reactive systems*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993.

[LEG 91] LE GUERNIC P., GAUTIER T., LE BORGNE M., LE MAIRE C., "Programming real-time applications with SIGNAL", *Proceedings of the IEEE*, vol. 79, num. 9, p. 1321–1336, September 1991.

[MAN 90] MANNA Z., PNUELI A., "A Hierarchy of Temporal Properties.", *PODC*, p. 377–410, 1990.

[RUD 93] RUDELL R., "Dynamic variable ordering for ordered binary decision diagrams", *ICCAD '93: Proceedings of the 1993 IEEE/ACM International conference on Computer-aided design*, Los Alamitos, CA, USA, IEEE Computer Society Press, p. 42–47, 1993.

[TOU 90] TOUATI H., SAVOJ H., LIN B., BRAYTON R. K., SANGIOVANNI-VINCENTELLI A., "Implicit state enumeration of finite state machines using BDDs", *International Conference on Computer Aided Design (ICCAD)*, November 1990.

Chapter 7

Synchronous Functional Programming with Lucid Synchrone

7.1. Introduction

LUCID SYNCHRONE is a programming language dedicated to the design of reactive systems. It is based on the synchronous model of LUSTRE [HAL 91a] which it extends with features usually found in functional languages such as higher-order or constructed data-types. The language is equipped with several static analysis, all expressed as special type-systems and used to ensure the absence of certain run-time errors on the final application. It provides, in particular, automatic type and clock inference and statically detects initialization issues or dead-locks. Finally, the language offers both data-flow and automata-based programming inside a unique framework.

This chapter is a tutorial introduction to the language. Its aim is to show that techniques can be applied, at the language level, to ease the specification, implementation, and verification of reactive systems.

7.1.1. *Programming reactive systems*

We are interested in programming languages dedicated to the design of reactive systems [HAR 85]. Such systems interact continuously with their external environment and have to meet strong temporal constraints: emergency braking systems, plane autopilots, electronic stopwatches, etc. In the mid-1970s, with the shift from mechanical and electronic systems to logical systems, the question of the computer-based

Chapter written by Paul CASPI, Grégoire HAMON and Marc POUZET.

implementation of reactive systems arose. At first, these implementations used available general-purpose programming languages, such as assembly, C, or ADA. Low-level languages were prominent as they give strong and precise control over the resources. Reactive systems often run with very limited memory and their reaction time needs to be statically known.

General-purpose, low-level languages quickly showed their limits. The low-level description they offer is very far from the specification of the systems, making implementation error prone and certification activities very difficult. A large number of embedded applications are targeting critical functions – fly-by-wire systems, control systems in nuclear power plants, etc. As such, they have to follow constraints imposed by certification authorities that evaluate the quality of a design before production use, for example the standards DO-178B in avionics and IEC-61508 in transport.

Better adapted languages were needed. The first idea was to turn to concurrent programming model with the goal of getting closer to the specification. Indeed, reactive systems are inherently concurrent: the system and its environment are evolving in parallel and concurrency is the natural way to compose systems from more elementary ones. Unfortunately, traditional models of concurrency come with numerous problems of system analysis, non-determinism, or compilation. These models have therefore been seldom used in domains where security is a concern. Moreover, even if the goal is to get closer to a specification, the obtained models are far from formalisms traditionally used by engineers (such as continuous control and finite state transition systems).

7.1.1.1. *The synchronous languages*

Critical industries (e.g., avionics, energy), driven by high security needs, were the firsts to actively research new development techniques. An answer was proposed at the beginning of the 1980s with the introduction of *synchronous languages* [BEN 03], the most famous of which being ESTEREL [BER 92], LUSTRE [HAL 91a] and SIGNAL [BEN 91]. These languages, based on a mathematical model of concurrency and time, are designed exclusively to describe reactive systems. The idea of dedicated programming languages with a limited expressive power, but perfectly adapted to their domain of application, is essential. The gain obtained lies in program analysis and the ability to offer compile-time guarantees on the run-time behavior of the program. For example, real-time execution and the absence of deadlocks are guaranteed by construction. Synchronous languages have been successfully applied in several industries and have helped in bringing formal methods to industrial developments.

Synchronous languages are based on the *synchronous hypothesis*. This hypothesis comes from the idea of separating the functional description of a system from the constraints of the architecture on which it will be executed. The functionality of the system can be described by making a hypothesis of instantaneous computations

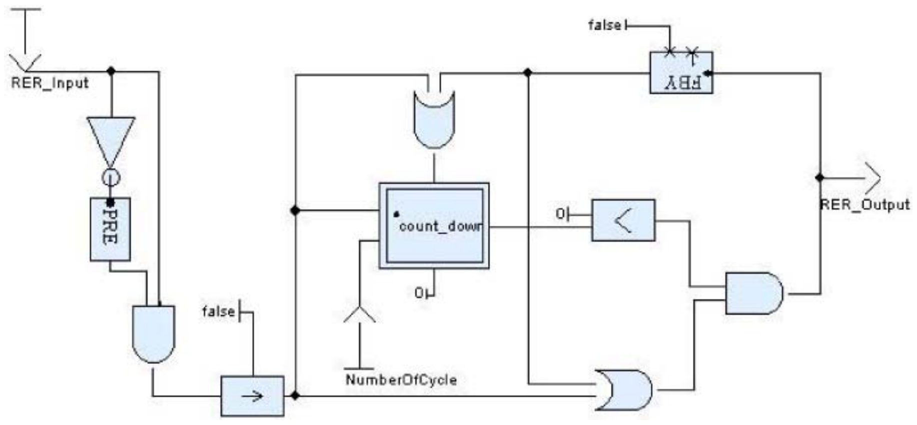


Figure 7.1. *A* SCADE *design*

and communications, as long as it can be verified afterward that the hardware is fast enough for the constraints imposed by the environment. This is the classical *zero delay* model of electronics or control theory. It allows for reasoning in logical time in the specification, without considering the real time taken by computations. Under the synchronous hypothesis, non-determinism associated with concurrency disappears, as the machine is supposed to have infinite resources. It is possible to design high-level programming languages that feature parallel constructions, but can be compiled to efficient sequential code. The correspondence between logical time and physical time is verified by computing the worst case reaction time of the programs.

LUSTRE is based on the idea that the restriction of a functional language on streams, similar to LUCID [ASH 85] and the Kahn process networks [KAH 74], would be well adapted to the description of reactive systems[1]. LUSTRE is based on a data-flow programming model. A system is described as a set of equations over sequences. These sequences, or flows, represent the communications between the components of the system. This programming model is close to the specifications used by engineers of critical software thus reducing the distance between specification and implementation. In LUSTRE, the specification is the implementation. Moreover, the language is very well adapted to the application of efficient techniques for verification (see Chapter 6), test [RAY 98] and compilation [HAL 91b]. The adequacy of the language with the domain and the efficiency of the technique that can be applied on it prompted the industrial success of LUSTRE and of its industrial version SCADE [SCA 07]. SCADE

1. LUSTRE is the contraction of *Lucid, Synchrone et Temps Réel* – Lucid, Synchronous and Real-Time.

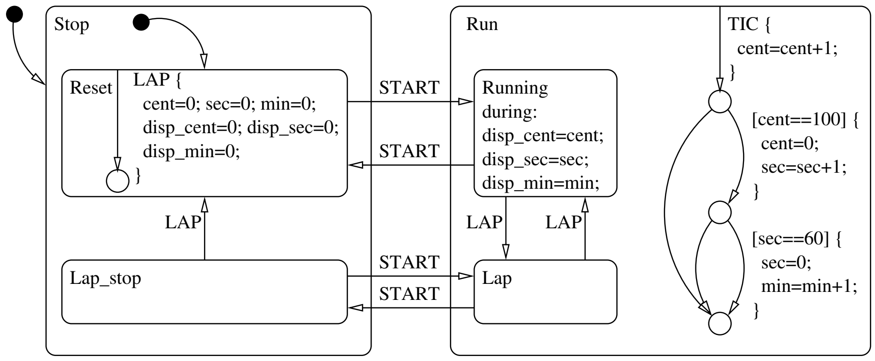


Figure 7.2. *A Chronometer in* STATEFLOW

offers a graphical modeling of the LUSTRE equations, close to the diagrams used by control or circuit designers (Figure 7.1). SCADE comes with formal validation tools and its code generator is certified (DO-178B, level A), reducing the need for testing. SCADE is now used by numerous critical industries. It is, for example, a key element in the development of the new Airbus A380.

7.1.1.2. *Model-based design*

Meanwhile, non-critical industries continued using general purpose languages, or low-level dedicated languages (norm IEC-1131). Security needs were much lower and did not justify the cost of changing development methods. Because of the increasing size and complexity of systems and the need to reduce development cycles, they finally looked for other methods. These concerns have led to widespread interest in *model-based development environments*, the growing use of SIMULINK/STATEFLOW and the introduction of UML. Integrated environments offer a description of systems using high-level graphical formalisms close to automata or data-flow diagrams. They provide extensive sets of tools for simulation, test, or code-generation. Again, the distance between specification and implementation has been reduced, the system being automatically generated from the model.

These tools have usually not been designed with formal development in mind. Their semantics are often partially and informally specified. This is a serious problem for verification or certification activities. It is also a growing problem for the development activity itself: as the complexity of the systems increases, the risk of misinterpreting the models increases accordingly. An example of such an environment is SIMULINK/STATEFLOW [MAT 03] (Figure 7.2), an industrial *de-facto* standard. SIMULINK/STATEFLOW does not have formal semantics and it is possible to write programs that will fail at run-time. On the other hand, the environment offers a complete set of tools, in which both the model and its environment can be modeled, simulated, and code can be automatically generated.

7.1.1.3. *Converging needs*

The needs of all industries dealing with reactive systems, critical or not, are converging. They are all facing problems due to the increasing size and complexity of the systems. Synchronous languages, created to answer the needs of specific domains (for example, critical software for LUSTRE), are not directly adapted to the description of complex systems mixing several domains. The needs for verification and system analysis, for a time specific to critical industries, are now everywhere. The size of the systems requires automation of verification or test generation activities, themselves required by the global diffusion of systems and the costs induced by conception errors that go unnoticed. These converging needs require in particular:

– abstraction mechanism at the language level that makes it possible to get even closer to the specification of the systems, better reusability of components and the creation of libraries;

– techniques of analysis that are automatic and modular, and techniques of code generation that can offer guarantees on the absence of run-time errors;

– a programming model in which sampled continuous models, such as LUSTRE models, and control-based description, such as automata, can be arbitrarily composed.

7.1.2. ***Lucid Synchrone***

The LUCID SYNCHRONE programming language was created to study and experiment extensions of the synchronous language LUSTRE with higher-level mechanisms such as type and clock inference, higher-order or imperative control-structures. It started from the observation that there was a close relationship between three fields, synchronous data-flow programming, the semantics of data-flow networks [KAH 74] and compilation techniques for lazy functional languages [WAD 90]. By studying the links between those fields, we were interested in answering two kinds of questions: (1) theoretical questions, on the semantics, the static analysis and the compilation of synchronous data-flow languages; (2) practical questions, concerning the expressiveness of LUSTRE and the possibilities of extending it to answer needs of SCADE users.

This chapter presents the actual development of the language and illustrates its main features through a collection of examples. This presentation is voluntarily made informal and technical details about the semantics of the language can be found in related references. Our purpose is to convince the reader of the various extensions that can be added to existing synchronous languages while keeping their strong properties for programming real-time systems. An historical perspective on the development of the language and how some of its features have been integrated into the industrial tool SCADE shall be discussed in section 7.3.

The core LUCID SYNCHRONE language is based on a synchronous data-flow model in the spirit of LUSTRE and embedded into a functional notation using *ML*. The language provides the main operations of LUSTRE with the ability to write, up to syntactic details, any valid LUSTRE program. It also provides other operators such as the `merge` that is essential to combine components evolving on different rates. These features are illustrated in sections 7.2.1–7.2.4. The language incorporates several features usually found in functional languages such as higher-order or data-types and pattern-matching. Higher-order, which means that a function can be parametrized by another function, makes it possible to define generic combinators that can be used as libraries. Combined with the compiler's support for separate compilation, this is a key to increase modularity. Higher-order is illustrated in section 7.2.5. To answer the growing need for abstraction, the language supports constructed data-types. Values of that type naturally introduce control behavior and we associate them with a pattern-matching operation, thus generalizing the *activation condition* construction (or *enable-subsystems*) in graphical tools such as SCADE or SIMULINK. Data-types and pattern-matching are presented in section 7.2.6. Concurrent processes need to communicate and, in a dataflow language, this can only be done through inputs and outputs (so-called *in-ports* and *out-ports* in graphical languages), which can be cumbersome. LUCID SYNCHRONE introduces the notion of a *shared memory* to ease the communication between several modes of executions. This mechanism is safe in the sense that the compiler statically checks non-interference between the uses of the shared memories, rejecting, in particular programs with concurrent writes. The language also supports *signals* as found in ESTEREL and which simplify the writing of control-dominated systems. Shared memory and signals are presented in sections 7.2.7 and 7.2.8. Maybe the most unique feature of LUCID SYNCHRONE is its direct support of hierarchical state machines. This opens up new ways of programming, as it allows the programmer to combine in any way s/he wants dataflow equations and finite state-machine. State machines are illustrated in sections 7.2.9 and 7.2.11. Finally, being a functional language, it was interesting to investigate more general features such as recursion. This part is discussed in section 7.2.12.

Besides these language extensions, the language features several static analysis in order to contribute to make programming both easier and safer. In LUCID SYNCHRONE, types are automatically inferred and do not have to be given explicitly given by the programmer. Automatic type inference and parametric polymorphism have been originally introduced in ML languages and have proved their interest for general purpose programming [PIE 02]. Nonetheless, they were not provided, as such, in existing synchronous tools. Type polymorphism makes it possible to define generic components and increase code reuse while type inference frees the programmer from declaring the type of variables. The language is also founded on the notion of clocks as a way to specify the various paces in a synchronous system. In LUCID SYNCHRONE, clocks are types and the *clock calculus* of synchronous languages which aims at checking the coherency between those paces is defined as a type inference problem. Finally, the compiler provides two other static analysis also defined by special type systems. The initialization analysis checks that the behavior of the system does not depend on the initial values of delays while the causality analysis detects deadlocks.

7.2. Lucid Synchrone

We cannot present the language without a word about its two parents OBJECTIVE CAML [LER 07] and LUSTRE. The functional programmer may consider LUCID SYNCHRONE as a subset of OCAML managing synchronous streams. The synchronous programmer will instead consider it as an extension of LUSTRE with functional features and control structures. The language has largely evolved since its first implementation in 1995. We present here the current version (V3)[2].

7.2.1. *An ML dataflow language*

7.2.1.1. *Infinite streams as basic objects*

LUCID SYNCHRONE is a *dataflow* language. It manages infinite sequences of streams as primitive values. Variable x stands for the infinite sequence of values $x_0, x_1, x_2, \ldots$ that x receives during the execution. In the same way, `1` denotes the infinite sequence $1, 1, 1, \ldots$. The type `int` denotes the type of integer sequences. Scalar functions (e.g., `+`, `*`) apply point-wisely to streams:

`c`	t	f	$\cdots$
`x`	x_0	x_1	$\cdots$
`y`	y_0	y_1	$\cdots$
`x+y`	$x_0 + y_0$	$x_1 + y_1$	$\cdots$
`if c then x else y`	x_0	y_1	$\cdots$
`if c then (x,y) else (0,0)`	(x_0, y_0)	$(0, 0)$	$\cdots$

`x` and `y` being two integer streams, `x+y` produces a stream obtained by adding pointwisely the current values of `x` and `y`. Synchrony finds a natural characterization here: at instant i, all the streams take their i-th value.

Tuples of streams can be built and are considered as streams.

7.2.1.2. *Temporal operations: delay and initialization*

The initialized delay `fby` (or *followed by*) comes from LUCID [ASH 85], the ancestor of dataflow languages. This operation takes a first argument that gives the initial value for the result and a second argument that is the delayed stream. In the following example, `x fby y` returns the stream `y` delayed and initialized with the first value of `x`.

It is often useful to separate the delay from the initialization. This is obtained by using the uninitialized delay `pre` (for *previous*) and the initialization operator

2. The manual and distributions are available at `www.lri.fr/~pouzet/lucid-synchrone`.

`->`. `pre x` delays its argument `x` and has an unspecified value *nil* at the first instant. `x -> y` returns the first value of `x` at its first instant then the current value of `y`. The expression `x -> pre y` is equivalent to `x fby y`.

`x`	x_0 x_1 x_2 $\cdots$
`y`	y_0 y_1 y_2 $\cdots$
`x fby y`	x_0 y_0 y_1 $\cdots$
`pre x`	nil x_0 x_1 $\cdots$
`x -> y`	x_0 y_1 y_2 $\cdots$

Because the `pre` operator may introduce undefined values (represented by nil), it is important to check that the program behavior does not depend on these values. This is done statically by the *initialization analysis*.

7.2.2. *Stream functions*

The language makes a distinction between two kinds of functions: combinatorial and sequential. A function is combinatorial if its output at the current instant only depends on the current value of its input. This is a stateless function. A sufficient condition for an expression to be combinatorial is that it does not contain any delay, initialization operator or automaton. This sufficient condition is easily checked during typing.

A one-bit adder is a typical combinatorial function. It takes three Boolean inputs `a`, `b` and a carry `c` and returns a result `s` and a new carry `co`.

```
let xor (a, b) = (a & not(b)) or (not(a) & b)

let full_add (a, b, c) = (s, co) where
       s = xor (xor (a, b), c)
   and co = (a & b) or (b & c) or (a & c)
```

When this program is given to the compiler, we get:

```
val xor :  bool * bool -> bool
val xor :: 'a * 'a -> 'a
val full_add :  bool * bool * bool -> bool * bool
val full_add :: 'a * 'a * 'a -> 'a * 'a
```

For every declaration, the compiler infers types (`:`) and clocks (`::`). The type signature `bool * bool -> bool` states that `xor` is a combinatorial function that returns a Boolean stream when receiving a pair of Boolean streams. The clock signature `'a * 'a -> 'a` states that `xor` is a length preserving function: it returns a value every time it receives an input. Clocks will be explained later.

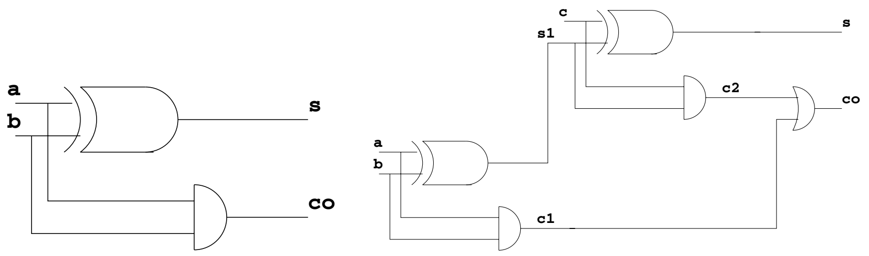


Figure 7.3. *Hierarchical definition of a* 1*-bit adder*

A more efficient adder can be defined as the parallel composition of two half-adders as illustrated in Figure 7.3.

```
let half_add (a,b) = (s, co)
  where s = xor (a, b) and co = a & b

let full_add(a,b,c) = (s, co) where
  rec (s1, c1) = half_add(a,b)
  and (s, c2) = half_add(c, s1)
  and co = c2 or c1
```

Sequential (or state-full) functions are functions whose output at time n may depend on their inputs' history. Sequential functions are introduced with the keyword `node` and they receive a different type signature. A front edge detector `edge` is written:

```
let node edge c = false -> c & not (pre c)

val edge :  bool => bool
val edge :: 'a -> 'a
```

A possible execution is given in the following diagram.

`c`	f f t t f t $\cdots$
`false`	f f f f f f $\cdots$
`c & not (pre c)`	nil f t f f t $\cdots$
`edge c`	f f t f f t $\cdots$

The type signature `bool => bool` states that `edge` is a function from Boolean streams to Boolean streams and that its result depends on the history of its input. The class of a function is verified during typing. For example, forgetting the keyword `node` leads to a type error.

```
let edge c = false -> c & not (pre c)

>let edge c = false -> c & not (pre c)
>              ^^^^^^^^^^^^^^^^^^^^^^^^
This expression should be combinatorial.
```

In a dataflow language, stream definitions can be mutually recursive and given in any order. This is the natural way to describe a system as the parallel composition of subsystems. For example, the function that decrements a counter according to an Boolean stream can be written:

```
let node count d t = ok where
  rec ok = (cpt = 0)
  and cpt = 0 -> (if t then pre cpt + 1 else pre cpt) mod d

val count : int -> int => bool
val count :: 'a -> 'a -> 'a
```

`count d t` returns the Boolean value true when `d` occurrences of `t` have been received.

d	3 2 2 2 2 4 ···
t	t t t t t t ···
cpt	3 2 1 2 1 4 ···
ok	t f f t f t ···

LUCID SYNCHRONE being a functional language, it is possible to write partial applications by fixing one parameter of a function.

```
let count10 t = count 10 t

val count10 :  int => int
val count10 :: 'a -> 'a
```

7.2.3. *Multi-sampled systems*

In all our previous examples, all the parallel processes share the same rate. At any instant n, all the streams take their n-th value. These systems are said to be single-clocked, a synchronous circuit being a typical example of a single-clocked system.

We now consider systems that evolve at different rates. We use a clock mechanism that was first introduced in LUSTRE and SIGNAL. Every stream s is paired with a Boolean sequence c, or *clock*, that defines the instants when the value of s is present (precisely c is true if and only if s is present). Some operations make it possible to produce a stream with a slower (sampling) or faster (oversampling) rate. To be valid, programs have to verify a set of clock constraints and this is done by the *clock calculus*. In LUCID SYNCHRONE, clocks are represented by types and the clock calculus is expressed as a type-inference system.

7.2.3.1. *The sampling operator* `when`

The operator `when` is a sampler making it possible to communicate values from fast processes to slower ones by extracting sub-streams according to a Boolean condition.

`c`	f	t	f	t	f	t	$\cdots$
`x`	x_0	x_1	x_2	x_3	x_4	x_5	$\cdots$
`x when c`		x_1		x_3		x_5	$\cdots$
`x whenot c`	x_0		x_2		x_4		$\cdots$

Stream `x when c` is slower than stream `x`. Assuming that `x` and `c` are produced at a clock ck, we say that `x when c` is produced at the clock ck **on** c.

The function `sum` that computes the sum of its input (i.e., $s_n = \Sigma_{i=0}^{n} x_i$) can be used at a slower rate by sampling its input:

```
let node sum x = s where rec s = x -> pre s + x
let node sampled_sum x c = sum (x when c)

val sampled_sum :  int -> bool => int
val sampled_sum :: 'a -> (_c0:'a) -> 'a on _c0
```

Whereas the type signature abstracts the value of a stream, the clock signature abstracts its temporal behavior: it characterizes the instants where the value of a stream is available. `sampled_sum` has a functional clock signature stating that for any clock `'a` and Boolean stream `_c0`, if the first argument `x` has clock `'a`, and the second argument is equal to `_c0` and has clock `'a`, then the result will receive the clock `'a on _c0` (variable `c` is renamed to avoid name conflicts). An expression with clock `'a on _c0` is present when both its clock `'a` is true and the Boolean stream `_c0` is present and true. Thus, an expression with clock `'a on _c0` has a slower rate than an expression with clock `'a`. Coming back to our example, the output of `sum (x when c)` is present only when `c` is true.

This sampled sum can be instantiated with a particular clock. For example:

```
let clock ten = count 10 true
let node sum_ten dx = sampled_sum dx ten

val ten :  bool
val ten :: 'a
val sum_ten :  int => int
val sum_ten :: 'a -> 'a on ten
```

The keyword `clock` introduces a clock name (here `ten`) from a Boolean stream. This clock name can in turn be used to sample a stream.

Clocks express control properties in dataflow systems. Filtering the inputs of a node according to a Boolean condition, for example, means that the node will be executed only when the condition is true.

`c`	f	t	f	t	f	f	t	$\cdots$
`1`	1	1	1	1	1	1	1	$\cdots$
`sum 1`	1	2	3	4	5	6	7	$\cdots$
`(sum 1) when c`		2		4			6	$\cdots$
`1 when c`		1		1			1	$\cdots$
`sum (1 when c)`		1		2			3	$\cdots$

Thus, sampling the input of a sequential function f is not equivalent, in general, to sampling its output, that is, $(f(x \texttt{ when } c) \neq (fx) \texttt{ when } c)$.

Clocks can be nested arbitrarily. For example, a watch can be written:

```
let clock sixty = count 60 true
let node hour_minute_second second =
  let minute = second when sixty in
  let hour = minute when sixty in
  hour,minute,second

val hour_minute_second ::
     'a -> 'a on sixty on sixty * 'a on sixty * 'a
```

A stream with clock `'a on sixty on sixty` is present only one instant over $3,600$, which is the expected behavior.

7.2.3.2. *The combination operator* `merge`

The operator `merge` allows slow processes to communicate with faster ones by combining two streams according to a condition. Two arguments of a `merge` must have a complementary clock.

`c`	f	t	f	f	f	t	$\cdots$
`x`		x_0				x_1	$\cdots$
`y`	y_0		y_1	y_2	y_3		$\cdots$
`merge c x y`	y_0	x_0	y_1	y_2	y_3	x_1	$\cdots$

Using the `merge` operator, we can define a holder (the operator `current` of LUSTRE) which holds the value of a stream between successive sampling. Here, `ydef` is a default value used when no value has yet been received:

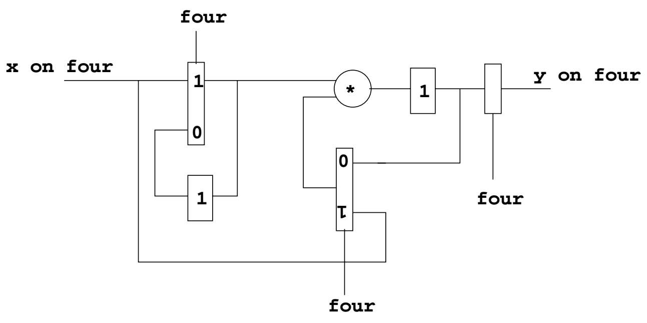


Figure 7.4. *A example of oversampling*

```
let node hold ydef c x = y
    where rec y = merge c x ((ydef fby y) whenot c)

val hold :  'a -> bool -> 'a => 'a
val hold :: 'a -> (_c0:'a) -> 'a on _c0 -> 'a
```

7.2.3.3. *Oversampling*

Using these two operators, we can define oversampling functions, that is, functions whose internal rate is faster than the rate of their inputs. In this sense, the language is strictly more expressive than LUSTRE and can be compared to SIGNAL.

Oversampling appears naturally when considering a long duration task made during several time steps. Consider the computation of the sequence $y_n = (x_n)^5$. This sequence can be computed by writing:

```
let power x = x * x * x * x * x
val power :: 'a -> 'a
```

The output is computed at the same rate as the input (as stated by the signature `'a -> 'a`). Four multiplications are necessary at every cycle. Suppose that only one multiplication is feasible at every instant. We can replace this instantaneous computation by an iteration through time by slowing the clock of `x` by a factor of four.

```
let clock four = count 4 true

let node iterate cond x = y where
  rec i = merge cond x ((1 fby i) whenot cond)
  and o = 1 fby (i * merge cond x (o whenot cond))
  and y = o when cond
```

```
let node spower x = iterate four x

val iterate :  bool -> int => int
val iterate :: (_c0:'a) -> 'a on _c0 -> 'a on _c0

val spower :  int => int
val spower :: 'a on four -> 'a on four
```

The corresponding dataflow network is given in Figure 7.4.

four	t	f	f	f	t	f	f	f	t
x	x_0				x_1				x_2
i	x_0	x_0	x_0	x_0	x_1	x_1	x_1	x_1	x_2
o	1	x_0^2	x_0^3	x_0^4	x_0^5	x_1^2	x_1^3	x_1^4	x_1^5
spower x	1				x_0^5				x_1^5
power x	x_0^5				x_1^5				x_2^5

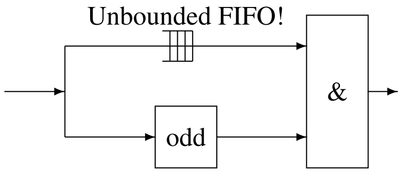
Since the function `power` has the polymorphic clock signature `'a -> 'a`, the sequence `power x` has the same clock as `x`. Thus, `spower x` produces the same sequence as `1 fby (power x)` but with a slower rate.

An important consequence of the use of clocks and automatic inference is the possibility of replacing every use of `(1 fby power x)` by `spower x` without modifying the rest of the program. The global system is automatically slowed-down to adapt to the clock constraints imposed by the function `spower`. This property contributes to the modular design and code reuse. Nonetheless, because clocks in LUCID SYNCHRONE can be built from any Boolean expression, the compiler is unable to infer quantitative information (which would make it possible to prove the equivalence between `(1 fby power x)` and `spower x`). `four` is considered by the compiler as a symbolic name and its periodic aspect is hidden. Recent works have considered an extension of synchronous language with periodic clocks [CAS 05, COH 06].

7.2.3.4. *Clock constraints and synchrony*

When stream functions are composed together, they must verify clock constraints. For example, the arguments of a `merge` must have complementary clocks and the arguments of an operator applied point-wisely must have the same clock. Consider for example the program:

```
let clock half = count 2 true
let node odd x = x when half
let node wrong x = x & (odd x)
```



x	x_0	x_1	x_2	x_3	x_4	x_5
half	t	f	t	f	t	f
x when half	x_0		x_2		x_4	
x & (odd x)	$x_0 \& x_0$		$x_1 \& x_2$		$x_2 \& x_4$	

Figure 7.5. *A non-synchronous program*

This program adds stream x to the sub-stream of x obtained by filtering one input over two[3]. This function would compute the sequence $(x_n \& x_{2n})_{n \in \mathbb{N}}$. Graphically, the computation of `x & (odd x)` corresponds to the Kahn network [KAH 74] depicted in Figure 7.5. In this network, input x is duplicated, one going through the function odd whose role is to discard one input over two. The two stream are in turn given to an & gate. If no value is lost, this network cannot be executed without a buffering mechanism. As time goes on, the size of the buffer will grow and will finally overflow. This program can not be efficiently compiled and boundedness execution can not be guaranteed. This is why we reject it statically. The goal of the clock calculus is to reject these types of programs that cannot be executed synchronously without any buffering mechanism [CAS 92].

In LUCID SYNCHRONE, clocks are types and the clock calculus is expressed as a type inference problem [CAS 96, COL 03]. When the previous program is given to the compiler, it displays:

```
> let node wrong x = x & (odd x)
>                      ^^^^^^^
This expression has clock 'b on half,
but is used with clock 'b.
```

Using the formal definition of the clock calculus, we can show the following correction theorem: every well clocked program can be executed synchronously [CAS 96, CAS 98]. This result can be compared to the type preservation theorem in ML languages [PIE 02]. Clocks being types, they can be used to abstract the temporal behavior of a system and be used as programming interfaces. Moreover, they play a fundamental role during the compilation process to generate efficient code. Intuitively, clocks become control-structures in the sequential code and the compiler gathers, as much as possible, all computations under the same clock (as soon as data-dependencies are preserved).

3. In control-theory, `x when half` is a half frequency sampler.

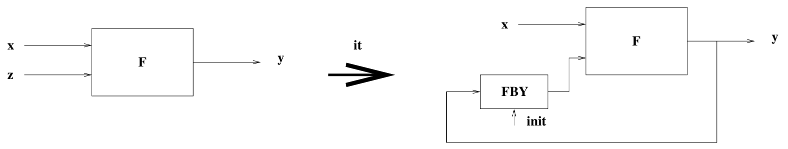


Figure 7.6. *An iterator*

7.2.4. *Static values*

Static values are constant values that are useful to define parametrized system, these parameters being fixed at the beginning of an execution.

```
let static m = 100.0
let static g = 9.81
let static mg = m *. g
```

The compiler checks that a variable which is declared to be static is bound to an expression that produces a constant stream.

The language considered so far is not that different from LUSTRE. The main differences are the automatic type and clock inference and the operation `merge` which is an essential operation in order to program multi-sampled systems. We now introduce several extensions which are specific to the language.

7.2.5. *Higher-order features*

Being a functional language, functions are first class objects which can be passed to functions or returned by functions. For example, the function `iter` is a serial iterator. Its graphical representation is given in Figure 7.6.

```
let node iter init f x = y where
  rec y = f x (init fby y)

val iter : 'a -> ('b -> 'a -> 'a) => 'a
val iter :: 'a -> ('b -> 'a -> 'a) -> 'a
```

such that:

```
let node sum x = iter 0 (+) x
let node mult x = iter 1 ( * ) x
```

Higher-order is compatible with the use of clocks and we can thus write a more general version of the serial iterator defined in section 7.2.3.3.

```
let node iterate cond x init f = y where
  rec i = merge cond x ((init fby i) whenot cond)
  and o = init fby (f i (merge cond x (o whenot cond)))
  and y = o when cond
```

```
let node spower x = iterate four x 1 ( * )

val iterate :  bool -> 'a -> 'a -> ('a -> 'a) => int
val iterate ::
  (_c0:'a) -> 'a on _c0 -> 'a -> ('a -> 'a -> 'a ) -> 'a on _c0

val spower :  int => int
val spower :: 'a on four -> 'a on four
```

Higher-order is a natural way to define new primitives from basic ones. For example, the “activation condition” is a primitive operator in graphical tools such as SCADE/LUSTRE or SIMULINK. It takes a condition `c`, a function `f`, a default value `default`, an input `input` and computes `f(input when c)`. It holds the last computed value when `c` is false.

```
let node condact c f default input = o where
  rec o = merge c (run f (input when c))
                  ((default fby o) whenot c)

val condact :  bool -> ('a => 'b) -> 'b -> 'a -> 'b
val condact :: (_c0:'a) -> ('a on _c0 -> 'a on _c0) -> 'a -> 'a -> 'a
```

The keyword `run` states that its first argument `f` is a stateful function and thus has a type of the form $t_1 \Rightarrow t_2$ instead of $t_1 \rightarrow t_2$.

Using the primitive `condact`, it is possible to program a classical operator both available in the SCADE library and in the *digital* library of SIMULINK. Its graphical representation in SCADE is given in Figure 7.1. This operator detects a rising edge (false to true transition). The output is true when a transition has been detected and is sustained for `numb_of_cycle` cycles. The output is initially false and a rising edge arriving while the output is true is still detected.

```
let node count_down (res, n) = cpt where
  rec cpt = if res then n else (n -> pre (cpt - 1))

let node rising_edge_retrigger rer_input numb_of_cycle = rer_output
  where
    rec rer_output = (0 < v) & (c or count)
    and v =
          condact count_down clk 0 (count, numb_of_cycle)
    and c = false fby rer_output
    and clock clk = c or count
    and count = false -> (rer_input & pre (not rer_input))
```

Higher-order is useful for the compiler writer as a way to reduce the number of basic primitives in the language. For the programmer, it brings the possibility of defining operators and to build generic libraries. In LUCID SYNCHRONE, all the static analysis (types, clocks, etc.) are compatible with higher-order. Note that higher-order programs we have considered so far can still be statically expanded into first-order programs.

7.2.6. *Datatypes and pattern matching*

Until now, we have only considered basic types and tuples. The language also supports structured datatypes. It is, for example, possible to define the type `number` whose value is either an integer or a float. The type `circle` defines a circle by its coordinates, its center and its radius.

```
type number = Int of int | Float of float
type circle = { center: float * float; radius: float }
```

Programs can be defined by pattern matching according to the structure of a value. Let us illustrate this on a wheel for which we want to detect the rotation. The wheel is composed of three colored sections: blue (`Blue`), red (`Red`) and green (`Green`). A sensor observes the color sequence and must determine from the observation if the wheel is immobile or not, as well as the direction of its rotation.

The direction is direct (`Direct`) for a sequence of `Red`, `Green`, `Blue`... and indirect (`Indirect`) for the opposite sequence. The direction may also be undetermined (`Undetermined`) or the wheel may be immobile (`Immobile`).

```
type color = Blue | Red | Green
type dir = Direct | Indirect | Undetermined | Immobile

let node direction i = d where
  rec pi = i fby i
  and ppi = i fby pi
  and d = match ppi, pi, i with
        (Red, Red, Red) | (Blue, Blue, Blue)
      | (Green, Green, Green) -> Immobile

      | (_, Blue, Red) | (_, Red, Green)
      | (_, Green, Blue) -> Direct

      | (_, Red, Blue) | (_, Blue, Green)
      | (_, Green, Red) -> Indirect

      | _ -> Undetermined
      end
```

The behavior is defined by pattern matching on three successive values of input `i`. Each case (possibly) defines a set of equations. At every instant, the construction `match/with` selects the first pattern (from top to bottom) that matches the value of `(pii, pi, i)` and executes the corresponding branch. Only one branch is executed during a reaction.

This example illustrates the interest of the pattern matching construct in ML language: a compiler is able to check its exhaustiveness and completeness. In the LUCID SYNCHRONE compiler, this analysis strictly follows that of OCAML [MAR 03a].

7.2.7. *A programming construct to share the memory*

The pattern-matching construct is a control structure whose behavior is very different from the `if/then/else` construct. During a reaction, only one branch is active. On the other hand, the `if/then/else` is a strict operator and all its argument execute at the same rate. The pattern-matching construct corresponds to a `merge` which combines streams with complementary clocks.

With control structures comes the problem of communicating between the branches of a control structure. This is a classic problem when dealing with several running modes in a graphical tool such as SCADE or SIMULINK. Each mode is defined by a block diagram and modes are activated in an exclusive manner. When two modes communicate through the use of some memory, this memory must be declared on the outside of the block in order to contain the last computed value. To ease this communication, we introduce constructions to declare, initialize and access a *shared memory* 4. The last computed value of a shared variable `o` can be accessed by writing `last o`. Consider the following system. It has two modes, the mode *up* increments a shared variable `o`, whereas the mode *down* decrements it.

```
let node up_down m step = o where
  rec match m with
        true -> do o = last o + step done
      | false -> do o = last o - step done
      end
  and last o = 0
```

In the above program, the `match/with` construction combines equations whereas it was applied to expressions in the previous example. The ability for a control-structure to combine equations eases the partial definition of some shared variables which may not have an explicit definition in one handler.

4. The SIMULINK tool provides a mechanism that eases the communication between several block diagram. This is mainly based on the use of imperative variables which can be read or written in different places.

The equation `last o = 0` defines a shared variable `o` with initial value `0`. The communication between the two modes is made with the construction `last o` which contains the last computed value of `o`.

step	1	1	1	1	1	1	1	1	1	1	1	···
m	*t*	*t*	*t*	*f*	*f*	*t*	*t*	*t*	*f*	*f*	*t*	···
last o	0	1	2	3	2	1	2	3	4	3	2	···
o	1	2	3	2	1	2	3	4	3	2	3	···

The above program has the same meaning as the one given below[5]. Here, the computation of the shared variable `last o` is done outside of the two control branches, meaning that `last o` does contain the last computed value of `o`.

```
let node up_down m step = o where
  rec match m with
        true -> do o = last_o + step done
      | false -> do o = last_o - step done
      end
  and last_o = 0 -> pre o
```

`last o` is another way to refer to the previous value of a stream and is thus similar to `pre o`. There is however a fundamental difference between the two. This difference is a matter of instant of observation. In a dataflow diagram, `pre` (e) denotes a local memory containing the value of its argument on the last time it has been observed. If `pre` (e) appears in a block that is executed from time to time, say on a clock c, this means that argument e is computed and stored only when clock c is true. On the other hand, `last` o denotes the last value of variable o at the clock where o is defined. `last` o is only defined on variables, not on expressions. `last` o is a way to communicate a value between two modes and this is why we call it a shared memory. The semantics of the previous program is precisely defined as:

```
let node up_down m step = o where
  rec o = merge c ((last_o when c) + (step when c))
                  ((last_o whenot c) - (step whenot c))
  and clock c = m
  and last_o = 0 -> pre o
```

In a graphical language, shared variables can be depicted differently to minimize the possible confusion between `last` o and `pre` o.

5. This is precisely how the LUCID SYNCHRONE compiler compiles it.

As a final remark, the introduction of shared memories makes it possible to implicitly complement streams with their last values. Omitting an equation for some variable `x` is equivalent to adding an equation `x = last x`. This is useful in practice because modes define several streams since it is enough to only give a definition to streams that have to change.

7.2.8. *Signals and signal patterns*

One difference between LUSTRE and ESTEREL is that the former manages streams whereas the latter manages signals. Streams and signals differ in the way they are accessed and produced. ESTEREL distinguishes two kinds of signals: pure and valued. A pure signal is essentially a Boolean stream, true when the signal is present and false otherwise. A valued signal is a more complex object: it can be either present or absent, and when present, it carries a value. Signals exhibit an interesting feature for the programmer: they only exist when explicitly emitted. A stream on the other hand, must be defined in every instant. This feature leads to a more natural description of control-dominated systems.

ESTEREL-like signals are a particular case of clocked streams and thus can be simulated into a data-flow calculus without any penalty in term of static analysis or code-generation [COL 06]. Signals are built and accessed through the use of two programming constructs, `emit` and `present`. A valued signal is simply a pair made of (1) a stream sampled on that condition c packed with (2) a Boolean stream c – its clock – giving the instant where the signal is present[6]. The clock signature of a signal x is a dependent pair $\Sigma(c : \alpha).\alpha$ `on` c made of a Boolean c with clock type α and a stream containing a value and whose clock type is α `on` c. We write α `sig` as a short-cut for the clock signature $\Sigma(c : \alpha).\alpha$ `on` c.

7.2.8.1. *Signals as clock abstractions*

A signal can be built from a sampled stream by abstracting its internal clock.

```
let node within min max x = o where
  rec clock c = (min <= x) & (x <= max)
  and emit o = x when c

val within :  'a -> 'a -> 'a => int sig
val within :: 'a -> 'a -> 'a -> 'a sig
```

This function computes a condition `c` and a sampled stream `x when c`. The equation `emit o = x when c` defines a signal `o` present and equal to `x` when

6. In circuit terminology, a signal is made of a value and an *enable* which indicate the instant where the value is valid [VUI 93].

c is true. The construction emit encapsulates the value with its clock, corresponding to a limited form of existential quantification. This enters exactly in the existing [COL 03] clock calculus based on the Laufer and Odersky type-system.

7.2.8.2. *Testing presence and pattern matching over signals*

The presence of signal x can be tested using the Boolean expression ?*x*. For example, the following program counts the number of occurrences of the signal x.

```
let node count x = cpt where
  rec cpt = if ?x then 1 -> pre cpt + 1 else 0 -> pre cpt

val count :  'a sig => int
val count :: 'a sig -> 'a
```

LUCID SYNCHRONE offers a more general mechanism to test the presence of several signals and access their value. This mechanism is similar to pattern matching and is reminiscent of join-patterns in the join-calculus [FOU 96].

The following program expects two input signals x and y and returns the sum of x and y when both signals are emitted. It returns the value of x when only x is emitted, the value of y when only y is emitted and 0 otherwise.

```
let node sum x y = o where
  present
    x(v) & y(w) -> do o = v + w done
  | x(v1) -> do o = v1 done
  | y(v2) -> do o = v2 done
  | _ -> do o = 0 done
  end

val sum :  int sig -> int sig => int
val sum :: 'a sig -> 'a sig -> 'a
```

Each handler is made of a filter and a set of equations. Filters are treated sequentially. The filter x(v) & y(w) is verified when both signals x and y are present. In that case, v and w are bounded to the values of the signals. If this filter is not satisfied, the second one is considered, etc. The last filter _ defines the default case. Note that x and y are signal expressions whereas v and w are patterns. Thus, a filter x(4) & y(w) is read: await the instant where x is present and carry the value 4 and y is present.

Using signals, it is possible to mimic the default construction of SIGNAL. The expression default x y emits the value of x when x is present, the value of y when x is absent and y is present. o, being a signal, does not keep implicitly its last value in the remaining case (x and y absent) and it is considered absent.

```
let node default x y = o where
  present
    x(v) -> do emit o = v done
  | y(v) -> do emit o = v done
  end

val default :  'a -> 'a => 'a
val default :: 'a sig -> 'a sig -> 'a sig
```

This is only a simulation since the clock information – the precise instant where `x`, `y` and `default x y` are emitted – is hidden. The compiler is not able to state that `o` is emitted only when `x` or `y` are present.

The use of signals comes naturally when considering control dominated systems. Moreover, pattern matching over signals is safe in the sense that it is possible to access the value of a signal only when the signal is emitted. This is an important difference with ESTEREL where the value of a signal is implicitly sustained and can be accessed even when the signal is not emitted, thus raising initialization issues.

7.2.9. *State machines and mixed designs*

In order to define control dominated systems, it is also possible to directly define finite state machines. These state machines can be composed with dataflow equations as well as other state machines and can be arbitrarily nested [COL 05].

An automaton is a set of states and transitions. A state is made of a set of equations and transitions. Two types of transitions, *weak* and *strong*, can be fired in a state and for each of them, the target state can be either entered by *history* or simply *reset*.

7.2.9.1. *Weak and strong preemption*

In a synchronous reaction, we can consider two types of transitions from a state: a transition is *weak* when it is inspected at the end of the reaction or *strong* when it is made immediately, at the beginning of the reaction. In the former case, the condition determines the active state of the next reaction. In the latter case, it determines the current active set of equations to be executed.

Here is an example of a two state automaton with weak transitions:

```
let node weak_switch on_off = o where
  automaton
    Off -> do o = false until on_off then On
  | On -> do o = true until on_off then Off
  end
```

A state is made of several states (the initial one being the first in the list) and each state defines a set of shared variables. This automaton has two states `Off` and `On`,

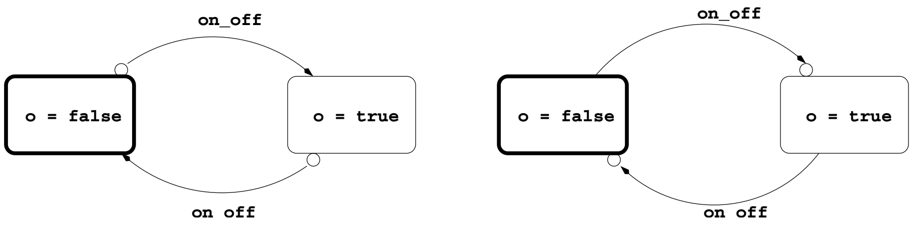


Figure 7.7. *Weak and strong transitions in an automaton*

each of them defining the current value of `o`. The keyword `until` indicates that `o` is false until the time (included) where `on_off` is true. Then, at the next instant, the active state becomes `On`. An automaton with weak transitions corresponds to a Moore automaton. On the other hand, the following function returns the value true as soon as the input `on_off` is true. Here, the value of `o` is defined by the equation `o = false` unless the condition `on_off` is verified. The condition is thus tested before executing the definitions of the state: it is called a strong transition.

```
let node strong_switch on_off = o where
  automaton
    Off -> do o = false unless on_off then On
  | On -> do o = true unless on_off then Off
  end
```

The graphical representation of these two automata is given in Figure 7.7. We borrow the notation introduced by Jean-Louis Colaço and inspired by the SYNCCHARTS [AND 96]: a strong transition is represented by an arrow starting with a circle which indicates that the condition is evaluated at the same instant as the target state. Reciprocally, a weak transition is represented by an arrow terminated by a circle, indicating that the condition is evaluated at the same instant as the source state.

`on_off`	*f*	*f*	*t*	*f*	*f*	*t*	*f*	*t*
`weak_switch on_off`	*f*	*f*	*f*	*t*	*t*	*t*	*f*	*f*
`strong_switch on_off`	*f*	*f*	*t*	*t*	*t*	*f*	*f*	*t*

We can notice that for any Boolean stream `on_off`, `weak_switch on_off` produces the same sequence as `strong_switch (false -> pre on_off)`.

7.2.9.2. *ABRO and modular reset*

Adding automata to a dataflow language gives ESTEREL-like features. We illustrate it on the ABRO example which illustrates the interest of synchronous composition and preemption [BER 99]. Its specification is the following:

> "Await the presence of events `a` and `b` and emit `o` at the precise instant where the two events have been received. Reset this behavior at every occurrence of `r`".

We first define a node `expect` that awaits for the presence of an event.

```
let node expect a = o where
  automaton
    S1 -> do o = false unless a then S2
  | S2 -> do o = true done
  end

let node abo a b = (expect a) & (expect b)
```

The node `abo` returns the value true and sustains it as soon as the inputs `a` and `b` are true. This is the parallel composition of two automata composed with an and gate. The node `abro` is obtained by making a new state automaton with a strong transition on the condition `r`. The target state is reset: every stream or automaton restarts in its initial configuration. This reset is indicated by the keyword `then`.

```
let node abro a b r = o where
  automaton
    S -> do o = abo a b unless r then S
  end
```

The construction `reset/every` is a short-cut for such an automaton.

```
let node abro a b r = o where
  reset
    o = abo a b
  every r
```

7.2.9.3. *Local definitions to a state*

Each state in an automaton is made of a set of equations defining shared and local variables. Local variables can be used to compute the values of shared variables and weak transitions only. They cannot be used to compute strong transitions since strong transitions are tested at the beginning of the reaction. This is checked automatically by the compiler.

The following program sustains the value true for a duration `d1`, the value false for a duration `d2`, then restarts.

```
let node alternate d1 d2 = status where
  automaton
    True ->
      let rec c = 1 -> pre c + 1 in
      do status = true
      until (c = d1) then False
  | False ->
      let rec c = 1 -> pre c + 1 in
      do status = false
      until (c = d2) then True
  end
```

The state `True` defines a local variable `c` that is used to compute the weak transition `c = d1`.

7.2.9.4. *Communication between states and shared memory*

In the above example, there is no communication between the values computed in each state. The need for such communication appears naturally when considering several running modes. Consider the following three states automaton.

```
let node up_down go = o where
  rec automaton
        Init ->
           do o = 0 until go then Up
      | Up ->
          do o = last o + 1
          until (o >= 5) then Down
      | Down ->
          do o = last o - 1
          until (o <= - 5) then Up
      end
```

This program computes the sequence:

go	f	f	t	t	t	t	t	t	t	t	t	t	t
o	0	0	0	1	2	3	4	5	4	3	2	1	0

Because the initial state is only weakly preempted, the value of `last o` is necessarily defined when entering the `Up` state. `last o` always contains the last computed value of `o`.

Every state defines the current value of a shared variable. In practice, it is heavy to define the value of a shared variable in every state. We need a mechanism to implicitly give a default value to those variables. A natural choice is to maintain the last computed value. Let us consider the example of a button to adjust an integer value from two Boolean values[7]:

```
let node adjust p m = o where
  rec last o = 0
  and automaton
        Idle ->
           do unless p then Incr unless m then Decr
      | Incr ->
           do o = last o + 1 unless (not p) then Idle
      | Decr ->
           do o = last o - 1 unless (not m) then Idle
      end
```

7. This example is due to Jean-Louis Colaço and Bruno Pagano.

The absence of equations in the initial state `Idle` means that `o` keeps its previous value, that is, an equation `o = last o` is implicitly added to the state.

7.2.9.5. *Resume or reset a state*

When a transition is fired, it is possible to specify whether the target state is reset (and this is what has been considered in the previous examples) or not. The language makes it possible to continue (or resume) the execution of a state in the configuration it had at its previous execution. For example:

```
let node up_down () = o where
  rec automaton
        Up -> do o = 0 -> last o + 1 until (o >= 2)
              continue Down
      | Down -> do o = last o - 1 until (o <= -2)
                continue Up
      end
```

The chronogram for an execution is given below:

o	0	1	2	1	0	-1	-2	-1	0	1	2	-1

Let us notice that `last o` is necessarily defined when entering the state `Down` since the state is only left with a weak transition. Replacing it with a strong transition (`unless`) raises an initialization error.

7.2.10. *Parametrized state machines*

We can now consider a more general class of automata where states can be parametrized. The actual value of the parameter is computed during the transition. This is useful to reduce the number of states and the communicate values between a source state and a target state. It also makes it possible to express in a uniform manner special treatments when entering a state (e.g., *transitions on entry* of STATEFLOW). The following program counts occurrences of `x`.

```
let node count x = o where
  automaton
    Zero -> do o = 0 until x then Succ(1)
  | Succ(v) -> do o = v until x then Succ(v+1)
  end
```

Consider now a mouse controller whose specification is the following:

> "Produce the event `double` when the two events `click` have been received in less than four `top`. Emit `simple` if only one event `click` has been received"

This corresponds to a three states automaton:

```
let node counting e = cpt where
  rec cpt = if e then 1 -> pre cpt + 1 else 0 -> pre cpt

let node controller click top = (simple, double) where
  automaton
    Await ->
     do simple = false and double = false
     until click then One
  | One ->
     do simple = false and double = false
     unless click then Emit(false, true)
     unless (counting top = 4) then Emit(true, false)
  | Emit(x1, x2) ->
     do simple = x1 and double = x2
     until true then Await
  end
```

The controller awaits the first occurrence of `click` then it enters in state `One` and counts the number of `top`. This state is strongly preempted when a second `click` is received or that the condition `counting top = 4` is true. For example, if `click` is true, the control goes immediately in state `Emit(false, true)`, giving the initial values `false` and `true` to the parameters `x1` and `x2`. Thus, at this instant, `simple = false` and `double = true`. At the next instant, the control goes to the initial state `Await`.

This example illustrates an important feature of state machines in LUCID SYNCHRONE: only one set of equations is executed during a reaction. Nonetheless, it is possible to combine (at most) one strong transition followed by a weak transition and this is exactly what has been illustrated above. As opposed to other formalisms such as the STATECHARTS [HAR 87] or the SYNCCHARTS [AND 96], it is impossible to cross an arbitrary number of states during a reaction leading to simpler design and debugging.

7.2.11. *Combining state machines and signals*

Weak or strong transitions are not only made of Boolean conditions, they can also test for the presence of signals as it was done with the `present` construct. We illustrate it on a system with two input signals `low` and `high` and an output stream `o`.

```
let node switch low high = o where
  rec automaton
        Init -> do o = 0 then Up(1)
      | Up(u) ->
          do o = last o + u
          unless low(v) then Down(v)
        | Down(v) ->
          do o = last o - v
          unless high(w) then Up(w)
        end

val switch :  'a sig -> 'a sig => 'a
val switch :: 'a sig -> 'a sig -> 'a
```

The condition `unless low(v) then Down(v)` reads: “enter the state `Down(v)` when the signal `low` is present with value `v`”. The construct `do o = 0 then Up(1)` is a short-cut for `do o = 0 until true then Up(1).`

high		3								2		
low				1		9			2	4		
o	0	1	2	1	0	-1	-2	-3	-4	-2	0	2

The specification of the mouse controller can be rewritten:

```
type e = Simple | Double

let node counting e = o where
  rec o = if ?e then 1 -> pre o + 1 else 0 -> pre o

let node controller click top = e where
  automaton
    Await ->
     do until click(_) then One
  | One ->
     do unless click(_) then Emit Double
     unless (counting top = 4) then Emit Simple
  | Emit(x) ->
     do emit e = x then Await
  end

val controller :  'a sig -> 'b sig => 'c sig
val controller :: 'a sig -> 'a sig -> 'a sig
```

Note that no value is computed in states `Await` and `One`. When writing `emit o = x`, the programmer states that `o` is a signal and thus, does not have to be defined in every state (or to implicitly complement its current value with `last o`). The signal `o` is only emitted in the state `Emit` and is absent otherwise.

The join use of signals and data-types exhibits an extra property with respect to the use of Boolean to represent events: here, the output `o` has only three possible values (`Simple`, `Double` or absent) whereas the Boolean encoding gives four values (with one meaningless).

7.2.12. *Recursion and non-real-time features*

LUCID SYNCHRONE also provides a way to define recursive functions thus leading to possible non-real-time executions. Nonetheless, this feature can be turned-off through a compilation frag, restricting the type system to only allow recursion on values with a statically bounded size.

A typical example of functional recursion is the sieve of *Eratosthene* as described in the seminal paper by Kahn [KAH 74] whose synchronous version has been given in [CAS 96]. These programs are still synchronous – parallel composition is the synchronous composition based on a global time scale – and streams can be efficiently compiled into scalar variables. Functional recursion models the dynamic creation of process and thus execution in bounded time and space is lost. Recursion can also be used through a static argument (as this is typically done in hardware functional languages such as LAVA [BJE 98]). In that case, the program executes in bounded time and memory. Nonetheless, the current version of the compiler does not distinguish them from the general case and they are thus rejected when the compilation flag is on. Several examples of recursive functions are available in the distribution [POU 06].

7.2.13. *Two classical examples*

We end this discussion with two classical examples. The first example is an *inverted pendulum* programmed in a purely dataflow style. The second example is a simple controller for a heater, and illustrates the combination of dataflow and automata.

7.2.13.1. *The inverted pendulum*

Consider an inverted pendulum with length l, its bottom part with coordinates (x_0, y_0) being manually controlled. θ is the angle of the pendulum. The physical low of the pendulum is given in Figure 7.8.

$$\begin{aligned}
l \times \tfrac{d^2\theta}{dt^2} &= (sin(\theta) \times (\tfrac{d^2y_0}{dt^2} + g)) - (cos(\theta) \times \tfrac{d^2x_0}{dt^2})\\
x &= x_0 + l \times sin(\theta)\\
y &= y_0 + l \times cos(\theta)
\end{aligned}$$

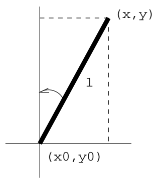


Figure 7.8. *The inverted pendulum*

We first define a module `Misc` to build an integrator and derivative (`*.` stands for the floating point multiplication).

```
(* module Misc *)
let node integr t x' =
  let rec x = 0.0 -> t *. x' +. pre x in x
let node deriv t x = 0.0 -> (x -.(pre x))/. t
```

The main module is written:

```
(* module Pendulum *)
let static dt = 0.05 (* step *)
let static l = 10.0  (* length *)
let static g = 9.81  (* acceleration *)

let node integr x' = Misc.integr dt x'
let node deriv x = Misc.deriv dt x

(* the equation of the pendulum *)
let node equation x0'' y0'' = theta where
  rec theta =
    integr (integr ((sin thetap) *. (y0'' +. g)
                    -. (cos thetap) *. x0'') /. l)
  and thetap = 0.0 fby theta

let node position x0 y0 =
  let x0'' = deriv (deriv x0)  in
  let y0'' = deriv (deriv y0)  in

  let theta = equation x0'' y0'' in

  let x = x0 +. l *. (sin theta)  in
  let y = y0 +. l *. (cos theta)  in
  Draw.make_pend x0 y0 x y

let node main () =
  let x0,y0 = Draw.mouse_pos () in
  let p = Draw.position x0 y0 in
  Draw.clear_pendulum (p fby p);
  Draw.draw_pendulum p;;
```

The dot notation `Misc.integr` denotes the function `integr` from the module `Misc`.

This example also illustrates the communication between LUCID SYNCHRONE and the host language (here OCAML) in which auxiliary functions are written. Here, the module `Draw` exports several functions (for example, `make_pend` creates a pendulum, `mouse_pos` returns the current position of the mouse).

7.2.13.2. *A heater*

The second example is a controller for a gas heater depicted in Figure 7.9.

The front of the heater has a green light indicating a normal functioning whereas a red light indicates that some problem has occurred (security stop). In the case of a

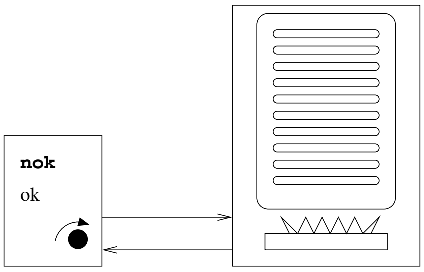


Figure 7.9. *The heater*

problem, the heater is stopped. It can be restarted by pressing a restart button. Finally, it is possible to set the desired water temperature.

The controller has the following inputs: the stream `res` is used to restart the heater; `expected_temp` is the expected temperature; `actual_temp` is the actual water temperature; `light_on` indicates that the gas is burning; and `dsecond` is a Boolean stream giving the base rate of the system. The outputs of the controller are the following: `open_light` lights the gas; `open_gas` opens the gas valve; `ok` is true for a normal functioning whereas `nok` indicates a problem.

The purpose of the controller is to keep the water temperature close to the expected temperature. When the water needs to be heated, the controller turns on the gas and the light for at most 500 milliseconds. When the light is on, only the gas valve is maintained open. If there is no light after 500 milliseconds it stops for 100 milliseconds and starts again. If after three tests there is still no light, the heater is blocked on a security stop. Only pushing the `res` button restarts the process.

```
let static low = 4
let static high = 4
let static delay_on = 500 (* in milliseconds *)
let static delay_off = 100

let node count d t = ok where
  rec ok = cpt = 0
  and cpt = (d -> pre cpt - 1) mod d

let node edge x = false -> not (pre x) & x
```

The following node decides whether the heater must be turned on. To avoid oscillations, we introduce an hysteresis mechanism. `low` and `high` are two thresholds. The first version is purely dataflow. The second, while equivalent, uses the automaton construction.

```
let node heat expected_temp actual_temp = on_heat where
  rec on_heat =
    if actual_temp <= expected_temp - low then true
    else if actual_temp >= expected_temp + high
         then false
         else false -> pre on_heat

let node heat expected_temp actual_temp = on_heat where
  rec automaton
        False ->
          do on_heat = false
          unless (actual_temp <= expected_temp - low)
          then True
      | True ->
          do on_heat = true
          unless (actual_temp >= expected_temp + high)
          then False
      end
```

Now, we define a node that turns on the light and gas for 500 milliseconds, then turn them off for 100 milliseconds and restarts:

```
let node command dsecond = (open_light, open_gas) where
  rec automaton
        Open ->
          do open_light = true
          and open_gas = true
          until (count delay_on dsecond) then Silent
      | Silent ->
          do open_light = false
          and open_gas = false
          until (count delay_off dsecond) then Open
      end
```

The program controlling the aperture of the light and gas is written below:

```
let node light dsecond on_heat light_on =
                    (open_light, open_gas, nok) where
  rec automaton
        Light_off ->
          do nok = false
          and open_light = false
          and open_gas = false
          until on_heat then Try
      | Light_on ->
          do nok = false
          and open_light = false
          and open_gas = true
          until (not on_heat) then Light_off
      | Try ->
          do
            (open_light, open_gas) = command dsecond
          until light_on then Light_on
          until (count 4 (edge (not open_light)))
          then Failure
      | Failure ->
          do nok = true
          and open_light = false
          and open_gas = false
          done
      end
```

Finally, the main function connects the two components.

```
let node main
      dsecond res expected_temp actual_temp light_on =
                  (open_light, open_gas, ok, nok) where
  rec reset
        on_heat = heat expected_temp actual_temp
      and
        (open_light, open_gas, nok) =
           light dsecond on_heat light_on
      and
        ok = not nok
      every res
```

In all the above examples, we only describe the reactive kernel of the application. From these definitions, the LUCID SYNCHRONE compiler produces a transition function written in OCAML which can in turned be linked to any other OCAML program.

7.3. Discussion

In this section we discuss related works and in particular the various embeddings of circuit description languages or reactive languages inside general purpose functional languages. Then, we give an historical perspective on LUCID SYNCHRONE and its use as a prototyping language for various extension of the industrial tool SCADE.

7.3.1. *Functional reactive programming and circuit description languages*

The interest of using a lazy functional language for describing synchronous circuits was identified in the early 1980s by Mary Sheeran in μFP [SHE 84]. Since then, various languages or libraries have been embedded inside the language HASKELL for describing circuits (HYDRA [O'D 95], LAVA [BJE 98]), the architecture of processors (for example, HAWK [MAT 98]), reactive systems (FRAN [ELL 99] and FRP [WAN 00]). Functional languages dedicated to circuit description have also been proposed (for example, JAZZ [VUI 93], REFLECT [KRS 04]). Circuits and dynamical systems can be modeled directly in HASKELL, using module defining basic operations (for example, registers, logical operations, stream transformers) in a way very similar to what synchronous languages such as LUSTRE or LUCID SYNCHRONE offer. The embedding of these domain specific languages inside HASKELL benefits from the expressive power of the host language (typing, data and control structures). The class mechanism of HASKELL [HAL 96] can also be used to easily change the representation of streams in order to obtain the verification of a property, a simulation or a compilation. *Multi-stage* [TAH 99] techniques are another way of describing domain-specific languages. This approach through an embedding inside a general purpose language is well adapted to circuit description language where the compilation result is essentially a net-list of Boolean operators and registers. This is nonetheless limited when a compilation to software code is expected (as is mainly the case for SCADE/LUSTRE). Even if a net-list can be compiled into sequential code, the obtained code is very inefficient due to code size increase. Moreover, when sampling functions are considered, as it is the case in FRAN or FRP, real-time (execution in bounded time and memory) cannot be statically guaranteed[8]. These works do not provide a notion of clock, control structures mixing dataflow systems and automata or compilation techniques with dedicated type-systems. The choice we have made in LUCID SYNCHRONE was to reject more programs in order to obtain more guarantees at compile time (e.g., synchronous execution, absence of deadlocks). Moreover, only software compilation has been considered in our work.

7.3.2. *Lucid Synchrone as a prototyping language*

The development of LUCID SYNCHRONE started around 1995 to serve as a laboratory for experimenting various extensions of synchronous languages. The first works showed that it was possible to define a functional extension of LUSTRE, combining the expressiveness of functional programming using lazy lists, with synchronous efficiency [CAS 95]. The clock calculus was defined as a dependent type system and generalized to higher-order [CAS 96]. The next question was to understand how to extend compilation techniques. This was answered by using co-algebraic techniques to formalize and to generalize the techniques already used to compile SCADE [CAS 98]. These results combined together lead to the first implementation of a compiler (V1) and had important practical consequences. Indeed, the definition of the clock calculus as a type system is the key to clock inference, which makes them much easier to use. The introduction of polymorphism is an important aspect of code reusability [COL 04a]. Automatic inference mechanisms are essential in a graphical tool like

8. Consider, for example, the program in Figure 7.5.

SCADE, in which programs are mainly drawn. They can also be found hidden in tools such as SIMULINK [CAS 05].

A partnership started in 2000 with the SCADE development team at ESTEREL-TECHNOLOGIES (TELELOGIC at the time), to write a new SCADE compiler. This compiler, RELUC (for *Retargetable Lustre Compiler*), uses the results already applied in LUCID SYNCHRONE, the clock calculus through typing, as well as new constructions of the language.

The basis of the language being there, works focused on the design of modular type-based analysis: causality loops analysis [CUO 01] and initialization analysis [COL 02, COL 04b]. The initialization analysis has been developed in collaboration with Jean-Louis Colaço at ESTEREL-TECHNOLOGIES and used at the same time in the LUCID SYNCHRONE compiler and the RELUC compiler. On real-size examples (more than 50,000 lines of codes), it proved to be fast and precise, reducing the number of wrong alarms. Work on the clock calculus also continued. As it was expressed as a dependent type system, it was natural to embed it in the COQ [COQ 07] proof assistant, thus getting both a correction proof [BOU 01] and some introspection on the semantics of the language. By looking at SCADE designs, we observed that this calculus could be also turned into an ML-like type system basing it on the Laüfer and Odersky extension [LÄU 92]. Although less expressive than the one based on dependent types, it is expressive enough in practice and can be implemented much more efficiently [COL 03]. It is used in the current version of the compiler (V3) and replace the dependent type based clock calculus of RELUC. At the same time, several language extensions were also studied: a modular reinitialization construct [HAM 00], the addition of sum types, and a pattern matching operation [HAM 04]. These works show the interest of the clock mechanism of synchronous languages, present in the beginning of both LUSTRE and SIGNAL. Clocks provide a simple and precise semantics for control structures that can be translated into the code language, thus reusing the existing code generator. This work was pursued in collaboration with ESTEREL-TECHNOLOGIES and led to the proposition of an extension of LUSTRE with hierarchical automata [COL 05, COL 06] in the spirit of the *mode-automata* of Maraninchi and Rémond [MAR 03b]. This extension is also based on the clock mechanism, and automata are translated into the core language. This extension is implemented in LUCID SYNCHRONE (V3) and in the RELUC compiler at ESTEREL-TECHNOLOGIES. All these developments are integrated in SCADE 6, the next release of SCADE.

7.4. Conclusion

This chapter has presented the actual development of LUCID SYNCHRONE. Based on the synchronous model of LUSTRE but reformulated in the framework of typed functional languages, it offers higher-order features, automatic type and clock inference and the ability to describe, in a uniform way, data and control dominated systems.

The LUCID SYNCHRONE experiment illustrates the various extensions that can be done in the field of synchronous languages to increase their expressive power and safety while retaining their basic properties for describing real-time systems. Two natural directions can be drawn from this work. One concerns the link with formal verification and proof systems (such as COQ) with certified compilation and proof of programs in mind. The other concerns the integration of some of the principles of synchronous programming as a general model of concurrency (not limited to real-time) inside a general purpose language.

7.5. Acknowledgment

The work on LUCID SYNCHRONE has been mainly developed at the Pierre and Marie Curie University and greatly benefited from discussions with Thérèse Hardin. It is also the result of a long collaboration with Jean-Louis Colaço from ESTEREL-TECHNOLOGIES. We thank them warmly.

7.6. Bibliography

[AND 96] ANDRÉ C., "Representation and analysis of reactive behaviors: a synchronous approach", *CESA*, Lille, IEEE-SMC, July 1996, available at: `www-mips.unice.fr/~andre/synccharts.html`.

[ASH 85] ASHCROFT E. A., WADGE W. W., "Lucid, The Data-flow Programming Language", APIC Studies in Data Processing, Academic Press, 1985.

[BEN 91] BENVENISTE A., LEGUERNIC P., JACQUEMOT C., "Synchronous programming with events and relations: the SIGNAL language and its semantics", *Science of Computer Programming*, vol. 16, p. 103–149, 1991.

[BEN 03] BENVENISTE A., CASPI P., EDWARDS S., HALBWACHS N., LE GUERNIC P., DE SIMONE R., "The synchronous languages 12 years later", *Proceedings of the IEEE*, vol. 91, num. 1, January 2003.

[BER 92] BERRY G., GONTHIER G., "The Esterel synchronous programming language, design, semantics, implementation", *Science of Computer Programming*, vol. 19, num. 2, p. 87–152, 1992.

[BER 99] BERRY G., The Esterel v5 Language Primer, Version 5.21 release 2.0, Draft book, 1999.

[BJE 98] BJESSE P., CLAESSEN K., SHEERAN M., SINGH S., "Lava: hardware design in Haskell", *International Conference on Functional Programming (ICFP)*, ACM, 1998.

[BOU 01] BOULMÉ S., HAMON G., "Certifying synchrony for free", *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, vol. 2250, La Havana, Cuba, Lecture Notes in Artificial Intelligence, Springer Verlag, December 2001, Short version of "A clocked denotational semantics for Lucid-Synchrone in Coq", available as a Technical Report (LIP6), at `www.lri.fr/~pouzet`.

[BRO 04] BROOKS C., LEE E. A., LIU X., NEUENDORFFER S., ZHAO Y., ZHENG H., "Heterogeneous concurrent modeling and design in Java", Memorandum UCB/ERL M04/27, EECS, University of California, Berkeley, CA 94720, USA, July 2004.

[BUC 94] BUCK J., HA S., LEE E., MESSERSCHMITT D., "Ptolemy: a framework for simulating and prototyping heterogeneous systems", *International Journal of computer Simulation*, 1994, special issue on Simulation Software Development.

[BUD 99] BUDDE R., PINNA G. M., POIGNÉ A., "Coordination of synchronous programs", *International Conference on Coordination Languages and Models*, num. 1594 Lecture Notes in Computer Science, 1999.

[CAS 92] CASPI P., "Clocks in dataflow languages", *Theoretical Computer Science*, vol. 94, p. 125–140, 1992.

[CAS 95] CASPI P., POUZET M., "A functional extension to Lustre", ORGUN M. A., ASHCROFT E. A., Eds., *International Symposium on Languages for Intentional Programming*, Sydney, Australia, World Scientific, May 1995.

[CAS 96] CASPI P., POUZET M., "Synchronous Kahn networks", *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pennsylvania, May 1996.

[CAS 98] CASPI P., POUZET M., "A co-iterative characterization of synchronous stream functions", *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998, extended version available as a VERIMAG tech. report no. 97–07 at `www.lri.fr/~pouzet`.

[CAS 05] CASPI P., CURIC A., MAIGNAN A., SOFRONIS C., TRIPAKIS S., "Translating discrete-time Simulink to Lustre", *ACM Transactions on Embedded Computing Systems*, 2005, Special Issue on Embedded Software.

[COH 06] COHEN A., DURANTON M., EISENBEIS C., PAGETTI C., PLATEAU F., POUZET M., "N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems", *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, USA, January 2006.

[COL 02] COLAÇO J.-L., POUZET M., "Type-based initialization analysis of a synchronous data-flow language", *Synchronous Languages, Applications, and Programming*, vol. 65, Electronic Notes in Theoretical Computer Science, 2002.

[COL 03] COLAÇO J.-L., POUZET M., "Clocks as first class abstract types", *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, October 2003.

[COL 04a] COLAÇO J.-L., GIRAULT A., HAMON G., POUZET M., "Towards a higher-order synchronous data-flow language", *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, September 2004.

[COL 04b] COLAÇO J.-L., POUZET M., "Type-based initialization analysis of a synchronous data-flow language", *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6, num. 3, p. 245–255, August 2004.

[COL 05] COLAÇO J.-L., PAGANO B., POUZET M., "A conservative extension of synchronous data-flow with state machines", *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey City, New Jersey, USA, September 2005.

[COL 06] COLAÇO J.-L., HAMON G., POUZET M., "Mixing signals and modes in synchronous data-flow systems", *ACM International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October 2006.

[COQ 07] "The Coq proof assistant", 2007, http://coq.inria.fr.

[CUO 01] CUOQ P., POUZET M., "Modular causality in a synchronous stream language", *European Symposium on Programming (ESOP'01)*, Genoa, Italy, April 2001.

[ELL 99] ELLIOTT C., "An embedded modeling language approach to interactive 3D and multimedia animation", *IEEE Transactions on Software Engineering*, vol. 25, num. 3, p. 291–308, May–June 1999.

[FOU 96] FOURNET C., GONTHIER G., "The reflexive chemical abstract machine and the join-calculus", *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, ACM, p. 372–385, January 1996.

[HAL 91a] HALBWACHS N., CASPI P., RAYMOND P., PILAUD D., "The synchronous dataflow programming language LUSTRE", *Proceedings of the IEEE*, vol. 79, num. 9, p. 1305–1320, September 1991.

[HAL 91b] HALBWACHS N., RAYMOND P., RATEL C., "Generating efficient code from dataflow programs", *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[HAL 96] HALL C. V., HAMMOND K., JONES S. L. P., WADLER P. L., "Type classes in Haskell", *ACM Trans. Program. Lang. Syst.*, vol. 18, num. 2, p. 109–138, ACM Press, 1996.

[HAM 00] HAMON G., POUZET M., "Modular resetting of synchronous data-flow programs", *ACM International Conference on Principles of Declarative Programming (PPDP'00)*, Montreal, Canada, September 2000.

[HAM 02] HAMON G., "Calcul d'horloge et Structures de Contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML", PhD thesis, Pierre and Marie Curie University, Paris, France, 14 November 2002.

[HAM 04] HAMON G., "Synchronous data-flow pattern matching", *Synchronous Languages, Applications, and Programming*, Electronic Notes in Theoretical Computer Science, 2004.

[HAR 85] HAREL D., PNUELI A., "On the development of reactive systems", *Logic and Models of Concurrent Systems*, vol. 13 of *NATO ASI Series*, p. 477–498, Springer Verlag, 1985.

[HAR 87] HAREL D., "StateCharts: a visual approach to complex systems", *Science of Computer Programming*, vol. 8–3, p. 231–275, 1987.

[JOU 94] JOURDAN M., LAGNIER F., RAYMOND P., MARANINCHI F., "A multiparadigm language for reactive systems", *5th IEEE International Conference on Computer Languages*, Toulouse, France, IEEE Computer Society Press, May 1994.

[KAH 74] KAHN G., "The semantics of a simple language for parallel programming", *IFIP 74 Congress*, North Holland, Amsterdam, 1974.

[KRS 04] KRSTIC S., MATTHEWS J., "Semantics of the reFLect Language", *PPDP*, ACM, 2004.

[LÄU 92] LÄUFER K., ODERSKY M., "An extension of ML with first-class abstract types", *ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, California, p. 78–91, June 1992.

[LER 07] LEROY X., "The Objective Caml system release 3.10. Documentation and user's manual", Report, INRIA, 2007.

[MAR 98] MARANINCHI F., RÉMOND Y., "Mode-automata: about modes and states for reactive systems", *European Symposium On Programming*, Lisbon, Portugal, Springer Verlag, March 1998.

[MAR 00] MARANINCHI F., RÉMOND Y., RAOUL Y., "MATOU: an implementation of mode-automata into DC", *Compiler Construction*, Berlin (Germany), Springer Verlag, March 2000.

[MAR 03a] MARANGET L., "Les avertissements du filtrage", *Actes des Journées Francophones des Langages Applicatifs*, Inria éditions, 2003.

[MAR 03b] MARANINCHI F., RÉMOND Y., "Mode-automata: a new domain-specific construct for the development of safe critical systems", *Science of Computer Programming*, num. 46, p. 219–254, Elsevier, 2003.

[MAT 98] MATTHEWS J., LAUNCHBURY J., COOK B., "Specifying microprocessors in Hawk", *International Conference on Computer Languages*, IEEE, 1998.

[MAT 03] "The Mathworks, stateflow and stateflow coder, user's guide", release 13sp1, September 2003.

[O'D 95] O'DONNELL J., "From transistors to computer architecture: Teaching functional circuit specification in Hydra", SPRINGER VERLAG, Ed., *Functional Programming Languages in Education*, p. 195–214, 1995.

[PIE 02] PIERCE B. C., *Types and Programming Languages*, MIT Press, 2002.

[POI 97] POIGNÉ A., HOLENDERSKI L., "On the combination of synchronous languages", ROEVER W., Ed., *Workshop on Compositionality: The Significant Difference*, vol. LNCS 1536, Malente, Springer Verlag, p. 490–514, September 8–12 1997.

[POU 02] POUZET M., Lucid Synchrone: un langage synchrone d'ordre supérieur, Paris, France, November 2002, Authorization to supervise research.

[POU 06] POUZET M., Lucid Synchrone, version 3. Tutorial and reference manual, Paris-Sud University, LRI, April 2006, `www.lri.fr/~pouzet/lucid-synchrone`.

[RAY 98] RAYMOND P., WEBER D., NICOLLIN X., HALBWACHS N., "Automatic Testing of Reactive Systems", *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[REY 98] REYNOLDS J. C., *Theories of Programming Languages*, Cambridge University Press, 1998.

[SCA 07] SCADE, "`http://www.esterel-technologies.com/scade/`", 2007.

[SHE 84] SHEERAN M., "muFP, a language for VLSI design", *ACM Conference on LISP and Functional Programming*, Austin, Texas, p. 104–112, 1984.

[TAH 99] TAHA W., Multi-Stage Programming: Its Theory and Applications, Technical Report num. CSE-99-TH-002, Oregon Graduate Institute of Science and Technology, November 1999.

[THE 05] 2005, `www.mathworks.com`.

[VUI 93] VUILLEMIN J., On circuits and numbers, Report, Digital, Paris Research Laboratory, 1993.

[WAD 90] WADLER P., "Deforestation: transforming programs to eliminate trees", *Theoretical Computer Science*, vol. 73, p. 231–248, 1990.

[WAN 00] WAN Z., HUDAK P., "Functional reactive programming from first principles", *International Conference on Programming Language, Design and Implementation (PLDI)*, 2000.

This page intentionally left blank

Chapter 8

Verification of Real-Time Probabilistic Systems

8.1. Introduction

In this chapter, we consider verification techniques for a particular class of real-time systems: those which incorporate *probabilistic* characteristics. This can come from a number of possible sources, for example, unpredictable behavior resulting from the failure of components of a system, or random choices made according to the execution of a probabilistic protocol. An application domain in which both of these aspects are important is that of probabilistic communication protocols, including for example Bluetooth, IEEE 802.11 Wireless LAN and IEEE 1394 FireWire, which often include potentially faulty communication channels and probabilistic algorithms such as randomized back-off.

Formal verification techniques for systems such as these need to incorporate not only probabilistic characteristics, but also *non-determinism* (e.g. due to concurrency between asynchronous components or underspecified system parameters) and *real-time* information. For this reason, *probabilistic timed automata* are an ideal modeling formalism in this context. In this chapter, we give an introduction to probabilistic timed automata, how to express properties of these models, and a range of model checking techniques that can be applied to them. We also include an example of their practical application to a case study: the IEEE 1394 FireWire root contention protocol.

Chapter written by Marta KWIATKOWSKA, Gethin NORMAN, David PARKER and Jeremy SPROSTON.

The structure of the remainder of this chapter is as follows. In section 8.2, we give some background material and notation, introduce the probabilistic timed automata formalism and then explain how properties of these models can be formalized. In section 8.3, we present four model checking techniques: the *region graph* method, the *forwards symbolic* and *backward symbolic* approaches and the *digital clocks* technique. Finally, in section 8.4, we present the FireWire root contention case study and, using this, summarize the relative performances of the various approaches.

8.2. Probabilistic timed automata

We model real-time probabilistic systems using probabilistic timed automata [JEN 96, KWI 02a], a formalism which features non-deterministic and probabilistic choices between transitions, and also contains constraints on the times at which transitions can or must be taken. The formalism is derived by extending classical timed automata [ALU 94, HEN 94] with discrete probability distributions over edges.

8.2.1. *Preliminaries*

A discrete probability *distribution* over a countable set Q is a function $\mu : Q \rightarrow [0,1]$ such that $\sum_{q \in Q} \mu(q) = 1$. Let $\mathsf{Dist}(Q)$ denote the set of all probability distributions over Q. For a distribution μ on Q, let $\mathsf{support}(\mu) = \{q \in Q \mid \mu(q) > 0\}$. For an uncountable set Q' we define $\mathsf{Dist}(Q')$ to be the set of functions $\mu : Q' \rightarrow [0,1]$, such that $\mathsf{support}(\mu)$ is a countable set and μ restricted to $\mathsf{support}(\mu)$ is a (discrete) probability distribution. For $q \in Q$, let μ_q be the *point distribution* at q which assigns probability 1 to q.

We use $\mathbb{R}$ to denote the non-negative reals, and $\mathbb{N}$ to denote the naturals. Let $\mathbb{T} \in \{\mathbb{R}, \mathbb{N}\}$ be a *time domain*, and let $\mathcal{X}$ be a finite set of variables called *clocks* which take values from $\mathbb{T}$. A function $v : \mathcal{X} \rightarrow \mathbb{T}$ is referred to as a *clock valuation*. The set of all clock valuations is denoted by $\mathbb{T}^{\mathcal{X}}$. Let $\mathbf{0} \in \mathbb{T}^{\mathcal{X}}$ be the clock valuation that assigns 0 to all clocks in $\mathcal{X}$. For any $v \in \mathbb{T}^{\mathcal{X}}$ and $t \in \mathbb{T}$, we use $v{+}t$ to denote the clock valuation defined as $(v{+}t)(x) = v(x){+}t$ for all $x \in \mathcal{X}$. We use $v[X := 0]$ to denote the clock valuation obtained from v by resetting all of the clocks in $X \subseteq \mathcal{X}$ to 0, and leaving the values of all other clocks unchanged; formally, $v[X := 0](x) = 0$ if $x \in X$, and $v[X := 0](x) = v(x)$ otherwise.

The set of *zones* of $\mathcal{X}$, written $Zones(\mathcal{X})$, is defined by the syntax:

$$\zeta ::= x \leqslant d \mid c \leqslant x \mid x{+}c \leqslant y{+}d \mid \neg\zeta \mid \zeta \vee \zeta$$

where $x, y \in \mathcal{X}$ and $c, d \in \mathbb{N}$. As usual, $\zeta_1 \wedge \zeta_2 \equiv \neg(\neg\zeta_1 \vee \neg\zeta_2)$, strict constraints can be written using negation, for example $x > 2 \equiv \neg(x \leqslant 2)$, and equality can be written as the conjunction of constraints, for example $x = 3 \equiv (x \leqslant 3) \wedge (3 \leqslant x)$. The clock valuation v *satisfies* the zone ζ, written $v \triangleleft \zeta$, if and only if ζ resolves to true after substituting each clock $x \in \mathcal{X}$ with the corresponding clock value $v(x)$ from v.

Intuitively, the semantics of a zone is the set of clock valuations (subset of $\mathbb{T}^{\mathcal{X}}$) which satisfies the zone.

Note that more than one zone may represent the same set of clock valuations (for example, $(x \leqslant 2) \wedge (y \leqslant 1) \wedge (x \leqslant y{+}2)$ and $(x \leqslant 2) \wedge (y \leqslant 1) \wedge (x \leqslant y{+}3)$). We henceforth consider only canonical zones, which are zones for which the constraints are as 'tight' as possible. For any valid zone $\zeta \in \mathit{Zones}(\mathcal{X})$, there exists an $O(|\mathcal{X}|^3)$ algorithm to compute the (unique) canonical zone of ζ [DIL 89]. This enables us to use the above syntax for zones interchangeably with semantic, set-theoretic operations.

We require the following classical operations on zones [HEN 94, TRI 98]. For zones ζ, ζ' and set of clocks $X \subseteq \mathcal{X}$:

$$\nearrow_{\zeta'} \zeta \overset{\text{def}}{=} \big\{v \mid \exists t \geqslant 0 \cdot \exists v' \triangleleft \zeta \cdot \big(v = v'{+}t \;\wedge\; \forall t' \leqslant t \cdot (v'{+}t' \triangleleft \zeta{\vee}\zeta')\big)\big\}$$

$$\swarrow_{\zeta'} \zeta \overset{\text{def}}{=} \big\{v \mid \exists t \geqslant 0 \cdot \big(v{+}t \triangleleft \zeta \;\wedge\; \forall t' \leqslant t \cdot (v{+}t' \triangleleft \zeta{\vee}\zeta')\big)\big\}$$

$$[X := 0]\zeta \overset{\text{def}}{=} \{v \mid v[X := 0] \in \zeta\}$$

$$\zeta[X := 0] \overset{\text{def}}{=} \{v[X := 0] \mid v \in \zeta\}.$$

Zone $\nearrow_{\zeta'} \zeta$ contains the clock valuations that can be reached from a clock valuation in ζ by letting time pass and remain in ζ' while time elapses. On the other hand, zone $\swarrow_{\zeta'} \zeta$ contains the clock valuations that can, by letting time pass, reach a clock valuation in ζ and remain in ζ' until ζ is reached. Zone $[X := 0]\zeta$ contains the clock valuations which result in a clock valuation in ζ when the clocks in X are reset to 0, while $\zeta[X := 0]$ contains the clock valuations which are obtained from clock valuations in ζ by resetting the clocks in X to 0.

In addition we will require the concepts of *c-equivalence* and *c-closure* [TRI 98, DAW 98].

DEFINITION 8.1.– *For $c \in \mathbb{N}$, two clock valuations v, v' are described as* c-equivalent *if the following conditions are satisfied:*

– for any $x \in \mathcal{X}$, either $v(x) = v'(x)$, or $v(x) > c$ and $v'(x) > c$;

– for any $x, y \in \mathcal{X}$, either $v(x){-}v(y) = v'(x){-}v'(y)$, or $v(x){-}v(y) > c$ and $v'(x){-}v'(y) > c$.

We define the c-closure *of a zone $\zeta \in \mathbb{R}^{\mathcal{X}}$, denoted by* $\mathsf{close}(\zeta, c)$*, to be the greatest zone $\zeta' \supseteq \zeta$ satisfying the following: for all $v' \in \zeta'$, there exists $v \in \zeta$ such that v and v' are c-equivalent.*

Intuitively, the c-closure of a zone is obtained by removing all of its boundaries that correspond to constraints referring to integers greater than c. Observe that, for a given c, there are only a finite number of c-closed zones.

8.2.2. *Syntax of probabilistic timed automata*

We now define formally probabilistic timed automata, in which, similarly to timed automata [ALU 94, HEN 94], the timing constraints are represented by zones over a finite set of clocks. The difference between timed automata and probabilistic timed automata lies in the edge relation: in timed automata the traversal of an edge corresponds to a single target location and, possibly, the resetting of some clocks; instead probabilistic timed automata allow (discrete) probabilistic choice over target locations and clock resets.

DEFINITION 8.2.– *A* probabilistic timed automaton *is a tuple* $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$ *where:*

– *L is a finite set of* locations *with an* initial location $\bar{l} \in L$*;*

– *$\mathcal{X}$ is a finite set of* clocks;

– *Σ is a finite set of* events;

– *$inv : L \rightarrow Zones(\mathcal{X})$ is a function called the* invariant condition;

– *$prob \subseteq L \times Zones(\mathcal{X}) \times \Sigma \times \mathsf{Dist}(2^{\mathcal{X}} \times L)$ is a finite set called the* probabilistic edge relation.

A *state* of a probabilistic timed automaton PTA is a pair $(l, v) \in L \times \mathbb{T}^{\mathcal{X}}$ such that $v \triangleleft inv(l)$. Informally, the behavior of a probabilistic timed automaton can be understood as follows. The model starts in the initial location $\bar{l}$ with all clocks set to 0, that is, in the state $(\bar{l}, \mathbf{0})$. In this, and any other state (l, v), there is a non-deterministic choice of either (1) making a *discrete transition* or (2) letting *time pass*. In case (1), a discrete transition can be made according to any probabilistic edge $(l, g, \sigma, p) \in prob$ with source location l which is *enabled*, where (l, g, σ, p) is said to be enabled if its zone g is satisfied by the current clock valuation v. Then the probability of moving to the location l' and resetting all of the clocks in X to 0 is given by $p(X, l')$. Case (2) has the effect of increasing the values of all clocks in $\mathcal{X}$ by the amount of time elapsed. We note that, in case (2), the option of letting time pass is available only if the invariant condition $inv(l)$ is satisfied continually while time elapses.

DEFINITION 8.3.– *An* edge *of* PTA *generated by $(l, g, \sigma, p) \in prob$ is a tuple of the form (l, g, σ, p, X, l') such that $p(X, l') > 0$. Let* $\mathsf{edges}(l, g, \sigma, p)$ *be the set of edges generated by (l, g, σ, p), and let* $\mathsf{edges} = \{\mathsf{edges}(l, g, \sigma, p) \mid (l, g, \sigma, p) \in prob\}$.

EXAMPLE 8.1.– *Consider the probabilistic timed automaton modeling a simple probabilistic communication protocol given in Figure 8.1. The nodes represent the locations, namely* di *(sender has data, receiver idle),* si *(sender sent data, receiver idle), and* sr *(sender sent data, receiver received). The automaton starts in location* di *in which data is ready to be sent by the sender (the double border indicates that* di *is the initial location). After between 1 and 2 time units, the sender attempts to send the*

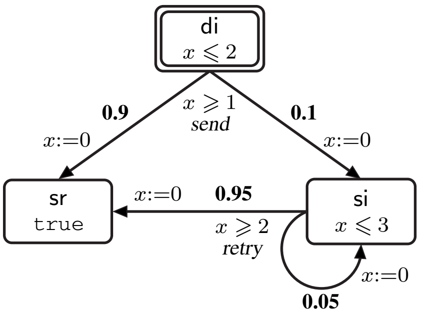


Figure 8.1. *A probabilistic timed automaton modeling a simple communication protocol*

data (event send*) and with probability 0.9 the data is received (location* sr *is reached), or with probability 0.1 the data is lost (location* si *is reached). In* si*, after 2 to 3 time units, the sender will attempt to resend the data (event* resend*), which again can be lost, this time with probability 0.05.*

Note that a timed automaton [ALU 94, HEN 94] corresponds to a probabilistic timed automaton for which every probabilistic edge $(l, g, \sigma, p) \in \mathit{prob}$ is such that $p = \mu_{(X,l')}$ (the point distribution assigning probability 1 to (X, l')) for some $(X, l') \in 2^{\mathcal{X}} \times L$.

We say that a probabilistic timed automaton is *well-formed* if, whenever a probabilistic edge is enabled, the target states resulting from all probabilistic outcomes satisfy the invariant condition. Formally, a probabilistic timed automaton $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, \mathit{inv}, \mathit{prob})$ is said to be well-formed if:

$$\forall (l, g, \sigma, p) \in \mathit{prob} \cdot \forall v \in \mathbb{T}^{\mathcal{X}} \cdot (v \lhd g) \\ \longrightarrow \big(\forall (X, l') \in \mathsf{support}(p) \cdot v[X := 0] \lhd \mathit{inv}(l')\big).$$

A probabilistic timed automaton can be transformed into a well-formed probabilistic timed automaton by simply replacing the guard g in each probabilistic edge $(l, g, \sigma, p) \in \mathit{prob}$ with

$$\left(\bigwedge_{(X,l') \in \mathsf{support}(p)} [X := 0]\mathit{inv}(l') \right) \wedge g \ .$$

For the remainder of the chapter we assume that all probabilistic timed automata are well-formed. Finally, we also assume that the set Σ of events of a probabilistic timed automaton is such that $\Sigma \cap \mathbb{R} = \emptyset$.

8.2.3. *Modeling with probabilistic timed automata*

To aid higher-level modeling, it is often useful to define complex systems as the *parallel composition* of a number of interacting sub-components. The definition of the parallel composition operator $\|$ of probabilistic timed automata uses ideas from the theory of (untimed) probabilistic automata [SEG 95b] and classical timed automata [ALU 94]. Let $\mathsf{PTA}_i = (L_i, \bar{l}_i, \mathcal{X}_i, \Sigma_i, inv_i, prob_i)$ for $i \in \{1,2\}$ and assume that $\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$.

DEFINITION 8.4.– *The* parallel composition *of two probabilistic timed automata* PTA_1 *and* PTA_2 *is the probabilistic timed automaton*

$$\mathsf{PTA}_1 \,\|\, \mathsf{PTA}_2 = (L_1 \times L_2, (\bar{l}_1, \bar{l}_2), \mathcal{X}_1 \cup \mathcal{X}_2, \Sigma_1 \cup \Sigma_2, inv, prob)$$

such that

– $inv(l, l') = inv_1(l) \wedge inv_2(l')$ *for all* $(l, l') \in L_1 \times L_2$*;*

– $((l_1, l_2), g, \sigma, p) \in prob$ *if and only if one of the following conditions holds:*

1) $\sigma \in \Sigma_1 \backslash \Sigma_2$ *and there exists* $(l_1, g, \sigma, p_1) \in prob_1$ *such that* $p = p_1 \otimes \mu_{(\emptyset, l_2)}$;

2) $\sigma \in \Sigma_2 \backslash \Sigma_1$ *and there exists* $(l_2, g, \sigma, p_2) \in prob_2$ *such that* $p = \mu_{(\emptyset, l_1)} \otimes p_2$;

3) $\sigma \in \Sigma_1 \cap \Sigma_2$ *and there exists* $(l_i, g_i, \sigma, p_i) \in prob_i$ *for* $i = 1, 2$ *such that* $g = g_1 \wedge g_2$ *and* $p = p_1 \otimes p_2$

where for any $l_1 \in L_1$*,* $l_2 \in L_2$*,* $X_1 \subseteq \mathcal{X}_1$ *and* $X_2 \subseteq \mathcal{X}_2$*:*

$$p_1 \otimes p_2(X_1 \cup X_2, (l_1, l_2)) = p_1(X_1, l_1) \cdot p_2(X_2, l_2)\,.$$

In addition (as in the timed automata model checking tool UPPAAL [LAR 97, BEH 04]), we allow the use of *urgent locations*, which when entered must be left before time can advance. These can be represented in the probabilistic timed automata framework by introducing an auxiliary clock z, setting the invariant condition of the urgent locations to $z \leqslant 0$, and resetting the value of z to 0 on entry to each urgent location. However, rather than introducing an auxiliary clock, it is generally more convenient from a modeling perspective to explicitly denote such locations as urgent.

8.2.4. *Semantics of probabilistic timed automata*

The semantics of timed automata is generally presented in terms of timed transition systems. To define the semantics of probabilistic timed automata, we employ timed Markov decision processes, which extend timed transition systems with (discrete) probabilistic choice. Timed Markov decision processes can also be seen as an extension of Markov decision processes (see Chapter 9) and a variant of Segala's probabilistic timed automata [SEG 95a].

DEFINITION 8.5.– *A* timed Markov decision process *is a tuple* $\mathsf{TMDP} = (S, \bar{s}, \Sigma, \mathbb{T}, Steps)$ *where:*

– S *is a set of* states*, including an* initial state $\bar{s} \in S$*;*

– *$Steps \subseteq S \times (\Sigma \cup \mathbb{T}) \times \mathsf{Dist}(S)$ is a* probabilistic transition relation *such that, if $(s, a, \mu) \in Steps$ and $a \in \mathbb{T}$, then μ is a point distribution.*

A *probabilistic transition* $s \xrightarrow{a,\mu} s'$ is performed from a state $s \in S$ by first non-deterministically selecting a pair (a, μ) such that $(s, a, \mu) \in Steps$, and then by making a probabilistic choice of target state s' according to the distribution μ, such that $\mu(s') > 0$. If $a \in \Sigma$, then the transition is interpreted as corresponding to the execution of an event, whereas if $a \in \mathbb{T}$, the transition is interpreted as corresponding to the elapse of time, the duration of which is a. We require that only event-distribution pairs can be probabilistic; that is, duration-distribution pairs always use a point distribution.

We now proceed to define the formal semantics of probabilistic timed automata. The definition is parametrized by both a time domain $\mathbb{T}$ and a *time increment operator* $\oplus$. A time increment operator is a binary operator which takes a clock valuation $v \in \mathbb{T}^{\mathcal{X}}$ and a time duration $t \in \mathbb{T}$, and returns a clock valuation $v \oplus t \in \mathbb{T}^{\mathcal{X}}$ which represents, intuitively, the clock valuation obtained from v after t time units have elapsed. The standard choice of time increment operator $\oplus$ corresponds to addition $+$, where $v+t$ is defined as in section 8.2.1. The semantics of probabilistic timed automata is typically described for the case in which $\mathbb{T} = \mathbb{R}$ and $\oplus$ is $+$. We also consider in section 8.3.4 an integer semantics in which $\mathbb{T} = \mathbb{N}$ and $\oplus$ is no longer standard addition.

DEFINITION 8.6.– *Let $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$ be a probabilistic timed automaton. The* semantics of PTA with respect to time domain $\mathbb{T}$ and time increment $\oplus$ *is the timed Markov decision process $[\![\mathsf{PTA}]\!]^{\oplus}_{\mathbb{T}} = (S, \bar{s}, \Sigma, \mathbb{T}, Steps)$ such that:*

– *$S \subseteq L \times \mathbb{T}^{\mathcal{X}}$ where $(l, v) \in S$ if and only if $v \triangleleft inv(l)$;*

– *$\bar{s} = (\bar{l}, \mathbf{0})$;*

– *$((l, v), a, \mu) \in Steps$ if and only if one of the following conditions holds:*

time transitions. *$a = t \in \mathbb{T}$ and $\mu = \mu_{(l, v \oplus t)}$ such that $v \oplus t' \triangleleft inv(l)$ for all $t' \in [0, t] \cap \mathbb{T}$;*

discrete transitions. *$a = \sigma \in \Sigma$ and there exists $(l, g, \sigma, p) \in prob$ such that $v \triangleleft g$ and for any $(l', v') \in S$:*

$$\mu(l', v') = \sum_{\substack{X \subseteq \mathcal{X}\ \& \\ v' = v[X:=0]}} p(X, l').$$

The summation in the definition of discrete transitions is required for the cases in which multiple clock resets result in the same target state.

8.2.5. *Probabilistic reachability and invariance*

In this section, we introduce *probabilistic reachability* and *invariance*, which are standard performance measures for probabilistic systems. Probabilistic reachability refers to the probability with which a certain set of states is reached (for example, states which correspond to the achievement of some goal or error states). On the other hand, probabilistic invariance refers to the probability of remaining in a certain set of states. For algorithms for the computation of such measures for finite state probabilistic systems see, for example [BIA 95, COU 98].

In order to introduce these measures, we first define more precisely the notion of the behavior of timed Markov decision processes. The behavior of a timed Markov decision process can be represented in two ways: using paths and adversaries. Formally, a path of a timed Markov decision process is a non-empty finite or infinite sequence of probabilistic transitions

$$\omega = s_0 \xrightarrow{a_0,\mu_0} s_1 \xrightarrow{a_1,\mu_1} s_2 \xrightarrow{a_2,\mu_2} \cdots .$$

We denote by $\omega(i)$ the $(i{+}1)$th state of ω, by $last(\omega)$ the last state of ω if ω is finite and by $step(\omega, i)$ the event or duration associated with the $(i{+}1)$-th transition (that is, $step(\omega, i) = a_i$). By abuse of notation, we say that a single state s is a path of length 0. The set of infinite paths starting in state s is denoted by $Path(s)$. For any infinite path $\omega = s_0 \xrightarrow{a_0,\mu_0} s_1 \xrightarrow{a_1,\mu_1} \cdots$, the accumulated duration up to the $(n{+}1)$-th state of ω is defined by:

$$dur(\omega, n{+}1) \overset{\text{def}}{=} \sum\{\!|a_i \mid 0 \leqslant i \leqslant n \wedge a_i \in \mathbb{T}|\!\} .$$

In contrast to a path, an adversary (or scheduler) represents a particular resolution of non-determinism *only*. More precisely, an adversary is a function which chooses an outgoing distribution in the last state of a path. Formally, we have the following definition.

DEFINITION 8.7.– *Let* TMDP $= (S, \bar{s}, \Sigma, \mathbb{T}, Steps)$ *be a timed Markov decision process. An* adversary A *of* TMDP *is a function mapping every finite path* ω *of* TMDP *to a pair* (a, μ) *such that* $(last(\omega), a, \mu)$ *is an element of Steps. For any state* $s \in S$*, let* $Path^A(s)$ *denote the subset of* $Path(s)$ *which correspond to* A*.*

The behavior of a timed Markov decision process TMDP $= (S, \bar{s}, \Sigma, \mathbb{T}, Steps)$ under a given adversary A is purely probabilistic. Using the measure construction of [KEM 76], for a state $s \in S$ and adversary A we can define a probability measure $Prob_s^A$ over the set of paths $Path^A(s)$. For further details see Chapter 9 or [RUT 04].

Models of real-time systems can contain unrealizable behaviors in which time does not exceed a certain bound. Generally, such behaviors are disregarded during the analysis of the model. In the following, we consider only time-divergent adversaries, which are adversaries that guarantee the divergence of time with probability 1. Formally, we say that an infinite path ω is *divergent* if, for any $t \in \mathbb{R}$, there exists $j \in \mathbb{N}$ such that $dur(\omega, j) > t$. An adversary A of a timed Markov decision process

TMDP is *divergent* if and only if, for each state s of TMDP, the probability $Prob_s^A$ assigned to the divergent paths of $Path^A(s)$ is 1. Let Adv_{TMDP} be the set of divergent adversaries of TMDP.

We now define probabilistic reachability and invariance at the level of timed Markov decision processes. For a timed Markov decision process $\mathsf{TMDP} = (S, \bar{s}, \Sigma, \mathbb{T}, Steps)$, state $s \in S$, sets $T, I \subseteq S$ of target states and invariant states, and adversary $A \in Adv_{\mathsf{TMDP}}$, let:

$$p^A_{\mathsf{TMDP}}(s, \Diamond\, T) \overset{\text{def}}{=} Prob_s^A\{\omega \in Path^A(s) \mid \exists i \in \mathbb{N} \,.\, \omega(i) \in T\}$$

$$p^A_{\mathsf{TMDP}}(s, \Box\, I) \overset{\text{def}}{=} Prob_s^A\{\omega \in Path^A(s) \mid \forall i \in \mathbb{N} \,.\, \omega(i) \in I\}\ .$$

DEFINITION 8.8.– Let $\mathsf{TMDP} = (S, \bar{s}, \Sigma, \mathbb{T}, Steps)$ be a timed Markov decision process. The *maximum and minimum reachability probabilities* of, from a state $s \in S$, reaching the set of target states T are defined as follows:

$$p^{\max}_{\mathsf{TMDP}}(s, \Diamond\, T) = \sup_{A \in Adv_{\mathsf{TMDP}}} p^A_{\mathsf{TMDP}}(s, \Diamond\, T)$$

$$p^{\min}_{\mathsf{TMDP}}(s, \Diamond\, T) = \inf_{A \in Adv_{\mathsf{TMDP}}} p^A_{\mathsf{TMDP}}(s, \Diamond\, T)\,.$$

The *maximum and minimum invariance probabilities* of, from a state $s \in S$, remaining in the set of states I are defined as follows:

$$p^{\max}_{\mathsf{TMDP}}(s, \Box\, I) = \sup_{A \in Adv_{\mathsf{TMDP}}} p^A_{\mathsf{TMDP}}(s, \Box\, I)$$

$$p^{\min}_{\mathsf{TMDP}}(s, \Box\, I) = \inf_{A \in Adv_{\mathsf{TMDP}}} p^A_{\mathsf{TMDP}}(s, \Box\, I)\,.$$

The minimum and maximum probabilities of reachability and invariance can be considered as dual properties. More precisely, for any state $s \in S$, sets of states T and I such that $T = S \setminus I$:

$$p^{\max}_{\mathsf{TMDP}}(s, \Diamond\, T) = 1 - p^{\min}_{\mathsf{TMDP}}(s, \Box\, I) \quad \text{and} \quad p^{\min}_{\mathsf{TMDP}}(s, \Diamond\, T) = 1 - p^{\max}_{\mathsf{TMDP}}(s, \Box\, I).$$

Because of this relationship, for the remainder of this chapter we focus on methods for computing probabilistic reachability properties.

In the context of the analysis of probabilistic timed automata, the set of target states is often expressed in terms of a set of target locations $T_L \subseteq L$. More precisely, given a set of target locations T_L, we take $T = \{(l, v) \in S \mid l \in T_L\}$ to be the set of target states and, for simplicity, write $p^{\max}_{[\![\mathsf{PTA}]\!]^{\oplus}_{\mathbb{T}}}(s, \Diamond\, T_L)$ and $p^{\min}_{[\![\mathsf{PTA}]\!]^{\oplus}_{\mathbb{T}}}(s, \Diamond\, T_L)$.

More generally, target states can be expressed by a set of location-zone pairs, rather than a set of locations. Location-zone pairs can be used, for example, to express that a target set of locations must be reached before or after a certain deadline. Using the construction of [KWI 02a], such reachability problems can be reduced to those referring to locations only by modifying syntactically the probabilistic timed automaton of interest.

The following types of properties can be expressed using probabilistic reachability:

Basic probabilistic reachability: the system reaches a certain set of states with a given maximum or minimum probability. For example, "with probability at least 0.999, a data packet is correctly delivered".

Probabilistic time-bounded reachability: the system reaches a certain set of states within a certain time deadline and probability threshold. For example, "with probability 0.01 or less, a data packet is lost within 5 time units".

Probabilistic cost-bounded reachability: the system reaches a certain set of states within a certain cost and probability bound. For example, "with probability 0.75 or greater, a data packet is correctly delivered with at most 4 retransmissions".

Invariance: the system does not leave a certain set of states with a given probability. For example, "with probability 0.875 or greater, the system never aborts".

Bounded response: the system inevitably reaches a certain set of states within a certain time deadline with a given probability. For example, "with probability 0.99 or greater, a data packet will always be delivered within 5 time units".

8.3. Model checking for probabilistic timed automata

In this section, we consider a number of automatic verification methods for probabilistic timed automata. The underlying semantics of a probabilistic timed automaton PTA is generally presented with respect to the time domain $\mathbb{R}$, and hence the resulting semantic timed Markov decision process $[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+}$ generally has an uncountable number of states and transitions. Therefore, the methods that we present are based on the construction of finite-state Markov decision processes, which are defined so that their analysis can be used to infer reachability probabilities and other performance measures of probabilistic timed automata. We present a different finite-state construction in each of the following four sections.

8.3.1. *The region graph*

Our first method is inspired by the classical region-graph technique for timed automata [ALU 94], in which a finite-state transition system is constructed from a timed automaton. Proofs of the key results can be found in [KWI 02a] where it is shown that the region graph can be used to model check the probabilistic timed temporal logic PTCTL which subsumes the checking of reachability properties.

The construction is based on a finite equivalence relation on the infinite set $\mathbb{R}^{\mathcal{X}}$ of clock valuations. We apply the same equivalence to the set of clock valuations of probabilistic timed automata in order to obtain a probabilistic region graph, which contains sufficient information for the calculation of reachability probabilities. In this section, we assume that probabilistic timed automata are *structurally non-Zeno* [TRI 05]: a probabilistic timed automaton is structurally non-Zeno if, for every sequence $X_0, (l_0, g_0, \sigma_0, p_0), X_1, (l_1, g_1, \sigma, p_1), \ldots, X_n, (l_n, g_n, \sigma, p_n)$, such that $p_i(X_{i+1}, l_{i+1}) > 0$ for $0 \leqslant i < n$, and $p_n(X_0, l_0) > 0$, there exists a clock $x \in \mathcal{X}$ and $0 \leqslant i, j \leqslant n$ such that $x \in X_i$ and $g_j \Rightarrow (x \geqslant 1)$ (that is, g_j contains a conjunct of the form $x \geqslant c$ or $x > c$ for some $c \geqslant 1$).

We recall the definition of the equivalence relation on clock valuations used to define the region graph of (non-probabilistic) timed automata [ALU 94]. For a real number $q \in \mathbb{R}$, let $\lfloor q \rfloor$ be the integer part of q.

DEFINITION 8.9.– *Let $c \in \mathbb{N}$ be a natural number and let $\mathcal{X}$ be a set of clocks. The* clock equivalence (with respect to c) *over $\mathcal{X}$ is defined as the relation $\equiv_c \subseteq \mathbb{R}^{\mathcal{X}} \times \mathbb{R}^{\mathcal{X}}$ over clock valuations, where $v \equiv_c v'$ if and only if:*

– for each clock $x \in \mathcal{X}$, either both $v(x) > c$ and $v(x') > c$ or:

- $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ *(clock values have the same integer part)*

- $v(x) - \lfloor v(x) \rfloor = 0$ *if and only if* $v(x') - \lfloor v(x') \rfloor = 0$ *(the fractional parts of the clocks' values are either all zero or all positive);*

– for each pair of clocks $x, y \in \mathcal{X}$, either $\lfloor v(x) - v(y) \rfloor = \lfloor v'(x) - v'(y) \rfloor$ (the integer part of the difference between clocks values is the same) or both $v(x) - v(y) > c$ and $v'(x) - v'(y) > c$.

Note that the final point of the definition implies that the ordering on the fractional parts of the clock values of x and y is the same, unless the difference between the clocks is above c. For example, in the case of $\mathcal{X} = \{x_1, x_2, x_3\}$, the clock valuations v and v', where $v(x_1) = 0.1$, $v(x_2) = 0.25$ and $v(x_3) = 0.8$, and $v'(x_1) = 0.5$, $v'(x_2) = 0.9$ and $v'(x_3) = 0.95$, are clock equivalent, but v'', where $v''(x_1) = 0.3$, $v''(x_2) = 0.75$ and $v''(x_3) = 0.6$, is not clock equivalent to v and v'. An example of the partition that clock equivalence induces on a space of valuations of two clocks x_1 and x_2 is shown in Figure 8.2, where c equals 2; each vertical, horizontal or diagonal line segment, open area and point of intersection of lines is a distinct clock equivalence class. Clock equivalence classes can be interpreted as zones: for example, the clock equivalence class α denoted in Figure 8.2 corresponds to the zone $(1 < x < 2) \wedge (1 < y < 2) \wedge (x > y)$, while the clock equivalence class β corresponds to the zone $(x = 1) \wedge (1 < y < 2)$.

We define the *time successor* of a clock equivalence class α to be the first distinct clock equivalence class reached from α by letting time pass. Formally, the time successor β of α is the clock equivalence class for which $\alpha \neq \beta$ and, for all $v \in \alpha$,

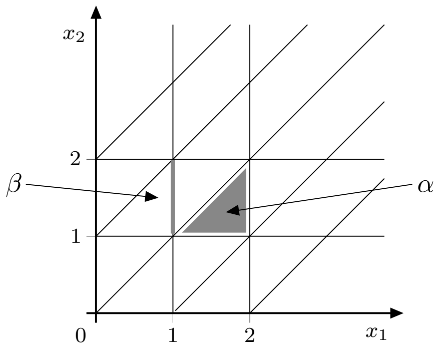


Figure 8.2. *The partition induced by clock equivalence with* $c = 2$

there exists $t \in \mathbb{R}$ such that $v + t \in \beta$ and $v + t' \in \alpha \cup \beta$ for all $0 \leqslant t' \leqslant t$. A clock equivalence class α is *unbounded* if, for all $v \in \alpha$ and $t \in \mathbb{R}$, we have $v + t \in \alpha$; in the case of an unbounded clock equivalence class α, we let the time successor of α be α itself. Because clock equivalence classes can be regarded as zones, for a clock equivalence class α and clock set $X \subseteq \mathcal{X}$, we use the notation $\alpha[X := 0]$ to denote the set $\{v[X := 0] \mid v \in \alpha\}$, as for zones. For a zone $\zeta \in \mathit{Zones}(\mathcal{X})$, we write $\alpha \lhd \zeta$ to denote that all clock valuations in α satisfy the zone ζ.

For the remainder of this section, assume that the probabilistic timed automaton $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, \mathit{inv}, \mathit{prob})$ is fixed. Let $c^{\mathsf{PTA}}_{\max}$ be the maximum constant against which any clock is compared in the invariant conditions and guards of PTA. A *region* (l, α) of PTA is a pair comprising a location $l \in L$ and a class α of the clock equivalence $\equiv_{c^{\mathsf{PTA}}_{\max}}$. Since the set of clock equivalence classes is finite, and therefore the number of regions is finite, we can define a finite-state Markov decision process, the states of which are regions, and the transitions of which are derived from the time constraints and probabilistic edges of the probabilistic timed automaton.

DEFINITION 8.10.– *The* region graph of $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, \mathit{inv}, \mathit{prob})$ *is a finite-state Markov decision process* $\mathsf{MDP} = (R, \bar{r}, \mathit{Steps}_R)$*, where R is the set of all regions of* PTA*, the initial region $\bar{r}$ is $(\bar{l}, \{\mathbf{0}\})$, and the probabilistic transition relation $\mathit{Steps}_R \subseteq R \times \mathsf{Dist}(R)$ is the smallest relation such that $((l, \alpha), \rho) \in \mathit{Steps}_R$ if either of the following conditions hold:*

time transitions: $\rho = \mu_{(l,\beta)}$*, where β is such that $\beta \lhd \mathit{inv}(l)$, and β is the time successor of α;*

discrete transitions: *there exists a probabilistic edge $(l, g, \sigma, p) \in \mathit{prob}$ such that $\alpha \lhd g$ and, for any region $(l', \beta) \in R$, we have:*

$$\rho(l', \beta) = \sum_{\substack{X \subseteq \mathcal{X}\ \& \\ \beta = \alpha[X:=0]}} p(X, l').$$

The following proposition states the correctness of the region graph for calculating minimum and maximum reachability probabilities.

PROPOSITION 8.1.– *For any probabilistic timed automaton* PTA *and corresponding timed Markov decision process* $[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+} = (S, \bar{s}, \Sigma, \mathbb{R}, \mathit{Steps})$*, if* $\mathsf{MDP} = (R, \bar{r}, \mathit{Steps}_R)$ *is the region graph of* PTA *and* T_R *is a set of target regions, then for any* $s \in S$:

$$p^{\max}_{[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+}}(s, \Diamond\, T) = p^{\max}_{\mathsf{MDP}}(r_s, \Diamond\, T_R) \quad \textit{and} \quad p^{\min}_{[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+}}(s, \Diamond\, T) = p^{\min}_{\mathsf{MDP}}(r_s, \Diamond\, T_R)$$

where if $s = (l, v)$*, then* $r_s = (l, \alpha) \in R$ *such that* $v \in \alpha$*, and* $T = \{(l, v) \mid r_{(l,v)} \in T_R\}$.

EXAMPLE 8.2.– *We now consider the* PTA *in Example 8.1 (see Figure 8.1) and use the region graph and Proposition 8.1 to calculate the minimum and maximum probability of reaching the location* sr *within* 6 *time units. Following the approach of [KWI 02a], we add an additional clock* z *to the automaton and compute the probability of reaching the target set of regions* $T_R = \{(\mathsf{sr}, \alpha) \mid \alpha \lhd (z < 6)\}$*. The Markov decision process representing the region graph has 115 states and 125 transitions. In Figure 8.3 we present a fragment of this Markov decision process. Transitions without probability labels correspond to probability 1. Computing the probability of reaching the target set* T_R *on the region graph, we find that the minimum and maximum probabilities of reaching the location* sr *within* 6 *time units equal 0.995 and 0.99525 respectively.*

8.3.2. *Forward symbolic approach*

In this section we consider an approach to approximating the maximum reachability probability based on a forward symbolic analysis. This approach allows us to compute an upper bound on the maximum probability of reaching a set of target locations. Before introducing the algorithm, we present a number concepts for representing and manipulating zones and state sets of a probabilistic timed automaton.

In this section and in section 8.3.3, we assume that probabilistic timed automata allow time to diverge with probability 1 from each state. That is, for each probabilistic timed automaton PTA, we assume that there exists a divergent adversary of $[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+}$.

Proofs of the key results presented in this section are available in [KWI 02a].

8.3.2.1. *Symbolic state operations*

In order to represent symbolically the state sets computed during the analysis, we use, both in this section and section 8.3.3, the concept of *symbolic state*. A symbolic state is a pair (l, ζ) comprising a location and a zone over the clocks of the probabilistic timed automaton under study. The set of states corresponding to a symbolic state (l, ζ) is $\{(l, v) \mid v \lhd \zeta\}$, while the state set corresponding to a set of symbolic states is the union of those corresponding to each individual symbolic state. Symbolic states are denoted by $\mathtt{u}, \mathtt{v}, \mathtt{z}, \ldots$ and sets of symbolic states by $\mathtt{U}, \mathtt{V}, \mathtt{Z}, \ldots$.

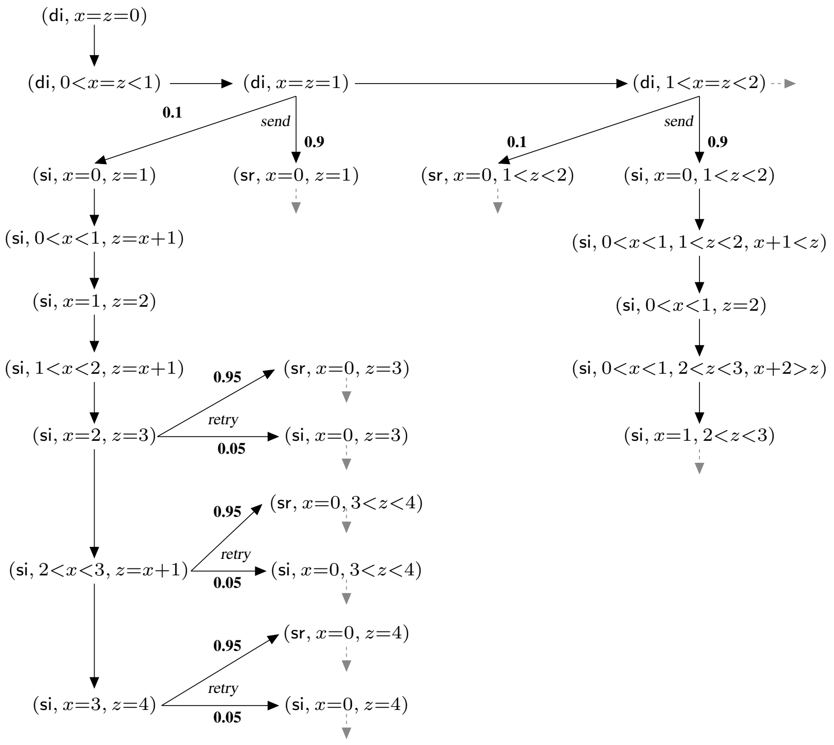


Figure 8.3. *Fragment of the region graph model of Example 8.1*

Consider a probabilistic timed automaton $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$. Our aim of forward exploration through the state space of PTA requires operations to return the successor states of all of the states in a particular set (where the set is represented as a symbolic state). More precisely, we introduce a *discrete successor* operation dpost which, given an edge $e = (l, g, \sigma, p, X, l')$ of PTA and a symbolic state (l, ζ), returns all of the states obtained by traversing e from a state in (l, ζ). Similarly, the *time successor* operation tpost computes, for the symbolic state (l, ζ), the set of states which can be reached from a state in (l, ζ) by letting time elapse. These two operations are then composed to define a generalized successor operation post. For a given symbolic state (l, ζ) and an edge $e = (l, g, \sigma, p, X, l') \in$ edges, the post operation returns the set of states that can be obtained from (l, ζ) by traversing the edge e and then letting time elapse. Note that we also parametrize post by an integer $c \in \mathbb{N}$, and only compute the c-closure of zones obtained by the time successor operation; this is to ensure the termination of the forward algorithm. For a symbolic state (l, ζ), edge $e = (l, g, \sigma, p, X, l')$

and constant $c \in \mathbb{N}$:

$$\begin{aligned}
\mathsf{tpost}(l,\zeta) &\overset{\text{def}}{=} (l, \nearrow_{inv(l)} \zeta)\\
\mathsf{dpost}(e,(l,\zeta)) &\overset{\text{def}}{=} (l', ((\zeta \wedge g)[X := 0]) \wedge inv(l'))\\
\mathsf{close}(l,\zeta,c) &\overset{\text{def}}{=} (l, \mathsf{close}(\zeta,c))\\
\mathsf{post}[e,c](l,\zeta) &\overset{\text{def}}{=} \mathsf{close}(\mathsf{tpost}(\mathsf{dpost}(e,(l,\zeta))),c)\ .
\end{aligned}$$

8.3.2.2. *Computing maximum reachability probabilities*

An algorithm for generating a finite representation of the state space of a probabilistic timed automaton PTA for a given set of target set T_L of locations is presented in Figure 8.4. Recall from section 8.3.1 that $c^{\mathsf{PTA}}_{\max}$ denotes the maximum constant against which any clock is compared in the invariant conditions and guards of PTA. The algorithm returns a Markov decision process $\mathsf{MDP} = (\mathsf{Z}, \bar{\mathsf{z}}, Steps)$, called the *forwards zone graph*, and a set of target states $Reached \subseteq \mathsf{Z}$.

The algorithm consists of two distinct computation steps: firstly, the generation of symbolic states Z that encode states of PTA reachable from the initial state with non-zero probability, and secondly, construction of the forwards zone graph generated from Z and the probabilistic edges of PTA. As in the case of similar algorithms in the non-probabilistic context [DAW 96, LAR 97], the algorithm searches forward through a reachable portion of the state space of the system by iterating the transition relation a finite number of times. Any symbolic state (l,ζ) such that $l \in T_L$ (i.e. where the target has been reached) is not explored, but instead added to the set $Reached$. Since, for a given $c \in \mathbb{N}$, there are a finite number of c-closed zones [DAW 98, TRI 98] and the number of locations is finite, the forwards zone graph generated by Figure 8.4 is guaranteed to be finite. This also implies the termination of the algorithm.

Finally, we can then obtain an upper bound on the maximum probability of reaching the target set T_L of locations as the maximum probability of reaching the target set $Reached$ in the forward zone graph. If $Reached = \emptyset$, then the forward search through the reachable state space has found that no location in T_L is reachable with positive probability, and therefore we conclude that the maximum probability is 0. The correctness of the algorithm follows from the following proposition.

PROPOSITION 8.2.– *Let* $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$ *be a probabilistic timed automaton and* T_L *be a set of target locations. If the algorithm* $\mathsf{ForwReach}(T_L)$ *returns* $\mathsf{MDP} = (\mathsf{Z}, \bar{\mathsf{z}}, Steps)$ *and Reached, then:*

$$p^{\max}_{[\![\mathsf{PTA}]\!]^{+}_{\mathbb{R}}}((\bar{l},\mathbf{0}), \Diamond\, T_L) \leqslant p^{\max}_{\mathsf{MDP}}(\bar{\mathsf{z}}, \Diamond\, Reached).$$

```
algorithm ForwReach($T_L$)

1.  $\mathsf{Z} := \emptyset$
2.  $\mathit{Reached} := \emptyset$
3.  $\bar{\mathsf{z}} := \mathsf{close}(\mathsf{tpost}(\bar{l}, \mathbf{0}), c^{\mathsf{PTA}}_{\max})$
4.  $\mathit{Fringe} := \{\bar{\mathsf{z}}\}$
5.  repeat
6.    choose $(l, \zeta) \in \mathit{Fringe}$
7.    $\mathit{Fringe} := \mathit{Fringe} \setminus \{(l, \zeta)\}$
8.    for each $(l, g, \sigma, p, l', X) \in \mathsf{edges}$ do
9.      let $(l', \zeta') := \mathsf{post}[(l, g, \sigma, p, l', X), c^{\mathsf{PTA}}_{\max}](l, \zeta)$
10.     if $\zeta' \neq \emptyset \wedge (l', \zeta') \notin \mathsf{Z} \wedge l' \in T_L$ then
11.       $\mathit{Reached} := \mathit{Reached} \cup \{(l', \zeta')\}$
12.     else if $\zeta' \neq \emptyset \wedge (l', \zeta') \notin \mathsf{Z}$
13.       $\mathit{Fringe} := \mathit{Fringe} \cup \{(l', \zeta')\}$
14.     end if
15.   end for each
16.   $\mathsf{Z} := \mathsf{Z} \cup \{(l, \zeta)\}$
17. until $\mathit{Fringe} = \emptyset$
18. construct $\mathsf{MDP} = (\mathsf{Z}, \bar{\mathsf{z}}, \mathit{Steps}_{\mathsf{Z}})$ where $((l, \zeta), \rho) \in \mathit{Steps}_{\mathsf{Z}}$ if and only if
      there exists $(l, g, \sigma, p) \in \mathit{prob}$ such that
        $- \zeta \cap g \neq \emptyset$
        – for any $(l', \zeta') \in \mathsf{Z}$: $\rho(l', \zeta')$ equals
            $\sum \{\!| \, p(X, l') \mid \mathsf{post}[(l, g, \sigma, p, l', X), c^{\mathsf{PTA}}_{\max}](l, \zeta) = (l', \zeta') \, |\!\}$
19. return $(\mathsf{MDP}, \mathit{Reached})$
```

Figure 8.4. *Algorithm* ForwReach(T_L) *for* $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, \mathit{inv}, \mathit{prob})$

Unfortunately, the probability obtained via this approach may be greater than the probability of reaching the target set via *any* adversary of the probabilistic timed automaton. This is illustrated by the example below.

EXAMPLE 8.3.– *Consider the probabilistic timed automaton presented in Figure 8.5 and suppose that the target set of interest consists of the single location l. There are only two classes of adversary with a non-zero probability of reaching location l: one in which the initial state is left when* $x = y = 0$*, and the other when the initial state is left when* $x = y = 1$*. In the former case, if the left-hand edge to location* l_1 *is taken, then the outgoing edge of* l_1 *can never be taken, and so we must remain in this location; however, if the right-hand edge to* l_2 *is taken, then the outgoing edge to l can be selected immediately. The case for the selection of the transition of* $\bar{l}$ *when* $x = y = 1$ *is symmetric. Therefore, for each of the two classes of adversaries, the probability of reaching the target set* $\{l\}$ *is 0.5. However, for the forward zone graph and set Reached generated by* ForwReach$(\{l\})$*, as shown in Figure 8.5, the corresponding reachability probability is 1.*

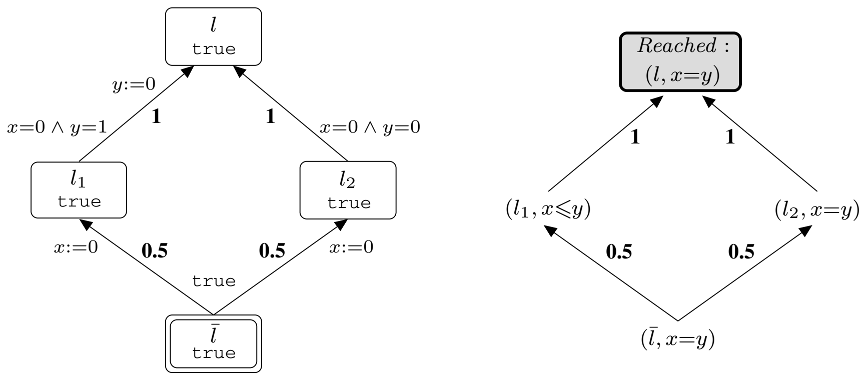
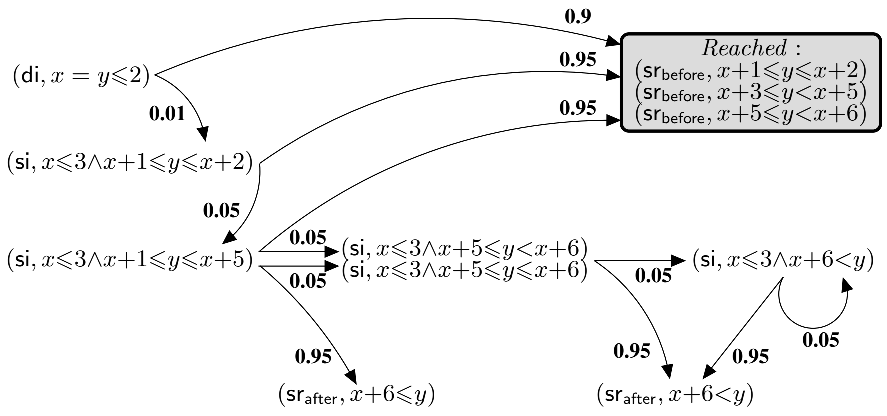


Figure 8.5. *Example demonstrating that* ForwReach *yields only upper bounds*

Recall that invariance properties can be stated in terms of reachability properties. The fact that we obtain upper bounds on maximum reachability probabilities implies that we obtain lower bounds on the minimum probability of invariance, and hence the forward reachability method is particularly appropriate for establishing the satisfaction of probabilistic, real-time invariance properties.

One important difference between the algorithm in Figure 8.4 and analogous algorithms in the non-probabilistic, real-time studies is the fact that the *on-the-fly* property of the latter algorithms is compromised in our context. This property refers to the fact that if a symbolic state which reaches the target set is computed, then the algorithm can terminate immediately with a "YES" answer to the reachability problem. Such a strategy is insufficient for probabilistic timed automata. For example, consider the case in which we have found a path of symbolic states reaching the target set T_L, and which corresponds to the probability λ. Then it may be possible to find another path to T_L, thus increasing the probability of reaching this target set.

EXAMPLE 8.4.– *Returning to Example 8.1 (see Figure 8.1), we now calculate the maximum probability of reaching the location* sr *within* 6 *time units. Following the approach of [KWI 02a], we add a distinct clock z to the automaton, and split the location* sr *into locations* $\mathsf{sr}_{\mathsf{before}}$ *and* $\mathsf{sr}_{\mathsf{after}}$. *Furthermore, we replace each probabilistic edge which has* sr *as a target location with two probabilistic edges where* sr *is replaced with* $\mathsf{sr}_{\mathsf{before}}$ *or* $\mathsf{sr}_{\mathsf{after}}$, *and where $z < 6$ or $z \geqslant 6$, respectively, is added to the guard. Applying* $\mathsf{ForwReach}(\{\mathsf{sr}_{\mathsf{before}}\})$ *to this automaton returns the Markov decision process given in Figure 8.6. From Proposition 8.3, it follows that the maximum probability of reaching the location* sr *within 6 time units is bounded above by 0.99525, corresponding to the maximum probability of reaching Reached from* $\bar{\mathsf{z}} = (\mathsf{di}, x = y \leqslant 2)$ *in the Markov decision process given in Figure 8.6. In fact, in this case, the computed bound equals the actual probability. Intuitively, this is because there is no conflict between*



$(\mathsf{di}, x = y \leqslant 2)$
$(\mathsf{si}, x \leqslant 3 \wedge x+1 \leqslant y \leqslant x+2)$
$(\mathsf{si}, x \leqslant 3 \wedge x+1 \leqslant y \leqslant x+5)$
$(\mathsf{si}, x \leqslant 3 \wedge x+5 \leqslant y < x+6)$
$(\mathsf{si}, x \leqslant 3 \wedge x+5 \leqslant y \leqslant x+6)$
$(\mathsf{si}, x \leqslant 3 \wedge x+6 < y)$
$(\mathsf{sr}_{\mathsf{after}}, x+6 \leqslant y)$
$(\mathsf{sr}_{\mathsf{after}}, x+6 < y)$
$Reached:$
$(\mathsf{sr}_{\mathsf{before}}, x+1 \leqslant y \leqslant x+2)$
$(\mathsf{sr}_{\mathsf{before}}, x+3 \leqslant y < x+5)$
$(\mathsf{sr}_{\mathsf{before}}, x+5 \leqslant y < x+6)$
0.9
0.95
0.95
0.01
0.05
0.05
0.05
0.05
0.95
0.95
0.95
0.05

Figure 8.6. *Markov decision process generated by* $\mathsf{ForwReach}(\{\mathsf{sr}_{\mathsf{before}}\})$

letting time pass on different edges: the maximum probability is obtained by always taking a discrete transition as soon as it is available.

8.3.3. *Backward symbolic approach*

In this section we consider an approach for computing both the maximum and minimum reachability probability based on a backwards symbolic analysis. Unlike the forward approach in the previous section, this technique yields exact results. Furthermore, as in the region graph approach, it computes the probabilities for all states of the probabilistic timed automaton rather than for a fixed initial state.

Proofs of the key results, and specialized algorithms for *qualitative* properties (finding states that have probability 0 or 1 of reaching the target set), for which verification can be performed through an analysis of the underlying graph, are available in [KWI 07]. In fact, [KWI 07] presents algorithms for model checking for the probabilistic timed temporal logic PTCTL [KWI 02a], which subsumes reachability properties.

8.3.3.1. *Symbolic state operations*

Consider a probabilistic timed automaton $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$. The backward exploration through the state space of PTA requires operations to return the predecessors of a set of symbolic states. More precisely, we introduce a *discrete predecessor* operation dpre which, given an edge e of PTA and a set of symbolic states V, returns the set of symbolic states that encodes the set of states which, when edge e is traversed, belongs to the set encoded by V. The *time predecessor* operation $\mathsf{tpre}_{\mathtt{U}}$ is parametrized by a set of symbolic states U, and computes, for a set of symbolic states V, the set of symbolic states encoding states that can reach V by letting time elapse and remaining in U at all intermediate times.

Formally, we extended the *time predecessor* and *discrete predecessor* functions tpre and dpre of [HEN 94, TRI 98] to probabilistic timed automata as follows. For any sets of symbolic states $\mathtt{U}, \mathtt{V}$ and edge $e = (l, g, \sigma, p, X, l')$:

$$\mathsf{tpre}_{\mathtt{U}}(\mathtt{V}) \overset{\text{def}}{=} \left\{ \, (l, \swarrow_{\zeta^l_{\mathtt{U}} \wedge inv(l)} (\zeta^l_{\mathtt{V}} \wedge inv(l))) \; \middle| \; l \in L \, \right\}$$

$$\mathsf{dpre}(e, \mathtt{U}) \overset{\text{def}}{=} \left\{ \, (l, g \wedge inv(l) \wedge ([X := 0]\zeta^{l'}_{\mathtt{U}})) \, \right\} .$$

where for any set $\mathtt{U}$ of symbolic states $\zeta^l_{\mathtt{U}} \overset{\text{def}}{=} \bigvee\{\zeta \mid (l, \zeta) \in \mathtt{U}\}$; that is, $\zeta^l_{\mathtt{U}}$ is the zone such that $v \triangleleft \zeta^l_{\mathtt{U}}$ if and only if $(l, v) \in \mathtt{u}$ for some $\mathtt{u} \in \mathtt{U}$.

We also extend clock reset, conjunction and disjunction operations to sets of symbolic states as follows. For any clock x and sets of symbolic states $\mathtt{U}, \mathtt{V}$:

$$x \cdot \mathtt{U} \overset{\text{def}}{=} \left\{ \, (l, [\{x\} := 0]\zeta^l_{\mathtt{U}}) \; \middle| \; l \in L \, \right\}$$

$$\mathtt{U} \wedge \mathtt{V} \overset{\text{def}}{=} \left\{ \, (l, \zeta^l_{\mathtt{U}} \wedge \zeta^l_{\mathtt{V}}) \; \middle| \; l \in L \, \right\}$$

$$\mathtt{U} \vee \mathtt{V} \overset{\text{def}}{=} \left\{ \, (l, \zeta^l_{\mathtt{U}} \vee \zeta^l_{\mathtt{V}}) \; \middle| \; l \in L \, \right\} .$$

Finally, let $[\![\mathtt{false}]\!] \overset{\text{def}}{=} \emptyset$ and $[\![\mathtt{true}]\!] \overset{\text{def}}{=} \{(l, inv(l)) \mid l \in L\}$ be the sets of symbolic states representing the empty and full state sets respectively. In addition, for any $\zeta \in Zones(\mathcal{X})$, let $[\![\zeta]\!] = \{(l, \zeta \wedge inv(l)) \mid l \in L\}$.

8.3.3.2. *Probabilistic until*

The algorithm for computing minimum reachability probabilities (which will be described in section 8.3.3.4) relies on the computation of (maximum) probabilities for two classes of properties: *probabilistic invariance* (see section 8.2.5) and *probabilistic until*. Probabilistic until corresponds to the probability of reaching a certain set of target states while remaining in another set of states until the target set is reached. For a timed Markov decision process $\mathsf{TMDP} = (S, \bar{s}, \Sigma, \mathbb{T}, Steps)$, state $s \in S$, sets $U, V \subseteq S$ of states, and adversary $A \in Adv_{\mathsf{TMDP}}$:

$$p^A_{\mathsf{TMDP}}(s, U \; \mathcal{U} \; V)$$
$$\overset{\text{def}}{=} Prob^A_s\{\omega \in Path^A(\bar{s}) \mid \exists i \in \mathbb{N} \cdot (\omega(i) \in V \wedge \forall j < i \cdot \omega(j) \in U)\} .$$

Note that probabilistic reachability is a special case of probabilistic until:

$$p^A_{\mathsf{TMDP}}(s, \Diamond \, V) = p^A_{\mathsf{TMDP}}(s, S \; \mathcal{U} \; V) .$$

DEFINITION 8.11.– *Let* $\mathsf{TMDP} = (S, \bar{s}, \Sigma, \mathbb{T}, Steps)$ *be a timed Markov decision process. The* maximum and minimum until probabilities *of, from a state* $s \in S,$

reaching a set of states V while remaining in a set of states U up until this point are defined as follows:

$$p^{\max}_{\mathsf{TMDP}}(s, U\ \mathcal{U}\ V) = \sup_{A \in Adv_{\mathsf{TMDP}}} p^{A}_{\mathsf{TMDP}}(s, U\ \mathcal{U}\ V)$$

$$p^{\min}_{\mathsf{TMDP}}(s, U\ \mathcal{U}\ V) = \inf_{A \in Adv_{\mathsf{TMDP}}} p^{A}_{\mathsf{TMDP}}(s, U\ \mathcal{U}\ V) .$$

8.3.3.3. *Computing maximum reachability probabilities*

In this section we present methods for calculating maximum reachability probabilities. In fact, we present a more general method which can compute maximum until probabilities, as introduced in the previous section.

In the case of computing maximum until probabilities and, in particular maximum reachability probabilities, we use the algorithm MaxU given in Figure 8.7 and adapted from [KWI 01]. The algorithm iteratively applies time predecessor, discrete predecessor and conjunction operations on symbolic states until a fixpoint is reached. The key observation is that to preserve probabilistic branching we must take the conjunctions of symbolic states generated by edges from the same distribution. More precisely, we need to identify the state sets from which *multiple edges* within the support of the *same distribution* of the probabilistic timed automaton can be used to reach previously generated state sets. Upon termination of the fixpoint algorithm, the set of generated symbolic states is used to construct a finite-state Markov decision process which has sufficient information to compute the maximum probability of interest.

We now explain the algorithm $\mathsf{MaxU}(\mathtt{U}, \mathtt{V})$ in more detail. Lines 1–4 deal with the initialization of $\mathtt{Z}$, which is set equal to the set of time predecessors of $\mathtt{V}$, and the set of edges $E_{(l,g,\sigma,p)}$ associated with each probabilistic edge $(l, g, \sigma, p) \in prob$. Lines 5–20 generate a finite-state graph, the nodes of which are symbolic states, obtained by iterating timed and discrete predecessor operations (line 8), and taking conjunctions (lines 12–16). The edges of the graph are partitioned into the sets $E_{(l,g,\sigma,p)}$ for $(l, g, \sigma, p) \in prob$, where any $(\mathsf{z}, (X, l'), \mathsf{z}') \in E_{(l,g,\sigma,p)}$ corresponds to a transition from any state in the symbolic state z to some state in the symbolic state z' when the outcome (X, l') of the probabilistic edge (l, g, σ, p) is chosen. The graph edges are added in line 11. Line 20 describes the manner in which the probabilistic edges of the probabilistic timed automaton are used in combination with the computed edge sets to construct the Markov decision process MDP. The states of MDP are the symbolic states generated by the previous steps of the algorithm, and the probabilistic transition relation of MDP is constructed by grouping the graph edges generated by the same probabilistic edge of the probabilistic timed automaton under study. The initial state of MDP is irrelevant, and therefore is omitted from the notation.

The following proposition states the correctness of our algorithm for calculating maximum until probabilities.

```
algorithm $\mathsf{MaxU}(\mathtt{U},\mathtt{V})$

1.  $\mathtt{Z} := \mathsf{tpre}_{\mathtt{U}\vee\mathtt{V}}(\mathtt{V})$
2.  for $(l,g,\sigma,p) \in prob$
3.    $E_{(l,g,\sigma,p)} := \emptyset$
4.  end for
5.  repeat
6.    $\mathtt{Y} := \mathtt{Z}$
7.    for $\mathtt{y} \in \mathtt{Y} \wedge (l,g,\sigma,p) \in prob \wedge e = (l,g,\sigma,p,X,l') \in \mathsf{edges}(l,g,\sigma,p)$
8.      $\mathtt{z} := \mathtt{U} \wedge \mathsf{dpre}(e, \mathsf{tpre}_{\mathtt{U}\vee\mathtt{V}}(\mathtt{y}))$
9.      if $(\mathtt{z} \neq \emptyset) \wedge (\mathtt{z} \notin \mathsf{tpre}_{\mathtt{U}\vee\mathtt{V}}(\mathtt{V}))$
10.       $\mathtt{Z} := \mathtt{Z} \cup \{\mathtt{z}\}$
11.       $E_{(l,g,\sigma,p)} := E_{(l,g,\sigma,p)} \cup \{(\mathtt{z},(X,l'),\mathtt{y})\}$
12.       for $(\bar{\mathtt{z}},(\bar{X},\bar{l}'),\bar{\mathtt{y}}) \in E_{(l,g,\sigma,p)}$
13.         if $(\mathtt{z} \wedge \bar{\mathtt{z}} \neq \emptyset) \wedge ((\bar{X},\bar{l}') \neq (X,l')) \wedge (\mathtt{z} \wedge \bar{\mathtt{z}} \notin \mathsf{tpre}_{\mathtt{U}\vee\mathtt{V}}(\mathtt{V}))$
14.           $\mathtt{Z} := \mathtt{Z} \cup \{\mathtt{z} \wedge \bar{\mathtt{z}}\}$
15.         end if
16.       end for
17.     end if
18.   end for
19. until $\mathtt{Z} = \mathtt{Y}$
20. construct $\mathsf{MDP} = (\mathtt{Z}, Steps_{\mathtt{Z}})$ where $(\mathtt{z},\rho) \in Steps_{\mathtt{Z}}$ if and only if
        there exists $(l,g,\sigma,p) \in prob$ and $E \subseteq E_{(l,g,\sigma,p)}$ such that
          – $\mathsf{z} \in \{\mathsf{z}' \mid (\mathsf{z}',\mathsf{e},\mathsf{z}'') \in E\}$
          – $(\mathsf{z}',\mathsf{e},\mathsf{z}'') \in E \;\Rightarrow\; \mathsf{z}' \supseteq \mathsf{z}$
          – $(\mathsf{z}'_1,\mathsf{e},\mathsf{z}') \neq (\mathsf{z}'_2,\mathsf{e}',\mathsf{z}'') \in E \;\Rightarrow\; \mathsf{e} \neq \mathsf{e}'$
          – $E$ is maximal
          – $\rho(\mathsf{z}') = \sum\{\!|\, p(X,l') \mid (\mathsf{z},(X,l'),\mathsf{z}') \in E \,|\!\}\;\; \forall \mathsf{z}' \in \mathtt{Z}$
21. return MDP
```

Figure 8.7. *Algorithm* $\mathsf{MaxU}(\mathtt{U},\mathtt{V})$ *for* $\mathsf{PTA} = (L,\bar{l},\mathcal{X},\Sigma,inv,prob)$

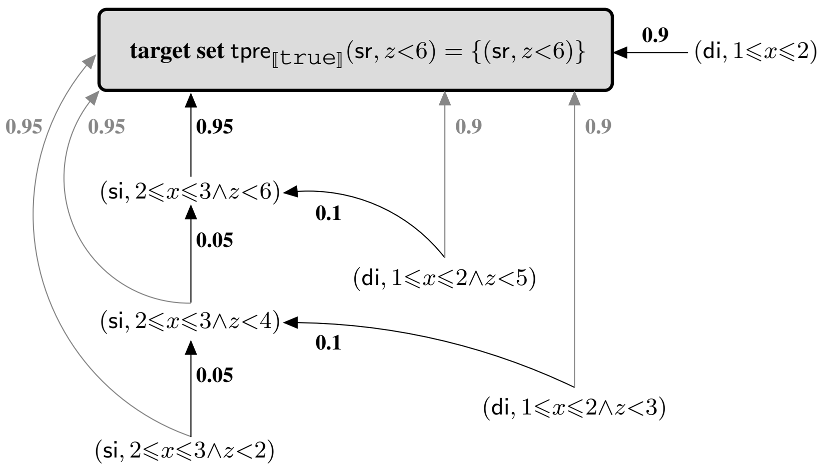
PROPOSITION 8.3.– *For any probabilistic timed automaton* PTA*, corresponding timed Markov decision process* $[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+} = (S,\bar{s},\Sigma,\mathbb{R},Steps)$ *and sets of symbolic states* $\mathtt{U}$ *and* $\mathtt{V}$*, if* $\mathsf{MDP} = (\mathtt{Z}, Steps_{\mathtt{Z}})$ *is the Markov decision process generated by* $\mathsf{MaxU}(\mathtt{U},\mathtt{V})$*, then for any* $s \in S$:

– $p^{\max}_{[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+}}(s, \mathtt{U}\;\mathcal{U}\;\mathtt{V}) > 0$ *if and only if* $s \in \mathsf{tpre}_{\mathtt{U}\vee\mathtt{V}}(\mathtt{Z})$*;*

– *if* $p^{\max}_{[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+}}(s, \mathtt{U}\;\mathcal{U}\;\mathtt{V}) > 0$*, then*

$$p^{\max}_{[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+}}(s, \mathtt{U}\;\mathcal{U}\;\mathtt{V}) = \max\left\{p^{\max}_{\mathsf{MDP}}(\mathtt{z}, \Diamond\, \mathsf{tpre}_{\mathtt{U}\vee\mathtt{V}}(\mathtt{V})) \;|\, \mathtt{z} \in \mathtt{Z} \wedge s \in \mathsf{tpre}_{\mathtt{U}\vee\mathtt{V}}(\mathtt{z})\right\}.$$

EXAMPLE 8.5.– *We now return to the probabilistic timed automaton in Example 8.1 (see Figure 8.1) and repeat the calculation from Example 8.4, i.e. the maximum probability of reaching the location* sr *within* 6 *time units. We therefore add a distinct*



target set $\mathsf{tpre}_{[\![\mathtt{true}]\!]}(\mathsf{sr}, z{<}6) = \{(\mathsf{sr}, z{<}6)\}$
0.9
$(\mathsf{di}, 1{\leqslant}x{\leqslant}2)$
0.95
0.95
0.95
0.9
0.9
$(\mathsf{si}, 2{\leqslant}x{\leqslant}3{\wedge}z{<}6)$
0.1
0.05
$(\mathsf{di}, 1{\leqslant}x{\leqslant}2{\wedge}z{<}5)$
$(\mathsf{si}, 2{\leqslant}x{\leqslant}3{\wedge}z{<}4)$
0.1
0.05
$(\mathsf{di}, 1{\leqslant}x{\leqslant}2{\wedge}z{<}3)$
$(\mathsf{si}, 2{\leqslant}x{\leqslant}3{\wedge}z{<}2)$

Figure 8.8. *Markov decision process generated by* $\mathsf{MaxU}([\![\mathtt{true}]\!], (\mathsf{sr}, z < 6))$

clock z *to the automaton and consider the probability of reaching the symbolic state* $(\mathsf{sr}, z < 6)$. *Applying* $\mathsf{MaxU}([\![\mathtt{true}]\!], \{(\mathsf{sr}, z < 6)\})$ *returns the Markov decision process given in Figure 8.8 where the darker arrows correspond to those edges generated by time and discrete predecessor operations (line 11 of Figure 8.7) and the lighter arrows are those generated in the construction of the Markov decision process (line 20 of Figure 8.7). From Proposition 8.3 it follows that, starting from the initial state, the maximum probability of reaching location* sr *within* 6 *time units is 0.99525, corresponding to the maximum probability of* $(\mathsf{di}, 1 \leqslant x \leqslant 2 \wedge z < 3)$ *reaching the target set in the Markov decision process given in Figure 8.8.*

8.3.3.4. *Computing minimum reachability probabilities*

As in the cases of (non-probabilistic) timed automata and (finite-state) Markov decision processes with fairness constraints, when considering properties which have universal quantification over paths or require the computation of minimum probabilities, standard algorithms can no longer be applied. For example, under divergent adversaries the minimum probability of letting one time unit elapse is 1; however, if we remove the restriction from time-divergent adversaries, this minimum probability becomes 0.

The results presented below are an extension of the results for (non-probabilistic) timed automata [HEN 94] which show that verifying $\exists\Box\ I$ ("there exists a divergent path for which all states along the paths are in the set I") reduces to computing the greatest fixpoint and, for certain sets of states U and V, computing the set of states satisfying $U\ \exists\mathcal{U}\ V$ ("there exists a divergent path for which a state in V is reached and it remains in the state U until a state in V is reached"). More precisely, for any set of symbolic states $\mathtt{I}$, the set of states that satisfy $\exists\Box\ \mathtt{I}$ reduces to computing the greatest fixpoint:

$$gfp\mathtt{X}.\big(\mathtt{I} \wedge z.(\mathtt{X}\ \exists\mathcal{U}\ [\![z > c]\!])\big) \tag{8.1}$$

for any $c(> 0) \in \mathbb{N}$ and clock z that does not appear in any of the guards, invariants or resets of the timed automata under study. An important point is that the expression requires that more than c time units elapse repeatedly.

Recall, from the duality between reachability and invariance, for any state s of $[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+}$:

$$p_s^{\min}(\Diamond\, T) = 1 - p_s^{\max}(\Box\ (L \setminus T))\,,$$

and hence, to calculate the minimum probability of reaching a set of target locations, it suffices to calculate the maximum probability of remaining in the states not in the target set, i.e. a maximum invariance probability. Note that, although we have reduced the problem to that of calculating a maximum probability, we cannot ignore time divergence when calculating such probabilities. For example, returning to the probabilistic timed automaton in Example 8.1 and the invariance set $\{(\mathsf{di}, 0 \leqslant x \leqslant 3)\}$, under all adversaries the maximum probability of remaining in this set is 1; however, as we cannot let more that 3 time units pass in di, under divergent adversaries this probability is 0.

Proposition 8.4 shows that we can reduce the computation of the maximum probability of invariance to that of computing the maximum probability of until within which a *qualitative* invariance is nested. As issues of time divergence are irrelevant to the computation of the maximum until probabilities, the proposition allows us to focus our attention on incorporating time divergence when finding the states which have maximum invariance probability 1.

PROPOSITION 8.4.– *For any probabilistic timed automaton* PTA*, corresponding timed Markov decision process* $[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+} = (S, \bar{s}, \Sigma, \mathbb{R}, \mathit{Steps})$*, state* $s \in S$ *and set of symbolic states* $\mathtt{I}$*:*

$$p^{\max}_{[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+}}(s, \Box\ \mathtt{I}) = p^{\max}_{[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+}}(s, \mathtt{I}\ \mathcal{U}\ [\![\Box\ \mathtt{I}]\!]_{=1})$$

where $[\![\Box\ \mathtt{I}]\!]_{=1} = \{s' \mid s' \in S \wedge p^{\max}_{[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+}}(s', \Box\ \mathtt{I}) = 1\}$.

Since we have already introduced an algorithm for calculating maximum until probabilities, it remains to consider a method for calculating the set $[\![\Box\ \mathtt{I}]\!]_{=1}$. Based on (8.1) we obtain the following proposition.

PROPOSITION 8.5.– *For any probabilistic timed automaton* PTA*, corresponding timed Markov decision process* $[\![\mathsf{PTA}]\!]_{\mathbb{R}}^{+} = (S, \bar{s}, \Sigma, \mathbb{R}, \mathit{Steps})$*, set of symbolic states* $\mathtt{I}$*,*

```
algorithm $\mathsf{MaxI}_{\geqslant 1}(c, \mathtt{I})$

$\mathtt{Z} := [\![\mathtt{true}]\!]$
repeat
   $\mathtt{Y} := \mathtt{Z}$
   $\mathtt{Z} := \mathtt{I} \wedge z.\mathsf{MaxU}_{\geqslant 1}(\mathtt{Y}, [\![z>c]\!])$
until $\mathtt{Z} = \mathtt{Y}$
return $\mathtt{Z}$
```

Figure 8.9. *Algorithm* $\mathsf{MaxI}_{\geqslant 1}(c, \mathtt{I})$ *for* $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$

```
algorithm $\mathsf{MaxU}_{\geqslant 1}(\mathtt{U}, \mathtt{V})$

$\mathtt{Z}_0 := [\![\mathtt{true}]\!]$
repeat
   $\mathtt{Y}_0 := \mathtt{Z}_0$
   $\mathtt{Z}_1 := [\![\mathtt{false}]\!]$
   repeat
      $\mathtt{Y}_1 := \mathtt{Z}_1$
      $\mathtt{Z}_1 := \mathtt{V} \vee (\mathtt{U} \wedge \mathsf{dpre1}(\mathtt{Y}_0, \mathtt{Y}_1))$
      $\mathtt{Z}_1 := \mathtt{Z}_1 \vee \mathsf{tpre}_{\mathtt{U}\vee\mathtt{V}}(\mathtt{Y}_0 \wedge \mathtt{Y}_1)$
   until $\mathtt{Z}_1 = \mathtt{Y}_1$
   $\mathtt{Z}_0 := \mathtt{Z}_1$
until $\mathtt{Z}_0 = \mathtt{Y}_0$
return $\mathtt{Z}_0$
```

```
algorithm $\mathsf{dpre1}(\mathtt{U}, \mathtt{V})$

$\mathtt{Y} := [\![\mathtt{false}]\!]$
for $(l, g, \sigma, p) \in prob$
   $\mathtt{Y}_0 := [\![\mathtt{true}]\!]$
   $\mathtt{Y}_1 := [\![\mathtt{false}]\!]$
   for $e \in \mathsf{edges}(l, g, p)$
      $\mathtt{Y}_0 := \mathsf{dpre}(e, \mathtt{U}) \wedge \mathtt{Y}_0$
      $\mathtt{Y}_1 := \mathsf{dpre}(e, \mathtt{V}) \vee \mathtt{Y}_1$
   end
   $\mathtt{Y} := (\mathtt{Y}_0 \wedge \mathtt{Y}_1) \vee \mathtt{Y}$
end
return $\mathtt{Y}$
```

Figure 8.10. *Algorithm* $\mathsf{MaxU}_{\geqslant 1}(\mathtt{U}, \mathtt{V})$ *for* $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$

constant $c(> 0) \in \mathbb{N}$*, if* $z \in \mathcal{X}$ *does not appear in any of the guards or invariant conditions of* PTA*, then the set* $[\![\Box\ \mathtt{I}]\!]_{=1}$ *is given by the greatest fixpoint*

$$gfp\mathtt{X} \cdot \big(\mathtt{I} \wedge z \cdot [\![\, \mathtt{X}\ \mathcal{U}\ [\![z > c]\!]\,]\!]_{=1}\big)$$

where $[\![\mathtt{U}\ \mathcal{U}\ \mathtt{V}]\!]_{=1} = \{s \mid s \in S \wedge p^{\max}_{[\![\mathsf{PTA}]\!]^+_{\mathbb{R}}}(s, \mathtt{U}\ \mathcal{U}\ \mathtt{V}) = 1\}$.

The algorithm $\mathsf{MaxI}_{\geqslant 1}(c, \mathtt{I})$ for calculating the set $[\![\Box\ \mathtt{I}]\!]_{=1}$ follows from Proposition 8.5 and is given in Figure 8.9. The algorithm called $\mathsf{MaxU}_{\geqslant 1}(\mathtt{U}, \mathtt{V})$, given in Figure 8.10, computes the set of states $[\![\mathtt{U}\ \mathcal{U}\ \mathtt{V}]\!]_{=1}$ and is based on a similar algorithm for finite-state Markov decision processes [ALF 97].

EXAMPLE 8.6.– *We now return to the probabilistic timed automaton in Example 8.1 (see Figure 8.1) and compute the* minimum *probability of a message being correctly delivered before 6 time units have elapsed. This is achieved by adding a clock* z *to the*

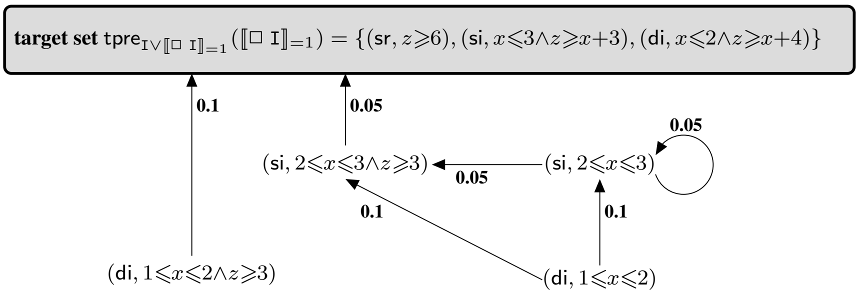

target set $\mathsf{tpre}_{\mathtt{I}\vee[\![\Box\ \mathtt{I}]\!]_{=1}}([\![\Box\ \mathtt{I}]\!]_{=1}) = \{(\mathsf{sr}, z\geqslant 6), (\mathsf{si}, x\leqslant 3\wedge z\geqslant x+3), (\mathsf{di}, x\leqslant 2\wedge z\geqslant x+4)\}$
0.1
0.05
0.05
$(\mathsf{si}, 2\leqslant x\leqslant 3\wedge z\geqslant 3)$
0.05
$(\mathsf{si}, 2\leqslant x\leqslant 3)$
0.1
0.1
$(\mathsf{di}, 1\leqslant x\leqslant 2\wedge z\geqslant 3)$
$(\mathsf{di}, 1\leqslant x\leqslant 2)$

Figure 8.11. *Markov decision process generated by* $\mathsf{MaxU}(\mathtt{I}, [\![\Box\ \mathtt{I}]\!]_{=1})$

probabilistic timed automaton and computing the maximum probability of remaining in the set of symbolic states

$$\mathtt{I} = [\![\mathtt{true}]\!] \setminus \{(\mathsf{sr}, z < 6)\} = \{(\mathsf{di}, 0 \leqslant x \leqslant 3), (\mathsf{si}, 0 \leqslant x \leqslant 2), (\mathsf{sr}, z \geqslant 6)\},$$

i.e. the states where either the message has not been delivered or clock z is greater than or equal to 6. Using Proposition 8.4 we have $p_s^{\max}(\Box\ \mathtt{I}) = p_s^{\max}(\mathtt{I}\ \mathcal{U}\ [\![\Box\ \mathtt{I}]\!]_{=1})$*, and hence we first find* $[\![\Box\ \mathtt{I}]\!]_{=1}$ *(the set of states for which the maximum probability of remaining in* $\mathtt{I}$ *is 1). Using Proposition 8.5 this set is given by* $\mathsf{MaxI}_{\geqslant 1}(c, \mathtt{I})$ *which returns:*

$$[\![\Box\ \mathtt{I}]\!]_{=1} = \{(\mathsf{sr}, z \geqslant 6), (\mathsf{si}, x \leqslant 3 \wedge z \geqslant x+3), (\mathsf{di}, x \leqslant 2 \wedge z \geqslant x+4)\}.$$

Next, applying $\mathsf{MaxU}(\mathtt{I}, [\![\Box\ \mathtt{I}]\!]_{=1})$ *returns the Markov decision process given in Figure 8.11. As* $(\mathsf{di}, 1 \leqslant x \leqslant 2)$ *is the only symbolic state in Figure 8.11 for which the time predecessor set includes* $(\mathsf{di}, x = 0 \wedge z = 0)$*, using Proposition 8.3, from the initial state, the maximum probability of remaining in* $\mathtt{I}$ *until a state in* $[\![\Box\ \mathtt{I}]\!]_{=1}$ *is reached equals the maximum probability of* $(\mathsf{di}, 1 \leqslant x \leqslant 2)$ *reaching* $\mathsf{tpre}_{\mathtt{I}\vee[\![\Box\ \mathtt{I}]\!]_{=1}}([\![\Box\ \mathtt{I}]\!]_{=1})$*, and hence equals 0.005.*

Finally, using Proposition 8.4 and the duality between probabilistic reachability of invariance, starting from di *with* x *equal to 0, the minimum probability of correctly delivering before 6 time units have elapsed equals* $1 - 0.005 = 0.995$.

8.3.4. *Digital clocks*

In this section we present the *integral semantics* (or "digital clocks") model where the time domain is the natural numbers as opposed to the real numbers. This leads to a finite-state Markov decision process which can therefore be model checked directly by employing the efficient symbolic method in tools such as PRISM [HIN 06, PRI]. This approach is based on the work of [HEN 92] which studies the question of when real-time properties of timed automata can be verified using only integral durations (digital clocks), and shows that such a reduction is possible for a large class of systems and properties, including time-bounded invariance and response. However, before we discuss the integral semantics, we introduce an additional measure for probabilistic timed automata, *expected reachability*, since this measure is, along with probabilistic reachability, preserved under certain restrictions by the integral semantics.

In this section, we assume that probabilistic timed automata are *structurally non-Zeno* (see section 8.3.1) and also *closed* and *diagonal-free*, that is, automata whose zones do not compare the values of clocks with each another or contain strict comparisons with constants. The correctness of the results presented in this section can be found in [KWI 06].

8.3.4.1. *Expected reachability*

Expected reachability is defined with respect to a set of target states and a cost function mapping state-event and state-duration pairs to real values (the cost of performing an event or letting a certain amount of time pass in the corresponding state, respectively). This measure corresponds to the expected cost (with respect to the cost function) of reaching the target set. More formally, for a timed Markov decision process $\mathsf{TMDP} = (S, \bar{s}, \Sigma, \mathbb{T}, \mathit{Steps})$, cost function $\mathcal{C} : S \times (\Sigma \cup \mathbb{T}) \to \mathbb{R}$, state $s \in S$, set $T \subseteq S$ of target states, and adversary $A \in \mathit{Adv}_{\mathsf{TMDP}}$, let $e^A_{\mathsf{TMDP}}(s, \mathit{cost}(\mathcal{C}, T))$ denote the usual expectation of the function $\mathit{cost}(\mathcal{C}, T)$ (which returns, for a given path $\omega \in \mathit{Path}(s)$, the total cost accumulated until a state in T is reached along ω) with respect to the measure Prob^A_s over $\mathit{Path}^A(s)$. That is:

$$e^A_{\mathsf{TMDP}}(s, \mathit{cost}(\mathcal{C}, T)) = \int_{\omega \in \mathit{Path}^A(s)} \mathit{cost}(\mathcal{C}, T)(\omega)\, d\mathit{Prob}^A_s$$

where for any $\omega \in \mathit{Path}^A(s)$:

$$\mathit{cost}(\mathcal{C}, T)(\omega) \overset{\text{def}}{=} \sum_{i=1}^{\min\{j \mid \omega(j) \in T\}} \mathcal{C}(\omega(i-1), \mathit{step}(\omega, i-1))$$

if there exists $j \in \mathbb{N}$ such that $\omega(j) \in T$, and $\mathit{cost}(\mathcal{C}, T)(\omega) \overset{\text{def}}{=} \infty$ otherwise.

Since cost of a path which does not reach T is set to ∞, even though the total cost of the path may not be infinite, the expected cost of reaching T from s is finite if and only if a state in T is reached from s with probability 1. *Expected time reachability* (the expected time with which a given set of states can be reached) is a special case of expected reachability, corresponding to the case when $\mathcal{C}(s, \sigma) = 0$ for all $s \in S$ and $\sigma \in \Sigma$ and $\mathcal{C}(s, t) = t$ for all $s \in S$ and $t \in \mathbb{T}$.

DEFINITION 8.12.– *Let* $\mathsf{TMDP} = (S, \bar{s}, \Sigma, \mathbb{T}, \mathit{Steps})$ *be a timed Markov decision process. The* maximum and minimum expected costs *of, from a state* $s \in S$*, reaching a set of states* T *under the cost function* $\mathcal{C}$ *are defined as follows:*

$$e^{\max}_{\mathsf{TMDP}}(s, \mathcal{C}, \Diamond\, T) = \sup_{A \in Adv_{\mathsf{TMDP}}} e^{A}_{\mathsf{TMDP}}(s, cost(\mathcal{C}, T))$$

$$e^{\min}_{\mathsf{TMDP}}(s, \mathcal{C}, \Diamond\, T) = \inf_{A \in Adv_{\mathsf{TMDP}}} e^{A}_{\mathsf{TMDP}}(s, cost(\mathcal{C}, T))\,.$$

Calculating expected reachability for finite-state Markov decision processes is equivalent to the *stochastic shortest path problem*; see for example [BER 91, ALF 99].

In practice, cost functions are defined not at the level of timed Markov decision processes, but in terms of probabilistic timed automata. At this level, cost functions can be defined using a pair (c_Σ, r), where $c_\Sigma : L \times \Sigma \to \mathbb{R}$ is a function assigning the cost, in each location, of executing each event in Σ, and $r : L \to \mathbb{R}$ is a function assigning to each location the rate at which costs are accumulated as time passes in that location. The associated cost function $\mathcal{C}_{c_\Sigma, r}$ is defined as follows, for each $(l, v) \in L \times \mathbb{T}^{\mathcal{X}}$ and $a \in \Sigma \cup \mathbb{T}$:

$$\mathcal{C}_{c_\Sigma, r}((l, v), a) \stackrel{\text{def}}{=} \begin{cases} c_\Sigma(l, a) & \text{if } a \in \Sigma \\ a \cdot r(l) & \text{otherwise.} \end{cases}$$

A probabilistic timed automaton equipped with a pair (c_Σ, r) is a probabilistic extension of priced timed automata (also known as weighted timed automata) [BEH 01, ALU 04].

Expected time reachability allows us to express, for example, “the expected time until a data packet is delivered is at most 20 ms” and “the expected time until a packet collision occurs is at least 100 seconds”. In general, expected reachability allows us to compute the maximum and an minimum values for measures including: “the expected number of retransmissions before the message is correctly delivered”, “the expected number of packets sent before failure” and “the expected number of lost messages within the first 200 seconds”. For example, in the case of the last property, to calculate this measure, we would first need to modify the probabilistic timed automaton under study by adding a distinct clock and location such that, from all locations, once this clock has reached 200 seconds the only transition is to this new location. The set of target states would then be the set containing only this new location and the cost function would equal 0 on all time transitions and events except the event(s) corresponding to a message being lost, whose cost would be set to 1.

In addition, using cost functions of the form $\mathcal{C}_{c_\Sigma, r}$, we can consider performance measures such as:

– the expected time the channel is free before N messages are sent (by setting $r(l)$ to be 1 if location l corresponds to a state in which the channel is free, and 0 otherwise);

– the expected time a sender spends waiting for an acknowledgment (by setting $r(l)$ to be 1 if location l corresponds to a state in which the sender is waiting for an acknowledgment, and 0 otherwise);

– the expected energy consumption within the first $T(\in \mathbb{N})$ seconds (by setting $r(l)$ to the power usage (Watts) of location $l \in L$ and $c_\Sigma(l, \sigma)$ to be the energy consumption associated with performing the event σ in location l).

8.3.4.2. *Integral semantics*

Recall that the semantics of a probabilistic timed automaton given in section 8.2.4 is parametrized by both the time domain $\mathbb{T}$ and time increment $\oplus$. In this section we set the time domain $\mathbb{T}$ equal to $\mathbb{N}$, we let the time increment operator $\oplus$ equal $\oplus_\mathbb{N}$ which is defined below, and refer to $[\![\mathsf{PTA}]\!]_\mathbb{N}^{\oplus_\mathbb{N}}$ as the integral semantics of PTA. To define $\oplus_\mathbb{N}$, first, for any $x \in \mathcal{X}$, let c_x denote the greatest constant that clock x is compared to in the clock constraints of PTA. If the value of the clock x exceeds c_x, then its exact value is not relevant when deciding which probabilistic edges are enabled. This means that $c_x{+}1$ is the maximum value of clock x that needs to be represented, because we can interpret this value as corresponding to all clock values greater than c_x, which leads us to the following definition of $\oplus_\mathbb{N}$. For any clock valuation $v \in \mathbb{N}^\mathcal{X}$ and time duration $t \in \mathbb{N}$, let $v{\oplus_\mathbb{N}}t$ be the clock valuation of $\mathcal{X}$ which assigns the value $\min\{v(x){+}t, c_x{+}1\}$ to all clocks $x \in \mathcal{X}$ (although the operator $\oplus_\mathbb{N}$ is dependent on PTA, we omit the sub- or superscript indicating this for clarity).

The definition of integral semantics for probabilistic timed automata is a generalization of the analogous definition for the classical model in [BEY 01]. The fact that the integral semantics of a probabilistic timed automaton is finite can be derived from the definitions.

The results below demonstrate that digital clocks are sufficient for calculating probabilistic reachability and expected reachability properties of probabilistic timed automata.

PROPOSITION 8.6.– *For any probabilistic timed automaton* PTA *and target set of symbolic states* $\mathtt{T}$ *in which all zones are closed and diagonal free:*

$$p^{\max}_{[\![\mathsf{PTA}]\!]^+_\mathbb{R}}((\bar{l}, \mathbf{0}), \Diamond\, \mathtt{T}) = p^{\max}_{[\![\mathsf{PTA}]\!]^{\oplus_\mathbb{N}}_\mathbb{N}}((\bar{l}, \mathbf{0}), \Diamond\, \mathtt{T})$$
$$p^{\min}_{[\![\mathsf{PTA}]\!]^+_\mathbb{R}}((\bar{l}, \mathbf{0}), \Diamond\, \mathtt{T}) = p^{\min}_{[\![\mathsf{PTA}]\!]^{\oplus_\mathbb{N}}_\mathbb{N}}((\bar{l}, \mathbf{0}), \Diamond\, \mathtt{T})\ .$$

Furthermore, for any (non-negative) cost function $\mathcal{C}_{c_\Sigma, r}$ *with rational coefficients, if all probability values appearing in* PTA *are rational, then:*

$$e^{\max}_{[\![\mathsf{PTA}]\!]^+_\mathbb{R}}((\bar{l}, \mathbf{0}), \mathcal{C}_{c_\Sigma, r}, \Diamond\, \mathtt{T}) = e^{\max}_{[\![\mathsf{PTA}]\!]^{\oplus_\mathbb{N}}_\mathbb{N}}((\bar{l}, \mathbf{0}), \mathcal{C}_{c_\Sigma, r}, \Diamond\, \mathtt{T})$$
$$e^{\min}_{[\![\mathsf{PTA}]\!]^+_\mathbb{R}}((\bar{l}, \mathbf{0}), \mathcal{C}_{c_\Sigma, r}, \Diamond\, \mathtt{T}) = e^{\min}_{[\![\mathsf{PTA}]\!]^{\oplus_\mathbb{N}}_\mathbb{N}}((\bar{l}, \mathbf{0}), \mathcal{C}_{c_\Sigma, r}, \Diamond\, \mathtt{T}).$$

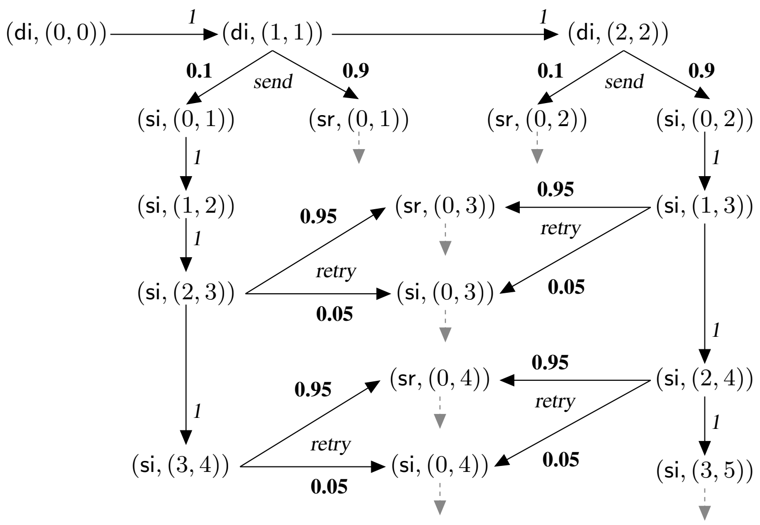


Figure 8.12. *Fragment of the integral semantic model of Example 8.1*

EXAMPLE 8.7.– *We return once more to the* PTA *in Example 8.1 (see Figure 8.1) and consider the integer semantic model of this automaton. Similarly to the cases before, since we are interested in the probability of reaching the location* sr *within* 6 *time units, we add an additional clock* z *to the automaton. Note that, since all zones must be closed (and diagonal-free), in this case we cannot consider a strict bound on the clock* z*, i.e. we can consider the target set* $(\mathsf{sr}, z \leqslant 6)$ *but not the target set* $(\mathsf{sr}, z < 6)$*. The Markov decision process obtained through the integer semantics has 32 states and 42 transitions. In Figure 8.12 we present a fragment of this Markov decision process in which a clock valuation* v *is written as a pair* (r_1, r_2) *rather than* $v(x) = r_1$ *and* $v(z) = r_2$*. We find that the minimum and maximum probabilities equal 0.995 and 0.9975 respectively. The fact that maximum probability differs from the previous cases is due to the fact that the target set now includes the case when the clock* z *equals 6.*

8.4. Case study: the IEEE FireWire root contention protocol

We illustrate the practical applicability of the techniques presented in this chapter with a real-life case study: the IEEE FireWire root contention protocol [IEE 95], which uses both randomization and timing delays to determine a leader among two contending processes. This section is based on results presented in [KWI 03, DAW 04, KWI 06].

8.4.1. *Overview*

The IEEE 1394 High Performance serial bus is used to transport digital video and audio signals within a network of multimedia systems and devices, such as PCs,

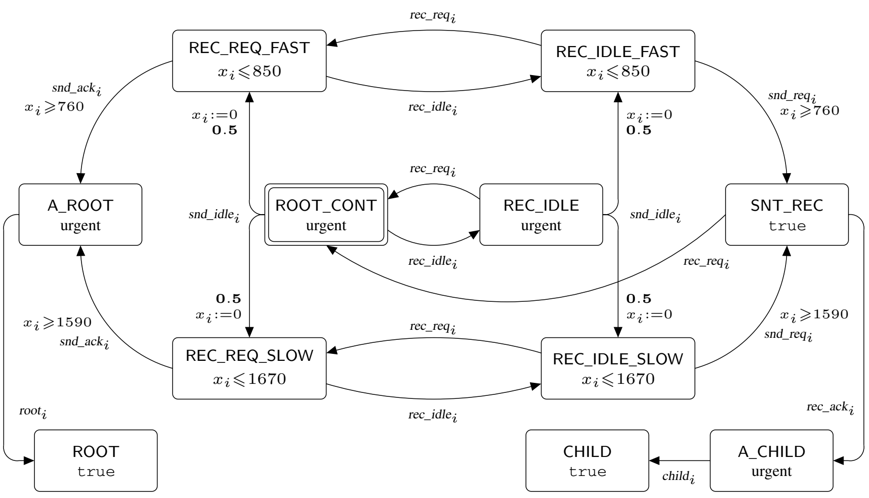


Figure 8.13. *The probabilistic timed automaton* $\texttt{Node}_i^{\texttt{P}}$

laptops and camcorders. It has a scalable architecture, and it is hot-pluggable, meaning that devices can be added to or removed from the network at any time, supports both isochronous and asynchronous communication and allows quick, reliable and inexpensive data transfer. It is currently one of the standard protocols for interconnecting multimedia equipment. The standard comprises various protocols. The one that we consider here is a leader election protocol called the *root contention protocol.*

The root contention protocol takes place after a bus reset in the network, i.e. when a node (device or peripheral) is added to or removed from the network. After a bus reset, all nodes in the network have equal status and know only to which nodes they are directly connected, so a leader must then be chosen. The aim of this protocol is to check whether the network topology is a tree and, if so, to construct a spanning tree over the network whose root is the leader elected by the protocol.

In order to elect a leader, nodes exchange “be my parent” requests with its neighbors. However, *contention* may arise when two nodes simultaneously send such requests to each other. The solution adopted by the standard to overcome this conflict, called *root contention*, is both probabilistic and timed: each node will flip a coin in order to decide whether to wait for a short or a long time before sending a request.

8.4.2. *Probabilistic timed automata model*

The model comprises four components: two contending nodes and two connecting wires. Figure 8.13 shows the probabilistic timed automaton $\texttt{Node}_i^{\texttt{P}}$ for a node. This model is a probabilistic extension of the timed automaton model presented in [SIM 01].

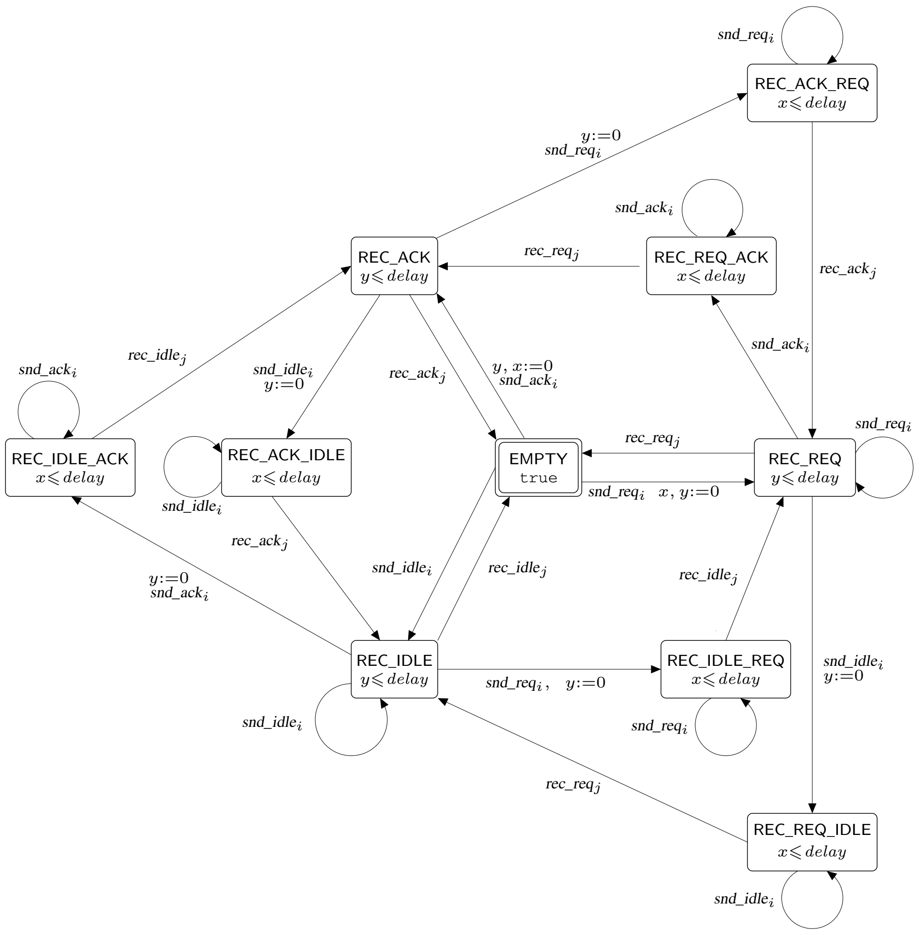


Figure 8.14. *The probabilistic timed automaton* Wire$_i$

The behavior of the model commences in location ROOT_CONT, which models the situation in which the node has detected root contention. Because this location is urgent, $\texttt{Node}_i^{\texttt{P}}$ is forced to select an outgoing probabilistic edge instantly. Consider the bifurcating probabilistic edge labeled by *snd_idle*$_i$, which corresponds to the node flipping a coin in order to determine whether it should wait for a short or long time. The *snd_idle*$_i$ event is sent by $\texttt{Node}_i^{\texttt{P}}$ to its communication medium, referring to a transmission of an idle signal across the node's wire to the other node. In both of the locations REC_REQ_FAST and REC_REQ_SLOW which may be reached after taking the probabilistic edge, the passage of time may mean that the value of clock x_i can reach a value enabling the edges, which means that an acknowledgment is sent (event *snd_ack*$_i$), and the node then declares itself to be leader (the event *root*$_i$ which labels

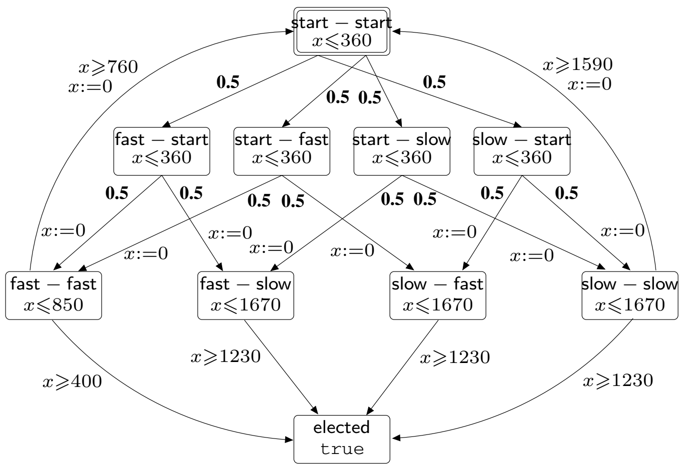


Figure 8.15. *The probabilistic timed automaton* $\mathtt{I}_1^{\mathtt{p}}$

the subsequent left-pointing edge to the location ROOT). In contrast, in the locations REC_REQ_FAST and REC_REQ_SLOW, if the node receives an idle signal from the other contending node (event *rec_idle*$_i$) before sending an acknowledgment, it is forced to move to the right to REC_IDLE_FAST or REC_IDLE_SLOW, respectively.

In this case, after a certain amount of time elapses, $\mathtt{Node}_i^{\mathtt{p}}$ can issue a request to the other node to be its parent by sending the event *snd_req*$_i$ to its wire. If the node then subsequently detects a parent request from the other node (event *rec_req*$_i$), it returns to the location ROOT_CONT, and restarts the root contention process. If, on the other hand, the node detects an acknowledgment from the other node (event *rec_ack*$_i$), it proceeds to declare itself as the child by sending a *child*$_i$ event.

The communication medium between the nodes comprises two wires modeled as two-place buffers, along which signals are driven continuously. These are represented by the timed automata $\mathtt{Wire}_1$ and $\mathtt{Wire}_2$. Figure 8.14 illustrates the general case, $\mathtt{Wire}_i$. This model is the classical (non-probabilistic) timed automaton taken from [SIM 01], with the interpretation that all transitions are made with probability 1. The parallel composition

$$\mathtt{Impl}^{\mathtt{p}} = \mathtt{Node}_1^{\mathtt{p}} \,\|\, \mathtt{Wire}_1 \,\|\, \mathtt{Wire}_2 \,\|\, \mathtt{Node}_2^{\mathtt{p}}$$

of the resulting probabilistic timed automata is then constructed using Definition 8.4.

We also study the abstract probabilistic timed automaton $\mathtt{I}_1^{\mathtt{p}}$ of the root contention protocol given in Figure 8.15. It is a probabilistic extension of the classical timed automaton $\mathtt{I}_1$ of [SIM 01], where each instance of bifurcating edges corresponds to a coin being flipped. For example, in the initial location start-start, there is a non-deterministic choice corresponding to node 1 (respectively node 2) starting the root contention protocol and flipping its coin, leading with probability 0.5 to each of slow-start and fast-start (respectively start-slow and start-fast). To simplify the presentation, the events of the probabilistic edges of $\mathtt{I}_1^{\mathtt{p}}$ have been omitted.

8.4.3. *Model checking statistics*

In this section we investigate the state spaces and timing statistics of the different approaches presented in this chapter when applied to the root contention protocol. The statistics were obtained by setting the maximum transmission delay equal to 360 nanoseconds and calculating the minimum probability of electing a leader by a deadline D and the minimum probability of eventually electing a leader (i.e. when $D = \infty$). The generation times for the symbolic backward approach are for the prototype implementation of [KWI 07] and in the case of the forward algorithm for the implementation based on KRONOS [DAW 96] presented in [DAW 04]. To apply the forward algorithm, which can only compute upper bounds on maximum reachability probabilities, the model transformations of $\mathtt{Impl}^{\mathtt{p}}$ and $\mathtt{I}_1^{\mathtt{p}}$ presented in [KWI 03] were employed. Since no model transformation is known to reduce the computation of the minimum probability of eventually electing a leader to a maximum reachability probability, the forward approach has not been applied to this case.

Tables 8.1 and 8.2 present the state spaces for all techniques and generation times for the backward and forward algorithms. The results demonstrate that the region graph and digital clock models have very large state spaces and that the region graph becomes prohibitively large very quickly (limiting its applicability to only the abstract version of the contention protocol $\mathtt{I}_1^{\mathtt{p}}$). The results also show that the forward and backwards methods lead to models of similar sizes and that the generation times are much larger for the backward algorithms. This difference in times can be attributed to the fact that the backward technique requires more complex operations on zones. This difference is increased by the fact that the forward implementation employs the established tool KRONOS. For the case $D = \infty$, note that the backward algorithm generates an empty state space. This is because after the first step (calculating $[\![\Box\ \mathtt{I}]\!]_{=1}$) there is no further work to be done.

Tables 8.3 and 8.4 report on the construction and model checking times for the (finite-state) Markov decision processes when using the PRISM tool [HIN 06, PRI]. Due to the size of the state spaces for the region graph and digital clocks models, the construction and model checking times are larger in these cases. Although the digital clocks approach generates much larger state spaces, we can see that this does not have a drastic effect on the model checking times. This is due to regularity in the model which is exploited in the symbolic data structures employed by PRISM. The fact that construction times for the forward models are much faster than those obtained for the

D (10^3ns)	Region graph	Forward	Backward	Digital clocks
2	423,016	53 (0.00)	15 (1.39)	68,056
4	1,395,390	131 (0.00)	25 (1.55)	220,565
8	3,375,390	372 (0.02)	81 (1.35)	530,965
10	4,365,390	526 (0.03)	126 (1.61)	686,165
20	9,315,390	1,876 (0.09)	528 (11.0)	1,462,165
30	14,265,390	4,049 (0.20)	1,206 (11.2)	2,238,165
40	19,215,390	7,034 (0.46)	2,168 (1,030)	3,014,165
50	24,165,390	10,865 (1.23)	3,426 (6,464)	3,790,165
60	29,115,390	15,511 (2.74)	4,964 (26,995)	4,566,165
∞	1,542	-	0 (0.55)	776

Table 8.1. *Model sizes (and generation times in seconds) for* $\mathtt{I}_1^{\mathtt{P}}$

D (10^3ns)	Forward	Backward	Digital clocks
2	965 (0.08)	551 (447)	6,719,773
4	2,599 (0.94)	2,220 (3,854)	44,366,235
6	4,337 (1.64)	4,264 (14,954)	86,813,479
8	7,831 (2.93)	8,075 (100,893)	129,267,079
10	11,119 (4.27)	13,428 (608,392)	171,720,679
20	41,017 (18.7)	-	383,988,679
30	89,283 (56.1)	-	596,256,679
40	155,675 (129)	-	808,524,679
∞	-	0 (97.2)	212,268

Table 8.2. *Model sizes (and generation times in seconds) for* $\mathtt{Impl}^{\mathtt{P}}$

backward models is due to the forward implementation optimizing the PRISM input [DAW 04]. Finally we note that, due to similarity between the model sizes, there is no significant difference between the model checking times of the forward and backward models.

8.4.4. *Performance analysis*

We now present an analysis of the performance of the FireWire root contention protocol based on the application of the techniques described in the previous section. Firstly, Figure 8.16(a) presents the minimum probability of electing a leader within a deadline as the communication delay (wire length) between the nodes varies. As expected, as the communication delay between the nodes increases, the probability of electing the root before a deadline decreases.

In the remaining analysis, we will consider two cases for the maximum transmission delay along the wires (the constant $delay$ in Figure 8.14): 360 nanoseconds (ns)

D	Region graph		Forward		Backward		Digital clocks	
(10^3ns)	const.	m/c	const.	m/c	const.	m/c	const.	m/c
2	14.3	5.90	0.08	0.02	0.08	0.03	2.90	0.53
4	29.2	45.2	0.10	0.03	0.07	0.02	8.67	3.81
8	72.4	215	0.15	0.03	0.30	0.03	31.4	22.0
10	100	332	0.16	0.04	0.59	0.03	49.0	34.1
20	314	1,140	0.70	0.10	11.5	0.05	183	134
30	898	2,535	1.83	0.16	44.7	0.12	406	280
40	1,186	2,786	4.66	0.29	268	0.29	746	312
50	1,906	2,861	11.0	0.50	448	0.53	1,191	302
60	2,666	2,938	19.6	0.71	1,125	0.73	1,754	300
∞	0.61	0.33	-	-	0	0	0.41	0.11

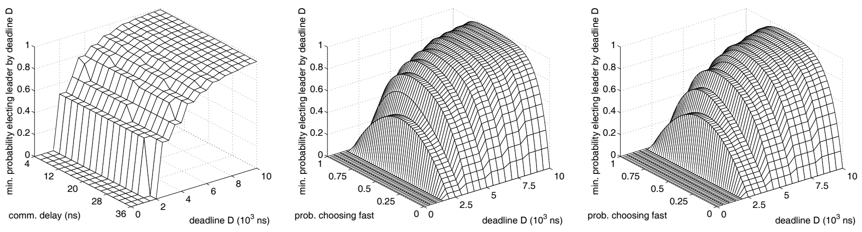
Table 8.3. *Model construction and model checking times in seconds for* $\mathtt{I}_1^{\mathtt{P}}$

D	Forward		Backward		Digital clock	
(10^3ns)	const.	m/c	const.	m/c	const.	m/c
2	3.02	0.08	10.8	0.04	264	2.49
4	8.98	0.11	157	0.08	1,627	300
6	14.1	0.15	646	0.09	2,967	13,812
8	41.3	0.23	2,065	0.15	5,690	26,583
10	79.5	0.34	5,081	0.30	7,930	34,885
20	1,303	1.11	-	-	16,748	104,112
30	6,893	4.54	-	-	26,969	141,785
40	25,417	8.38	-	-	32,325	199,102
∞	-	-	0	0	23.0	68.9

Table 8.4. *Model construction and model checking times in seconds for* $\mathtt{Impl}^{\mathtt{P}}$

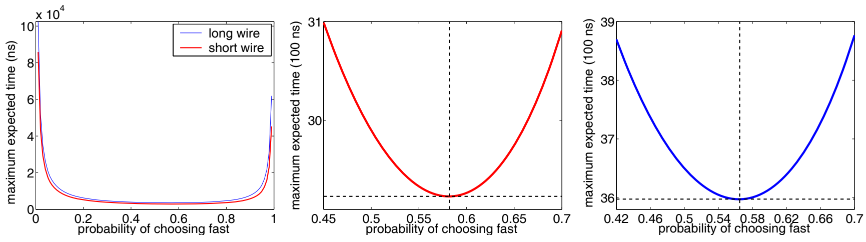
and 30 ns. This models the distance between the two nodes, i.e. the length of the connecting wires. A delay of 360 ns represents the assumption that the nodes are separated by a distance close to the maximum required for the correctness of the protocol (from the analysis of [SIM 01]). A delay of 30 ns corresponds more closely to the maximum separation of nodes specified in the actual IEEE standard; this value is 22.725 ns, and therefore our figure of 30 ns is an over-approximation. This is for efficiency reasons: it allows us to use a time granularity of 10 ns when we consider probabilistic model checking using digital clocks. In the following text, we will refer to the two cases (360 ns and 30 ns) as “long wire” and “short wire”, respectively.

Figures 8.16(b), 8.16(c) and 8.17 report on the effect of using a biased coin with respect to the minimum probability of electing a leader within a deadline and the maximum expected time to elect a leader, respectively. Note that we assume that the nodes in root contention use coins of the same bias. Although it is possible to improve the performance of the protocol by assuming that the nodes’ coins have different biases,



(a) Communication delay (b) Biased coin (short wire) (c) Biased coin (long wire)

Figure 8.16. *Minimum probability of electing a leader by deadline D*



(a) both wire lengths (b) short wire (c) long wire

Figure 8.17. *Maximum expected time to elect a leader*

i.e. one coin is biased towards fast and the other towards slow, this is not feasible in practice since each node follows the same procedure and it is not known in advance which nodes of the network will take part in the root contention protocol. Furthermore, assuming that two nodes enter the contention protocol, to decide which node should flip which sort of coin is equivalent to electing a leader.

The results demonstrate that the (timing) performance of the root contention protocol can be improved using a biased coin which has a higher probability of flipping "fast". The idea behind this result is that, although the use of such a biased coin decreases the likelihood of the nodes flipping different values, when nodes flip the same values there is a greater chance that less time passes before they flip again (i.e. when both flip "fast") [STO 02]. There is a compromise here, because as the coin becomes more biased towards "fast", the probability of the nodes actually flipping different values (which is required for a leader to be elected) decreases even though the delay between coin flips will on average decrease. The results in Figure 8.17 further demonstrate that, for a shorter wire length, there is a greater advantage when using a biased coin (for the short wire length the minimum occurs for a coin which flips "fast" with a probability near 0.58 while for the long wire length the minimum is near 0.56). The reasoning behind this result is that for the short wire length more time is saved when both nodes flip fast than for a longer wire length, since the time required when both nodes flip fast is dependent on a constant delay given by the protocol specification plus a delay dependent on the wire length.

8.5. Conclusion

In this chapter we gave an introduction to the probabilistic timed automata formalism, which is suitable for the modeling and analysis of systems exhibiting both real-time and probabilistic characteristics. We described four different techniques for model checking probabilistic timed automata. In brief these can be summarized as follows:

– The *region graph* approach is applicable to a large class of probabilistic timed automata and full PTCTL, but its application can be prohibitively expensive.

– The *forward reachability* approach is applicable to general probabilistic timed automata, but is restricted to computing upper bounds on maximum reachability probabilities.

– The *backward reachability* approach is applicable to general probabilistic timed automata and full PTCTL, but is more expensive than the forward approach and, at the time of writing, cannot be applied to expected reachability.

– The *digital clocks* approach is applicable to closed and diagonal free probabilistic timed automata; it can be used to compute probabilistic and expected reachability measures, but cannot be used to verify full PTCTL.

This chapter also demonstrated the applicability of these methods to the analysis of the IEEE FireWire root contention protocol. Additional case studies using these methods include Bluetooth Device Discovery [DUF 06], IEEE 802.11 Wireless LAN [KWI 02b], IPV4 Zeroconf [KWI 06], IEEE 802.15.4 CSMA-CA (ZigBee) [FRU 06] and IEEE 802.3 CSMA/CD [KWI 07]. See also [PRI] for further details.

8.6. Bibliography

[ALF 97] DE ALFARO L., "Formal verification of probabilistic systems", PhD thesis, Stanford University, 1997.

[ALF 99] DE ALFARO L., "Computing minimum and maximum reachability times in probabilistic systems", BAETEN J., MAUW S., Eds., *Proc. 10th Int. Conf. Concurrency Theory (CONCUR'99)*, vol. 1664 of *Lecture Notes in Computer Science*, Springer Verlag, p. 66–81, 1999.

[ALU 94] ALUR R., DILL D., "A theory of timed automata", *Theoretical Computer Science*, vol. 126, num. 2, p. 183–235, 1994.

[ALU 04] ALUR R., LA TORRE S., PAPPAS G., "Optimal paths in weighted timed automata", *Theoretical Computer Science*, vol. 318, num. 3, p. 297–322, 2004.

[BEH 01] BEHRMANN G., FEHNKER A., HUNE T., LARSEN K., PETTERSSON P., ROMIJN J., VAANDRAGER F., "Minimum-cost reachability for linearly priced timed automata", BENEDETTO M. D., SANGIOVANNI-VINCENTELLI A., Eds., *Proc. 4th Int. Workshop on Hybrid Systems: Computation and Control (HSCC'01)*, vol. 2034 of *Lecture Notes in Computer Science*, Springer Verlag, p. 147–162, 2001.

[BEH 04] BEHRMANN G., DAVID A., LARSEN K. G., "A tutorial on UPPAAL", BERNARDO M., CORRADINI F., Eds., *Formal Methods for the Design of Real-Time Systems: 4th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, vol. 3185 of *Lecture Notes in Computer Science*, Springer Verlag, p. 200–236, 2004.

[BER 91] BERTSEKAS D., TSITSIKLIS J., "An analysis of stochastic shortest path problems", *Mathematics of Operations Research*, vol. 16, num. 3, p. 580–595, 1991.

[BEY 01] BEYER D., "Improvements in BDD-based reachability analysis of timed automata", OLIVEIRA J., ZAVE P., Eds., *Proc. Symp. Formal Methods Europe (FME'01)*, vol. 2021 of *Lecture Notes in Computer Science*, Springer Verlag, p. 318–343, 2001.

[BIA 95] BIANCO A., DE ALFARO L., "Model checking of probabilistic and nondeterministic systems", THIAGARAJAN P., Ed., *Proc. 15th Conf. Foundations of Software Technology and Theoretical Computer Science*, vol. 1026 of *Lecture Notes in Computer Science*, Springer Verlag, p. 499–513, 1995.

[COU 98] COURCOUBETIS C., YANNAKAKIS M., "Markov decision processes and regular events", *IEEE Transactions on Automatic Control*, vol. 43, num. 10, p. 1399–1418, 1998.

[DAW 96] DAWS C., OLIVERO A., TRIPAKIS S., YOVINE S., "The tool KRONOS", ALUR R., HENZINGER T., SONTAG E., Eds., *Hybrid Systems III, Verification and Control*, vol. 1066 of *Lecture Notes in Computer Science*, Springer Verlag, p. 208–219, 1996.

[DAW 98] DAWS C., TRIPAKIS S., "Model checking of real-time reachability properties using abstractions", STEFFEN B., Ed., *Proc. 4th Int. Conf. Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, vol. 1384 of *Lecture Notes in Computer Science*, Springer Verlag, p. 313–329, 1998.

[DAW 04] DAWS C., KWIATKOWSKA M., NORMAN G., "Automatic verification of the IEEE 1394 root contention protocol with KRONOS and PRISM", *Int. Journal on Software Tools for Technology Transfer*, vol. 5, num. 2–3, p. 221–236, 2004.

[DIL 89] DILL D., "Timing assumptions and verification of finite-state concurrent system", SIFAKIS J., Ed., *Proc. Int. Workshop Automatic Verification Methods for Finite State Systems*, vol. 407 of *Lecture Notes in Computer Science*, Springer Verlag, p. 197–212, 1989.

[DUF 06] DUFLOT M., KWIATKOWSKA M., NORMAN G., PARKER D., "A formal analysis of Bluetooth device discovery", *Int. Journal on Software Tools for Technology Transfer*, vol. 8, num. 6, p. 621–632, 2006.

[FRU 06] FRUTH M., "Probabilistic model checking of contention resolution in the IEEE 802.15.4 low-rate wireless personal area network protocol", *Proc. 2nd Int. Symp. Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'06)*, 2006.

[HEN 92] HENZINGER T., MANNA Z., PNUELI A., "What good are digital clocks?", KUICH W., Ed., *Proc. 19th Int. Colloquium on Automata, Languages and Programming (ICALP'92)*, vol. 623 of *Lecture Notes in Computer Science*, Springer Verlag, p. 545–558, 1992.

[HEN 94] HENZINGER T., NICOLLIN X., SIFAKIS J., YOVINE S., "Symbolic model checking for real-time systems", *Information and Computation*, vol. 111, num. 2, p. 193–244, 1994.

[HIN 06] HINTON A., KWIATKOWSKA M., NORMAN G., PARKER D., "PRISM: A tool for automatic verification of probabilistic systems", HERMANNS H., PALSBERG J., Eds., *Proc. 12th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, vol. 3920 of *Lecture Notes in Computer Science*, Springer Verlag, p. 441–444, 2006.

[IEE 95] IEEE 1394-1995, *High Performance Serial Bus Standard*, 1995.

[JEN 96] JENSEN H., "Model checking probabilistic real time systems", BJERNER B., LARSSON M., NORDSTRÖM B., Eds., *Proc. 7th Nordic Workshop on Programming Theory*, Report 86, Chalmers University of Technology, p. 247–261, 1996.

[KEM 76] KEMENY J., SNELL J., KNAPP A., *Denumerable Markov Chains*, Springer Verlag, 2nd edition, 1976.

[KWI 01] KWIATKOWSKA M., NORMAN G., SPROSTON J., "Symbolic computation of maximal probabilistic reachability", LARSEN K., NIELSEN M., Eds., *Proc. 13th Int. Conf. Concurrency Theory (CONCUR'01)*, vol. 2154 of *Lecture Notes in Computer Science*, Springer Verlag, p. 169–183, 2001.

[KWI 02a] KWIATKOWSKA M., NORMAN G., SEGALA R., SPROSTON J., "Automatic verification of real-time systems with discrete probability distributions", *Theoretical Computer Science*, vol. 282, p. 101–150, 2002.

[KWI 02b] KWIATKOWSKA M., NORMAN G., SPROSTON J., "Probabilistic model checking of the IEEE 802.11 wireless local area network protocol", HERMANNS H., SEGALA R., Eds., *Proc. 2nd Joint Int. Workshop Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'02)*, vol. 2399 of *LNCS*, Springer, p. 169–187, 2002.

[KWI 03] KWIATKOWSKA M., NORMAN G., SPROSTON J., "Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol", *Formal Aspects of Computing*, vol. 14, p. 295–318, 2003.

[KWI 06] KWIATKOWSKA M., NORMAN G., PARKER D., SPROSTON J., "Performance analysis of probabilistic timed automata using digital clocks", *Formal Methods in System Design*, vol. 29, p. 33–78, 2006.

[KWI 07] KWIATKOWSKA M., NORMAN G., SPROSTON J., WANG F., "Symbolic model checking for probabilistic timed automata", *Information and Computation*, vol. 205, num. 7, 2007.

[LAR 97] LARSEN K., PETTERSSON P., YI W., "UPPAAL in a nutshell", *Int. Journal on Software Tools for Technology Transfer*, vol. 1, num. 1-2, p. 134–152, 1997.

[PRI] PRISM web page, www.theprismmodelchecker.org/.

[RUT 04] RUTTEN J., KWIATKOWSKA M., NORMAN G., PARKER D., *Mathematical techniques for analyzing concurrent and probabilistic systems,* P. Panangaden and F. van Breugel (eds.), vol. 23 of *CRM Monograph Series*, American Mathematical Society, 2004.

[SEG 95a] SEGALA R., "Modeling and verification of randomized distributed real-time systems", PhD thesis, Massachusetts Institute of Technology, 1995.

[SEG 95b] SEGALA R., LYNCH N. A., "Probabilistic simulations for probabilistic processes", *Nordic Journal of Computing*, vol. 2, num. 2, p. 250–273, 1995.

[SIM 01] SIMONS D., STOELINGA M. I. A., "Mechanical verification of the IEEE 1394a root contention protocol using UPPAAL2k", *Int. Journal on Software Tools for Technology Transfer*, vol. 3, num. 4, p. 469–485, Springer Verlag, 2001.

[STO 02] STOELINGA M., "Alea jacta est: verification of probabilistic, real-time and parametric systems", PhD thesis, University of Nijmegen, The Netherlands, 2002.

[TRI 98] TRIPAKIS S., "The formal analysis of timed systems in practice", PhD thesis, Joseph Fourier University, 1998.

[TRI 05] TRIPAKIS S., YOVINE S., BOUAJJANI A., "Checking timed Büchi automata emptiness efficently", *Formal Methods in System Design*, vol. 26, num. 3, p. 267–292, 2005.

Chapter 9

Verification of Probabilistic Systems Methods and Tools

9.1. Introduction

Historically, functional verification and performance evaluation have been two distinct stages in the development of applications. Each one had its own models and methods. For 15 years, numerous works covered both areas. These works are now referred to as *probabilistic verification* or, more accurately, by *verification of probabilistic systems*.

This direction of research is prompted by the new needs of the modelers. They wish, for instance, to compute the probability that a property expressed as some logical formula is satisfied. They also wish to analyze a system including both non-deterministic and probabilistic features. The goal of this chapter is to introduce three important topics related to this research:

– the definition of high level stochastic models;

– the verification of Markov chains (MC);

– the verification of Markov decision processes (MDP).

We will always follow the same organization for these topics:

– the detailed presentation of one approach;

– an overview of the other approaches;

– the description of an analysis tool related to this topic.

Chapter written by Serge HADDAD and Patrice MOREAUX.

The first part of this chapter consists of preliminary notions related to stochastic processes and Markov chains. The second part is devoted to high level formalisms. The third part covers the verification of Markov chains and the chapter ends with a discussion of the verification methods for Markov decision processes.

9.2. Performance evaluation of Markovian models

9.2.1. *A stochastic model of discrete event systems*

We assume that the reader is familiar with the basics of probability theory. For more details, see the following books [FEL 68, FEL 71, TRI 82].

In this chapter, we use the following notations:

– $\Pr(E)$ denotes the probability of an event E and $\Pr(A \mid B)$ the probability of A given B;

– the adverb "almost", in expressions such as "almost everywhere" or "almost surely", means for a set of probability 1;

– $\mathbb{R}$ (respectively $\mathbb{R}^+, \mathbb{R}^{+*}$) denotes the set of real numbers (respectively non negative real numbers, positive real numbers). If x is a real number then $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x;

– if $E \subseteq \mathbb{R}$ then $\mathrm{Inf}(E)$ (respectively $\mathrm{Sup}(E)$) denotes the greatest lower bound (respectively least upper bound) of E.

An execution of a discrete event system (DES) is characterized by a sequence (*a priori* infinite) of events $\{e_1, e_2, \ldots\}$ separated by time delays. Only the occurrence of an event changes the state of the system.

More formally, the stochastic behavior of a DES is defined by two families of random variables:

– $X_0, \ldots, X_n, \ldots$ ranging over the (discrete) space of states, denoted S. In the sequel, unless explicitly mentioned, we assume that this space is finite. X_0 represents the initial state of the system and X_n $(n > 0)$ the current state after the occurrence of the n^{th} event. The occurrence of an event does not necessarily change the state of the system, hence X_{n+1} may be equal to X_n;

– $T_0, \ldots, T_n, \ldots$ ranging over $\mathbb{R}^+$ where T_0 represents the delay before the first event and T_n $(n > 0)$ represents the time elapsing between the n^{th} and the $(n+1)^{th}$ event. Observe that these delays may be zero (e.g., a sequence of instructions considered as instantaneous compared to database transactions including I/O operations).

When initial distribution X_0 is concentrated in a single state s, we say that the process starts in s (i.e., $\Pr(X_0 = s) = 1$).

A priori, there is no restriction on these families of random variables. However, for the categories of processes that we study, a DES cannot execute an infinite number of actions in a finite time. Otherwise stated:

$$\sum_{n=0}^{\infty} T_n = \infty \text{ almost surely.} \tag{9.1}$$

This property allows us to define the state of the system at any instant. Let $N(\tau)$ be the random variable defined by:

$$N(\tau) =_{def} \operatorname{Inf}\left(\left\{n \mid \sum_{k=0}^{n} T_k > \tau\right\}\right)$$

Due to (9.1), $N(\tau)$ is defined almost everywhere. As can be seen in Figure 9.1, $N(\tau)$ presents jumps strictly greater than 1. The state $Y(\tau)$ of the system at instant τ, is now $X_{N(\tau)}$. Observe that $Y(\tau)$ is not equivalent to the stochastic process, but that it allows, in most cases, to proceed to standard analyses. The scheme in Figure 9.1 presents a possible *realization* of the process and illustrates the meaning of the random variables introduced above. In this example, the process is initially in state s_4 and remains in this state until τ_0 where it visits s_6. At time $\tau_0 + \tau_1$, the system successively visits in zero time, states s_3 and s_{12} before reaching s_7 where it remains some non-zero amount of time. Observing $Y(\tau)$ in continuous time hides the vanishing states s_3 and s_{12} of the process.

The performance evaluation of a DES leads to two kinds of analysis:

– the study of the transient behavior, i.e. the computation of measures depending on the time elapsed since the initial state. This study aims to analyze the initialization stage of a system and the terminating systems. For instance, dependability and reliability [LAP 95, MEY 80, TRI 92] require transient analysis;

– the study of steady-state behavior of the system. In numerous applications, the modeler is interested by the behavior of the system once it is stabilized.

Obviously, this requires that such a steady-state behavior exists. This condition can be expressed, denoting $\boldsymbol{\pi}(\tau)$ the distribution of $Y(\tau)$, by:

$$\lim_{\tau \to \infty} \boldsymbol{\pi}(\tau) = \boldsymbol{\pi} \tag{9.2}$$

where $\boldsymbol{\pi}$ is also a distribution called the *steady-state distribution.*

Transient and steady-state distributions are often only intermediate values useful to compute *performance indices*. For instance, the steady-state probability that a server is available, the probability that at time τ a connection is established or the mean number of clients waiting for a service are such indices.

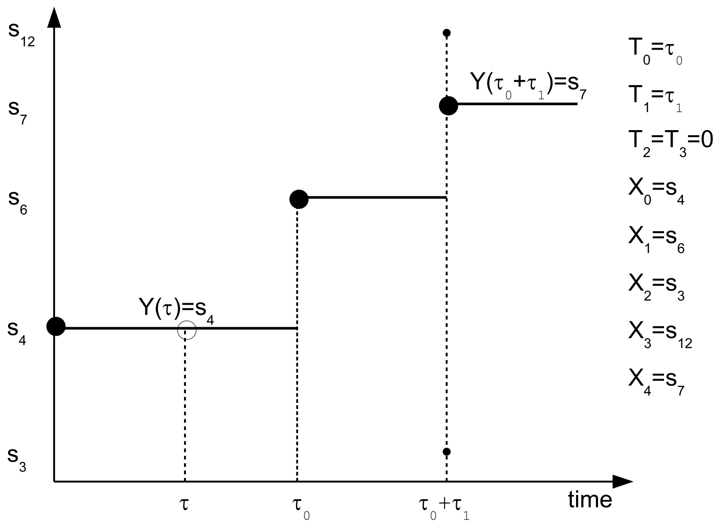


Figure 9.1. *A realization (trajectory) of the stochastic process*

In order to reason in a generic way about DES we assume in the following that we are given a set of functions defined on the set of states and ranging over $\mathbb{R}$. Such a function f may be interpreted as a performance index and, given a distribution $\boldsymbol{\pi}$, quantity $\sum_{s \in S} \boldsymbol{\pi}(s) \cdot f(s)$ represents the measure of this index.

When the index is a function ranging over $\{0, 1\}$, it can be viewed as an atomic proposition satisfied in a state if the function is equal to 1. In the following, we note $\mathcal{P}$, the set of atomic propositions and $s \models \phi$, and the fact that s satisfies ϕ, with s a state and ϕ an atomic proposition. In this case, given a distribution $\boldsymbol{\pi}$, quantity $\sum_{s \models \phi} \boldsymbol{\pi}(s)$ represents the measure of this index.

9.2.2. *Discrete-time Markov chains*

9.2.2.1. *Presentation*

A discrete-time Markov chain (DTMC) presents the following characteristics:

– the delay between successive instants T_n is the constant value 1;

– the successor of the current state depends only on this state and the transition probabilities are time-invariant[1]:

$$\begin{aligned}&\Pr(X_{n+1} = s_j \mid X_0 = s_{i_0}, \ldots, X_n = s_i)\\ &\quad = \Pr(X_{n+1} = s_j \mid X_n = s_i) = p_{ij} =_{def} \mathbf{P}[i, j].\end{aligned}$$

1. Hence the name *homogenous* chain used in studies about more general definitions of Markov chains.

In the following sections, we use both notations for state transitions.

9.2.2.2. *Transient and steady-state behaviors of DTMC*

In this section we recall classical results while providing intuitive justifications which are not mathematical proofs.

The analysis of the transient behavior does not present any difficulty. State changes occur at time $\{1, 2, \ldots\}$. Given an initial distribution $\boldsymbol{\pi}_0$ and a transition matrix $\mathbf{P}$, then $\boldsymbol{\pi}_n$ the distribution of X_n (i.e. the state of the chain at time n) is given by formula $\boldsymbol{\pi}_n = \boldsymbol{\pi}_0 \cdot \mathbf{P}^n$ which is obtained with the help of an elementary recurrence.

The analysis of the asymptotic behavior of a DTMC (in the case of a countable or finite set of states) leads to the following classification of states:

– a state s is *transient* if the probability that it occurs again once it has occurred is strictly less than 1. Consequently, its occurrence probability $\Pr(X_n = s)$ goes to 0 when n goes to ∞. A state is called *recurrent* if it is not transient;

– a state is *null recurrent* if the mean time between two successive occurrences of this state is infinite. Intuitively, once reached, the mean delay between occurrences of this state will go to ∞ as the number of occurrences goes to ∞ and consequently, once again, the occurrence probability will go to 0. This intuitive reasoning is mathematically sound;

– a state is *non-null recurrent* if the mean delay between two successive occurrences of this state is finite. If a stationary distribution exists then it is concentrated on the non-null recurrent states.

We discuss this analysis when the state space is finite. Let us consider the graph defined as follows:

– the set of vertices is the set of states of the Markov chain;

– there is an edge from s_i to s_j if $p_{ij} > 0$.

Let us study the strongly connected components (SCC) of this graph. If an SCC has an outgoing edge then necessarily the states of this SCC are transient. Conversely, all the states of a *terminal* SCC (i.e. without an outgoing edge) are non-null recurrent. In the particular case where a terminal SCC is reduced to a state s (i.e. $\mathbf{P}[s, s] = 1$), we say that s is an *absorbing* state.

When the graph is strongly connected, we say that the chain is *irreducible*. In the general case, every terminal SCC is an irreducible subchain.

Let us study the existence of a steady-state distribution when the chain is irreducible. First notice that it may not exist. For instance, a chain with two states s_0 and s_1, an initial distribution concentrated in a state and where $p_{0,1} = p_{1,0} = 1$, alternates between the two states and thus does not converge to a steady-state distribution. By generalization, an irreducible chain is *periodic* with period $k > 1$ if we can partition the states in subsets $S_0, S_1, \dots, S_{k-1}$ such that from states of S_i the chain only reaches, in one step, states of $S_{(i+1) \bmod k}$.

It turns out that an irreducible and aperiodic chain (called *ergodic*) yields a steady-state distribution and that it is *independent from the initial distribution*. Computing this distribution is relatively easy. Indeed, we have $\boldsymbol{\pi}_{n+1} = \boldsymbol{\pi}_n \cdot \mathbf{P}$. With the limit (which is mathematically sound), we obtain $\boldsymbol{\pi} = \boldsymbol{\pi} \cdot \mathbf{P}$. Moreover, $\boldsymbol{\pi}$ is the single distribution of:

$$\mathbf{X} = \mathbf{X} \cdot \mathbf{P} \tag{9.3}$$

Let us note that an initial distribution, the solution of this equation, is *invariant*: whatever an observation time, the current distribution is identical to the initial distribution. In order to solve equation (9.3), we can proceed to a direct computation by substituting the normalization equation ($\mathbf{X} \cdot \mathbf{1}^T = 1$ where $\mathbf{1}^T$ denotes the column vector with all 1s) for any other equation.

However, iterative computations are more interesting if the state space is huge. The simplest one consists of iterating $\mathbf{X} \leftarrow \mathbf{X} \cdot \mathbf{P}$ [STE 94].

Let us tackle the general case assuming only that the terminal SCC (denoted $\{\mathcal{C}_1, \dots, \mathcal{C}_k\}$) are aperiodic with steady-state distributions $\{\boldsymbol{\pi}_1, \dots, \boldsymbol{\pi}_k\}$. In this case, the chain also generates a stationary distribution (which here depends on the initial distribution). This distribution is given by formula $\boldsymbol{\pi} = \sum_{i=1}^{k} \Pr(\text{to reach } \mathcal{C}_i) \cdot \boldsymbol{\pi}_i$. Thus, it remains to compute the probability of reaching a terminal SCC. We evaluate this quantity starting from a fixed state and then condition it following the initial distribution: $\Pr(\text{to reach } \mathcal{C}_i) = \sum_{s \in S} \boldsymbol{\pi}_0(s) \cdot \boldsymbol{\pi}'_{\mathcal{C}_i}(s)$ where $\boldsymbol{\pi}'_{\mathcal{C}_i}(s) = \Pr(\text{to reach } \mathcal{C}_i \mid X_0 = s)$. Let $\mathbf{P}_{T,T}$ be the transition submatrix of the chain restricted to the transient states and $\mathbf{P}_{T,i}$ the transition submatrix from transient states to states of $\mathcal{C}_i$, then $\boldsymbol{\pi}'_{\mathcal{C}_i} = (\sum_{n \geqslant 0} (\mathbf{P}_{T,T})^n) \cdot \mathbf{P}_{T,i} \cdot \mathbf{1}^T = (\mathbf{I} - \mathbf{P}_{T,T})^{-1} \cdot \mathbf{P}_{T,i} \cdot \mathbf{1}^T$. The first equality is obtained by conditioning the reachability of $\mathcal{C}_i$ by the possible length of the associated path whereas the second can be straightforwardly checked.

9.2.3. *Continuous-time Markov chains*

9.2.3.1. *Presentation*

A continuous-time Markov chain (CTMC) presents the following characteristics:

– the delay between successive instants T_n is a random variable whose distribution is a negative exponential with a rate only depending on state X_n. Otherwise stated:

$$\begin{aligned} &\Pr(T_n \leqslant \tau \mid X_0 = s_{i_0}, \dots, X_n = s_i, T_0 \leqslant \tau_0, \dots, T_{n-1} \leqslant \tau_{n-1}) \\ &\qquad = \Pr(T_n \leqslant \tau \mid X_n = s_i) = 1 - e^{-\lambda_i \cdot \tau}; \end{aligned}$$

– the successor state of the current state only depends on this state and transition probabilities are time-invariant[2]:

$$\begin{aligned}\Pr(X_{n+1} = s_j \mid X_0 = s_{i_0}, \ldots, X_n = s_i, T_0 \leqslant \tau_0, \ldots, T_{n-1} \leqslant \tau_{n-1}) \\ = \Pr(X_{n+1} = s_j \mid X_n = s_i) = p_{ij} =_{def} \mathbf{P}[i,j].\end{aligned}$$

The discrete chain defined by matrix $\mathbf{P}$ is called *embedded chain*. It "observes" the state changes of the CTMC without taking into account time elapsing. A state of the CTMC is *absorbing* if it is absorbing with respect to the embedded DTMC.

9.2.3.2. *Transient and steady-state behaviors of CTMC*

In a CTMC, due to the memoryless characteristic of the exponential law, at any time the evolution of DES only depends on its current state.

More precisely, the process is characterized by its initial distribution $\boldsymbol{\pi}(0)$, matrix $\mathbf{P}$ and the family of λ_is. Let us call $\boldsymbol{\pi}(\tau)$ the distribution of $Y(\tau)$ and $\pi_k(\tau) = \boldsymbol{\pi}(t)(s_k)$. If δ is small then the probability that more than one event occurs between τ and $\tau + \delta$ is negligible and the probability that one event occurs and triggers a state change from k to k' is approximatively equal to $\lambda_k \cdot \delta \cdot p_{kk'}$ (by definition of the exponential law).

$$\pi_k(\tau + \delta) \approx \pi_k(\tau) \cdot (1 - \lambda_k \cdot \delta) + \sum_{k' \neq k} \pi_{k'}(\tau) \cdot \lambda_{k'} \cdot \delta \cdot p_{k'k}$$

Thus:

$$\frac{\pi_k(\tau + \delta) - \pi_k(\tau)}{\delta} \approx \pi_k(\tau) \cdot (-\lambda_k) + \sum_{k' \neq k} \pi_{k'}(\tau) \cdot \lambda_{k'} \cdot p_{k'k}$$

And finally:

$$\frac{d\pi_k}{d\tau} = \pi_k(\tau) \cdot (-\lambda_k) + \sum_{k' \neq k} \pi_{k'}(\tau) \cdot \lambda_{k'} \cdot p_{k'k}.$$

Let us define matrix $\mathbf{Q}$ by: $q_{kk'} = \lambda_k \cdot p_{kk'}$ for $k \neq k'$ and $q_{kk} = -\lambda_k (= -\sum_{k' \neq k} q_{kk'})$. Then the previous equation can be rewritten as:

$$\frac{d\boldsymbol{\pi}}{d\tau} = \boldsymbol{\pi} \cdot \mathbf{Q}. \tag{9.4}$$

2. Here again, we say that the chain is *homogenous*.

Matrix $\mathbf{Q}$ is called the *infinitesimal generator* of the CTMC. From (9.4), this generator entirely specifies the chain evolution.

If this equation establishes the memoryless feature of a CTMC, it does not provide a practical means to compute the transient behavior of the chain. For this, we describe a second CTMC equivalent to the first one from a probabilistic point of view (a technique introduced in [JEN 53] and known as uniformization). Let us choose a value $\mu \geqslant \mathrm{Sup}(\{\lambda_i\})$. Whatever the current state, the delay before the next event follows an exponential law with (uniform) parameter μ. The state change is then triggered by transition matrix $\mathbf{P}^\mu$ defined by $\forall i \neq j, \mathbf{P}^\mu[s_i, s_j] = (\mu)^{-1} \cdot \lambda_i \cdot \mathbf{P}[s_i, s_j]$. The DTMC associated with matrix $\mathbf{P}^\mu$ is called the *uniformized chain* (for μ) of the original CTMC. The (straightforward) computation of the infinitesimal generator of the second CTMC shows that it is equal to that of the first chain. So this is the same stochastic process (forgetting the vanishing states). The transient distribution $\boldsymbol{\pi}(\tau)$ is obtained as follows. We compute the probability of being in s at time τ knowing that there have been n state changes during $[0, \tau]$. This probability is given by the uniformized chain and more precisely by $\boldsymbol{\pi}(0) \cdot (\mathbf{P}^\mu)^n$. Afterwards, we “uncondition” this probability by computing the probability of n state changes knowing that the delay between two changes follows the exponential law. This probability is given by $e^{-\mu\cdot\tau} \cdot (\mu \cdot \tau)^n / n!$. Thus:

$$\boldsymbol{\pi}(\tau) = \boldsymbol{\pi}(0) \cdot \left(e^{-\mu\cdot\tau} \sum_{n \geqslant 0} \frac{(\mu \cdot \tau)^n (\mathbf{P}^\mu)^n}{n!} \right).$$

In practice, this infinite sum does not raise any difficulties since it converges very quickly and the summation may be stopped as soon as the required precision is greater than $e^{-\mu\cdot\tau} \cdot (\mu \cdot \tau)^n / n!$.

Let us examine the asymptotic behavior of a CTMC. The simplest way to analyze its behavior consists of studying the embedded DTMC of the uniformized chain. As observed during the presentation of this approach, this chain is not unique. Let us select a DTMC obtained with a choice of $\mu > \mathrm{Sup}(\{\lambda_i\})$. In this case, every state s fulfills $\mathbf{P}^\mu[s, s] > 0$ and thus every terminal SCC of this chain is ergodic. This implies that it yields a steady-state distribution. This distribution measures the steady-state probability of the occurrence of a state. However, since the uniform description of the CTMC implies a mean sojourn time identical for every state ($1/\mu$), it also provides the steady-state distribution of the CTMC.

In the particular (and frequent) case where the embedded chain is ergodic, this distribution is obtained by solving the equation $\mathbf{X} = \mathbf{X} \cdot \mathbf{P}^\mu$. We observe that $\mathbf{P}^\mu = \mathbf{I} + (1/\mu)\mathbf{Q}$. Thus, the distribution is also the unique solution of the equation:

$$\mathbf{X} \cdot \mathbf{Q} = 0 \quad \text{and} \quad \mathbf{X} \cdot \mathbf{1}^T = 1. \tag{9.5}$$

By analogy, we say that the CTMC is ergodic.

9.3. High level stochastic models

The notion of a stochastic process provides a probabilistic framework for DES, and Markov chain are a subclass of processes whose analysis is feasible. However, the modeler wishes to use a specification model at a higher level whose semantics given by a stochastic process then allows a quantitative analysis. It turns out that the stochastic semantics of a high level model raises specific problems. The goal of this section is to show how to identify and solve these problems.

We have chosen to illustrate these semantical features through the model of Petri nets. We first introduce a general model and then a Markovian sub-model in order to show how to obtain the characteristics of the corresponding CTMC. We assume that the reader is aware of the model of ordinary Petri nets. Otherwise, we advise consulting [GIR 03].

9.3.1. *Stochastic Petri nets with general distributions*

The stochastic feature of Petri nets is introduced by considering that a transition has a variable *firing delay* once it is enabled, obtained by sampling some distribution (ranging over $\mathbb{R}^+$) and that firing conflicts are solved by some probabilistic sampling depending on the transition *weights*. Different subclasses of stochastic Petri nets are defined by restricting these kinds of distributions. In the next definition, we do not restrict these distributions in any way.

DEFINITION 9.1.– *A (marked) stochastic Petri net with general laws (GLSPN)* $N = (P, T, Pre, Post, \Phi, w, m_0)$ *is defined by:*

– *P, the finite set of places;*

– *T, with $P \cap T = \emptyset$, the finite set of transitions;*

– *Pre (respectively $Post$), the backward (respectively forward) incidence matrix from $P \times T$ to $\mathbb{N}$;*

– *Φ, a function from T to the set of distributions ranging over $\mathbb{R}^+$, the law of transition delays;*

– *w, a function from T to $\mathbb{R}^{+*}$, the transition weights;*

– *m_0, a vector of $\mathbb{N}^P$, the initial marking.*

The introduction of distributions and weights is not enough to specify the stochastic process associated with the GLSPN. We are going to study the problems related to this specification.

NOTE.– Most of the parameters of this process could depend on the current marking. For sake of readability, we will not consider it in the following discussion.

9.3.1.1. *Choice policy*

Given that some marking has been reached, we need to determine the next transition to fire among the enabled ones. There are two possibilities:

– a probabilistic choice according to a distribution proportional to the weight of enabled transitions. This is a *pre-selection* since the choice takes place before the sampling of the delay;

– conversely, an independent sampling of the delay for every enabled transition followed by the choice of the shortest delay (with a possible re-use of some previous sampling; see below). When there are multiple shortest delays, we perform an additional probabilistic choice according to the weights, called *post-selection*.

The second solution is generally chosen since on the one hand it corresponds to most of the modeling interpretations and on the other hand, with the help of immediate transitions (see section 9.3.4), the pre-selection can be simulated by the post-selection. Observe that, except when the distributions are continuous, the specification of a distribution for a post-selection is required (here by the weight of transitions).

9.3.1.2. *Service policy*

If, given a marking m, a transition t has enabling degree $e = \lfloor \mathrm{Inf}(\{m(p)/ Pre[p,t]\}) \rfloor > 1$, we may consider that the marking *provides* e clients to the transition viewed as a server. So when sampling the delay, there are three options depending on the modeled event:

– a single sampling is performed, the transition accepts clients one by one (*single-server* mode);

– e samplings are performed, the transition accepts all the clients (*infinite-server* mode);

– $\mathrm{Inf}(\{e, deg(t)\})$ samplings are performed, the transition accepts no more than $deg(t)$ clients simultaneously; this case generalizes the previous ones (with $deg(t) = 1$ or ∞) (*multiple-server* mode). Here the modeler must specify $deg(t)$ for every transition.

9.3.1.3. *Memory policy*

Once transition t is fired, what is the effect of the sampling of another transition t' for its next firings?

The first possibility consists of forgetting this sampling. If transition t' is still enabled, a new sampling is performed (*resampling memory* mode). With such a semantic, t could model a crash (immediately repaired) of a service specified by t'.

The second possibility consists of storing the sampling decremented by the shortest sampling, only if t' is still enabled (enabling memory PRD (*Preemptive Repeat*

Different) mode). If t' is disabled, this mechanism could model a *time-out* (t') canceled by the firing of t.

The third possibility is identical when the transition is still enabled but keeps the sampling unchanged if t' is disabled. This sampling will be used again when t' becomes enabled (*enabling memory* PRI (*Preemptive Repeat Identical*) mode). Transition t' when disabled could model a job aborted by t in order to be achieved later in the same conditions.

The fourth possibility consists of storing the sampling decremented by the shortest sampling. A transition t' once disabled could model a job suspended by t (*age memory* mode, also called PRS (*Preemptive ReSume*)).

In order to achieve the specification of this policy, we must take into account the case of multiple-server transitions, which requires selecting which samplings should be kept, suspended or forgotten. The simplest solution is a First In First Out (FIFO) policy for samplings. The last sampling is the first one to be suspended or forgotten. Other policies (such as suspend or forget the least advanced "client") may be incompatible with some analysis methods.

It is clear that once these three policies are defined, the stochastic process is determined without ambiguity. Let us now restrict the kinds of delay distributions.

9.3.2. *GLSPN with exponential distributions*

In the original model [FLO 85, MOL 81] (called *stochastic Petri net* or SPN), every transition t has a delay distribution which is a negative exponential with rate $\lambda(t)$ (assuming an enumeration of T, we denote $\lambda_k = \lambda(t_k)$).

Let us examine the stochastic process generated by such a net with *single-server* mode. Let m be a marking, $t_1, \ldots, t_k$ the enabled transitions from m. We check that:

– the sojourn time in m is an exponential law with rate $\lambda_1 + \cdots + \lambda_k$;

– the probability of firing t_i before the other transitions is equal to $\frac{\lambda_i}{\sum \lambda_j}$ and it does not depend on the elapsed sojourn time;

– the distribution of the remaining firing delay of t_i knowing that t_j is fired is identical to its initial distribution (memoryless law).

Otherwise stated, the current marking fully determines the future behavior of the stochastic process. Thus this process is a continuous-time Markov chain, "isomorphic" to the reachability graph of the net, whose parameters are deduced from the states (i.e. the reachable markings). This reasoning can be generalized to the other modes. Assuming that the graph is finite, the transient and steady-state analyses of this model are performed as described in section 9.2.3.

9.3.3. *Performance indices of SPN*

Since the state of a Petri net is a marking, the functions associated with indices are expressed by numerical expressions where variables are the possible place markings (x_p representing the marking of place p). For instance, assume there is a place cli counting the number of clients and a place off witnessing the fact that the server is unavailable. The expression x_{cli} provides the number of clients whereas $x_{cli} \geqslant 1 \wedge x_{\mathit{off}} = 1$ means that at least one client is waiting for a service during the unavailability of the server.

Observe that, using some ad hoc reasoning, it is possible to compute indices related to transitions. For instance, assume that t represents the arrival of a client and that we want to compute the probability that a client arrives when the server is unavailable. First we identify markings where t is enabled. For any such marking m, we compute the probability that t is the next transition to be fired. In SPNs, this probability that we denote $\pi_{fire}(m,t)$ is obtained as the ratio between its rate and the sum of the rates of enabled transitions from m. Let $\boldsymbol{\pi}$ be the state distribution for which the index has to be measured, then the searched value is:

$$\frac{\sum_{m \xrightarrow{t} m' \wedge m'(\mathit{off})=1} \boldsymbol{\pi}(m) \cdot \pi_{fire}(m,t)}{\sum_{m \xrightarrow{t} m'} \boldsymbol{\pi}(m) \cdot \pi_{fire}(m,t)}.$$

9.3.4. *Overview of models and methods in performance evaluation*

The domain of performance evaluation is extremely vast due to the long history of telecommunications. Thus, we limit ourselves to skipping through this area, referring to books for the interested reader.

The first evaluation models were the queues and then the queuing networks [KLE 75, KLE 76]. This model mainly focuses on the kinds of laws, the type and the number of clients, the service policies and the routing between queues. However, although appropriate for studying telecommunication networks, it misses generic mechanisms in order to model systems (such as operating systems) which include synchronization between components. We can add ad hoc mechanisms but it is safer to add probabilistic features to functional models of concurrent systems.

Stochastic Petri nets have been the support of numerous modelings with ordinary Petri nets [AJM 95]. Their elementary firing rule makes it possible to easily transform them into a stochastic model as we have done in the previous section. Furthermore, high-level Petri nets which provide support for data structure and parametrization of actions have also be enlarged with a stochastic semantics [CHI 93b].

Process algebra integrate compositional features and thus are appropriate for a hierarchical design. So, they have also been enlarged with a stochastic semantics [HIL 96]. Contrary to Petri nets, this semantic raises subtle problems like, for instance, the quantitative specification of action synchronization.

The evaluation methods are generally classified with respect to the complexity of computations. We follow this order restricting the references to the ones related to Petri nets for sake of conciseness.

When the structure of a model is really simple, it is possible to obtain a formula expressing the steady-state distribution with respect to the numerical parameters. In the case of a formula obtained by sub-formulae associated with the system components, we call such formula a product form. In the framework of Petri nets, a product form is difficult to obtain due to the synchronization required for firing a transition. However, for subclasses of Petri nets such formulae have been established [HEN 90, HAD 05].

In the general case, it is necessary to generate the CTMC associated with the SPN and to compute its steady-state distribution. When the size of the chain is too large, an analysis of the structure of the net makes it possible to compute bounds [CHI 93a] or to design approximate methods [CAM 94]. In the case of high-level nets, the aggregation techniques *a priori* generate a smaller CTMC equivalent to the CTMC of the net [CHI 93b]. When the Petri net is unbounded (i.e. when the place markings may be arbitrarily large), the associated CTMC is infinite. However, if a single place is unbounded, it is still possible to obtain the steady-state distribution [HAV 95].

If exponential laws are suitable to model events whose time distribution is unknown, some operations have a duration belonging to some time interval or even close to be constant. In such cases, the choice of an exponential law leads to loose approximations. Thus, a study of nets including transitions with deterministic laws (also called Dirac laws) has been undertaken [AJM 87, LIN 98]. Numerous alternative approaches have then been proposed depending on the kind of laws and the occurrence of the corresponding transitions [DON 98, GER 99, LIN 98, LIN 99].

9.3.5. *The GreatSPN tool*

GreatSPN [AJM 95] is one of the most prominent tools for the qualitative and quantitative analysis of Petri nets, developed since the beginning of the 1980s by the performance evaluation team of Torino University. First analyzing SPNs, it has gradually integrated the semantical extensions related to the transition distributions. It is also the only tool to cope with the model of the stochastic well-formed Petri net (SWN), a high-level model which takes advantage of behavioral symmetries in order to obtain a lumped CTMC directly from the definition of the net. GreatSPN is available from the authors, free for academic use. The software includes a graphical interface for the definition of the net, which also handles non-graphical properties (like delays) and triggers the computations. Most of the results are presented in the graphical interface.

9.3.5.1. *Supported models*

GreatSPN manages ordinary Petri nets whose transitions have negative exponential distributions (standard SPN), but also phase type distribution (like ERLANG), deterministic distribution (i.e. constant) and the null Dirac distribution (immediate transitions). GreatSPN also manages stochastic well-formed Petri nets, the most widely used high-level stochastic Petri net model. Both models are analyzed using the graphical interface.

9.3.5.2. *Qualitative analysis of Petri nets*

The tool integrates most of the standard analysis methods for Petri nets: boundedness checking, linear invariant computations, trap and deadlock computations, etc. When the net is bounded, GreatSPN computes and saves on disk its reachability graph. For an SWN, it computes a symbolic reachability graph which can be transformed into a lumped Markov chain.

9.3.5.3. *Performance analysis of stochastic Petri nets*

GreatSPN computes the transient and steady-state distributions of bounded nets. The user can also specify complex performance indices that the tool computes on demand. The software also provides distributions and performance indices for SWN at the lumped level and if necessary at the ordinary level.

Despite numerous improvements for the state representation (i.e. the markings), the size of the reachability graph may forbid the exact resolution of very large models. Thus, GreatSPN includes a stochastic simulator in order to cope with such situations.

9.3.5.4. *Software architecture*

The implementation of GreatSPN is based on a modular structure which follows the analysis process for Petri nets (definition, qualitative analysis, performance evaluation). Every step is realized by programs written in C, whose executable code can also be triggered by commands. This makes it possible to write scripts in order to combine the different algorithms implemented in the tool. The reader can refer to the site `www.unito.it/~GreatSPN` for more details.

9.4. Probabilistic verification of Markov chains

The detailed discussion refers to CTMC.

9.4.1. *Limits of standard performance indices*

Performance indices defined above give valuable information to the system designer. However, they do not express all significant measures. Let us illustrate this point with the help of a service availability. Some properties related to this concept are as follows:

– *instantaneous* availability guarantee in transient mode. That is, the probability, at a given time τ, of the service availability;

– instantaneous availability guarantee in steady-state. That is, the probability, at any time, of the service availability in steady-state;

– sustained availability guarantee in transient mode. That is, the probability that the system is permanently available between times τ and τ';

– sustained availability guarantee in steady-state. That is, the probability that the service is permanently available between two instants in steady-state. Since the process is in steady-state, this index depends only on the duration between these two instants;

– availability and response time guarantee in steady-state. That is, the probability that, after a request, the service stays on until the response and that the response time is lower than a given upper bound.

Although the first two properties may be easily deduced from the transient and steady-state distributions, this is not the case for the other properties. We could figure out an ad hoc algorithm for each of these. It is however more judicious to introduce a new logic to express complex performance indices and to design a general evaluation algorithm for this logic.

9.4.2. *A temporal logic for Markov chains*

The continuous stochastic logic (CSL) is an adaptation of the computation tree logic (CTL) [EME 80] to continuous Markov chains. It expresses formulae evaluated on states with the following syntax. Here we mainly refer to the approach of [BAI 03a].

DEFINITION 9.2.– *A CLS formula is inductively defined as:*

– *if $\phi \in \mathcal{P}$, then ϕ is a CSL formula;*

– *if ϕ and ψ are CSL formulae, then $\neg\phi$ and $\phi \wedge \psi$ are CSL formulae;*

– *if ϕ is a CSL formula, $a \in [0,1]$ is a real number, $\bowtie \in \{<, \leqslant, >, \geqslant\}$, then $S_{\bowtie a}\phi$ is a CSL formula;*

– *if ϕ and ψ are CSL formulae, $a \in [0,1]$ is a real number, $\bowtie \in \{<, \leqslant, >, \geqslant\}$ and I is an interval of $\mathbb{R}_{\geqslant 0}$, then $P_{\bowtie a}\mathcal{X}^I\phi$ and $P_{\bowtie a}\,\phi\mathcal{U}^I\psi$ are CSL formulae.*

Only the two last items require some explanation. Formula $S_{\bowtie a}\phi$ is satisfied by a state s of the chain if, for the process started in s, the stationary cumulated probability (say p) of the states satisfying ϕ verifies $p \bowtie a$. The value of this formula is well defined since a finite CTMC has a stationary distribution. Let us note that this value does not depend on state s if the chain is ergodic.

A sample of the stochastic process satisfies $\mathcal{X}^I\phi$ if the first state change occurs in the interval I and if the reached state verifies ϕ. State s satisfies $P_{\bowtie a}\mathcal{X}^I\phi$ if the probability (say p) that a sample of the process started in s satisfies the given condition fulfills $p \bowtie a$.

A sample of the stochastic process satisfies $\phi\mathcal{U}^I\psi$ if there is a time $\tau \in I$ such that ψ is satisfied and at all previous times ϕ is satisfied. State s satisfies $P_{\bowtie a}\phi\mathcal{U}^I\psi$ if the probability (say p) that a sample of the process, started in s satisfies the given condition fulfills $p \bowtie a$.

To exemplify these definitions, let us give the formal expressions of the availability properties expressed above.

– 99% instantaneous availability guarantee in transient mode:

$$P_{\geqslant 0.99} true\mathcal{U}^{[\tau,\tau]} disp$$

where $disp$ is an atomic proposition meaning that the service is available.

– 99% instantaneous availability guarantee in steady-state:

$$S_{\geqslant 0.99} disp$$

– 99% sustained availability guarantee in transient mode:

$$P_{<0.01} true\mathcal{U}^{[\tau,\tau']} \neg disp$$

– 99% sustained availability guarantee in steady-state:

$$S_{<0.01} true\mathcal{U}^{[\tau,\tau']} \neg disp$$

– 99% availability and response time (3 time units) guarantee in steady-state:

$$S_{\geqslant 0.99}(req \Rightarrow P_{\geqslant 0.99}(disp\mathcal{U}^{[0,3]} ack))$$

where req is an atomic proposition meaning a receiving request and ack is an atomic proposition meaning an answer to a request. Let us note that the two 99% occurrences do not have the same meaning. The internal operator occurrence is a requirement on the behavior of the process started in a specific state, while the second one is a global requirement on the states weighed with a stationary distribution. *A priori*, different required values could have been specified.

9.4.3. *Verification algorithms*

Given a CTMC and a CSL formula ϕ, the verification algorithm proceeds by the successive evaluation of the sub-formulae of ϕ, "upwards" in the syntactic tree of the formula ϕ, from leaves to the root, labeling each state with the sub-formulae this state verifies. Thus, every step of the algorithm evaluates a formula viewing the operands of the most external operator as atomic propositions.

This leads us to study each operator.

$\boxed{\phi = \neg\psi}$ The algorithm labels each state with ϕ if it is not labeled with ψ.

$\boxed{\phi = \psi \wedge \chi}$ The algorithm labels each state with ϕ if it is labeled with ψ and χ.

$\boxed{\phi = S_{\bowtie a}\psi}$ The algorithm computes the steady-state distribution of the process started in s (as mentioned in section 9.2.3). Then it sums up the probabilities of the states labeled with ψ and labels s with ϕ if the computed value (say p) verifies $p \bowtie a$. Let us note that, for all states of a sink SCC, only one computation is required. Also, if the CTMC has a unique stationary distribution, then the truth value of the formula is state independent.

$\boxed{\phi = P_{\bowtie a}\mathcal{X}^I\psi}$ Let s be a state. The occurrence of the next transition inside the interval I and the satisfaction of ψ by the reached state are two independent events. Thus, the searched probability is the product of the probabilities of these events. Let us denote $I = [\tau, \tau']$; we assume without loss of generality that intervals are closed. Indeed, since distributions are continuous, taking or not taking interval bounds into account does not matter with regard to the value of the formula. If $\mathbf{Q}$ is the infinitesimal generator of the chain and $\mathbf{P}$ the transition matrix of the embedded Markov chain, then the probability of the first event is $e^{\tau\mathbf{Q}[s,s]} - e^{\tau'\mathbf{Q}[s,s]}$ and the probability of the second event is $\sum_{s' \models \psi} \mathbf{P}[s, s']$.

$\boxed{\phi = P_{\bowtie a}\psi\mathcal{U}^I\chi}$ The evaluation of this formula mainly consists of transient analyses of chains derived from the original chain by elementary transformations. For a chain X, we denote by X^ϕ the chain derived by transforming all states verifying ϕ as absorbing states. To simplify matters, we study the various kinds of intervals.

– $\phi = P_{\bowtie a}\psi\mathcal{U}^{[0,\infty[}\chi$. In this case, the sample of the process must stay in states verifying ψ until reaching a state verifying χ, without time constraint. In other words, we track the behavior of the chain until a state verifying $\neg\psi \vee \chi$. Let us study the chain $X^{\neg\psi\vee\chi}$. If a terminal SCC of this chain holds a state verifying χ, then it is reduced to a single state and the requested probability is 1, otherwise this probability is zero for all states of the SCC since they cannot reach a state verifying χ. Let us call "good" a SCC associated with probability 1. So, the requested probability for the remaining states is the probability of reaching a state of a good SCC. This probability depends only on the embedded chain of $X^{\neg\psi\vee\chi}$ and its computation has been described in section 9.2.2.

– $\phi = P_{\bowtie a}\psi\mathcal{U}^{[0,\tau]}\chi$. In this case, the sample of the process must stay in states verifying ψ until reaching a state verifying χ no later than time τ. In other words, we track the behavior of the chain until reaching a state verifying $\neg\psi \vee \chi$. Thus, the probability to be computed is $\Pr(X^{\neg\psi\vee\chi}(\tau) \vDash \chi \mid X^{\neg\psi\vee\chi}(0) = s)$.

– $\phi = P_{\bowtie a}\psi\mathcal{U}^{[\tau,\tau]}\chi$. In this case, the sample of the process must stay in states verifying ψ during the interval $[0, \tau]$ and moreover these states must verify χ at time τ. We overlook state changes at time τ since its probability is zero. Thus, the probability to be computed is $\Pr(X^{\neg\psi}(\tau) \vDash \psi \wedge \chi \mid X^{\neg\psi}(0) = s)$.

– $\phi = P_{\bowtie a}\psi\mathcal{U}^{[\tau,\infty[}\chi$. In this case, the sample of the process must stay in states verifying ψ during the interval $[0, \tau]$, then it must verify the formula $\psi\mathcal{U}^{[0,\infty[}\chi$ from state s reached at time τ. The requested probability is then $\sum_{s'\vDash\psi} \Pr(X^{\neg\psi}(\tau) = s' \mid X^{\neg\psi}(0) = s) \cdot \boldsymbol{\pi}(s')$ where $\boldsymbol{\pi}(s')$ is computed as in the first case.

– $\phi = P_{\bowtie a}\psi\mathcal{U}^{[\tau,\tau']}\chi$. The same reasoning as above leads to the following formula for the requested probability: $\sum_{s'\vDash\psi} \Pr(X^{\neg\psi}(\tau) = s' \mid X^{\neg\psi}(0) = s) \cdot \Pr(X^{\neg\psi\vee\chi}(\tau' - \tau) \vDash \chi \mid X^{\neg\psi\vee\chi}(0) = s')$.

9.4.4. *Overview of probabilistic verification of Markov chains*

Historically, the verification of discrete-time chains happened before the verification of continuous-time chains. The first attempt for verifying LTL formulae on DTMC [VAR 85] is conceptually easy: translate the formula into a Büchi automaton; then determinize this automaton into a Rabin automaton; build the synchronized product of this automaton with the DTMC, leading to a new DTMC; and apply a variant of the analysis presented in section 9.2. Unfortunately, the complexity of this algorithm is doubly exponential with respect to the size of the formula. In [COU 95], the authors also build a new DTMC by iterative refinement of the initial DTMC analyzing the formula operators. This leads to a simple exponential procedure. Moreover, they show that this is the optimal complexity. A third algorithm [COU 03] also translates the formula into a Büchi automaton. However, the chosen translation algorithm allows us to directly evaluate the probability related to the formula on the synchronized product of the automaton and the DTMC. This method also presents an optimal theoretical complexity and behaves better than the previous method in effective modeling cases.

A standard analysis technique for performance models is to combine "rewards" to states and/or to transitions of a chain and to compute performance indices related to these rewards. In order to extend the scope of probabilistic verification to such models, a new logic, PRCTL, is introduced in [AND 03] together with a formula evaluation algorithm.

The first significant works on CTMC were established in [AZI 96, AZI 00]. They proved that verification of CSL formulae on CTMC is decidable. However, the corresponding algorithm is very complicated since it forbids the approximations we have implicitly used in the computations of the previous section.

In fact, even with the above method, computation may become intractable for large Markov chains. An efficient approach to cope with this problem is to profit from the modularity of the specification. In this context, we try to replace a module with a smaller but equivalent one with respect to the formula to be checked. Then we check the model made up of reduced modules. This approach, initialized in [BAI 03a], was generalized in [BAI 03b] which studies numerous kinds of equivalence.

A completely different approach to reduce complexity of the verification is proposed in [YOU 06]. Assuming we have to check the formula $P_{\leqslant a}\phi$, we can generate random executions and compute the ratio of executions satisfying ϕ; in accordance with classical probability results, this value converges on the requested probability. This method is very efficient when the evaluation of the formula ϕ only requires a time bounded execution.

9.4.5. *The ETMCC tool*

ETMCC (Erlangen-Twente Markov chain model checker) is the first model checking tool (2000) for CSL formulae on CTMC. It is developed by several German universities, originally Erlangen, and the Deutch university of Twente. A limited version is available from the authors. ETMCC allows analysis of CTMC with properties given in CSL extended with several convenient features for the modeler, including rewards. Tool usage is standard: model specification with a specific formalism, definition of the properties with probabilistic temporal logic and results analysis. Even if ETMCC also analyzes discrete-time models, we present functionalities for continuous-time systems for which it was specifically designed. Moreover, the discrete-time case is the subject of common work with the PRISM tool team (see below).

9.4.5.1. *Language of system models*

ETMMC has chosen a very simple solution: models are CTMC, provided by the user as the rate matrix given through textual sparse version. Let us note this allows us to use ETMCC from various higher-level models (process algebras, Petri nets, etc.) as soon we are able to generate their (finite) state spaces. It is enough to translate these descriptions into the textual input format of ETMCC. The list of atomic properties satisfied on states must also be provided through a text file.

9.4.5.2. *Language of properties*

ETMCC allows us to define properties with the CSL logic detailed above. They are given through the graphical user interface. CSL is extended to mean reward computations in transient or steady-state mode, from state or event reward functions; this allows the practitioner to obtain useful indices such as the mean cost of a transaction during a given duration.

9.4.5.3. *Computed results*

The software analyzes the model with respect to required properties and shows results, truth values of CSL formulae and reward indices, through the user interface.

9.4.5.4. *Software architecture*

ETMCC is implemented in Java. The most involved part is the evaluation of bounded time (Next and Until) operators. It makes use of numerical integration solvers or CTMC uniformization solvers. The application also involves graph algorithms especially for computation of maximal SCC. The graph of reachable states is implemented with a sparse representation. This representation is also used for all numerical computations. The reader can visit the ETMCC web site `http://www7.informatik.uni-erlangen.de/etmcc` for a detailed description of the tool.

9.5. Markov decision processes

9.5.1. *Presentation of Markov decision processes*

Let us assume that we have to analyze executions of a set of transactions such that each one can be modeled with a CTMC. Seeking for a model of the global system, we are then faced to the fact that we do not know the scheduler of the transactional system. We could model choices of the scheduler as probabilistic actions and thus, come down to a global CTMC. However, this solution reduces significantly the scope of computed measures. Indeed, such results would be interpreted as performance indices of an "average" scheduler. However, in practice, we are looking for extremal indices such as maximal probability of a transaction abort, for all (an infinite number of) schedulers.

It is thus necessary to use a more expressive formalism than DTMC. More precisely, this formalism must allow us to express probabilistic choices and non-deterministic choices. This naturally leads us to *Markov decision process* (MDP).

DEFINITION 9.3.– *A Markov decision process* $\Sigma = (S, \mathcal{P}, V, next)$ *is defined by:*

– S*, the (finite) set of states;*

– $\mathcal{P}$*, the (finite) set of atomic propositions;*

– V*, the state characteristic function which associates with each state* s*, the subset* $V(s)$ *of* $\mathcal{P}$ *of true propositions in* s*;*

– $next$*, the function which associates with each state* s*, the set* $next(s) = \{\boldsymbol{\pi}_1^s, \ldots, \boldsymbol{\pi}_{k_s}^s\}$ *of distributions with support* S*.*

The fundamental element of this definition is the $next$ function which controls the evolution of the process. In state s, the process non-deterministically chooses a distribution $\boldsymbol{\pi}_i^s \in next(s)$, and then samples this distribution, which defines the next state.

In agreement with this interpretation, we introduce a (immediate) succession relation ρ defined by $\rho(s, s') \Leftrightarrow \exists \boldsymbol{\pi}_i^s,\ \boldsymbol{\pi}_i^s(s') > 0$. An execution path is then a sequence of states such that every pair of consecutive states fulfills this relation.

We wish to put this system in a full probabilistic setting. To do so, we define so called strategies. A strategy t is a function providing for every execution path $s_0, s_1, \ldots, s_n$, $St(s_0, s_1, \ldots, s_n)$, a distribution from $next(s_n)$. For a given strategy and a given initial state, the Markov decision process behaves like a discrete-time stochastic process so that the probability of an event Ev of this process is well defined and will be denoted by $\mathrm{Pr}^{St}(Ev)$.

9.5.2. *A temporal logic for Markov decision processes*

Probabilistic computation tree logic (pCTL), the temporal logic that we will present, is an adaptation of CTL to Markov decision processes. We refer here mainly to the approach of [BIA 95]. Formulae of pCTL are evaluated on states with the following syntax.

DEFINITION 9.4.– *A pCTL formula is inductively defined by:*

– if $\phi \in \mathcal{P}$, then ϕ is a pCTL formula;

– if ϕ and ψ are pCTL formulae, then $\neg\phi$ and $\phi \wedge \psi$ are pCTL formulae;

– if ϕ and ψ are pCTL formulae, $a \in [0, 1]$ is a real number and $\bowtie \in \{<, \leqslant, >, \geqslant\}$, then $A\phi\mathcal{U}\psi$, $E\phi\mathcal{U}\psi$ and $P_{\bowtie a}\phi\mathcal{U}\psi$ are pCTL formulae.

Only the last point requires some explanation. The first two operators do not involve numerical values of the distributions. $A\phi\mathcal{U}\psi$ (respectively $E\phi\mathcal{U}\psi$) is true in state s if and only if every (respectively at least one) execution path starting in s comprises a prefix made of states satisfying ϕ followed by a state satisfying ψ. The last operator involves strategies in the following way: $P_{\bowtie a}\phi\mathcal{U}\psi$ is true in s if *for every strategy*, the probability (say p) for the corresponding stochastic process that an execution path stating in s comprises a prefix made of states satisfying ϕ, followed by a state satisfying ψ, satisfies $p \bowtie a$.

9.5.3. *Verification algorithms*

Given a Markov decision process, and a pCTL formula ϕ, the model checking algorithm proceeds by successive evaluation of sub-formulae of ϕ, "going upwards" the syntactic tree of the formula ϕ, from leaves to the root, and labeling each state with the sub-formulae it satisfies. Thus, each step of the algorithm evaluates a formula viewing the operands of the most external operator as atomic propositions.

This leads us to study each operator.

$\boxed{\phi = \neg\psi}$ The algorithm labels each state with ϕ if it is not labeled with ψ.

$\boxed{\phi = \psi \wedge \chi}$ The algorithm labels each state with ϕ if it is labeled with ψ and χ.

$\boxed{\phi = E\psi\mathcal{U}\chi}$ In a first step, the algorithm labels states labeled with χ. Then, going back up from these states using the precedence relation (ρ^{-1}) it labels states labeled with ψ. It iterates this step starting from the newly labeled states until saturation.

$\boxed{\phi = A\psi\mathcal{U}\chi}$ The algorithm uses a recursive function tagging visited states. When it evaluates a state labeled with χ, it labels it with ϕ and returns true. When it evaluates a state not labeled with χ or ψ it returns false. When it evaluates a state not labeled with χ but labeled with ψ and *not yet visited*, it calls the function for each successor of the state and labels it with ϕ if all calls return true. When it evaluates an *already visited* and not labeled state, it returns false. The reader could check that this procedure returns false if there is a path starting in the state, which comprises a state not labeled with ϕ or χ before every state labeled with χ, or a (infinite) path made of states labeled with ϕ but not with χ. This last case is detected by the presence of a circuit, thanks to tagging states.

$\boxed{\phi = P_{\geqslant a}\psi\mathcal{U}\chi}$ We only study this probabilistic operator since the other ones are similar. The algorithm computes simultaneously for all states the minimal probability (say $\boldsymbol{\pi}_{\min}(s) = \mathrm{Inf}(\{\mathrm{Pr}^{St}(s \vDash \psi\mathcal{U}\chi)\})$) of "good" paths and then compares it with a to find states to be labeled. If a state verifies χ, then whatever the strategy St, $\mathrm{Pr}^{St}(s \vDash \psi\mathcal{U}\chi) = 1$ and so $\boldsymbol{\pi}_{\min}(s) = 1$. Let us call S_{good} this subset of states. From S_{good}, we get the set of states where $\boldsymbol{\pi}_{\min}(s) > 0$. This set, denoted by $S_{>0}$, is first initialized to S_{good} and it is iteratively enlarged with states which have, whatever the strategy, a non-zero probability of reaching it in one step: $S_{>0} \leftarrow S_{>0} \cup \{s \mid \forall \boldsymbol{\pi}_s^i, \exists s' \in S_{>0}, \boldsymbol{\pi}_s^i(s') > 0\}$. This procedure necessary terminates. Let us call $S_{bad} = S \setminus S_{>0}$. We can easily check that $\forall s \in S_{bad}, \boldsymbol{\pi}_{\min}(s) = 0$. It remains to evaluate $S_{int} = S_{>0} \setminus S_{good}$. Since this evaluation is at the heart of the method and its extensions, we explain it in detail below and we prove its correctness.

First claim. Vector $\boldsymbol{\pi}_{\min}$ is a solution of equation (9.6) where vector $\mathbf{x}$ is the unknown.

$$\forall s \in S_{int}, \mathbf{x}(s) = \mathrm{Inf}\left(\left\{ \sum_{s' \in S_{int}} \boldsymbol{\pi}_s^i(s')\mathbf{x}(s') + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^i(s') \right\}_{1 \leqslant i \leqslant k_s}\right). \tag{9.6}$$

Proof. We state the equality, proving the two inequalities.
($\geqslant$) Let St be a strategy for the process starting in s, then:

$$\mathrm{Pr}^{St}(s \vDash \psi\mathcal{U}\chi) = \sum_{s' \in S_{int}} \boldsymbol{\pi}_s^{St(s)}(s')\mathrm{Pr}^{St_{s'}}(s' \vDash \psi\mathcal{U}\chi) + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^{St(s)}(s')$$

with $St_{s'}$ the strategy defined by $St_{s'}(s', \ldots, s_n) = St(s, s', \ldots, s_n)$.

So,

$$\begin{aligned}\Pr^{St}(s \vDash \psi\mathcal{U}\chi) &\geqslant \sum_{s' \in S_{int}} \boldsymbol{\pi}_s^{St(s)}(s')\boldsymbol{\pi}_{\min}(s') + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^{St(s)}(s') \\ &\geqslant \operatorname{Inf}\left(\left\{\sum_{s' \in S_{int}} \boldsymbol{\pi}_s^i(s')\boldsymbol{\pi}_{\min}(s') + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^i(s')\right\}_{1 \leqslant i \leqslant k_s}\right)\end{aligned}$$

With this final inequality being true for all St, we get:

$$\forall s \in S_{int}, \boldsymbol{\pi}_{\min}(s) \geqslant \operatorname{Inf}\left(\left\{\sum_{s' \in S_{int}} \boldsymbol{\pi}_s^i(s')\boldsymbol{\pi}_{\min}(s') + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^i(s')\right\}_{1 \leqslant i \leqslant k_s}\right).$$

($\leqslant$) Now, let $\epsilon > 0$ given, then, by definition of $\boldsymbol{\pi}_{\min}$, for all s' there is a strategy $St_{s'}$ such that $\Pr^{St_{s'}}(s' \vDash \psi\mathcal{U}\chi) \leqslant \boldsymbol{\pi}_{\min}(s') + \epsilon$. Given a state s, we build a strategy St for the process starting in s in the following way. St chooses the distribution $\boldsymbol{\pi}_s^i$ which minimizes $\sum_{s' \in S_{int}} \boldsymbol{\pi}_s^i(s')\Pr^{St_{s'}}(s \vDash \psi\mathcal{U}\chi) + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^i(s')$, then applies to the next reached state $s'$the strategy $St_{s'}$. By construction,

$$\begin{aligned}\forall i, \Pr^{St}(s \vDash \psi\mathcal{U}\chi) &\leqslant \sum_{s' \in S_{int}} \boldsymbol{\pi}_s^i(s')\Pr^{St_{s'}}(s' \vDash \psi\mathcal{U}\chi) + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^i(s') \\ &\leqslant \sum_{s' \in S_{int}} \boldsymbol{\pi}_s^i(s')(\boldsymbol{\pi}_{\min}(s') + \epsilon) + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^i(s') \\ &\leqslant \epsilon + \sum_{s' \in S_{int}} \boldsymbol{\pi}_s^i(s')\boldsymbol{\pi}_{\min}(s') + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^i(s')\end{aligned}$$

Thus,

$$\begin{aligned}\boldsymbol{\pi}_{\min}(s) &\leqslant \Pr^{St}(s \vDash \psi\mathcal{U}\chi) \\ &\leqslant \epsilon + \operatorname{Inf}\left(\left\{\sum_{s' \in S_{int}} \boldsymbol{\pi}_s^i(s')\boldsymbol{\pi}_{\min}(s') + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^i(s')\right\}_{1 \leqslant i \leqslant k_s}\right)\end{aligned}$$

This last equality being true for arbitrary small ϵ, the second equality is established and concludes the proof. □

Second claim. Vector $\boldsymbol{\pi}_{\min}$ is the *unique* solution of (9.6).

Proof. To establish uniqueness, we study *Markovian* strategies, that is to say strategies for which the chosen distribution depends only on the last state of the execution. Let us denote $St(s)$ the chosen distribution when this state is s. Let us also note that the behavior of the process is then a DTMC. The equation satisfied by a Markovian strategy is:

$$\begin{aligned}\forall s \in S_{int}, \quad \Pr^{St}(s \vDash \psi\mathcal{U}\chi) = &\sum_{s' \in S_{int}} \boldsymbol{\pi}_s^{St(s)}(s')\Pr^{St}(s' \vDash \psi\mathcal{U}\chi) \\ &+ \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^{St(s)}(s')\end{aligned} \tag{9.7}$$

To simplify notations, $\mathbf{x}(s)$ will stand for $\Pr^{St}(s \vDash \psi\mathcal{U}\chi)$, $\mathbf{A}[s, s']$ will stand for $\boldsymbol{\pi}_s^{St(s)}(s')$ and $\mathbf{b}(s)$ will stand for $\sum_{s' \in S_{good}} \boldsymbol{\pi}_s^{St(s)}(s')$. Equation (9.7) is then rewritten in vector notation $\mathbf{x} = \mathbf{A}{\cdot}\mathbf{x}{+}\mathbf{b}$. Quantity $\mathbf{A}^n[s, s']$ is the probability of being in s' starting from s after n steps without leaving S_{int}. From the definition of S_{int}, for every strategy, the probability of staying forever in S_{int} is zero, which is expressed in DTMC context by the convergence of the series $\sum_{n \geqslant 0} \mathbf{A}^n$ (see section 9.2.2). Replacing (n times) $\mathbf{x}$ with its expression in the right-hand side of the equality, we get $\mathbf{x} = \sum_{i \leqslant n} \mathbf{A}^i\mathbf{b} + \mathbf{A}^{n+1}\mathbf{x}$. Taking the limit, $\mathbf{x} = \sum_{n \geqslant 0} \mathbf{A}^n\mathbf{b}$, which means that (9.7) has a unique solution.

Now, let $\mathbf{x}$ be a solution of (9.6). Let us denote by St the Markovian strategy which chooses for a state s, the distribution $\boldsymbol{\pi}_s^i$ for which the minimum is reached with the solution $\mathbf{x}$. Then $\mathbf{x}$ satisfies (9.7) corresponding to this strategy. Thus, $\mathbf{x}(s) = \Pr^{St}(s \vDash \psi\mathcal{U}\chi)$. We deduce that $\forall s, \mathbf{x}(s) \geqslant \boldsymbol{\pi}_{\min}(s)$. Now, let St' be the Markovian strategy leading to $\boldsymbol{\pi}_{\min}$. We note that, for every $s \in S_{int}$:

$$\begin{aligned}&\sum_{s' \in S_{int}} \boldsymbol{\pi}_s^{St'(s)}(s')(\mathbf{x}(s') - \boldsymbol{\pi}_{\min}(s')) \\ &\quad = \left(\sum_{s' \in S_{int}} \boldsymbol{\pi}_s^{St'(s)}(s')\mathbf{x}(s') + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^{St'(s)}(s')\right) \\ &\quad - \left(\sum_{s' \in S_{int}} \boldsymbol{\pi}_s^{St'(s)}(s')\boldsymbol{\pi}_{\min}(s') + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^{St'(s)}(s')\right) \\ &\quad \geqslant \mathbf{x}(s) - \boldsymbol{\pi}_{\min}(s)\end{aligned}$$

Rewritten with previous vector notations, this is given by $\mathbf{A}(\mathbf{x} - \boldsymbol{\pi}_{\min}) \geqslant \mathbf{x} - \boldsymbol{\pi}_{\min} \geqslant \mathbf{0}$. By iteration, we get $\mathbf{A}^n(\mathbf{x} - \boldsymbol{\pi}_{\min}) \geqslant \mathbf{x} - \boldsymbol{\pi}_{\min} \geqslant \mathbf{0}$. Finally, taking the limit, $\mathbf{x} - \boldsymbol{\pi}_{\min} = \mathbf{0}$ which proves the claim. □

Third claim. Vector $\boldsymbol{\pi}_{\min}$ is the unique solution of the linear programming problem:

$$\text{Maximize} \sum_{s \in S_{int}} \mathbf{x}(s) \text{ under the constraints:}$$

$$\forall s \in S_{int}, \ \forall 1 \leqslant i \leqslant k_s, \ \mathbf{x}(s) \leqslant \sum_{s' \in S_{int}} \boldsymbol{\pi}_s^i(s')\mathbf{x}(s') + \sum_{s' \in S_{good}} \boldsymbol{\pi}_s^i(s').$$

Proof. If $\mathbf{x}$ satisfies the constraints of this problem, then $\mathbf{x}$ satisfies the version of (9.6) where equalities are replaced with inequalities. Let us assume moreover that $\mathbf{x}$ is an optimal solution and that one of the inequalities (say for $\mathbf{x}(s)$) of (9.6) is a strict inequality. Then we may replace $\mathbf{x}(s)$ with the right-hand side of this specific inequality and get a better solution. Thus, an optimal solution verifies (9.6) and from the second claim, $\boldsymbol{\pi}_{\min}$ is the unique solution of this problem. $\square$

So, evaluation consists of solving this linear programming problem.

Complexity. By construction, the algorithm has a linear complexity with respect to the size of the formula. The complexity as a function of the size of the process depends on operators. For non-probabilistic operators, the above description should convince the reader that the complexity is again linear. For probabilistic operators, the most costly step is the resolution of a linear programming problem whose time complexity is polynomial with interior point methods [ROO 97].

9.5.4. *Overview of verification of Markov decision processes*

Markov decision processes were introduced mainly for modeling and solving optimization problems [PUT 94]. One of the pioneering works ([COU 95]) on MDP model checking refers to probabilistic satisfaction of linear temporal logic properties, and establishes exact complexity bounds for evaluation of formulae in (propositional) LTL or expressed as Büchi automata. The approach of the previous section has been extended to various branching-time logics: pCTL* [BIA 95], a variant of CTL*, and pTL and pTL* introducing observation clocks [ALF 97]. Another extension concerns logic semantics. For instance, going back to our introductory example, the modeler does not wish to take all schedulers into account. Indeed, a "real life" scheduler satisfies (even weak) fairness properties with regard to the choice of the transaction to be executed. However, the usual method which modifies the formula to take into account these hypotheses cannot be used in this context. Thus, we must introduce fair operators into the logics and design adapted algorithms [BAI 98, ALF 00]. Finally, to cope with the inaccuracy of numerical parameters of the MDP, other logics have been introduced and analyzed [ALF 04].

Temporal logic evaluation methods for MDP may also be combined with other methods to deal with more expressive models. As an example, probabilistic timed automata specify continuous-time systems where non-determinism corresponds both to the choice of the next event and its occurrence time. When these automata are non-probabilistic, the usual method builds a finite abstraction of their (usually infinite) timed transition system, which is sufficient to model check formulae of some temporal logics (such as, for instance, TCTL). This abstraction leads to several representations: region graph, zone graph, etc. In the probabilistic case, abstraction is also feasible, but the result is then an MDP which is analyzed with methods explained above [KWI 01, KWI 02b, KWI 02a]. Other more expressive models can only be analyzed with approximation techniques [KWI 00].

9.5.5. *The PRISM tool*

PRISM (Probabilistic Symbolic Model Checker) is the reference tool for model checking of probabilistic systems, developed at Birmingham University, UK, since the beginning of 2000, and available under the GPL Licence. It allows us to analyze discrete-time systems (Markov chains and Markov decision processes, PCTL logic) and continuous-time systems (Markov chains, CSL logic). PRISM usage is standard: model definition with a specific formalism, definition of the properties with probabilistic temporal logic, then computation and analysis of the results. We focus here on discrete-time system functionalities, functionalities for continuous-time systems being connected to those of the ETMCC tool presented above.

9.5.5.1. *Language of system models*

System descriptions in PRISM are based on reactive modules, close to the Alur and Henzinger [ALU 99] model used in the CTL model checking tool SMV. Modules correspond to active entities of the system and a state is defined as values of (global) variables declared and handled by modules. Synchronization is carried out by guards on values of variable. Stochastic parameters (transition probabilities) are defined in modules and the user indicates in the model if this is a DTMC or a MDP. Extending the basic model, state and transition reward functions may be defined in PRISM; it is then possible to get mean performance indices (cumulated profits or losses), in finite time horizon or in steady-state.

9.5.5.2. *Properties language*

For DTMC and MDP, PRISM allows us to define properties to be checked in the PCTL logic presented above. PRISM also computes the probability that a given PCTL formula is satisfied by all the states of a DTMC, without prefixing a probability threshold.

9.5.5.3. *Computed results*

The software analyzes the model in accordance with requested properties and returns results in the user interface, in the form of a set of graphical representations giving a syntactic view of the properties of the system. Moreover, we get mean parameters defined by reward functions of the model.

9.5.5.4. *Software architecture*

From the internal point of view, implementation is programmed with C++ for numerical code and with Java for the user interface and the overall structure of the software. An important feature of PRISM is the management of symbolic data structures based on multi-valued binary decision diagrams (MTBDD, [CLA 93]) allowing analysis of large systems (more than 10^{10} states in suitable cases). The non-probabilistic model checker systematically manages BDD structures. For numerical solvers, PRISM uses three "engines" based on MTBDD, sparse matrix representations and a mixed model of both. Indeed, experiments and many studies carried out with the tool show that MTBDD representations tend to notably slow down numerical with respect to, now well optimized, (iterative) methods, based on sparse representations.

9.6. Bibliography

[AJM 87] AJMONE MARSAN M., CHIOLA G., "On Petri nets with deterministic and exponentially distributed firing times", ROZENBERG G., Ed., *Advances in Petri Nets 1987*, num. 266 LNCS, p. 132–145, Springer Verlag, 1987.

[AJM 95] AJMONE MARSAN M., BALBO G., CONTE G., DONATELLI S., FRANCESCHINIS G., *Modeling with Generalized Stochastic Petri Nets*, Wiley series in Parallel Computing, John Wiley & Sons, England, 1995.

[ALF 97] DE ALFARO L., "Model checking of probabilistic and non-deterministic systems", *STACS'97*, num. 1200 LNCS, Springer Verlag, p. 165–176, 1997.

[ALF 00] DE ALFARO L., "From fairness to chance", *Electronic Notes on Theoretical Computer Science*, vol. 22, 2000.

[ALF 04] DE ALFARO L., FAELLA M., HENZINGER T., MAJUMDAR R., STOELINGA M., "Model checking discounted temporal properties", *TACAS 2004: 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, num. 2988 LNCS, Springer Verlag, p. 77–92, 2004.

[ALU 99] ALUR R., HENZINGER T. A., "Reactive modules", *Formal Methods in System Design*, vol. 15, num. 1, p. 7–48, 1999.

[AND 03] ANDOVA S., HERMANNS H., KATOEN J.-P., "Discrete-time rewards model-checked", *Formal Modeling and Analysis of Timed Systems (FORMATS 2003)*, num. 2791 LNCS, Marseille, France, Springer Verlag, p. 88–103, 2003.

[AZI 96] AZIZ A., SANWAL K., V.SINGHAL, BRAYTON R., "Verifying continuous-time Markov chains", *8th Int. Conf. on Computer Aided Verification (CAV'96)*, num. 1102 LNCS, New Brunswick, NJ, USA, Springer Verlag, p. 269–276, 1996.

[AZI 00] AZIZ A., SANWAL K., V.SINGHAL, BRAYTON R., "Model checking continuous-time Markov chains", *ACM Transactions on Computational Logic*, vol. 1, num. 1, p. 162–170, 2000.

[BAI 98] BAIER C., KWIATKOWSKA M., "Model checking for a probabilistic branching-time logic with fairness", *Distributed Computing*, vol. 11(3), p. 125–155, 1998.

[BAI 03a] BAIER C., HAVERKORT B., HERMANNS H., KATOEN J.-P., "Model-checking algorithms for continuous time Markov chains", *IEEE Transactions on Software Engineering*, vol. 29, num. 7, p. 524–541, July 2003.

[BAI 03b] BAIER C., HERMANNS H., KATOEN J.-P., WOLF V., "Comparative branching-time semantics for Markov chains", *Concurrency Theory (CONCUR 2003)*, num. 2761 LNCS, Marseille, France, Springer Verlag, p. 492–507, 2003.

[BIA 95] BIANCO A., DE ALFARO L., "Model checking of probabilistic and non-deterministic systems", *FST TCS 95: Foundations of Software Technology and Theoretical Computer Science*, num. 1026 LNCS, Bangalore, India, Springer Verlag, p. 499–513, 1995.

[CAM 94] CAMPOS J., COLOM J., JUNGNITZ H., SILVA M., "Approximate throughput computation of stochastic marked graphs", *IEEE Transactions on Software Engineering*, vol. 20, num. 7, p. 526–535, July 1994.

[CHI 93a] CHIOLA G., ANGLANO C., CAMPOS J., COLOM J., SILVA M., "Operationnal analysis of timed Petri nets and application to computation of performance bounds", *Proc. of the 5th International Workshop on Petri Nets and Performance Models*, Toulouse, France, IEEE Computer Society Press, p. 128–137, October 19–22 1993.

[CHI 93b] CHIOLA G., DUTHEILLET C., FRANCESCHINIS G., HADDAD S., "Stochastic well-formed colored nets and symmetric modeling applications", *IEEE Transactions on Computers*, vol. 42, num. 11, p. 1343–1360, November 1993.

[CLA 93] CLARKE E., FUJITA M., MCGEER P., MCMILLAN K., YANG J., ZHAO X., "Multi-terminal binary decision diagrams: an efficient data structure for matrix representation", *Proc. Int. Workshop on Logics Synthesis (IWLS'93)*, p. 1–15, 1993, also available in Formal Methods in System Design, 10(2/3): 149–169, 1997.

[COU 95] COURCOUBETIS C., YANNAKAKIS M., "The complexity of probabilistic verification", *Journal of the ACM*, vol. 42(4), p. 857–907, July 1995.

[COU 03] COUVREUR J.-M., SAHEB N., SUTRE G., "An optimal automata approach to LTL model checking of probabilistic systems", in *Proc. 10th Int. Conf. Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'2003)*, num. 2850 LNAI, Almaty, Kazakhstan, Springer Verlag, p. 361–375, September 2003.

[DON 98] DONATELLI S., HADDAD S., MOREAUX P., "Structured characterization of the Markov chains of phase-type SPN", *Proc. of the 10th International Conference on Computer Performance Evaluation. Modeling Techniques and Tools (TOOLS'98)*, num. 1469 LNCS, Palma de Mallorca, Spain, Springer Verlag, p. 243–254, September 14–18 1998.

[EME 80] EMERSON E. A., CLARKE E. M., "Characterizing correctness properties of parallel programs using fixpoints", *7th International Colloquium on Automata, Languages and Programming, (ICALP)*, Noordweijkerhout, The Netherlands, p. 169–181, 1980.

[FEL 68] FELLER W., *An Introduction to Probability Theory and its Applications. Volume I*, John Wiley & Sons, 1968 (third edition).

[FEL 71] FELLER W., *An Introduction to Probability Theory and its Applications. Volume II*, John Wiley & Sons, 1971 (second edition).

[FLO 85] FLORIN G., NATKIN S., "Les réseaux de Petri stochastiques", *TSI*, vol. 4, num. 1, p. 143–160, 1985.

[GER 99] GERMAN R., TELEK M., "Formal relation of Markov renewal theory and supplementary variables in the analysis of stochastic Petri nets", BUCHHOLZ P., SILVA M., Eds., *Proc. of the 8th International Workshop on Petri Nets and Performance Models*, Zaragoza, Spain, IEEE Computer Society Press, p. 64–73, September 8–10 1999.

[GIR 03] GIRAULT C., VALK R., Eds., *Petri Nets for Systems Engineering*, Springer Verlag, 2003.

[HAD 05] HADDAD S., MOREAUX P., SERENO M., M.SILVA, "Product-form and stochastic Petri nets: a structural approach", *Performance Evaluation*, vol. 59/4, p. 313–336, 2005.

[HAV 95] HAVERKORT B., "Matrix-geometric solution of infinite stochastic Petri nets", *Proc. of the International Performance and Dependability Symposium*, IEEE Computer Society Press, p. 72–81, 1995.

[HEN 90] HENDERSON W., PEARCE C., TAYLOR P., VAN DIJK N., "Closed queueing networks with batch services", *Queueing Systems*, num. 6, p. 59–70, 1990.

[HIL 96] HILLSTON J., *A Compositional Approach to Performance Modeling*, Cambridge University Press, 1996.

[JEN 53] JENSEN A., "Markov chains as an aid in the study of Markov processes", *Skand. Aktuarietidskrift*, vol. 3, p. 87–91, 1953.

[KLE 75] KLEINROCK L., *Queueing Systems. Volume I: Theory*, Wiley-Interscience, New York, 1975.

[KLE 76] KLEINROCK L., *Queueing Systems. Volume II: Computer Applications*, Wiley-Interscience, New York, 1976.

[KWI 00] KWIATKOWSKA M., NORMAN G., SEGALA R., SPROSTON J., "Verifying quantitative properties of continuous probabilistic timed automata", *CONCUR 2000 – Concurrency Theory*, num. 1877 LNCS, Springer Verlag, p. 123–137, August 2000.

[KWI 01] KWIATKOWSKA M., NORMAN G., SPROSTON J., "Symbolic computation of maximal probabilistic reachability", *Proc. 13th International Conference on Concurrency Theory (CONCUR'01)*, num. 2154 LNCS, Springer Verlag, p. 169–183, August 2001.

[KWI 02a] KWIATKOWSKA M., NORMAN G., SEGALA R., SPROSTON J., "Automatic verification of real-time systems with discrete probability distributions", *Theoretical Computer Science*, vol. 282, p. 101–150, June 2002.

[KWI 02b] KWIATKOWSKA M., NORMAN G., SPROSTON J., "Probabilistic model checking of the IEEE 802.11 wireless local area network protocol", *Proc. PAPM/PROBMIV'02*, num. 2399 LNCS, Springer Verlag, p. 169–187, July 2002.

[LAP 95] LAPRIE J., Ed., *Guide de la sûreté de fonctionnement*, Cépaduès – Éditions, Toulouse, France, 1995.

[LIN 98] LINDEMANN C., *Performance Modeling with Deterministic and Stochastic Petri Nets*, John Wiley & Sons, New York, 1998.

[LIN 99] LINDEMANN C., REUYS A., THÜMMLER A., "DSPNexpress 2000 performance and dependability modeling environment", *Proc. of the 29th Int. Symp. on Fault Tolerant Computing*, Madison, Wisconsin, June 1999.

[MEY 80] MEYER J., "On evaluating the performability of degradable computing systems", *IEEE Transactions on Computers*, vol. 29, num. 8, p. 720–731, August 1980.

[MOL 81] MOLLOY M. K., "On the integration of delay and throughput in distributed processing models", PhD thesis, University of California, Los Angeles, CA, USA, September 1981.

[PUT 94] PUTERMAN M., *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, John Wiley & Sons Inc., New York, 1994.

[ROO 97] ROOS C., TERLAKY T., VIAL J.-P., *Theory and Algorithms for Linear Optimization. An Interior Point Approach*, Wiley-Interscience, John Wiley & Sons Ltd, West Sussex, England, 1997.

[STE 94] STEWART W. J., *Introduction to the Numerical Solution of Markov Chains*, Princeton University Press, Princeton, NJ, USA, 1994.

[TRI 82] TRIVEDI K. S., *Probability & Statistics with Reliability, Queueing, and Computer Science Applications*, Prentice Hall, Englewood Cliffs, NJ, USA, 1982.

[TRI 92] TRIVEDI K. S., MUPPALA J. K., WOOLE S. P., HAVERKORT B. R., "Composite performance and dependability analysis", *Performance Evaluation*, vol. 14, num. 3–4, p. 197–215, February 1992.

[VAR 85] VARDI M., "Automatic verification of probabilistic concurrent finite-state programs", *FOCS 1985*, p. 327–338, 1985.

[YOU 06] YOUNES H., KWIATKOWSKA M., NORMAN G., PARKER D., "Numerical vs. statistical probabilistic model checking", *Int. Journal on Software Tools for Technology Transfer (STTT)*, vol. 8, num. 3, p. 216–228, 2006.

Chapter 10

Modeling and Verification of Real-Time Systems using the IF Toolset

This chapter presents an overview on the IF toolset which is an environment for the modeling and validation of heterogenous real-time systems. The toolset is built upon a rich formalism, the IF notation, allowing structured automata-based system representations. Moreover, the IF notation is expressive enough to support real-time primitives and extensions of high-level modeling languages such as SDL and UML by means of structure preserving mappings.

The core part of the IF toolset consists of a syntactic transformation component and an open exploration platform. The syntactic transformation component provides language level access to IF descriptions and has been used to implement static analysis and optimization techniques. The exploration platform gives access to the graph of possible executions. It has been connected to different state-of-the-art model checking and test case generation tools.

A methodology for the use of the toolset is presented using a case study concerning the Ariane 5 flight program for which both an SDL and a UML model have been validated.

The technical work presented in this chapter has been carried out between 1997 and 2005. This presentation is an updated translation of a chapter in [BOZ 06].

Chapter written by Marius BOZGA, Susanne GRAF, Laurent MOUNIER and Iulian OBER.

10.1. Introduction

Modeling plays a central role in software engineering for real-time distributed systems. The use of models in the design process of a software is used to express both the expected behavior of the system under development and that of its environment – consisting of the physical environment and the execution infrastructure or platform – and offers numerous advantages:

– Abstract descriptions using high-level concepts can be analyzed efficiently using appropriate formal methods, even at a time point at which no actual execution environment is available yet.

– The use of executable models, whose behavior is defined by a formal operational semantics, make it possible to experiment with large models just like with an implementation, and in addition, it allows the inclusion of legacy code for analysis purposes.

– Moreover, accurate executable models of the environment may offer more accurate observation and control capabilities than those provided by a real experimentation platform. Indeed, a model makes it possible to control physical parameters that are not always controllable in real-life (for example, time progress), and avoids probe effects and disturbances due to experimentation.

Building faithful models of complex systems useful for efficient analysis is a non-trivial problem and a prerequisite for the application of formal analysis techniques. Traditionally, modeling techniques are applied at early phases of system development and at a high abstraction level. Then, several models may be used, one for each type of analysis required. Nevertheless, methods are then needed to guarantee the consistency between these models during the entire design process and also with the final implementation. In restricted contexts, and when a global view of the system under development is available, some solutions have been proposed: for instance, the B method [ABR 88] provides a stepwise and tool guided model refinement technique, and the synchronous approach [BEN 93, BEN 03] allows the automatic production of executable code or integrated circuit layouts from abstract design models.

More recently, new model-based approaches have been proposed to better take into account other kinds of applications, which are able to cope with existing software components and complex distributed execution platforms with various scheduling policies. Among them are the OMG MDA initiative [OMG 03a] which proposes to generate code from a description of the execution platform, a functional description of the software behavior, and a non-functional description of its (assumed and guaranteed) properties. Architecture description languages such as AADL [SAE 04] focus on the description of existing execution architectures and allow analytic analysis and the generation of an implementation through the integration of existing pieces of code. Similarly, but on a more abstract level and more formally, a new paradigm for component composition or coordination [SIF 01, SIF 03, SCH 04, GÖS 05] offers a notion of *rich* component interfaces, including non-functional aspects.

These approaches rely heavily on the existence of modeling methods and tools to provide support and guidance for system design and validation, particularly regarding coordination and interaction between heterogenous components.

The IF toolset is an environment developed for the modeling and validation of heterogenous real-time systems. It is characterized by the following features:

– Support for modeling with user-level formalisms, such as SDL or UML, in commercial editing or CASE tools to describe both abstract and concrete models (used to generate an actual implementation). This is essential for easing usability for practitioners and allows them to stay with state-of-the-art modeling technology. Furthermore, the use of high level formalisms allows validating realistic models, which can be simplified if necessary by using automated tools. This avoids starting with (over) simplified models constructed in an ad hoc manner, as is often the case for other tools based on low level description languages, e.g., finite automata.

– Transformations from high level modeling formalisms into an intermediate representation, the IF notation, that serves as a semantic model by still preserving the essential structure for allowing efficient analysis and verification. This representation combines the composition of extended timed automata and dynamic priorities to encompass heterogenous interaction. Priorities play an important role for the description of scheduling policies as well as the restriction of asynchronous behavior to model run-to-completion execution modes.

We consider a class of timed automata which are well-timed by construction. This representation is rich and expressive enough to describe the main concepts and constructs of source languages by nevertheless preserving structural concepts required to simplify models for validation. In particular, the concurrency level and the data structure of the initial model are explicitly preserved in the IF representation. Thus, the translation methods we developed for SDL and UML preserve the overall structure of the source model, and the size of the generated IF description increases linearly with the size of the source model.

– Support for a wide range of abstraction and validation techniques, including model checking, static analysis, test case generation and simulation. A methodology has been studied at Verimag for complex real-time applications. Model simplifications based on static analysis techniques are applied directly on IF system descriptions, whereas dynamic verification techniques are applied on a global, possibly abstracted, execution graph, generated by an exploration platform from the IF description. Other simplifications such as partial order reductions are applied during this exploration process.

– Expression of requirements to be validated on models by using observers. These observers can be considered as a special class of models equipped with primitives for monitoring and checking for divergence from some nominal behavior. Our choice to use observers rather than declarative formalisms such as temporal logic or Live Sequence charts [DAM 99] is motivated by our concern to be close to industrial practice and to avoid inconsistency in requirements as much as possible.

This chapter is organized as follows. Section 10.2 presents the overall architecture of the IF toolset. Sections 10.3 and 10.4 are the heart of the chapter. The first section gives an overview on IF, including its main concepts and constructs and their semantics. The second section provides a detailed description of the functionality of the toolset, and in particular the simulation, analysis and verification tools. The UML to IF translation principle is also explained in section 10.5, showing how the main UML concepts are mapped into IF. Section 10.6 presents an example illustrating the application of the toolset to the modeling and validation of the Ariane 5 flight program. For this non-trivial case study, we provide a validation methodology and results. Section 10.7 presents concluding remarks about the toolset and the underlying modeling and validation methodology.

10.2. Architecture

Figure 10.1 describes the overall architecture of the toolset, the most important components as well as their inter-connections. We distinguish four different description levels: the user description level (UML, SDL), the intermediate description level (IF), the exploration platform level and the explicit labeled transition system level.

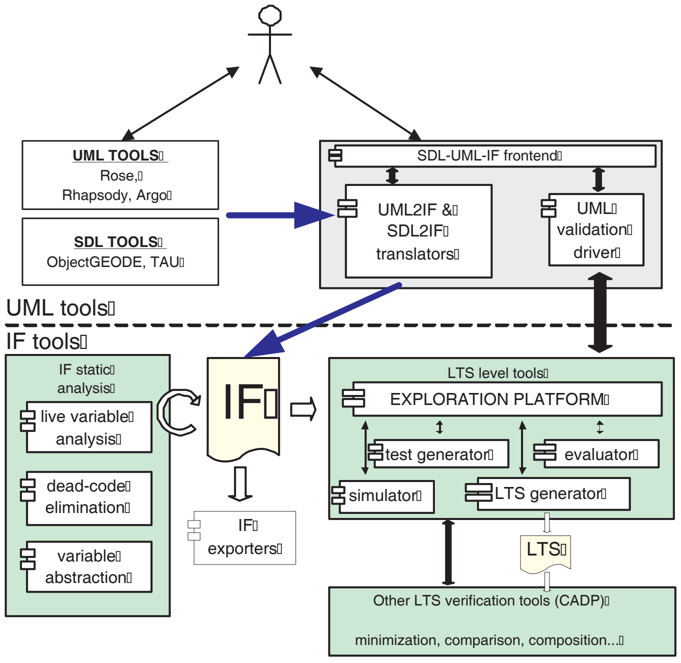


Figure 10.1. *IF toolset architecture*

User description level

This level corresponds to the description provided by the user in some existing specification language. To be processed, descriptions are automatically translated into their IF descriptions. Currently, the main input specification formalisms are UML and SDL.

Regarding UML, any UML tool can be used as long as it can export the model in XMI 1.0 [OMG 01], the standard XML format for exchanging UML models at the time we had built our tools, which is still used by a number of UML tools today. The IF toolset includes a translator which for an UML model in the XMI format produces an IF description. The translator has been tested for specifications produced by RATIONAL ROSE [IBM], RHAPSODY [ILO] or ARGO UML [RAM]. Regarding SDL, the translator takes as input textual SDL'96 specifications and produces IF descriptions.

Intermediate description level (IF)

IF is an intermediate representation based on timed automata extended with discrete data variables, communication primitives, and dynamic process creation and destruction. This representation is expressive enough to describe the basic concepts of modeling and programming languages for distributed real-time systems.

The abstract syntax tree of an IF description can be accessed through an application programming interface (API). Since all the data (variables, clocks) and the communication structure are still explicit, high-level transformations based on static analysis [MUC 97] or program slicing [WEI 84, TIP 94] can be applied. All these techniques can be used to transform the initial IF description into a simpler one while preserving safety properties [BOZ 03b, FER 03]. Moreover, this API is well-suited to implement exporters from IF to other specification formalisms.

Exploration platform and labeled transitions system level

At exploration platform level, a system is seen as a collection of running components which, in any global state, propose a set of possible steps. An exploration API makes it possible to represent and store explored states, as well as to compute on demand the successors of any given state. This API can be used by modules performing any kind of on-the-fly analysis.

Using this exploration API, several validation tools have been developed. They cover a broad range of functionalities: interactive/random/guide simulation, on-the-fly model checking using observers, on-the-fly temporal logic model checking, exhaustive state space generation, scheduling analysis, test case generation [BOZ 02a, BOZ 04]. Moreover, through this API are connected the CADP toolbox [FER 96b] for the validation of finite models and TGV [FER 96a, JÉR 99], a tool for test case generation using on-the-fly techniques.

At a semantic level, the state space containing all possible executions of the system can be explicitly represented as a labeled transition system (LTS) where the labels correspond to observable actions in the IF transitions. Such transition systems can be compared or reduced various modulo simulation and bisimulation equivalences, and they may be further composed or visualized, e.g., using the Aldebaran tool [BOZ 97] of the CADP toolbox.

10.3. The IF notation

IF is a notation for systems of components (called processes), running in parallel and interacting either through global variables or via asynchronous signals. Processes describe sequential behaviors including data transformations, communications and process creation. Furthermore, the behavior of a process may be subject to timing constraints. The number of processes may change over time: they may be created and deleted dynamically.

The semantics of a system is the labeled transition system obtained by interleaving the behavior of its processes. However, the interleaving can be restricted using dynamic priorities, in order to enforce a scheduling policy or any other user specific execution policy.

In this section, we present the concepts provided by the IF notation to describe the behavior of systems, by distinguishing between the functional and non-functional ones.

10.3.1. *Functional features*

The behavior of a process is described as a timed automaton, extended with data. A process has a unique process identifier (*pid*) and local memory consisting of variables (including clocks), control states and a queue of pending messages (received and not yet consumed).

A process moves from one control state to another by executing a transition. Transitions may be triggered by signals in the input queue or may be spontaneous. In any case, transitions may be guarded by predicates on variables, where a guard is a conjunction of a data guard and a time guard. A transition is enabled in a state if its trigger is present and its guard evaluates to true. Signals in the input queue are *a priori* consumed in a FIFO fashion, but it is possible to specify in transitions which signals should be "saved" in the queue for later use.

Transition bodies are sequential programs consisting of elementary actions (variable or clock assignments, message sending, process creation/destruction, resource requirement/release, etc.) and structured using elementary control-flow statements (such as if-then-else, while-do, etc.). In addition, transition bodies can contain calls to functions or procedures written in an external programming language (C/C++).

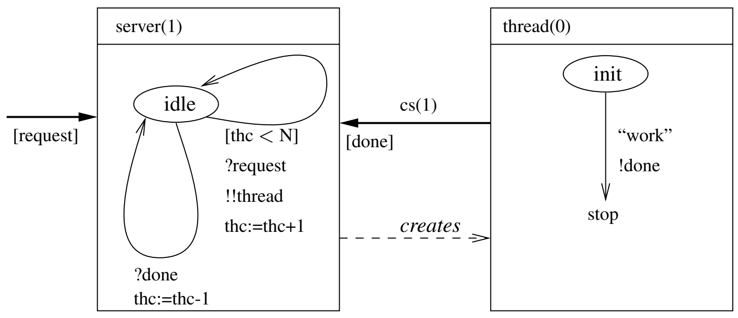


Figure 10.2. *Illustration of the multi-threaded server example*

As for state charts [HAR 87, HAR 98], control states can be hierarchically structured to factorize common behavior. Control states can be stable or unstable. A sequence of transitions between two stable states defines a step. The execution of a step is atomic, meaning that it corresponds to a single transition in the LTS representing the semantics. Notice that several transitions may be enabled at the same time, in which case the choice is made non-deterministically.

Signals are typed and can have data parameters. Signals can be addressed directly to a process (using its *pid*) and/or to a "signalroute" which delivers it to zero or more processes. The destination process stores received signals in its message queue. signalroutes represent specialized communication media transporting signals between processes. The behavior of a signalroute is defined by its delivery policy (FIFO or multi-set), its connection policy (peer to peer, unicast or multicast), its delaying policy and finally its reliability (reliable or lossy). More complex communication media must be specified explicitly as IF processes.

Concerning data, the IF notation provides the predefined basic types bool, integer, real, pid and clock, where clock is used for variables measuring time progress. Structured data types are built using the type constructors enumeration, range, array, record and abstract. Abstract data types can be used for manipulating external types and code.

EXAMPLE.– The IF description below describes a system consisting of a `server` process creating up to N `thread` processes for handling `request` signals. A graphical representation of the system is given in Figure 10.2.

```
system Server;
// parametrized signals
signal request();
signal done(pid);
// signalroutes and their signals
signalroute entry(1)
   from env to server
   with request;
signalroute cs(1) #delay[1,2]
   from thread to server
```

```
  with done;
// processus thread // definition
// initial number of instances : 0
process thread(0);
  // its formal parameters
  fpar parent pid, route pid;
  // its initial control state
  state init #start ;
    // a spontaneous transition containing
    // an informal action, a signal output
    // then, the destruction of the instance
    informal "work";
    output done() via route to parent;
    stop;
  endstate;
endprocess;
// process server // definition
// initial number of instances : 1
process server(1);
  // local variables;
  var thc integer;
  // its control state
  state idle #start ;
    // first transition, triggered by
    // reception of the done // signal
    input done();
      task thc := thc - 1;
      nextstate idle;
    // second transition, guarded
    // and triggered, create a new instance
    // of the thread // process
    provided thc < N;
    input request();
      fork thread(self, {cs}0);
      task thc := thc + 1;
      nextstate idle;
  endstate;
endprocess;
endsystem;
```

The semantics associate a global LTS with a system. At any point of time, its state is defined as the tuple of the states of its living components: the states of a process are the possible evaluations of its attributes (control state, variables and signal queue content). The state of a signalroute is a list of signals "in transit". The transitions of the global LTS representing a system are steps of processes and signal deliveries from signalroutes to signal queues where in any global state there is an outgoing transition for all enabled transitions of all components (interleaving semantics). The formal definition of the semantics can be found in [BOZ 02b].

10.3.2. *Non-functional features*

The time model of IF is that of timed automata with urgency [BOR 98, BOR 00]. As for regular timed automata [ALU 94], the execution of a transition is instantaneous, and time is progressing only in states. Moreover, transitions have an urgency attribute to indicate their priority with respect to time progress. We distinguish between "eager", "lazy" and "delayable" transitions. Eager transitions are urgent, that is they have to be executed at the point of time at which they become enabled – except if they are disabled by executing another transition. Lazy transitions are never urgent and they may be disabled by time progress. A delayable transition is not urgent, except at a time point at which time progress would disable it.

As in timed automata, the time distance between events is measured by variables of type “clock”. Clocks can be created, set to some value or reset (deleted) in any transition. They can be used in time guards to restrict the time points at which transitions can be taken. Local clocks allow the specification of timing constraints, such as durations of tasks (modeled by time passing in a state associated with this task, see example below), and deadlines for events in the same process. Global time constraints, such as end-to-end delays, can be expressed by means of global clocks or by observers (explained in the next section).

EXAMPLE.– A timed version of the `thread` process of the previous example is given. An extra state `work` introduced to identify the instants at which work starts and the instant at which work ends, and to constrain the duration between these events. The intention is to model an execution time of “*work*” of 2 to 4 time units. The *thread* process goes immediately to the `work` state – the start transition is **eager** – and the clock *wait* is set to 0 in order to start measuring time progress. The transition exiting the `work` state is **delayable** with a time guard expressing the constraint that the time since the clock `wait` has been set should be at least 2 but not more than 4.

```
process thread(0);
  fpar parent pid, route pid;
  var wait clock;
  state init #start;
    urgency eager;
    informal "work";
    set wait:= 0;
      nextstate work;
  endstate;
state work;
  urgency delayable;
  when wait >= 2 and wait <= 4;
    output done()
      via route to parent;
    stop;
  endstate;
endprocess;
```

System models may be highly non-deterministic, due to the non-determinism of the environment which is considered as open and due to the concurrency between processes. For the validation of functional properties, both types of explicit non-determinism are important in order to verify the correctness for a variation of environment behaviors and independently of any particular execution order. Nevertheless, going towards a specific implementation means resolving a part of this non-determinism and choosing an execution order satisfying time-related and other non-functional constraints.

In IF, such additional restrictions can be enforced by dynamic priorities defined by rules specifying that whenever for two process instances some condition (state predicate) holds, then one has less priority than the other. An example is

$$p1 \prec p2 \text{ if } p1.group = p2.group \text{ and } p2.counter < p1.counter$$

which, for any process instances which are part of some “*group*”, gives priority to those with the smallest values of the variable *counter* (e.g., the less frequently served).

Finally, in order to express mutual exclusion, it is possible to declare shared *resources*. These resources can be used through particular actions of the form "**require** `some-resource`" and "**release** `some-resource`".

10.3.3. *Expressing properties with observers*

An observer expresses in an operational way a safety property of a system by characterizing its acceptable executions. Observers also provide a simple and flexible mechanism for controlling model generation. They can be used to select parts of the model to explore and to cut off execution paths that are irrelevant with respect to given criteria. In particular, they can be used to restrict the environment of the system.

Observers are described in the same way as IF processes, i.e., as extended timed automata. They differ from IF processes in that they can react synchronously to events and conditions occurring in the observed system. Observers are classified into:

– *pure* observers, which express requirements to be checked on the system;

– *cut* observers, which in addition to monitoring, guide simulation by selecting execution paths. For example, they may be used to restrict the behavior of the environment;

– *intrusive* observers, which may also alter the system's behavior by sending signals and changing variables.

In order to monitor the system state, observers can use primitives for retrieving values of variables, the current state of the processes, the contents of queues, etc. To monitor actions performed by a system, observers use constructs for retrieving events together with data associated with them. Events are generated whenever the system executes one of the following actions: signal output, signal delivery, signal input, process creation and destruction, and informal statements. Observers can also monitor time progress, by using their own clocks or by monitoring the clocks of the system.

In order to express properties, observer states can be marked as `ordinary`, `error` or `success`. Error and success are both terminating states. Reaching a success state (an error state) means satisfaction (non-satisfaction). Cut observers use a cut action which stops exploration beyond the reached state.

EXAMPLE.– The following example illustrates the use of observers to express a simple safety property of a protocol with one transmitter and one receiver, such as the alternating bit protocol. The property is: *whenever a **put(m)** message is received by the **transmitter** process, the **transmitter** does not return to state **idle** before a **get(m)** with the same **m** is issued by the **receiver** process.*

```
pure observer safety1;
   var m data;
   var n data;
   var t pid;
   state idle #start ;
      match input put(m) by t;
         nextstate wait;
   endstate;
   state wait;
      provided ({transmitter}t)
         instate idle;
         nextstate err;
      match output put(n)
         nextstate err;
      match output get(n);
         nextstate decision;
   endstate;
   state decision #unstable ;
      provided n = m;
         nextstate idle;
      provided n <> m;
         nextstate wait;
   endstate;
   state err #error ;
   endstate;
endobserver;
```

10.4. The IF tools

10.4.1. *Core components*

The core components of the IF toolset are shown in Figure 10.3. The syntactic transformation component deals with the construction and the exploration of an abstract syntax tree (AST) from an IF description. This tree is a collection of C++ objects representing all the syntactic elements that may be present in an IF description: a system has processes, signalroutes and types; a process has states and variables; states include their outgoing transitions and so on. The core component has an interface giving access to this abstract syntax tree. Primitives are available to traverse the tree, to consult and modify its elements, and to print the tree (in the original syntax). The syntactic transformation component has been used to build several applications. The most important ones that we have built are code generators (either simulation code or application code), static analysis transformations (operating at syntactic level) and pretty printers.

Notice that this API has also been intended as an interface with other verification tools. There have indeed been several external tools connected to IF via this API. For example, there are translations to Promela, the language of SPIN [HOL 91], an explicit state model-checker such as IF but with a different performance profile. These translations have been defined in [BOS 00] and later in [PRI 02].

There is a translation from IF to LASH, the Liège Automata-based Symbolic Handler [BOI 02], a tool for exact symbolic reachability analysis for specification with integer variables. There is also a translation to TREX [ANN 01], a similar tool that handles specifications with queues. A translation to the PVS automatic abstraction tool INVEST [BEN 98] has been defined.

There exist translations to test case generators and test execution tools. There are transformations from IF to the input languages of the testing tools AGATHA [LUG 01]

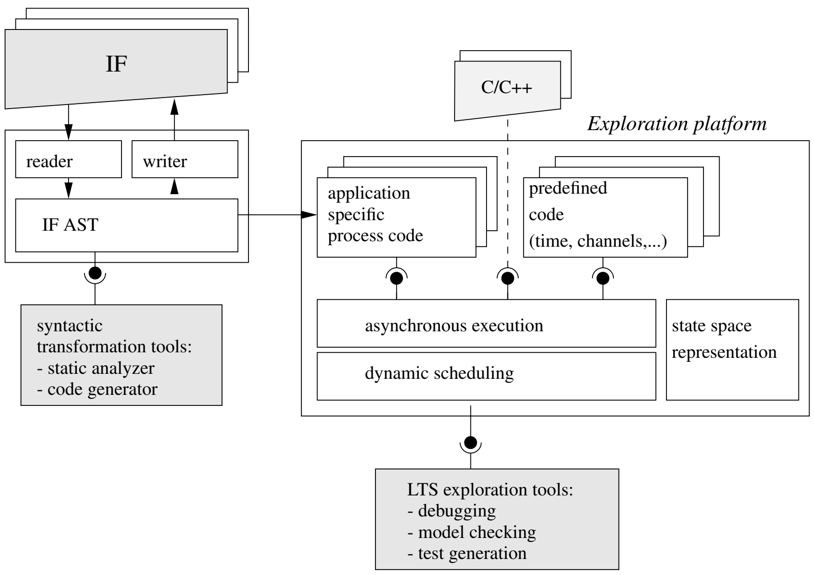


Figure 10.3. *Functional view of the IF core components*

and SPIDER [HAR 04]. The test case generation tool TGV [FER 96a, JÉR 99] has been modified to be able to work on IF specifications.

The exploration platform component has an API providing access to the LTS corresponding to an IF description. The interface offers primitives for representing and accessing states and labels as well as basic primitives for traversing an LTS: an *init* function which gives the initial state, and a *successor* function which computes the set of enabled transitions and successor states from a given state. These are the key primitives for implementing any on-the-fly forward enumerative exploration or validation algorithm.

The main features of the platform are simulation of the process execution, resolution of non-determinism, management of simulation time and representation of the state space. In other words, the exploration platform can be seen as an operating system where process instances are plugged-in and jointly executed. Process instances are either application specific (coming from IF descriptions) or generic (such as time or channel handling processes). More precisely, the platform composes active instances in order to obtain global states and global behavior of the system. Moreover, the platform stores the set of reachable states, as soon as they are reached.

As shown in Figure 10.3, the exploration platform consists of two layers sharing a common state representation:

– The asynchronous execution layer implements the general interleaving execution of processes. The platform successively asks each process to execute its enabled steps. During a process execution, the platform manages all inter-process operations: message delivery, time constraints checking, dynamic process creation and destruction, and tracking of events. After the completion of a step by a process, the platform takes a snapshot of the performed step, stores it and delivers it to the second layer.

– The dynamic scheduling layer collects all the enabled steps. It uses a set of dynamic priority rules to filter them. The remaining ones, which are maximal with respect to the priorities, are delivered to the user application via the exploration API.

Simulation time is handled by a specialized process which manages clock allocation/deallocation, computes time progress conditions and fires timed transitions. There are two implementations available, one for discrete-time and one for dense time. For discrete-time, clock values are explicitly represented by integers. Time progress is computed with respect to the next enabled deadline. For dense time, clock valuations are represented using "difference bound matrices" (DBMs) of variable size, as they are used in tools dedicated to timed automata, such as KRONOS [YOV 97] and UPPAAL [LAR 98].

Global states are implicitly stored by the platform. The internal state representation is shown in Figure 10.4. It preserves the structural information and seeks maximal sharing. The layered representation involves a unique message table. Queues are lists of messages, represented using suffix sharing. On top of them, there is a table of process states, all of them sharing queues in the table of queues. Processes are then grouped into fixed-size state chunks and, finally, global states are variable-size lists of chunks. Tables can be represented either by hash tables with collision or by binary trees. This scheme makes it possible to explicitly represent several million large structured states.

This architecture provides features for validating heterogenous systems. Exploration is not limited *a priori* to IF descriptions: any components with an adequate interface can be executed – in parallel to IF components – by the exploration platform. It is indeed possible to use C/C++ code (either directly, or instrumented accordingly) of already implemented components. Moreover, another advantage of the architecture is that it can be extended by adding new interaction primitives and exploration strategies. Presently, the exploration platform supports asynchronous (interleaved) execution and asynchronous point-to-point communication between processes. Different execution modes, for example "synchronous" or "run-to-completion", or additional interaction mechanisms, such as broadcast or rendezvous, are obtained by using dynamic priority rules [ALT 00].

Concerning the exploration strategies, reduction heuristics such as partial-order reduction or some form of symmetry reduction are already incorporated in the exploration platform. More specific heuristics may be added depending on a particular application domain.

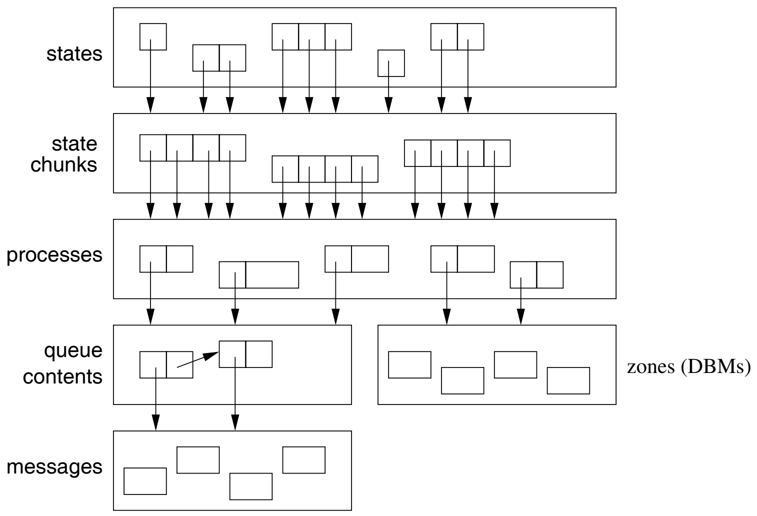


Figure 10.4. *Internal state representation*

The exploration platform and its interface has been used as back-end of debugging tools (interactive or random simulation), model checking (including exhaustive model generation, on-the-fly μ-calculus evaluation, model checking with observers), test case generation and optimization (shortest path computation). These back-end tools are described in section 10.4.3.

10.4.2. *Static analysis*

Practical experience with IF has shown that simplification by means of static analysis is crucial for dealing successfully with complex specifications. Even simple analysis such as live variables analysis or dead-code elimination can significantly reduce the size of the state space of a model. The available static analysis techniques are as follows:

– Live variable analysis transforms an IF description into an equivalent smaller one by removing globally dead variables and signal parameters and by *resetting* locally dead variables [MUC 97]. Initially, all the local variables of the processes and signal parameters are considered to be dead, unless otherwise specified by the user. Shared variables are considered to be always live. The analysis alternates local (standard) live variables computation on each process and inter-process liveness attribute propagation through input/output signal parameters until a (global) fixpoint is reached.

– Dead-code elimination transforms an IF description by removing unreachable control states and transitions under some user-given assumptions about the environment. It solves a simple static reachability problem by computing, for each process separately, the set of control states and transitions which can be statically proven to be unreachable from the initial control state. The analysis computes an upper approximation of the set of processes that can be effectively created.

– Variable abstraction makes it possible to obtain an over-approximation of the state space by eliminating variables declared as "irrelevant" by the user and all their dependencies. The computation proceeds just as for live variable analysis: processes are analyzed separately, and then the results are propagated between processes using the input/output dependencies. Contrary to the previous techniques which are exact, simplification by variable abstraction may introduce additional behaviors. Nevertheless, it always reduces the size of the state representation. Variable abstraction makes it possible to automatically extract system descriptions for symbolic verification tools accepting only specific types of data, such as the TREX tool [ANN 01] which only accepts counters, clocks and queues. Moreover, this technique makes it possible to obtain finite-state abstractions for model checking.

10.4.3. *Validation*

Simulation and validation with observers: the exploration platform (see section 10.4.1) includes several options allowing for exhaustive simulation, random or guided simulation, with or without breakpoints. In the presence of observers expressing properties (see section 10.3.3) the platform allows for the exhaustive exploration of the state space and stops whenever an error state is reached (the current explored path sequence is then given as a witness trace for the error). Moreover, the exploration can be guided using cut or intrusive observers.

EVALUATOR implements an on-the-fly model checking algorithm for the alternation free μ-calculus [KOZ 83]. This is a branching-time logic, based upon propositional calculus with fixpoint operators.

ALDEBARAN [BOZ 97] is a tool for the comparison of LTSs modulo behavioral preorder or equivalence relations. Usually, one LTS represents the system behavior, and the other its requirements. Moreover, ALDEBARAN can also be used to reduce a given LTS modulo a behavioral equivalence, possibly by taking into account an observation criterion. The preorders and equivalences available in ALDEBARAN include the usual simulation and bisimulation relations such as strong bisimulation [PAR 81], observational bisimulation [MIL 80], branching bisimulation [GLA 89], safety bisimulation [BOU 91], etc. The choice of the relation depends on the class of properties to be preserved.

TGV [FER 96a, JÉR 99] is a tool for test generation developed by IRISA and VERIMAG. It is used to automatically generate test cases for conformance testing of distributed reactive systems. It generates test cases from a formal specification of the system and a test purpose.

Let us note that on-the-fly verification is not systematically more efficient than global methods requiring the complete construction of the state space. The main reason is that on-the-fly verification explores a product space of the system and an observer or a formula. Moreover, the explicit construction of the sole system state space allows for the use of minimization techniques such as those implemented within Aldebaran, which usually cannot be applied on-the-fly.

10.4.4. *Translating UML to IF*

The toolset supports generation of IF descriptions from both SDL [BOZ 99] and UML [OBE 04, GRA 06b]. We describe the principles of the translation from UML to IF which has been developed in the OMEGA project [OME 05, GRA 04].

10.4.4.1. *UML modeling*

We have considered a subset of UML 1.4 including its object-oriented features and it is expressive enough for the specification of real-time systems. The elements of models are classes with structural features and relationships (associations, inheritance), and behavior descriptions through state machines and operations.

The translation tool adopts particular semantics for concurrency based on the UML distinction between active and passive objects. Informally, a set of passive objects together with an active object form an *activity group*. Activity groups are executed in a run-to-completion fashion, which means that there is no concurrency between the objects of the same activity group. Requests (asynchronous signals or method calls) coming from outside an activity group are queued and treated one by one. More details on these semantics can be found in [DAM 02, ZWA 06].

Additionally, we used a specialization of the standard UML 1.4 profile for scheduling, performance and time [OMG 03b]. Our profile, formally described in [GRA 06a], provides two kinds of mechanisms for timing: imperative mechanisms including timers, clocks and timed transition guards, and declarative mechanisms including linear constraints on time distances between events.

To provide connectivity with existing CASE tools such as RATIONAL ROSE [IBM], RHAPSODY [ILO] or ARGO UML [RAM], the toolset reads models using the standard XML representation for UML (XMI [OMG 01]).

10.4.4.2. *The principles of the mapping from UML to IF*

UML run-time entities (objects, call stacks, pending messages, etc.) are identifiable as a part of the system state in IF. This allows tracing back to UML specifications from simulation and verification.

Every UML class X is mapped to a process P_X with a local variable for each attribute or association of X. As inheritance is flattened, all inherited attributes and associations are replicated in the processes corresponding to each subclass. The class state machine is translated into the process behavior.

Each activity group is managed at run-time by a special IF process, of type *group manager*, which is responsible of sequentializing requests coming from objects outside the activity group, and of forwarding them to the objects inside when the group is stable. Run-to-completion is implemented by using the dynamic priority rule

$$y \prec x \text{ if } x \cdot leader = y$$

which means that *all objects of a group have higher priorities than their group manager.* For every object x, $x \cdot leader$ points to the manager process of the object's activity group. Thus, as long as at least one object inside an activity group can execute, its group manager will not initiate a new run-to-completion step. Notice that adopting a different execution mode can be easily done by just eliminating or adding new priority rules.

Concerning operations, the adopted semantics distinguishes between *primitive operations* – described by a method with an associated action – and *triggered operations* – described directly in the state machine of their owner class. Triggered operations are mapped to actions embedded directly in the state machine of the class. Each primitive operation is mapped to a handler process whose run-time instances represent the activations and the stack frames corresponding to calls. An operation call (either primitive or triggered) is expressed in IF by using three signals: a *call* signal carrying the call parameters, a *return* signal carrying the return value, and a *completion* signal indicating completion of computation of the operation, which may be different from *return*. Therefore, the action of invoking an operation is represented in IF by sending a *call* signal. If the caller is in the same activity group, then the *call* is directed to the target object and is handled immediately. Alternatively, if the caller is in a different group, the *call* is directed to the object's *group manager* and is handled in a subsequent run-to-completion step.

The handling of incoming primitive calls by an object is modeled as follows: in every state of the callee object (process), upon reception of a call signal, the callee creates a new instance of the operation's handler. The callee then waits until completion, before re-entering the same stable state in which it received the call.

Representing an operation activation as an object creation has several advantages. First, it provides a simple solution for handling *polymorphic* (dynamically bound) calls in an inheritance hierarchy. The receiver object knows its own identity, and can answer any *call* signal by creating the appropriate version of the operation handler from the hierarchy. Moreover, it allows for extensions to other types of calls than the ones currently supported by the semantics (e.g. non-blocking calls). It also preserves modularity and readability of the generated model. Finally, it makes it possible to distinguish the relevant instants in the context of timing analysis.

10.5. An overview on uses of IF in case studies

The IF toolset has been used in a number of case studies by using different input languages and the front-ends connecting them to IF. We give here an overview of the most significant ones:

– We have used the SDL front-end mainly for modeling and validating communication protocols, such as the SSCOP protocol [BOZ 00]. In this case study, the challenge was to handle a large amount of data, where most of it could be eliminated using static analysis.

The SDL front-end was also used for verifying a part of the management layer of the MASCARA protocol which is an extension of ATM to wireless connections. Here, a combination of partial order reduction, static analysis and compositional verification made it possible to verify the protocol [GRA 01].

We also have modeled in SDL the European Ariane 5 Launcher mission management and validated it with IF [BOZ 01]. This case study is presented in more detail in the next section.

– We have developed two UML front-ends. The UML front-end developed in the AGEDIS project [AGE 03, HAR 04] is not described here; in AGEDIS, we used IF as a intermediate formate for generating test cases with TGV for UML specifications. We applied the AGEDIS tool chain for generating test cases for an mq-broker protocol.

The UML interface developed in the OMEGA project has been used for several case studies, an overview can be found in [OME 05, GRA 05]:

– A first case study is a flight control mechanism that implements "sensor voting" and "sensor monitoring" operations in a typical flight control system.

– A second one involves the "medium altitude reconnaissance system" (MARS), which counteracts the image quality degradation caused by the forward motion of an aircraft [OBE 06b].

– A third application is a telecom service built on top of an embedded platform of service components which has been modeled and analyzed using both IF and Live sequence charts [COM 07].

– Finally, we have taken up on our initial Ariane 5 case study, and made a more complete UML model of the flight software by focusing on the relevant real-time behaviors [OBE 06a] which is presented in more detail in the next section.

– We used the IF toolset to analyze specific real-time systems modeling directly in the IF language. In [BOZ 03a], we modeled the production capacity of a chemical plant, and then used a new module based on the IF exploration engine integrating specific optimizations to extract optimal schedules for a specific production task. In [SAL 03], we used IF to implement a compositional method for timing analysis of combinational circuits.

– The IF toolset has been used to verify and test the K9 Rover Executive, an experimental NASA software for autonomous wheeled vehicles targeted for the exploration of Martian surface [AKH 04, BEN 04].

– IF has also been used by other groups who have developed some specific tools either generating IF specifications or using the existing APIs which then have been used in case studies. For example, [DHA 06] discusses an experiment on the validation of the Air Traffic Control (ATC) data link protocol used in ACARS or ATN networks. This protocol is originally fully specified in SDL, and the authors verify an IF version against properties expressed by observers, and within contexts that are also defined by observers. The authors have developed tools for automatic generation of such observers from higher-level descriptions, such as particular temporal logics.

[HÉD 05] reports on a use of IF for Quality of Service (QoS) evaluation of a driver for a data acquisition system. Non-probabilistic QoS criteria such as the minimum/maximum data acquisition delay or the maximum loss rate are derived by exhaustive simulation from a timed model of the driver.

10.6. Case study: the Ariane 5 flight program

Here, we select one of the more complete case studies with respect to the analysis and also the used features, which is the modeling and validation of the flight software of Ariane 5 by using the IF toolset. The study consisted of modeling a part of the existing software in a formalism amenable to validation with IF, and specifying a set of critical properties to be verified on that model.

For historical reasons, this study was performed on two similar models provided by EADS SPACE Transportation, one in SDL and one in UML. The two models were verified for different properties, and using different techniques. To simplify the presentation, except when stated, we do not distinguish between the two models.

We summarize the relevant results of both experiments, and we give the principles of a verification methodology that can be used in connection with the IF toolset. For such large examples, push-button verification is not sufficient and some iterative combination of analysis and validation is necessary to cope with complexity.

10.6.1. *Overview of the Ariane 5 flight program*

Ariane 5 is the European heavy-lift launcher, capable of placing in orbit payloads up to 9.5 tons. The flight software controls the launcher's mission from lift-off to payload release. It operates in a completely autonomous mode and has to handle both external disturbances (e.g. wind) and different hardware failures that may occur during the flight.

This case study takes into account the most relevant features of such an embedded application and focuses on the real time critical behavior by abstracting from complex functionality (control algorithms) and implementation details, such as specific hardware and operating system dependencies. Nevertheless, it is fully representative of an operational space system. A typical characteristic of such systems is that they implement two kinds of behavior:

– *cyclic synchronous algorithms*, mainly dedicated to control-command algorithms. In the remainder of the section they are called GNC for “Guidance, Navigation and Control”. The algorithms and their reactivity constraints are defined by the control engineers based on discretization of continuous physical laws;

– *aperiodic, event driven algorithms*. These algorithms manage the mission phases and perform particular tasks when the spacecraft changes from one permanent mode to another (engine ignition and stop, stage release, etc.), or when hardware failures occur (alternative or abortion maneuvers).

The software components implementing this functionality are physically deployed on a single processor and share a common bus for acquiring sensor data and sending commands to the equipment (actually, a set of replicated processors is used for guaranteeing some fault tolerance, but this is beyond the scope of our study).

We identified a set of properties, which include functional ones concerning the correct execution of command sequences and flight phases, and non-functional ones mostly concerning the task architecture and the scheduling of the system.

The functional model is independent of the task architecture. It is structured around six main components, modeled by singleton classes in UML and by processes in SDL:

– *Acyclic* is the main mission management component, which handles the start of the flight sequence and the switching from one phase to another. Its behavior is described by a state machine reacting to event receptions from the GNC algorithms (e.g., end of thrust detection) or from the environment, and to time conditions (e.g., time window protections ensuring that the treatment associated with an external event is performed within a predefined time window even in the case of failure of the event detection mechanism).

– A set of specific components handle each the acyclic activities related to a particular launcher stage. They react to events received from *Acyclic* or to internal time constraints. In the study, we considered only two stages: *EAP* (lateral boosters) and *EPC* (main stage of the Ariane 5 launcher).

– *Cyclics* is a component which manages the activation of the cyclic control/command algorithms (GNC). The algorithms are executed in a predefined order, depending on the current state of the launcher, which is tracked by the *Acyclic* class. We consider in more detail two of the algorithms activated by *Cyclics*, each one implemented by a separate object: *Thrust_Monitor*, responsible for the monitoring of the *EAP* thrust, and *Guidance_Task* which has an activation frequency that is lower than that of the other GNC algorithms. All other GNC algorithms are only represented by their non-functional characteristics, namely their worst case execution time (WCET).

This is sufficient for the purpose of timing analysis, and the numerical correctness of the algorithms was not considered in our study.

In order to validate the software, a part of the environment needs to be modeled. In our case, it includes two kinds of spacecraft equipment – *valves* and *pyrotechnic commands* (the model includes possible hardware failures), the external environment – namely the ground control center, as well as abstractions of parts of the software which are not described in the model, such as a numerical algorithm or the 1553MIL bus.

The verification was aimed to prove the feasibility of the scheduling policy adopted and the absence of certain types of clashes in the use of the shared bus, as well as some twenty functional safety properties identified by the EADS engineers, which can be classified as follows:

– *general requirements*, not necessarily specific to Ariane 5 but common to all critical real-time systems. They include basic untimed properties such as the absence of deadlocks, livelocks or signal loss, and basic timed properties such as the absence of timelocks and Zeno behaviors;

– *overall system requirements*, specific to the flight software and concerning its global behavior. For example, the global sequencing of the flight phases is respected: ground, vulcain ignition, booster ignition, etc.;

– *local component requirements*, specific to the flight software and regarding the functionality of some of its subsystems. This category includes, for example, checking the occurrence of some actions in some component (e.g, payload separation occurs eventually during an attitudinal positioning phase, or the stop sequence no. 3 can occur only after lift-off, or the state of engine valves conforms to the flight phase, etc.).

10.6.2. *Verification of functional properties*

Validation is a complex activity, involving the iterated application of verification and analysis phases as depicted in Figure 10.5.

Translation to IF and basic static analysis. This provides a first sanity check of the model. In this step, the user can find simple compile-time errors in the model (name errors, type errors, etc.) but also more elaborate information (uninitialized or unused variables, unused signals, dead code).

Model exploration. The validation process continues with a debugging phase. Without being exhaustive, the user begins to explore the model in a random or guided manner. Simulation states need not be stored because error detection, and not exhaustive exploration of the model is aimed for in this phase. Another aim is to inspect and validate known nominal scenarios of the specification. Moreover, the user can *test*

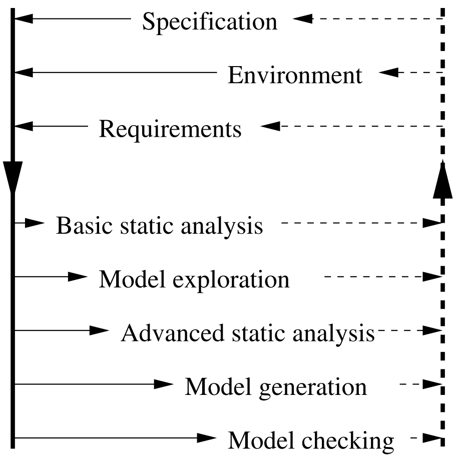


Figure 10.5. *Validation methodology in IF*

simple safety properties, which must hold on all execution paths. We usually test only generic properties, such as absence of deadlock and signal loss, or local assertions (invariants).

Advanced static analysis. The aim is to simplify the IF description and to reduce state space explosion during later verification. We use the following static analysis techniques to reduce both the state vector and the state space, while completely preserving its behavior (with respect to an observability criterion defined in terms of active variables and communication occurrences):

– A specific analysis technique is the elimination of redundant clocks [DAW 96]. Two clocks are *dependent* in a control state if their difference is constant and can be statically computed at that state. The initial SDL version of the flight program used no less than 130 timers. Using our static analysis tool we were able to reduce them to only 55 functionally independent timers.

– A second optimization identifies live equivalent states by introducing systematic resets for dead variables in certain states of the specification. For this case study, the live reduction has not been particularly effective due to the reduced number of variables (other than clocks). Nevertheless, our initial attempts to generate the model without live reduction failed, whereas using live reduction we were able to build the model with about $2 \cdot 10^6$ states and $18 \cdot 10^6$ transitions; but this size made subsequent analysis still unmanageable.

– The last static optimization is dead-code elimination. We used this technique to automatically eliminate some components which do not perform any relevant action.

State space generation. The LTS generation phase aims at building a state graph of the specification by exhaustive simulation. In order to cope with the complexity, the user may choose an adequate state representation – e.g., discrete or dense representation of time – and an exploration strategy – e.g., a pre-defined traversal order, use of partial order reductions, scheduling policies, etc.

The use of partial order reduction has been necessary and determining for constructing tractable models. We applied a simple *static* partial order reduction which eliminates spurious interleaving between internal steps occurring in different processes at the same time. Internal steps are those which do not perform visible communication actions, neither signal emission or access to shared variables. This partial order reduction imposes a fixed exploration order between internal steps and preserves *all* the properties expressed in terms of visible actions.

EXAMPLE.– By using partial order reduction on internal steps, we reduced the size of the model by 3 orders of magnitude, that is, from $2 \cdot 10^6$ states and $18 \cdot 10^6$ transitions to $1.6 \cdot 10^3$ states and $1.65 \cdot 10^3$ transitions, a model which can be easily handled by any model checker.

We considered two different models of the environment. A *time-deterministic* one, where actions take place at exactly defined time points and a *time non-deterministic* one where actions take place within predefined time intervals. Table 10.1 presents in each case the sizes of the models obtained depending on the generation strategy used. The number of states of the computed state graphs may be considered as relatively small with respect to the millions of states reached by some model checkers. It should however be noticed that each state is not just a few words size but about 10 kB; we can handle such a model with an ordinary workstation only thanks to the efficient state space storage strategy explained in section 10.4.1.

		time deterministic	time non-deterministic
model generation	− live reduction − partial order	state explosion	state explosion
	+ live reduction − partial order	2201760 st. 18706871 tr.	state explosion
	+ live reduction + partial order	1604 st. 1642 tr.	195718 st. 278263 tr.
model verification	model minimization	$\sim$ 1 sec.	$\sim$ 20 sec.
	model checking	$\sim$ 15 sec.	$\sim$ 120 sec.

Table 10.1. *Verification results. The model minimization and model checking experiments are performed on the smallest available models, obtained using live and partial order reduction*

Model checking. Once the model has been generated, three model checking techniques have been applied to verify requirements on the specification:

1. Model checking of μ-calculus formulae using EVALUATOR.

EXAMPLE.– The requirement expressing that "*the stop sequence no. 3 occurs only during the flight phase, and never on the ground phase*" can be expressed by the following μ-calculus formula, verified with EVALUATOR:

$$\neg\, \mu X \cdot < EPC!Stop\_3 > T \vee\ < \overline{EAP!Fire} > X$$

This formula expresses the requirement that the system does not execute the *stop sequence no. 3* without firing first EAP.

2. Construction of reduced models using ALDEBARAN. A second approach, usually much more intuitive for a non-expert end-user, consists of computing an abstract model (with respect to a given observation criterion) of the overall behavior of the specification. Possible incorrect behaviors can be detected by visualizing the resulting model (if it is small enough).

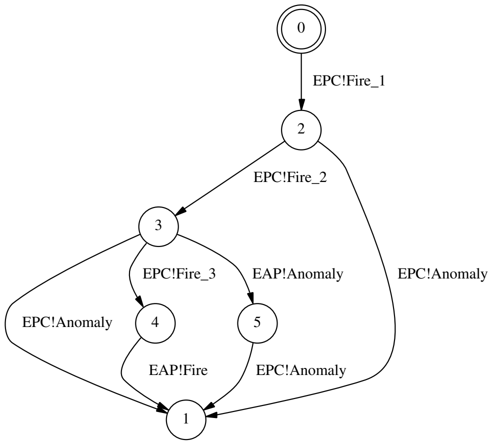
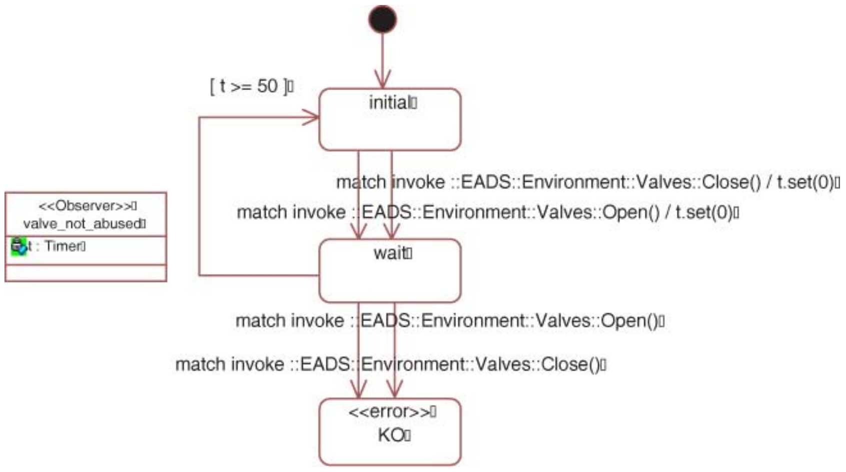


Figure 10.6. *Minimal model*

EXAMPLE.– All safety properties involving the firing actions of the two principal stages, EAP and EPC, and the detection of anomalies are preserved on the LTS in Figure 10.6 generated by ALDEBARAN. It is the quotient model with respect to safety equivalence [BOU 91] while keeping observable only the actions above. For instance, it is easy to check on this abstract model that, whenever an anomaly occurs *before* action *EPC!Fire_3* (ignition of the Vulcain engine), neither this action nor *EAP!Fire* action are executed and therefore the entire launch procedure is securely aborted.



[t >= 50]
initial
<<Observer>>
valve_not_abused
t : Timer
match invoke ::EADS::Environment::Valves::Close() / t.set(0)
match invoke ::EADS::Environment::Valves::Open() / t.set(0)
wait
match invoke ::EADS::Environment::Valves::Open()
match invoke ::EADS::Environment::Valves::Close()
<<error>>
KO

Figure 10.7. *A timed safety property of the Ariane 5 model*

Table 10.1 gives the average time required for verifying each kind of property by temporal logic model checking and model minimization respectively.

3. Model checking with observers. We also used *UML observers* to express and check requirements. Observers allow us to express in a much simpler manner most safety requirements of the Ariane 5 specification. Additionally, they make it possible to express *quantitative* timing properties, something which is difficult to express with μ-calculus formulae. Consequently, observers are usually preferred over μ-calculus by non-experts in verification.

EXAMPLE.– Figure 10.7 shows one of the timed safety properties that was verified against the UML model: "between any two commands sent by the flight software to the valves, at least 50 ms must elapse".

10.6.3. *Verification of non-functional properties*

The IF toolset may also be used for modeling and validating some non-functional properties, specifically properties related to time and schedulability. One objective of the Ariane 5 study was to validate the system under different hypotheses concerning the task architecture and scheduling policy.

The policy used in the system is based on a cyclic fixed-priority preemptive scheme, with priorities assigned by rate monotonic analysis (RMA [LAY 73]).

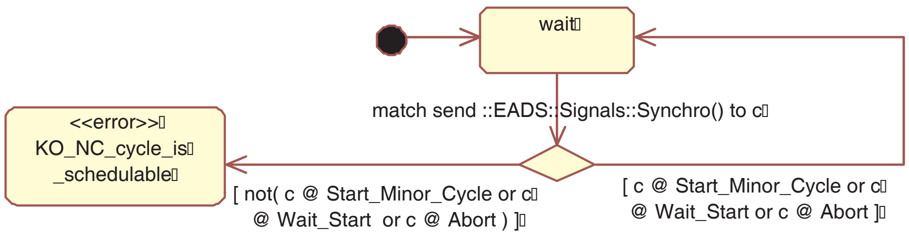

wait
match send ::EADS::Signals::Synchro() to c
<<error>>
KO_NC_cycle_is
_schedulable
[not(c @ Start_Minor_Cycle or c
@ Wait_Start or c @ Abort)]
[c @ Start_Minor_Cycle or c
@ Wait_Start or c @ Abort]

Figure 10.8. *A scheduling property of the Ariane 5 model*

Acyclic events are handled by cyclic sampling, and certain precautions are taken, for example, delaying the bus writing operations of each task until the end of the task's cycle. In turn, this statically ensures some important properties such as overall schedulability and mutual exclusion of writes on the bus, at least as long as the WCET estimations are correct.

Nevertheless, the task architecture is considered by the engineers as too restrictive, in particular in the presence of acyclic events requiring very short response times, or in the presence of cyclic algorithms requiring to read or write data during their cycle. Moreover, rate monotonic analysis is done under very pessimistic hypotheses for the execution times, which, for certain tasks of the Ariane 5 system, vary a lot depending on the flight phase.

One goal in our study was to loosen the hypotheses, namely those concerning the sampling of acyclic events and concerning the reading and writing of the bus. The hypotheses for RMA are therefore no longer fulfilled, and schedulability and mutual exclusion is to be verified based on (an abstraction of) the functional model. In order to do this, we integrated a model of a fixed priority preemptive scheduler in the system model, using the timed modeling artifacts offered by IF.

Once this is done, scheduling and mutual exclusion goals become safety properties of the obtained model, and can be formalized and verified using observers. This is the case for the schedulability of the NC task which can be expressed as: "the computation of the NC component finishes before the end of every period", which means that "the *Cyclics* component receives the *Synchro* signal only while being either in state *Start_Minor_Cycle*, in *Wait_Start* or in *Abort* (the other states are intermediate computation states)". This is formalized by the observer in Figure 10.8.

10.6.4. *Modular verification and abstraction*

The verification of large models such as that of Ariane 5 is generally subject to combinatorial explosion of the state space. The explosion cannot always be avoided using only strongly behavior-preserving reduction techniques such as partial order reduction or the aforementioned static analysis methods. One way to deal with this issue is the use of compositional abstractions or compositional verification: when a property concerns only a part of the system, we can replace all other parts – which then play the role of an environment – by a simplified model representing an abstraction of it. Such a simplified model can be obtained either automatically – for example, using the variable abstraction tools of IF, or building abstractions manually. If the abstract model satisfies a safety property based on observations preserved by the abstraction, we can conclude that the initial model also satisfies the property. On the other hand, if the abstract model does not satisfy a property, the result may represent a false negative, meaning that a more precise abstraction would make verification possible.

To illustrate this verification based on modular abstraction, consider the safety properties of the acyclic part of the Ariane 5 model. To verify them, the cyclic part (GNC) is replaced by an abstraction which discards all behavior details of the GNC component, and sends messages to the acyclic part at arbitrary time points (within some pre-determined intervals) – rather than at moments precisely computed by the GNC. This abstraction, although very coarse and easy to construct, proved sufficient to verify the safety properties of the acyclic part.

More details on the non-functional analysis of the Ariane 5 model can be found in [OBE 06a].

10.7. Conclusion

The IF toolset is the result of a long term research effort for theory, methods and tools for model-based development. It offers a unique combination of features for modeling and validation including support for high level modeling, static analysis, model checking and simulation. It has been designed with special care for openness to modeling languages and validation tools thanks to the definition of appropriate APIs. As mentioned in section 10.4.1, it has been connected to explicit model checking tools, to symbolic and regular model checkers and abstraction tools, as well as to the automatic test generation and test execution tools.

The IF notation is expressive and rich enough to map in a structural manner most UML concepts and constructs such as classes, state machines with action, and activity groups with run-to-completion semantics. The mapping flattens the description only for inheritance and synchronous calls and this is necessary for validation purposes. It preserves all relevant information about the structure of the model. This provides a basis for compositional analysis and validation techniques that should be further investigated.

The IF notation relies on a framework for modeling real-time systems based on the use of priorities and of types of urgency studied at Verimag [BOR 98, BOR 00, ALT 02]. The combined use of behavior and priorities naturally leads to layered models and allows compositional modeling of real-time systems, in particular of aspects related to resource sharing and scheduling. Scheduling policies can be modeled as sets of dynamic priority rules. The framework supports the composition of scheduling policies and provides composability results for deadlock freedom of the scheduled system. Priorities are also an elegant mechanism for restricting non-determinism and controlling execution. Run-to-completion execution and mutual exclusion can be modeled in a straightforward manner. Finally, priorities prove to be a powerful tool for modeling both heterogenous interaction and heterogenous execution as advocated in [GÖS 03]. The IF toolset fully supports this framework. It embodies principles for structuring and enriching descriptions with timing information as well as expertise gained through its use in several large projects such as the IST projects OMEGA [OME 05, GRA 04], AGEDIS [AGE 03] and ADVANCE [ADV 04].

The combination of different verification techniques enlarges the scope of application of the IF toolset. Approaches can differ according to the characteristics of the model. For data intensive models, static analysis techniques can be used to simplify the model before verification, while for control intensive models partial order techniques and observers are very useful to cope with state explosion. The combined use of static analysis and model checking by skilled users proves to be a powerful means to break the complexity. Clearly, the use of high level modeling languages involves some additional cost in complexity with respect to low level modeling languages, e.g., languages based on automata. Nevertheless, this is a price to pay for verification of detailed models of real-life systems where faithful models include dynamical changes of structure and a potentially infinite state space. In our methodology, abstraction and simplification is carried out automatically by static analysis.

The use of observers for requirements proves to be very convenient and easy to use compared to logic-based formalisms. They allow for a natural description, especially of real-time properties relating timed occurrences of several events. The "operational" description style is much more easy to master and understand by practitioners. The limitation to safety properties is not a serious one for well-timed systems. In fact, IF descriptions are by construction well-timed, as time can always progress due to the use of urgency types rather than invariants for expressing urgency. In this context, liveness properties are expressible as bounded response, that is, safety properties.

The IF toolset is unique in that it supports rigorous high level modeling of real-time systems and their properties as well as a complete validation methodology. Compared to commercially available modeling tools, it offers more powerful validation features. For graphical editing and version management, it needs a front end that generates either XMI or SDL. We are currently using RATIONAL ROSE and OBJECTGEODE. We have also connections from RHAPSODY and ARGO UML. Compared to other validation tools, the IF toolset presents many similarities with SPIN. Both tools offer features such as a high level input language, integration of external code, use of enumerative model checking techniques as well as static optimizations. In addition, IF allows the modeling of real-time concepts, an efficient state space storage algorithm for handling models with large states, and the toolset has an open architecture which eases the connection with other tools.

10.8. Bibliography

[ABR 88] ABRIAL J.-R., "The B Tool (Abstract).", *VDM '88, VDM – The Way Ahead, 2nd VDM-Europe Symposium, Dublin, 1988*, vol. 328 of *LNCS*, p. 86–87, 1988.

[ADV 04] ADVANCE CONSORTIUM, http://www.liafa.jussieu.fr/~advance, IST ADVANCE project, 2000–2004.

[AGE 03] AGEDIS CONSORTIUM, http://www.agedis.de, IST AGEDIS project, 2000–2003.

[AKH 04] AKHAVAN A., BOZGA M., BENSALEM S., ORFANIDOU E., "Experiment on Verification of a Planetary Rover Controller", *4th Intl. Workshop on Planning and Scheduling for Space, IWPSS'04*, Darmstadt, 2004.

[ALT 00] ALTISEN K., GÖSSLER G., SIFAKIS J., "A Methodology for the Construction of Scheduled Systems", *FTRTFT'00*, vol. 1926 of *LNCS*, Springer, p. 106–120, 2000.

[ALT 02] ALTISEN K., GÖSSLER G., SIFAKIS J., "Scheduler Modeling Based on the Controller Synthesis Paradigm", *Journal of Real-Time Systems*, special issue on "Control-theoretical approaches to real-time computing", vol. 23, num. 1/2, p. 55–84, 2002.

[ALU 94] ALUR R., DILL D., "A Theory of Timed Automata", *Theoretical Computer Science*, vol. 126, p. 183–235, 1994.

[ANN 01] ANNICHINI A., BOUAJJANI A., SIGHIREANU M., "TReX: A Tool for Reachability Analysis of Complex Systems", *Int. Conf. CAV'01*, Paris, France, vol. 2102 of *LNCS*, Springer, 2001.

[BEN 93] BENVENISTE A., LE GUERNIC P., CASPI P., HALBWACHS N., "A Decade of Concurrency, Reflexions and Perspectives", *REX Symposium*, vol. 803 of *LNCS*, 1993.

[BEN 98] BENSALEM S., LAKHNECH Y., OWRE S., "Computing Abstractions of Infinite State Systems Compositionally and Automatically", *Int. Conf. CAV'98*, Vancouver, Canada, vol. 1427 of *LNCS*, Springer, p. 319–331, 1998.

[BEN 03] BENVENISTE A., CASPI P., EDWARDS S. A., HALBWACHS N., GUERNIC P. L., DE SIMONE R., "The Synchronous Languages 12 Years Later", *Proc. of the IEEE*, vol. 91, num. 1, p. 64–83, 2003.

[BEN 04] BENSALEM S., BOZGA M., KRICHEN M., TRIPAKIS S., "Testing Conformance of Real-Time Software by Automatic Generation of Observers", *Runtime Verification Workshop, RV'04*, 2004.

[BOI 02] BOIGELOT B., LATOUR L., "The Liège Automata-based Symbolic Handler LASH", http://www.montefiore.ulg.ac.be/~boigelot/research/lash, 2002.

[BOR 98] BORNOT S., SIFAKIS J., TRIPAKIS S., "Modeling Urgency in Timed Systems", *Int. Symp.: Compositionality – The Significant Difference*, vol. 1536 of *LNCS*, 1998.

[BOR 00] BORNOT S., SIFAKIS J., "An Algebraic Framework for Urgency", *Information and Computation*, vol. 163, p. 172–202, 2000.

[BOS 00] BOSNACKI D., DAMS D., HOLENDERSKI L., SIDOROVA N., "Model Checking SDL with SPIN.", GRAF S., SCHWARTZBACH M. I., Eds., *Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems, TACAS'00, Berlin*, vol. 1785 of *LNCS*, Springer, p. 363–377, 2000.

[BOU 91] BOUAJJANI A., FERNANDEZ J., GRAF S., RODRIGUEZ C., SIFAKIS J., "Safety for Branching Time Semantics", *ICALP'91*, vol. 510 of *LNCS*, Springer, 1991.

[BOZ 97] BOZGA M., FERNANDEZ J., KERBRAT A., MOUNIER L., "Protocol Verification with the Aldebaran Toolset", *STTT, Journal for Software Tools for Technology Transfer*, vol. 1, num. 1–2, p. 166–184, 1997.

[BOZ 99] BOZGA M., FERNANDEZ J., GHIRVU L., GRAF S., KRIMM J., MOUNIER L., SIFAKIS J., "IF: An Intermediate Representation for SDL and its Applications", *SDL FORUM'99, Montreal, Canada*, Elsevier, p. 423–440, 1999.

[BOZ 00] BOZGA M., FERNANDEZ J.-C., GHIRVU L., JARD C., JÉRON T., KERBRAT A., MOREL P., MOUNIER L., "Verification and Test Generation for the SSCOP Protocol", *J. of Science of Computer Programming, Special Issue on Formal Methods in Industry*, vol. 36, num. 1, p. 27–52, Elsevier, 2000.

[BOZ 01] BOZGA M., LESENS D., MOUNIER L., "Model-Checking Ariane 5 Flight Program", *FMICS'01, Paris*, INRIA, p. 211–227, 2001.

[BOZ 02a] BOZGA M., GRAF S., MOUNIER L., "IF-2.0: A Validation Environment for Component-Based Real-Time Systems", *Conf. on Computer Aided Verification, CAV'02, Copenhagen*, vol. 2404 of *LNCS*, Springer, 2002.

[BOZ 02b] BOZGA M., LAKHNECH Y., "IF-2.0: Common Language Operational Semantics", Technical Report, Verimag, 2002.

[BOZ 03a] BOZGA M., MALER O., "Timed Automata Approach for the Axxom Case Study", Technical Report, Verimag, 2003.

[BOZ 03b] BOZGA M., FERNANDEZ J.-C., GHIRVU L., "Using Static Analysis to Improve Automatic Test Generation", *STTT, Int. J. on Software Tools for Technology Transfer*, vol. 4, num. 2, p. 142–152, 2003.

[BOZ 04] BOZGA M., GRAF S., OBER I., OBER I., SIFAKIS J., "The IF Toolset", *4th Int. School on Formal Methods for the Design of Computer, Communication and Software Systems: Real-Time, SFM-04:RT, Bologna*, vol. 3185 of *LNCS Tutorials*, Springer, 2004.

[BOZ 06] BOZGA M., GRAF S., MOUNIER L., OBER I., *La boîte à outils IF pour la modélisation et la vérification de systèmes temps réel*, Chapter 9, Hermes, Lavoisier, 2006.

[COM 07] COMBES P., HAREL D., KUGLER H., "Modeling and Verication of a Telecommunication Application using Live Sequence Charts and the Play-Engine Tool", forthcoming in *SoSym Int. Journal*, Springer, 2007.

[DAM 99] DAMM W., HAREL D., "LSCs: Breathing Life into Message Sequence Charts", *FMOODS'99 IFIP TC6/WG6.1 Int. Conf. on Formal Methods for Open Object-Based Distributed Systems*, Kluwer Academic Publishers, 1999.

[DAM 02] DAMM W., JOSKO B., PNUELI A., VOTINTSEVA A., "Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML", *Symposium FMCO'02*, LNCS, Springer, 2002.

[DAW 96] DAWS C., YOVINE S., "Reducing the Number of Clock Variables of Timed Automata", *RTSS'96*, Washington DC, IEEE Computer Society Press, p. 73–82, 1996.

[DHA 06] DHAUSSY P., ROGER J., BONIN H., SAVES E., HONNORÉ J., "Experimentation of Timed Observers for Validation of an Avionics Software", *Int. Conf. on Embedded Real Time Software (ERTS), Toulouse*, 2006.

[FER 96a] FERNANDEZ J., JARD C., JÉRON T., VIHO C., "Using On-the-fly Verification Techniques for the Generation of Test Suites", *CAV'96*, vol. 1102 of LNCS, 1996.

[FER 96b] FERNANDEZ J., GARAVEL H., KERBRAT A., MATEESCU R., MOUNIER L., SIGHIREANU M., "CADP: A Protocol Validation and Verification Toolbox", *CAV'96 (New Brunswick, USA)*, vol. 1102 of *LNCS*, Springer, p. 437–440, 1996.

[FER 03] FERNANDEZ J.-C., BOZGA M., GHIRVU L., "State Space Reduction Based on Live Variables Analysis.", *Sci. Comput. Program.*, vol. 47, num. 2–3, p. 203–220, 2003.

[GLA 89] VAN GLABBEEK R., WEIJLAND W., "Branching-Time and Abstraction in Bisimulation Semantics", Report num. CS-R8911, CWI, Amsterdam, 1989.

[GÖS 03] GÖSSLER G., SIFAKIS J., "Composition for Component-Based Modeling", *Proc. FMCO'02*, vol. 2852 of *LNCS*, Springer, 2003.

[GÖS 05] GÖSSLER G., SIFAKIS J., "Composition for Component-Based Modeling", *Science of Computer Programming*, p. 161–183, 2005.

[GRA 01] GRAF S., JIA G., "Verification Experiments on the Mascara Protocol", *SPIN Workshop 2001, Toronto*, vol. 2057 of *LNCS*, Springer Verlag, p. 123–142, 2001.

[GRA 04] GRAF S., HOOMAN J., "Correct Development of Embedded Systems", *European Workshop on Software Architecture: Languages, Styles, Models, Tools, and Applications (EWSA 2004), co-located with ICSE 2004, St Andrews, Scotland*, LNCS, 2004.

[GRA 05] GRAF S., DE BOER F., COMBES P., HOOMAN J., KUGLER H., KYAS M., LESENS D., OBER I., VOTINTSEVA A., YUSHTEIN Y., ZENOU M., "Final Project Report of the OMEGA IST project", 2005.

[GRA 06a] GRAF S., OBER I., OBER I., "Timed Annotations in UML", *STTT, Software Tools for Technology Transfer*, vol. 8, num. 2, 2006.

[GRA 06b] GRAF S., OBER I., OBER I., "Validating Timed UML Models by Simulation and Verification", *STTT, Software Tools for Technology Transfer*, vol. 8, num. 2, 2006.

[HAR 87] HAREL D., "Statecharts: A Visual Formalism for Complex Systems", *Sci. Comput. Programming 8, 231–274*, 1987.

[HAR 98] HAREL D., POLITI M., *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, 1998.

[HAR 04] HARTMAN A., NAGIN K., "The AGEDIS Tools for Model Based Testing", *ISSTA'2004*.

[HÉD 05] HÉDIA B. B., JUMEL F., BABAU J.-P., "Formal Evaluation of Quality of Service for Data Acquisition Systems", *FDL'05, Lausanne*, 2005.

[HOL 91] HOLZMANN G. J., *Design and Validation of Computer Protocols*, Prentice Hall Software Series, 1991.

[IBM] IBM, "Rational ROSE Development Environment", http://www-306.ibm.com/software/rational/.

[ILO] ILOGIX, "Rhapsody Development Environment", http://argouml.tigris.org/.

[JÉR 99] JÉRON T., MOREL P., "Test Generation Derived from Model Checking", *CAV'99 (Trento, Italy)*, vol. 1633 of *LNCS*, Springer, p. 108–122, 1999.

[KOZ 83] KOZEN D., "Results on the Propositional μ-Calculus", *Theoretical Computer Science*, 1983.

[LAR 98] LARSEN K., PETTERSSON P., YI W., "UPPAAL in a Nutshell", *Journal on Software Tools for Technology Transfer, STTT*, vol. 1, p. 134–152, 1998.

[LAY 73] LAYLAND J., LIU C., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, vol. 20, num. 1, 1973.

[LUG 01] LUGATO D., RAPIN N., GALLOIS J., "Verification and Test Generation for SDL Industrial Specifications with the AGATHA Toolset", *Real-Time Tools Workshop with CONCUR 2001, Aalborg*, 2001.

[MIL 80] MILNER R., *A Calculus of Communication Systems*, vol. 92 of *LNCS*, 1980.

[MUC 97] MUCHNICK S., *Advanced Compiler Design Implementation*, Morgan Kaufmann Publishers, 1997.

[OBE 04] OBER I., GRAF S., OBER I., "Model Checking of UML Models via a Mapping to Communicating Extended Timed Automata", *SPIN Workshop on Model Checking of Software*, vol. 2989 of *LNCS*, p. 127–145, 2004.

[OBE 06a] OBER I., GRAF S., LESENS D., "A Case Study in UML Model-Based Validation: The Ariane 5 Launcher Software", *FMOODS 2006*, vol. 4037 of *LNCS*, 2006.

[OBE 06b] OBER I., GRAF S., YUSHTEIN Y., "Using an UML Profile for Timing Analysis with the IF Validation Toolset", *Model-Based Development of Embedded Systems, MBEES, Dagstuhl, Germany*, num. 2006/01 Tech. Rep. SSE, U. Braunschweig, 2006.

[OME 05] OMEGA CONSORTIUM, http://www-omega.imag.fr, IST OMEGA project, 2002–2005.

[OMG 01] OMG, "Unified Modeling Language Specification (Action Semantics)", OMG Adopted Specification, December 2001.

[OMG 03a] OMG, "Model Driven Architecture", http://www.omg.org/mda, 2003.

[OMG 03b] OMG, "Standard UML Profile for Schedulability, Performance and Time, v. 1.0", OMG formal document, September 9, 2003.

[PAR 81] PARK D., "Concurrency and Automata on Infinite Sequences", *Theoretical Computer Science*, vol. 104, p. 167–183, 1981.

[PRI 02] PRIGENT A., CASSEZ F., DHAUSSY P., ROUX O., “Extending the Translation from SDL to Promela.”, *Int. SPIN Workshop on Model Checking of Software, Grenoble*, vol. 2318 of *LNCS*, p. 79–94, 2002.

[RAM] RAMIREZ A., VANPEPERSTRAETE P., RUECKERT A., ODUTOLA K., BENNETT J., TOLKE L., “ArgoUML Environment”, http://argouml.tigris.org/.

[SAE 04] SAE, “Architecture Analysis & Design Language (AADL)”, 2004.

[SAL 03] SALAH R. B., BOZGA M., MALER O., “On Timing Analysis of Large Combinational Circuits”, *Worksh. on Formal Modeling and Analysis of Systems, FORMATS*, 2003.

[SCH 04] SCHOLTEN J., ARBAB F., DE BOER F., BONSANGUE M. M., “Mocha-pi: an Exogenous Coordination Calculus based on Mobile Channels”, *ACM Symposium on Applied Computing (SAC 2005), ACM Press*, 2004.

[SIF 01] SIFAKIS J., “Modeling Real-Time Systems – Challenges and Work Directions”, *EMSOFT’01*, vol. 2211 of *LNCS*, Springer, 2001.

[SIF 03] SIFAKIS J., TRIPAKIS S., YOVINE S., “Building Models of Real-Time Systems from Application Software”, *Proc. IEEE*, vol. 91, num. 1, p. 100–111, 2003.

[TIP 94] TIP F., “A Survey of Program Slicing Techniques”, Report num. CS-R9438, CWI, Amsterdam, 1994.

[WEI 84] WEISER M., “Program Slicing”, *IEEE Transactions on Software Engineering*, vol. SE-10, num. 4, p. 352–357, 1984.

[YOV 97] YOVINE S., “KRONOS: A Verification Tool for Real-Time Systems”, *Software Tools for Technology Transfer*, vol. 1, num. 1+2, p. 123–133, 1997.

[ZWA 06] VAN DER ZWAAG M., HOOMAN J., “A Semantics of Communicating Reactive Objects with Timing”, *STTT, Int. Journal on Software Tools for Technology Transfer*, vol. 8, num. 2, Springer Verlag, 2006.

This page intentionally left blank

Chapter 11

Architecture Description Languages: An Introduction to the SAE AADL

11.1. Introduction

Software design received a great deal of attention in the 1970s. This research arose in response to the unique problems of developing large-scale software systems. The premise of the research was that architectural design (then referred to as programming-in-the-large) is an activity separate from implementation (also known as programming-in-the-small) [REM 76]. The notion of software architecture as a high-level model of the structure of software systems actually appeared in the 1990s and was then considered an important subfield of software engineering [GAR 93]. However, even if it is agreed that a (software) architecture deals with the coarse-grain elements and their overall interconnection structure, there is not a global consensus in the research community on what a software architecture is. For some people, it is a description (with special notations) of the software elements that compose a given system together with their interactions, while for others it designates an area of study.

Some of the benefits usually expected to be gained from the emergence of software architecture as a major discipline are:

– the architecture is a framework for tackling the increasing size and complexity of software systems and for satisfying all their requirements (functional as well as non-functional);

– the architecture is a technical basis for design but also a managerial one for cost estimation and process management;

Chapter written by Anne-Marie DÉPLANCHE and Sébastien FAUCOU.

– the architecture is an effective support for software use, reuse and evolution;

– the architecture is a sound basis for high-level reasoning in analysis and validation purposes.

These arguments show that the description of the architecture of a software system plays a key role in its design process. While being intrinsically at the core of the architectural design step, it also provides a reference point for the next development stages: they find information for their inputs (for example, for building formal models intended for specific analyses, or for code generation) and, in return, they expand it too. Thus, depending on the development step, the architecture description can be seen either as an abstract specification or as a detailed description of the operational system.

To support the architectural approach, various notations have been provided. They usually fall into three generations:

– first, the module interconnection languages or MILs: they provide grammar constructs for identifying software system modules (through required and provided resources) and for defining the interconnection specifications needed to assemble a complete program. MIL support during software development occurs after a system has been designed. Once a system structure is determined, it may be coded in an MIL to be checked and verified for completeness and inconsistencies. MIL code must be maintained during implementation and then used for high-level maintenance during system operation and enhancement. In [PRI 86] a panorama of such languages is presented;

– next, the configuration languages that can be viewed as extended MILs: these languages have in common the notion of the (type of) component as the basic element from which systems are constructed. Components have well-defined interfaces made of interaction points (in a broad sense). Complex components can be built by composing (instances of) more elementary components and, as a result, the overall structure of a system is described as a hierarchical composition of primitive components. Composition through the formal binding of component interfaces is a fundamental mechanism that these languages have introduced. Their additional objective was to provide support for configuration programming i.e. the ability not only to initially specify the configuration of a system but also to modify it dynamically (by adding, removing, or replacing components). Thus, changes with their rules can be specified declaratively at the configuration level and can be applied to the system in such a way that its consistency is preserved and the disruption to the running application is minimized. Languages such as Conic [KRA 89] at the origin of Darwin [MAG 95, MAG 96], or Durra [BAR 93] were the most famous configuration languages;

– progressively, the configuration languages have provided more conceptual and formal facilities at the component level as well as for the connections, thus increasing the reasoning capabilities at the architectural level. Formal syntax and semantics have replaced the intuitive and simple notations. In the 1990s, the generic term of architecture description language or ADL appeared. At that time, a number of researchers in industry and academia proposed ADLs: Wright, Darwin, Unicon, Rapide, Aesop, C2 SADL, MetaH, etc. [MED 00]. A negative element of this proliferation of ADLs is that typically each ADL operates in a stand-alone fashion, making it difficult to combine the facilities of one ADL with those of another. In an attempt at this, the ACME language [GAR 97] was developed as a joint effort of the software architecture research community. However, it only provides a common interchange format for architecture design tools. Nowadays, recent advances in model-driven engineering with model transformations appear to be a promising response to such a problem. Lastly, the standardization of the AADL language or the introduction into UML2 [OMG 04] of some concepts issuing from ADLs (such as the connector one) show that this approach is now quite mature.

The architectural design approach is now perceived as key for developing advanced real-time systems that exhibit specific issues: complexity due to multiple interacting functionalities, distributed platforms, non-functional constraints (timing, quality of service, dependability) that make software and hardware elements intrinsically dependent, scarce resources (including processing power, memory and communication bandwidth) which must be optimized, dynamic reconfiguration, predictability and thus verification needs *as soon as possible* in the design process, etc. Moreover, the increasing use of real-time embedded systems within industry, such as the automotive, avionics and aerospace fields (in which products come in a variety of forms and are increasingly built from components provided by external suppliers) gives rise to new requirements such as flexibility, reusability, portability, interoperability, etc. In order to be met, such requirements must be taken into account from the high-level design. Research has been and still is conducted into an "architecture-based" development process intended for real-time systems, and some dedicated ADLs have been proposed such as MetaH [BIN 96, VES 98], Clara [DUR 98, FAU 02, FAU 05], Cotre [FAR 03], or EAST-ADL [DEB 05]. Recently, the Society of Automotive Engineers (SAE) has standardized a "real-time ADL" for avionics applications: AADL (Architecture Analysis and Design Language) [SAE 04].

The purpose of this chapter is to present the architectural approach in the context of real-time system design. In section 11.2, the fundamental concepts that characterize a (general) ADL are introduced. Next, the real-time domain is targeted. In section 11.3, some of the main features of such real-time systems are recalled; we cannot ignore them when designing the architecture of a real-time system and thus a "real-time ADL" must offer facilities to support them. The difficulty in encompassing all these specificities naturally leads to the notion of architectural view; it is briefly discussed there too. The section ends with a short overview of the main real-time ADL proposals. Since AADL has now emerged as a standard in the real-time domain, we go deeper into its presentation in section 11.6. An example is used to illustrate its major possibilities, as well as some of its analysis capabilities.

11.2. Main characteristics of the architecture description languages

ADLs have been developed as languages for expressing the high-level structure of the overall application rather than the implementation details. As mentioned in the introduction, a number of languages belong to this category. Moreover, some of them are quite different due to their syntax, their semantics, their expressive power or their application domain. Trying to give a precise definition of an ADL on which the research community can come to a consensus is a sensitive problem. As a solution, Medvidovic and Taylor [MED 00] preferred to establish a classification and comparison framework for the ADLs. In this context, they proposed a guide composed of those features that determine whether or not a particular notation is an ADL. Four essential concepts are identified: component, interface, connector and configuration. We summarize their definitions as follows:

– A component is a unit of computation or a data store. Its state changes with time and depends on its interactions with its environment. With the connectors, the components are the building blocks of an architectural description. Since software reuse is in one of the primary goals of architecture-based development, ADLs distinguish component types from instances. Component types can then be instantiated multiple times in an architectural specification and each instance may correspond to a different implementation of the component. Here, the focus is merely on the way the component may be used and not on the way it is implemented. Moreover, in order to perform useful analyses, a component semantics that describes component behavior (i.e. relationships and processing between input and output messages in its interface) should be modeled. When the component is atomic, its behavior is modeled in natural, algorithmic or formal language. When it is not atomic, it is defined hierarchically as an architecture of sub-components (architectural configuration).

– A component's interface is a set of interaction points (in a broad sense) between it and the external world. These interaction points are named, typed and directed: input or output port, required or provided service, etc. Furthermore, a set of constraints and properties can be associated with an interface specifying some kind of contract for using it.

– Connectors are architectural building blocks used to model interactions among components and the rules that govern those interactions. To abstract away the complex architecture-level communication protocols and make them reusable, an ADL should distinguish connector type from instance. In order to enable proper connectivity of components and their communications, a connector should own an interface composed of interface points called roles (in reference to the role to be played by the component attached to it). Connectors have to be connected with component interaction points (subject to compatibility between the interaction point and the role). How the binding between the roles is achieved is the concern of the glue: it is a behavioral specification of the connector similar to the semantic description of a component. The fact that connectors may be modeled as first-class objects is an innovative progress due to research into ADL.

– Architectural configurations are connected graphs of components and connectors that describe a part or the overall structure of a system. They rely on composition either “vertically” (hierarchical composition), or “horizontally” (interconnection) according to different points of view. Thus, a configuration may: abstract away a complex structure of individual components and connectors; describe the dependencies among components and connectors; depict the concurrent flows of control through these components and connectors; specify non-functional properties such as processing unit(s) on which the system will execute, etc. The design of an architecture has to follow some rules ensuring adherence to the intended good properties: typing of data flows, deadlock avoidance, conformance with some design heuristics (in that case, we may talk about an architectural style such as pipe and filter, dataflow, communicating processes, etc.). Some of these properties are intrinsically part of the language (typing of data flows), while others have to be explicitly expressed as a set of interaction constraints among components and connectors (adherence to a style) and/or checked *a posteriori* (deadlock avoidance).

11.3. ADLs and real-time systems

11.3.1. *Requirement analysis*

As stated in the introduction, modern real-time systems exhibit an increasing complexity due to their functional requirements as well as their non-functional ones (hardware architecture, run-time support, various constraints of timeliness, dependability, or power consumption, and mutual dependence of all these aspects). This explains why the architectural approach is an important step for their design and why they need appropriate languages such as ADLs that contribute to a global and abstract approach. However, due to the specificities of real-time systems, the basic concepts of ADLs have to be specialized and/or supplemented in order to include relevant information that is essential for the following development steps. Hereafter, we propose to highlight some of these points that seem of major importance to us.

Real-time systems are reactive ones with regard to events occurring in the controlled environment, to time signals (periodic timer, watchdog, etc.), or to internal events (synchronization between activities, exception handling, mode switch, etc.). Therefore, to conduct behavioral analyses, it is essential to be able to describe at the architectural level how the control flows and these events are in relation in the system.

The description of the physical controlled environment, or at least its observable interactions with the real-time system, has to be considered too. On the one hand, it enables the operational context to be taken into account in the validation process. On the other hand, since the interface with the controlled environment is often not yet precisely known at the beginning of the design, it enables the evaluation of how different kinds of instrumentation should act upon the structure and the behavior of the overall system.

Time is undoubtedly a fundamental dimension of these specific systems. Its handling at the architectural level is prerequisite. As already mentioned, it is part of the control flows of the system. Furthermore, it is essential for the quantification of properties and constraints that apply to real-time systems. Coming originally from the requirements, such properties and constraints are progressively refined during the development process. Being able to specify them from the architectural design suits well the documentation and the traceability throughout the development steps, and also guides the implementation choices (partitioning, allocation, scheduling, etc.).

The hardware architecture together with the run-time support are important components of a real-time system that cannot be excluded from its architectural description. On the one hand, for many applications, the definition as well as the implementation of these components are not free of constraints (technological, cost-saving, interoperability with existing systems, scarce resources, etc.). This is a problem in itself. On the other hand, the applicative software and the run-time platform are closely connected. The behavior of a real-time system from the functional and operational point of view is dependent not only on the executive and communication services, but also on the way it is distributed over the platform, the processing power of the computing units, the communication channel bandwidth, the memory management mechanisms, etc. Unless (rough and senseless) approximations are made, it is impossible to conduct analyses on a system architecture about timing, quality of service or dependability without jointly taking into account software and hardware (including the physical environment).

Even if the architecture validation made off-line and *a priori* is a means for fault-avoidance in critical systems, it is not sufficient to satisfy all forms of dependability constraints (including reliability, safety and security). Thus, online mechanisms for fault-tolerance based on software or/and hardware redundancy have to be introduced. On the one hand, the capabilities of the ADLs in structuring and composing facilitate the specification and implementation of these mechanisms. On the other hand, since the redundancy patterns used for fault tolerance are generally well-known ones, tools able to build a fault-tolerant architecture by applying such patterns to some selected components in an automatic or semi-automatic way are conceivable.

This is often the case where the configuration of a real-time system has to be modified through its running. It may be due to successive operating phases offering different functionalities such as those of launching, flight or landing for an avionics system. It may also appear in response to the detection and recovery of some exception situations. These various configurations are distinct operating modes that have to be activated or deactivated according to specific condition occurrences. Such modes with their switches take sense from the architectural level. It is desirable that a real-time ADL provides dedicated abstractions for their specification.

Lastly, is it necessary to mention that the design of real-time systems requires not only languages with an adequate expressiveness but also associated tools. The potential of verifying system properties at the architectural level appears very interesting since it may gain a lot from its overall knowledge of the system and should enable design errors to be detected as soon as possible in the development process. In other respects, the architectural description of a system, like the one given by an ADL, is the starting point of the deployment step. Depending on its level of abstraction and independence with regard to the target platform, the deployment of a description may be a simple translation or a complex building task. The architect must be aided by tools for exploring the implementation space and for generating and configuring the code. Using an ADL to derive, from a unique root, various models for V&V activities as well as deployment is a promising way to reduce the semantic gap between the different design levels, or at least to make traceability easier.

11.3.2. *Architectural views*

We have just listed a set of features of real-time systems that have to be supported at the architectural level. It is obvious that a unique description including all this information is not easily conceivable and would be quite difficult to understand. It is thus preferable to operate slice cuts through an architectural description that bring out particular information. Thus, an architectural view is a description that is seen from a given perspective and omits entities that are not relevant to this perspective. The goal is to make the design easier by considering the various enclosed sub-problems separately. There is no agreed standard set of views, or a standard that defines their names. However, one of the most widely accepted models for architectural views in software engineering is the “4+1 view model” (and its variants) [KRU 95]. Its success is mainly due to its close relation to the Unified Modeling Language (UML) which has become a *de facto* standard in the industry. This model defines five views: logical view, implementation view, process view, deployment view, and finally the “+1” view, which is called use-case view.

In his thesis [WAL 03], A. Wall underlines that it is not easy, or maybe not possible, to define and decompose all important issues of a software system into distinct views. For this reason, he introduces the notion of view aspects. A view aspect emphasizes a particular important issue or a view, and may very well exist in different forms in several views. He presents three different aspects that are important in the real-time domain: “temporal aspect”, “communication aspect”, and “synchronization aspect”.

Analogously, in [FAU 02], S. Faucou examines the three important parts of the gross structure of a real-time system, i.e. the software architecture, the run-time architecture and the operational architecture. For each of them, he identifies some relevant views:

– for the software architecture: the “component view” is concerned with the individual definition of the components of the system, via their interfaces as well as their internal behavior; the “structural view” shows the overall organization of the system, via the interconnection of components and connectors; the “reactive view” describes how the system reacts to those event occurrences that are observable at the architecture level, thus specifying the control flows into the system;

– for the run-time architecture: the “software run-time view” covers the run-time services that can be called by the application such as those of an RTOS, a middleware, or an application communication protocol, etc.; the “hardware run-time view” defines and characterizes the physical composition of the system made of hardware resources;

– for the operational architecture: the “implementation view” deals with the mapping of the software architecture onto the software run-time architecture, i.e. the way the high-level entities and mechanisms of the first one are implemented with the concrete objects and services of the second one; the “system view” is concerned with the deployment of the implementation elements upon the hardware run-time components, i.e. the allocation of tasks, data or messages, the scheduling and optimized use of hardware resources (processors, networks), the configuration of the software run-time platform, etc. It is only from this last view (the first one that incorporates altogether software, hardware and environmental aspects) that a significant timing analysis can be performed.

The EAST-EEA project[1] [DEB 05] has developed a similar point of view for embedded real-time systems in the automotive field. A cornerstone of the project has been the EAST-ADL, an ADL that enables all information needed during development to be captured, from early analysis to implementation. Five architectural artifacts have been proposed that describe the functionalities inside the vehicle with their relationships, from a very high level (that of the vehicle-user) down to a very low one (that of the tasks and messages). Some particular aspects of these artifacts are further emphasized, giving rise to several views: vehicle view, functional analysis architecture, functional design architecture, logical architecture, hardware architecture, technical architecture and operational architecture.

11.4. Outline of related works

As mentioned in the introduction, ADLs have received and still receive a great deal of attention from the research community. The systems concerned are mainly “general” software ones. However, their adoption for the development of real-time systems is significantly slower and until recently (i.e. before the emerging AADL proposal), works on MetaH were the major contribution in the domain.

1. EAST-EEA: Electronic Architecture and Software Technology – Embedded Electronic Architecture. ITEA project from June 2001 to June 2004, with multiple partners (vehicle manufacturers, automotive suppliers and research institutes). http://www.east-eea.net.

MetaH is a language and toolset for the development of real-time embedded systems (with high assurance requirements). It stems from research performed by S. Vestal and his team at Honeywell Systems and Research Center (with the support of DARPA and the US Army) in the late 1980s. Starting about mid-way through a DSSA (Domain Specific Software Architecture) program for avionics systems, it began to be used on other programs [BIN 96]. The MetaH language supports the specification of the software and the hardware architectures as well as how they are combined to form an overall system for real-time, time-and-space partitioned, fault-tolerant, scalable multi-processor systems. A toolset operates from the MetaH language. It consists of a series of tools such as textual and graphical editors, software/hardware binder, schedulability, reliability or safety/security modeling and analysis, executive generator and application builder. The generators for the analytic models and the implementation code preserve structure in a way that allow elements of the requirements, specifications and analysis results to be easily traced between each other and to elements of the implementation code. A number of industrial case studies (in particular the pilot application of a missile guidance system [MCC 98]) have used MetaH in prototypical system developments. They demonstrated the practicality of using an architecture-oriented development and an ADL as a core modeling notation for providing analysis capabilities of several performance-critical quality attribute dimensions as well as automatic generation of glue code. They demonstrated design and implementation cost savings too. For this reason, the MetaH's authors were encouraged to promote and extend it by leading a standardization work: AADL is based on the MetaH approach.

In the same period, an ADL named Clara (which stands for Configuration LAnguage for Real-time Applications) was defined by the real-time research team of IRCCyN [DUR 98, FAU 05] dedicated to the design of the functional high-level architecture of real-time systems. Clara provides textual and graphical syntaxes. It describes a real-time application as a set of concurrent coarse-grain components interacting through connectors in order to perform system functionalities. This description can be annotated so as to express application timeliness requirements and properties. While defining Clara, special attention has been paid to the description of the reactive aspects, i.e. the control flows. Compared to other ADLs, it enables complex synchronization and triggering conditions to be expressed. Clara has been considered the core of different academic works conducted by the real-time research team of IRCCyN. Unfortunately, those tools that have been developed around Clara have remained prototypical, only being used for feasibility purposes. In particular, the following points have been studied:

– the compilation into timed Petri nets for the validation of Clara architectures under the assumption of unlimited hardware resources [DUR 98];

– the mapping of Clara architectures onto OSEK/VDX compliant distributed platforms together with a validation approach based on the simulation of the SDL model of the resulting operational architecture [FAU 02];

– the use of Romeo (`http://romeo.rts-software.org`), Tina (see Chapter 1) and CADP (see Chapter 5) tools for checking some functional properties and the consistency between the timing constraints and the allocated time budgets [FAU 05].

More recently, the Cotre project has studied how to integrate real-time validation techniques and tools into an ADL-based approach. Having been conducted mainly around AADL, this is presented in the following sections. Due to space limitation we conclude this section by only citing other works dealing with "real-time ADLs" such as Unicon [ZEL 96], Basement [HAN 96] or EAST-ADL [DEB 05] (already introduced in section 11.3.2).

11.5. The AADL language

AADL (Architecture Analysis and Design Language) is an ADL intended for real-time systems embedded in avionics applications, flight management systems, space applications, automotive applications, robotics systems, industrial process control equipment, medical devices, etc., that have challenging resource constraints (size, weight, power), requirements for real-time response, fault-tolerance and specialized input/output hardware, and that must be certified to high levels of assurance. As stated above, its development is based on the experiences of using MetaH. Thus, in 2001, MetaH was taken as the basis of a standardization effort under the authority of the SAE (International Society for Automotive Engineers) and its ASD (Avionics System Division). The standardization committee is composed of US and European industries from avionics and space fields (Honeywell, Rockwell Collins, Airbus, Dassault Aviation, ESA, EADS, etc.). In November 2004, the SAE released the aerospace standard AS5506, named the AADL [SAE 04]. Since then, some complements have been brought in and others are currently under review. The SAE AADL standard consists mainly of:

– the specification of the core language with textual syntax, semantics and a graphical representation;

– a UML profile of the SAE AADL;

– an XML/XMI specification as an SAE AADL model interchange format;

– an Ada and C language compliance and program interface annex;

– an error model annex that extends the core language to support reliability modeling.

Because of the richness of this language and due to space limitations, it is not possible to give an exhaustive presentation here. Therefore, our intent is only to introduce its main principles and constructs. By illustrating the modeling of a control system, some specific points will be described. Interested readers may want to supplement this chapter with technical notes [FEI 04, HUD 05, FEI 06] or by reviewing the AADL standard [SAE 04] for details about exact syntax and allowable keywords and their semantics.

11.5.1. *An overview of the AADL*

The AADL provides modeling concepts to describe the run-time architecture of application systems in terms of concurrent threads and their interactions as well as their mapping onto an execution platform. An architecture is described in terms of (a hierarchy of) components and interactions between them. The component abstractions of the AADL are separated into three categories:

1) *application software*

a) thread: this represents a sequential flow that executes instructions within a binary image produced from source text. It models a schedulable unit. A thread always executes within the virtual address space of a process;

b) thread group: this represents an organizational component to logically group thread, data, and thread group components within a process;

c) process: this represents a virtual address space whose boundaries are enforced at run-time;

d) data: this represents data type and static data in the source text. Components (like threads or processes) can have shared access to data subcomponents according to a specified concurrency protocol (that has to be specified elsewhere);

e) subprogram: this represents an execution entry-point in the source text. It can be called from threads and from other subprograms;

2) *execution platform (hardware)*

a) processor: this is an abstraction of hardware and software that is responsible for scheduling and executing threads;

b) memory: this consists of hardware such as RAM, ROM or more complex permanent storage that stores binary images of code and data;

c) device: this represents sensors, actuators or other components that interface with the external environment;

d) bus: this represents a communication channel that can exchange control and data between memories, processors and devices;

3) *composite*

a) system: this is a hierarchical assembly of interacting application software, execution platform and system components.

Each component is described in AADL with two phases. The first one consists of specifying its type. A component type declaration defines a component's functional interface, which is visible by other components. The notion of interface has a broad meaning since it itself is composed of:

1) features: these specify how that component interfaces with other components in the system. There are four categories of features:

a) port: this is a communication interface for the exchange of data and events between components;

b) subprogram: this is a call interface for a service that is accessible to other components. A data subprogram feature represents a subprogram through which the associated data is manipulated, whereas a server subprogram represents an entry-point for a synchronous remote procedure call;

c) parameter: this represents a data value that can be passed into and out of subprograms;

d) subcomponent access: an inner data or bus component may be specified to be accessible to components outside using a provides access feature declaration. A component may also indicate that it requires access to a data or a bus component declared outside utilizing a requires access feature declaration;

2) flow specifications: these are logical flows through the component's ports. The purpose of specifying flows is to support various forms of analysis;

3) properties: these are valued attributes and characteristics of the component.

The second part of the description of a component is concerned with its contents, i.e. its implementation. It defines the component's internal structure in terms of subcomponents and subcomponent connections. Moreover, it specifies how the flow specifications are realized through the subcomponents. It also gives some property values to express non-functional attributes, and it enables the representation of the operational states of the component and the possible run-time passages from one state to another with modes and mode transitions.

Each implementation is associated with a type of the same component category. Several implementations may be associated with each type. Type and implementation declarations have to meet specific rules according to each component category.

There are extension and refinement mechanisms to describe the components. They allow a description to be refined by modifying an already existing component, or by adding characteristics to a partially defined one. This is very useful in an incremental design process in which the architectural description is progressively built. Moreover, AADL enables collections of component declarations to be organized into separate units with their own namespaces thanks to packages.

AADL has very interesting extension capabilities enabling every specificity of a particular application to be taken into account. On the one hand, as mentioned above, valued properties can be associated with each component. Some of them are predefined in the core AADL (for instance, the period between successive dispatches of a thread, or the size of a memory). However, new property sets can also be added by the user tailored to his/her specific needs (no specific notational capability is provided as part of the AADL to describe the semantic meaning of such properties). These additions can be used to accommodate specialized modeling and analysis which can be defined in AADL annexes. An annex declaration enables extensions to the core language concepts and syntax. Annex subclauses and libraries can use any vocabulary word. The standard AADL-compliant tools are not required to process the content of such annex declarations; however, ad hoc processing methods have to be developed for them.

To achieve an architecture specification, the AADL standard does not prescribe any design process. Thus, an AADL specification can be designed top-bottom, bottom-up, following a component based-approach, etc. Readers looking for details about the design process of an AADL specification can refer to [HUD 05], where the architecture of an automotive cruise-control system is specified following a top-bottom approach. In the next sections, we highlight some key issues of real-time embedded system design, and we illustrate how AADL helps in tackling these issues. Once a valid high-level architecture specification has been obtained, AADL can serve as a basis for the detailed design step. Indeed, specific constructs are offered to perform architecture refinement down to a sufficient level of detail to enable automatic code generation and binary image building. These constructs can also be used to ease reusability of AADL design. This last point will be discussed in section 11.6.3.

11.6. Case study

11.6.1. *Requirements*

In order to illustrate the use of AADL, we will use a very simple case study: the design of a single-input single-output sampled control system (see Figure 11.1). Despite its simplicity, it will allow us to illustrate the main concepts of the language and present some of the analysis possibilities that can be run on an AADL specification. More complex examples can be found in other works or on the Internet (see for instance [HUD 05], where AADL is used to develop the architecture of an automotive cruise control system).

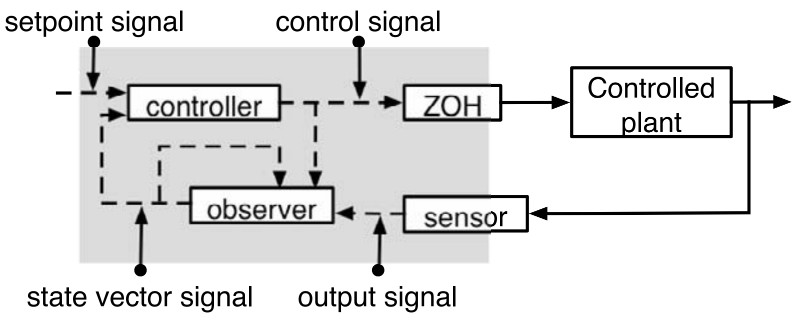


Figure 11.1. *Block diagram of the case study. Dashed lines represent discrete-time signals, while continuous lines represent continuous-time signals. The gray box defines the boundaries of that part of the system that can be designed with AADL*

The main goal of such a system is to control the plant, so as to drive its state "close to" a specified setpoint. In order to do so, it has to periodically sample the value of some output signal (of the plant), and produce the value of a control signal, sent back to the plant. Such a system is intrinsically a real-time system. Indeed, the required speed of the control system is derived from the analysis of the dynamics of the plant and the signals. For a detailed presentation of the real-time requirements of control systems, see [TÖR 98].

Following an example given in [TÖR 98], we break down the control system (the boundaries of which are shown by the gray box) into 4 functional blocks:

– the controller: this is in charge of producing the control signal. Its inputs are the desired setpoint and the state vector of the plant. It is a periodic activity, usually implemented through software components;

– the observer: this is in charge of producing the state vector signal. Its inputs are the (sampled version of the) plant output signal, the control signal and the previous state. It is a periodic activity, usually implemented through software components;

– the ZOH (zero-order hold): this is in charge of the digital-to-analog conversion of the control signal. It is a periodic activity, usually implemented through a dedicated hardware device (and the corresponding device driver software);

– the sensor: this is in charge of the analog-to-digital conversion of the output signal. It is a periodic activity, usually implemented through a dedicated hardware device (and the corresponding device driver software).

Let us consider that preliminary control-related studies have shown that an acceptable quality of control is obtained if the control loop is executed every 20 ms, with a maximal input-output latency (delay between a sampling instant and its corresponding actuation instant) of 15 ms. These same studies have shown that, due to the presence of noise on the sampled output signal, the observer must be executed every 10 ms in order to accurately track the system state. Thus, we have to specify a system architecture for a multi-rate control loop.

11.6.2. *Architecture design and analysis with AADL*

The case study used to illustrate the following sections has been developed within the OSATE/Topcased AADL open-source studio (version 1.4.3, available from `http://www.aadl.info`).

11.6.2.1. *High-level design*

The translation between the functional blocks of the block diagram of Figure 11.1 and the AADL component categories is straightforward:

– the sensor and the ZOH actuator are mapped on devices;

– the observer and controller are mapped on threads.

Each AADL thread must be associated with a process (a process is associated with an address space and a thread is associated with a control flow). Thus, we have to define a process `proc` that will contain the two threads. By reproducing the structure of the block diagram for the specification of its implementation `proc.i`, we obtain the graphical specification of Figure 11.2.

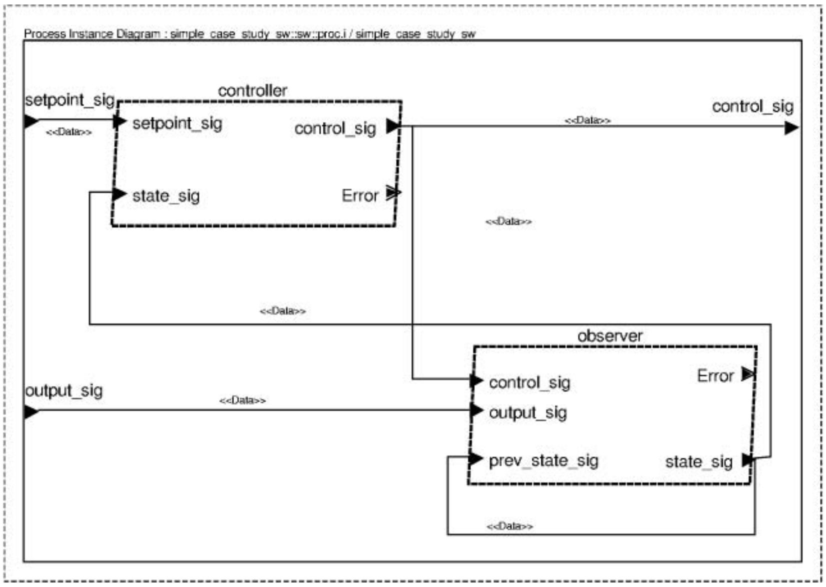


Figure 11.2. *A gross-grain software architecture for the case study*

In an operational AADL system specification, the binding between software and hardware components must be defined: each thread must be executed on a processor, and each process must be hosted on a memory, and each connection between a device and a thread must be mapped onto a bus. Thus, we also have to define a processor component (containing a memory component) to host the software architecture, and a bus component to interconnect the devices and the processor. By defining and connecting all these components, we obtain the system architecture of Figure 11.3 (due to some limitations of the OSATE/Topcased studio, bindings between software components and execution platform components are not shown; similarly, flow specifications are not shown).

The corresponding textual AADL specification is given in Figure 11.4. A system implementation definition is composed of the following clauses: `subcomponents`, `connections`, `flows`, `modes` (not present here) and `properties`.

`subcomponents`: this part describes the subcomponents contained by this implementation. Each subcomponent is defined by its instance name (for instance `the_process` or `the_sampler`), its category (process, processor, device, etc.)

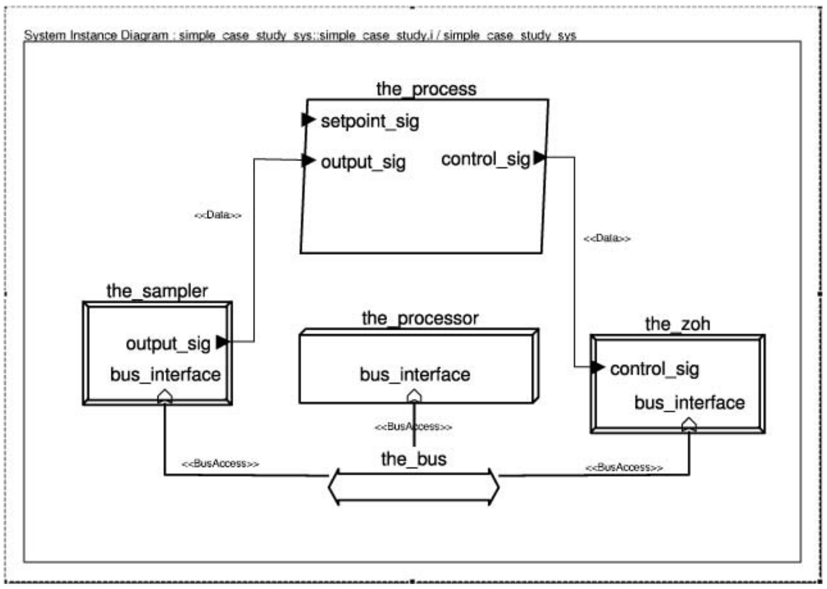


Figure 11.3. *A system architecture for the case study (graphical syntax)*

```
system implementation simple_case_study.i
  subcomponents
    the_process: process sw::proc.i;
    the_processor: processor ep::simple_proc.i;
    the_sampler: device ep::simple_sampler.i;
    the_zoh: device ep::simple_zoh.i;
    the_bus: bus ep::simple_bus.i;
  connections
    sample: data port the_sampler.output_sig -> the_process.output_sig;
    actuate: data port the_process.control_sig -> the_zoh.control_sig;
    proc_to_bus: bus access the_bus -> the_processor.bus_interface;
    sampler_to_bus: bus access the_bus -> the_sampler.bus_interface;
    zoh_to_bus: bus access the_bus -> the_zoh.bus_interface;
  flows
    control_loop: end to end flow
      the_sampler.samp -> sample -> the_process.samp_to_actu
      -> actuate -> the_zoh.actu {Expected_Latency => 15 Ms;};
  properties
    Actual_Processor_Binding => reference the_processor applies to the_process;
    Actual_Connection_Binding => reference the_bus applies to sample;
    Actual_Connection_Binding => reference the_bus applies to actuate;
end simple_case_study.i;
```

Figure 11.4. *A system architecture for the case study (textual syntax)*

and its classifier (for instance `sw::proc.i` or `ep::simple_sampler.i`). Depending on the level of detail, the classifier can be either a type or an implementation. Here, we have used implementations (for instance, we use the implementation `proc.i` of the process interface `proc`, defined in the package `sw`).

`connections`: this describes the connections between the ports of the subcomponents, or between the ports of the container and the ports of the subcomponents. Each connection can be named (for instance, `sample` or `actuate`), has a type (`data port` connection, `bus access` connection, etc.), a source and a destination. It can optionally be associated with properties (for instance, a connection latency).

`flows`: this describes the logical flows in the system. At the top-level of the specification, only end-to-end flows can be defined (an end-to-end flow is a flow fully contained in the component). An end-to-end flow is defined as a sequence of flow elements and connections. Flow elements can be flow source, flow path or flow sink. Here, we define an end-to-end flow corresponding to the input-output flow of the control loop: its source is the sampler device (flow source `samp`), it goes through the software realization of the observer and controller activities (flow path `samp_to_actu`) and its sink is the ZOH (flow sink `actu`). We associate this flow definition with a property, stating that the expected (maximal) latency of this flow is 15 ms. Each of the flow elements referenced here has to be defined in the corresponding elements. Thus, the definition of the device type `simple_sampler` must contain a flow source specification named `samp`, and the definition of the device implementation `simple_sampler.i` must contain a flow source realization named `samp`.

`properties`: this describes the properties of the system implementation. Here, we specify the binding between the software architecture components and the execution platform components.

11.6.2.2. *Thread and communication timing semantics*

This approximate version of the `proc.i` process implementation of Figure 11.2 must now be detailed. As we are designing a real-time sampled-data system, one of the crucial points is to achieve predictable flow latencies. In a multi-rate sampled control system, the sources of latency are phasing effects due to sampling, computation of data transformations within threads and devices, scheduling of concurrent threads, etc. AADL helps to overcome this difficulty by defining precise thread and communication timing semantics, and thus reasoning about the latencies. The idea is first to derive latency bounds based on high-level architecture requirements (periods, deadlines and communication patterns). Secondly, timing analysis techniques (e.g. schedulability analysis) must be used to check that a system instance meets these requirements (i.e. that the deadlines are met). The algorithm used to compute flow latencies of an AADL specification is described in [SEI 06] and implemented as a plug-in of the OSATE studio. In order to help the interested reader understand the results given below, we provide some technical explanations in section 11.6.2.3.

Let us go back to our specification. We use data port connections that provide "blackboard" semantics. This choice is natural for sampled-data systems that must be able to tolerate some sample loss (e.g. due to transient disturbance affecting the medium in the case of a distributed implementation). Thus, the correctness and timing predictability of data flows must be achieved by setting appropriate values for thread and communication timing properties. Moreover, as can be seen in Figure 11.2, there is a communication cycle that must be broken in order to obtain an implementable specification.

In AADL, periodic threads are dispatched when their associated periodic clock expires. However, data port connections between periodic threads may involve precedence constraints that can delay the release of a thread. AADL supports two kinds of data port connections: immediate and delayed. With both kinds, the values of the data consumed by a thread on its input ports are frozen when this thread is dispatched for execution. The difference lies in the date when the produced data is transmitted (by the run-time system) from the output port of the producer to the input port of the consumer. With an immediate connection, the data is transmitted as soon as the producer terminates its computation. Moreover, each time the periodic triggers of the producer and the consumer coincide, the consumer release is delayed until the completion of the producer. With a delayed connection, the data is transmitted only when the deadline of the producer is reached. The consumer is never delayed and uses the last received data.

In order to break the communication cycle of Figure 11.2, we must use a delayed connection, either for the state signal connection or for the control signal connection. The diagram of Figure 11.5 shows the scenario obtained if the state signal connection is delayed. Figure 11.6 shows the scenario obtained if the control signal connection is delayed (part of the corresponding textual AADL specification is given in Figure 11.7). As can be seen, the logical ordering of production and consumption of a data item is independent of thread response time.

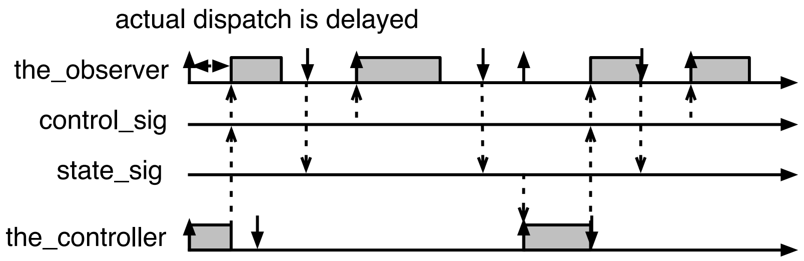


Figure 11.5. *Scenario 1: the state signal connection is delayed; the control signal connection is immediate. Solid arrows show thread dispatch (up) and deadline (down); dashed arrows show data item production and consumption*

We must now refine the device components in order to be able to compute the flow latency. In AADL, each device is implicitly associated with a device driver. The driver

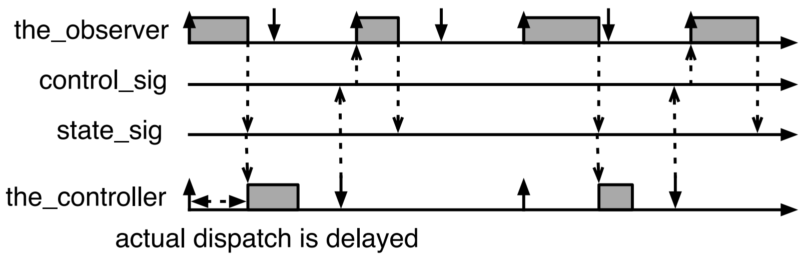


Figure 11.6. *Scenario 2: the state signal connection is immediate; the control signal connection is delayed. Solid arrows show thread dispatch (up) and deadline (down); dashed arrows show data item production and consumption*

```
process implementation proc.i
   subcomponents
     controller: thread controller.i;
     observer: thread observer.i;
   connections
     ...
     -- delayed data port connection
     control2: data port controller.control_sig ->> observer.control_sig;
     -- immediate data port connection
     state1: data port observer.state_sig -> controller.state_sig;
     ...
   flows
     samp_to_actu: flow path output_sig -> output
        -> observer.samp_to_state -> state1
        -> controller.state_to_actu -> control1
        -> control_sig;
 end proc.i;
```

Figure 11.7. *Part of the textual specification of the* `proc.i` *process implementation*

is a thread that reads the values made available on the device input data ports, and produces values on the device output data ports. The overhead induced by this thread can be part of the processor execution overhead or, if necessary, can be explicitly modeled. In the latter case, it is possible to define the behavior of the device driver thread through the following properties (non-exhaustive):

– `Device_Dispatch_Protocol`: can be set to periodic or aperiodic. Periodic means that the driver thread is periodically activated. This corresponds to the traditional polling device management policy. Aperiodic means that the thread is aperiodically activated by an external event source. This corresponds to the traditional interrupt-driven policy (notice that AADL 1.0 does not allow the behavior of the external event source to be defined);

– `Period`: if the device driver thread is periodic, this property defines its period;

– `Deadline`: the deadline of the device driver thread.

In order to achieve predictability of the control flows, we have to use a periodic sampler device. To avoid oversampling, its period must be equal to the period of the observer thread. Then, we have to choose between an immediate and a delayed data port connection between the device and the observer thread. Once again, this choice must be based on flow latency and schedulability analysis results. For the ZOH device, we will consider that its overhead is negligible and is part of the processor execution overhead.

Interacting with devices does not only involve device drivers. It also involves data port connections onto buses. The induced communication delays might be negligible or not (for our example, we will consider that they are negligible). If not, a way to take these delays into account is to set the `Latency` property of the connection. Notice that the standard properties of a bus do not allow the latency caused by data transmission to be deduced. We can set the propagation delay and the transmission time, but there is no way to compute the end-to-end transmission delay. This would require specifying the communication protocols, together with the timing properties of the transmitted streams.

Once all these properties have been set, it is possible to instantiate the system and run an end-to-end flow latency. The formulae corresponding to the different data port connection configurations are given in Table 11.1 (notations: D_x denotes the deadline of thread x with x being the first letter of the component name, and L_{con} denotes the value of the latency property of the data port connection con, where `sample` (respectively `state1`) is the data port connection between port `sampler.output_sig` (respectively `observer.state_sig`) and port `observer.output_sig` (respectively `controller.state_sig`)). See section 11.6.2.3 for some technical explanations.

`sample`	`state1`	latency
I	I	$D_c + L_{actuate}$
D	I	$\lceil \frac{D_s + L_{sample}}{T_o} \rceil T_o + D_c + L_{actuate}$
I	D	$\lceil \frac{D_s + L_{sample} + D_o}{T_c} \rceil T_c + D_c + L_{actuate}$
D	D	$\lceil \frac{D_s + L_{sample}}{T_o} \rceil T_o + \lceil \frac{D_o}{T_c} \rceil T_c + D_c + L_{actuate}$

Table 11.1. *Flow latency computation for different data port connection configurations. I stands for "immediate connection", D stands for "delayed connection"*

A tool to compute the flow latency analysis of an architecture is a great help to the architect. Of greater help would be a tool capable of performing a parametric analysis of the architecture, in order to derive constraints on property values that would ensure the respect of requirements. As our system is a very small one, we can perform this analysis "by hand". According to the requirements given in section 11.6.1, we have T_o = 10 ms, T_c = 20 ms, and we want the total latency to be less than or equal to 15 ms. As stated above, we consider that the communication latency is negligible (hence, $L_{sample} = L_{actuate} = 0$). Considering these values, only the first two architectures meet the latency bound: the third and the fourth architectures will lead to latency greater than 20 ms (because of the $\lceil \frac{x}{20} \rceil \times 20$ term), thus greater than 15 ms. After simplification of the latency equation of Table 11.1, we have the following constraints:

– $D_c \leqslant 15$ ms for the first line. We take $D_c = 15$ ms. Of course, the deadline of the observer and sampler driver threads must take precedence constraints into account. We take (for instance) $D_s = 3$ ms and $D_o = 8$ ms.

– $D_c \leqslant 5$ ms and $D_s \leqslant 10$ for the second line. We take $D_c = 5$ ms and $D_s = 10$ ms. There is a precedence constraint between the observer and the controller thread that we must take into account. Thus, we take (for instance) D_o = 2 ms.

Obviously, the flow latency analysis holds if the threads meet their deadlines. Thus, to validate the flow latency analysis, it is mandatory to perform a schedulability analysis of each eligible architecture. The next step would thus be the refinement of the specification, in order to perform this analysis. It consists of setting the appropriate properties (`Compute_Time` for threads and devices, `Scheduling_Protocol` for the processor). Subsequently, dedicated tools can extract the relevant information from the specification and assess its schedulability. To the best of our knowledge, there are currently two initiatives concerning AADL model schedulability analysis: the Cheddar tool [SIN 05], developed by the University of Brest, France, and the Furness toolset [SOK 06] developed jointly by the University of Pennsylvania, USA, and Fremont associates.

Of course, for the schedulability analysis results to be valuable, it is essential to take into account all the threads hosted on the processor. This means that each time we reuse our model inside a bigger system, a new schedulability analysis has to be performed, potentially invalidating the flow latency analysis.

11.6.2.3. *Technical overview of the flow latency analysis algorithm*

The algorithm described in Figure 11.8 is a simplified version of the algorithm used to compute flow latency. It is explained in [SEI 06] and implemented as a plug-in of the OSATE studio. It is based on a walk along the flow, starting from the flow source and ending at the flow sink.

The algorithm uses two variables:

– *tpl*: the total processing latency, used to store the total latency from the flow source to the last sampling point (or to the current flow element if no sampling point has been met yet). A sampling point corresponds to the consumption of a data item by a thread whose associated input port is connected to the producer through a delayed connection: in this case, the consumer thread dispatch, and hence the data consumption, is not synchronized with the data flow;

```
input $seq$ : sequence of flow elements
output latency of the flow
$seq \leftarrow aggregate(seq)$
$tpl \leftarrow 0$
$apl \leftarrow 0$
while $seq \neq []$ do
    $e \leftarrow head(seq)$
    if $samples(e)$ then
        $tpl \leftarrow tpl + \left\lceil \frac{apl}{e.period} \right\rceil e.period$
        $apl \leftarrow e.processing$
    else
        $apl \leftarrow apl + e.processing$
    end if
end while
return $tpl + apl$
```

Figure 11.8. *Simplified algorithm for flow latency computation*

– apl: the accumulated processing latency, used to store the latency from the last sampling point (or from the flow source if no sampling point has been met yet) to the current flow element.

The processing latency induced by a flow element is given through an AADL property: either the `Latency` property of the flow element, or the `Deadline` property of a thread or device hosting the element. To simplify the algorithm, we use a function that aggregates chains of threads with harmonic periods, connected through immediate data connections into one flow element, the processing latency of which equals the deadline of the last thread of the chain. This corresponds to the semantics of immediate data connection explained below.

As stated above, the sampling latency is induced by a periodic thread whose dispatch is independent of the arrival of data. Thus, when sampling occurs, the sampling latency is computed by rounding the accumulated processing latency up to the next multiple of the sampling period, given through the `Period` property of the sampling thread. In the algorithm, the presence of a sampling point at the beginning of a flow element is supposed to be detected by the $sample$ predicate.

11.6.2.4. *Modeling fault-tolerance mechanisms*

We will now see how AADL can be used to model fault and fault-tolerance mechanisms. First, we need to give some complementary information about error handling and operational modes in AADL.

Let us start with the run-time error handling semantics. When a thread performs a computation, its program can detect a functional error. This error can be either recoverable or unrecoverable. A recoverable error can be completely handled by the thread, for instance using an exception handling mechanism, or by the run-time system if the thread outputs data through the `Error` predefined event-data port, by using the `Raise_Error` service. In the latter case, the ongoing computation is aborted and the thread is given a chance to recover by executing a specific program set through its `Recover_Entrypoint` property. The data outputted through the `Error` event data port can be collected by another thread in charge of handling errors and mode switch at the process level, or propagated at the upper level. If we consider for instance the `controller` thread, we can imagine that the `Recover_Entrypoint` outputs on port `control_sig` the last computed value. A recoverable error can also be raised by the run-time system if the deadline associated with the ongoing computation is not met.

In the case of an unrecoverable error, the thread must output data using the `Error` event-data port. It can be set by the program executed by the thread, or by the run-time system in the case of a deadline violation of the recovery program. In order to recognize a recoverable error from an unrecoverable one, it is mandatory to access the data associated with the event.

Let us now explain the semantics of operational modes in AADL. Most AADL component implementations can embed a `modes` clause, describing the operational modes of the component and the associated mode switch logic. A mode is associated with a configuration. Thus, when a component enters a mode, its contained configuration changes. Typically these changes affect a small subset of the configuration, hence, instead of describing one configuration for each mode, the designer merely needs to label each affected object (component, connection, flow, property) by an `in modes` clause where he can list the modes where the object is present. If this clause is not given, the object is supposed to be present in all modes.

Mode switch logic is described by a transition system whose states represent modes, and whose transitions represent mode switches. Each transition must be associated with an event port: either an input event port of the container (the mode switch is triggered by an external source) or an input event port of a subcomponent (the mode switch is triggered by an internal source). System-level mode switch logic is obtained by recursively composing the modes among the component tree. When a mode switch is triggered, the system enters a "mode transition in progress" state, the duration of which is determined by different properties. It is especially possible to define the precise effects of mode switch on threads (see [SAE 04] for more details), for instance allowing the synchronization of periodic threads in order to achieve predictability of data flows.

Let us go back to our case study and assume that some run-time errors may occur: the sampler might become faulty, the bus might be subject to disturbances, the software might contain residual bugs, the load of the processor might be too high to ensure timeliness, etc. These faults can be transient or persistent. However, we want the system to be fail-safe: it must tolerate transient faults and stop in a safe state in the case of persistent ones. To achieve this requirement, we have to modify the system architecture.

We begin with the definition of the operational modes of the `proc` process embedding the software architecture. The corresponding software architecture specification is given in graphical AADL Figure 11.9. A part of the textual specification is given Figure 11.10. When loaded, it is in the mode `ok`. Each time a recoverable error is raised, the thread executes its `Recover_Entrypoint` and the error is recorded. If too many recoverable errors occur in a certain time-window, the component is considered as faulty and a mode switch has to be triggered.

A dedicated thread is used to embed this logic and appropriately trigger the mode switch: `error_handler`. Unfortunately, AADL 1.0 cannot describe the functional behavior of `error_handler`. This limitation should be overcomed in AADL 2.0, with the standardization of a "behavior" annex. The new mode is called `fallback`. In the `fallback` mode, the `controller` thread and the `observer` thread are disabled and the `fallback_control` thread is enabled. Its role is to control the plant to a safe fallback state. These changes do not affect the external behavior of the component: service continuity is ensured. Thus, it is an internal mode switch.

Having handled a recoverable error, let us now look at unrecoverable errors. They can affect either the `observer` or the `controller` thread (in mode `ok`), but they can also affect the `fallback_control` thread (in mode `fallback`). When such an error occurs, the component can not ensure service continuity. Thus, its external (observable) behavior changes and this can be handled only at the upper level. This is achieved by signaling an event using the `failure_sig` out event data port of the process, which will trigger a mode switch at the system level. Following the same idea, when the fallback policy has led the plant to a safe state, the process component stops its service and signals this using the `done` out event data port.

Once a fault-tolerant system specification has been designed, it must be validated. The error model annex of AADL [SAE 06] defines means to specify and evaluate the qualitative and quantitative safety, reliability and integrity properties of an architecture. It also enables redundancy management to be specified in an architecture. However, it is beyond the scope of this chapter to describe this annex.

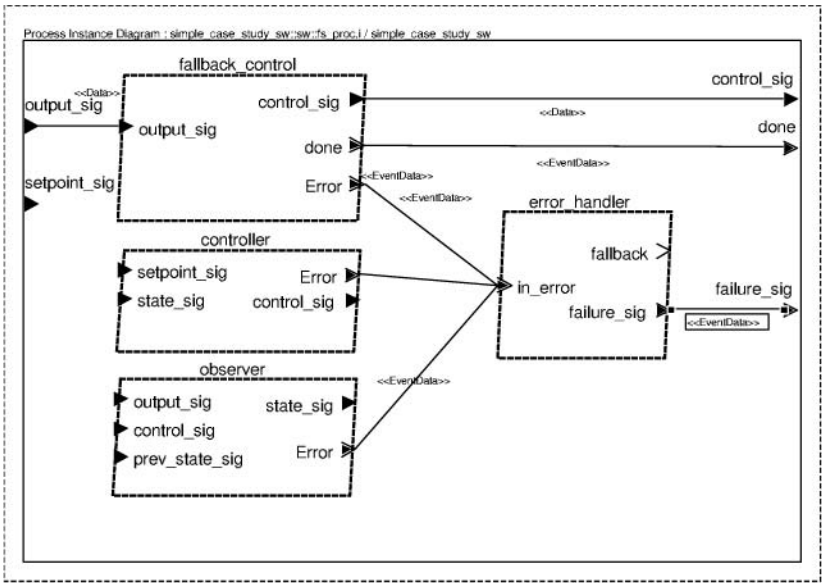


Figure 11.9. *Updated specification of software architecture for the case study. Only those connections that are not in Figure 11.2 are drawn. The* `error_handler` *thread interprets the data associated with the* `Error` *event data item to execute the appropriate reaction: triggering a local mode switch in the case of a recoverable error, or propagating the error at the upper level in the case of an unrecoverable error*

11.6.3. *Designing for reuse: package and refinement*

AADL can precisely define software and system architecture patterns (for instance the multi-rate control loop pattern used above), and offers an entry point to analysis tools capable of assessing some properties of these patterns. Considering the domain of time-critical embedded systems, we may wish to reuse such patterns, for obvious safety and economic reasons. AADL offers (limited) support to achieve this goal.

First, it supports the classical concept of a package. A package has a name and defines a namespace. It can contain component type and implementation definitions. As an illustration, our specification of the case study is structured into two packages:

– the `sw` package contains the definition of the software components;

– the `ep` package contains the definition of execution platform components.

As it can contain all the type and implementation definitions relative to a specific architectural pattern, an AADL package can be used as a "reuse unit". An example of the textual syntax for the (re)use of package is given in Figure 11.4, where the

```
process implementation proc.i
  subcomponents
    controller: thread controller.i in modes (ok);
    observer: thread observer.i in modes (ok);
    error_handler: thread error_handler.i in modes (ok, fallback);
    fallback_control: thread fallback_control.i in modes (fallback);
  connections
    ...
    output2: data port output_sig ->
      fallback_control.output_sig in modes (fallback);
    control3: data port fallback_control.control_sig ->
      control_sig in modes (fallback);
    error: event data port fallback_control.error ->
      error_handler.in_error in modes (fallback);
    failure: event data port error_handler.failure_sig ->
      failure_sig in modes (ok, fallback);
    done: event data port fallback_control.done ->
      done in modes (fallback);
  flows
    samp_to_actu: flow path output_sig -> odatacon_sample
       ...
       -> control_sig in modes (ok);
    samp_to_actu: flow path output_sig -> odatacon_sample_fb
       -> fallback_control.samp_to_actu -> odatacon_control_fb
       -> control_sig in modes (fallback);
  modes
    ok: initial mode ;
    fallback: mode ;
    ok -[error_handler.fallback]-> fallback;
end proc.i;
```

Figure 11.10. *Updated software architecture for the case study: specification of the process implementation*

operational model of the system is built from components defined in the `sw` and `ep` packages.

Secondly, AADL offers some (also classical) ways to extend and refine architectural pattern specifications. This enables a pre-defined pattern to be modified in order to fit with a specific use. Thus, AADL can define a component type (respectively implementation) as an extension of another component type (respectively implementation). The extending component inherits all the definitions of the extended component, but can also add new definitions. Moreover, inherited definitions can be refined by giving more details.

To illustrate these mechanisms, we can refactor our specification: the "failsafe" components can be defined as extensions of the "not-failsafe" versions. Examples of the corresponding textual syntax are given in Figure 11.11.

```
-- type extension: adding new features
process fs_proc extends proc
  features
    failure_sig: out event data port;
    done: out event data port;
end fs_proc;

-- implementation extension: adding subcomponents and modes
process implementation fs_proc.i extends proc.i
  subcomponents
    -- subcomponent refinements
    controller: refined to thread controller.i in modes (ok);
    observer: refined to thread observer.i in modes (ok);
    -- new subcomponents
    fallback_control: thread fallback_control in modes (fallback);
    error_handler: thread error_handler.i in modes (ok, fallback);
  connections
    -- connection refinements
    setpoint: refined to data port  in modes (ok);
    output: refined to data port  in modes (ok);
    -- new connections
    output2: data port output_sig -> fallback_control.output_sig
      in modes (fallback);
    control3: data port fallback_control.control_sig -> control_sig
      in modes (fallback);
    done1: event data port fallback_control.done -> done
      in modes (fallback);
    failure1: event data port error_handler.failure_sig -> failure_sig
      in modes(fallback);
    error1: event data port controller.Error -> error_handler.in_error
      in modes (ok);
    ...
  flows
    -- flow refinements
    samp_to_actu: refined to flow path  in modes (ok);
    samp_to_actu: flow path output_sig -> output2
      -> fallback_control.samp_to_actu
      -> control3 -> control_sig in modes (fallback);
  modes
    ok: initial mode ;
    fallback: mode ;
    ok -[ error_handler.fallback ]-> fallback;
end fs_proc.i;
```

Figure 11.11. *Using the architecture extension capabilities of AADL*

This extension mechanism is mainly a syntax-level construct to help a designer specify a system family or an assembly of pre-specified patterns. On the one hand, it is clearly useful for the modeling step. On the other hand, it does not provide many guaranteed-by-construction properties: most of the analysis still has to be run on the final operational system specification that will be used to obtain the implementation. Let us underline that this limitation is not specific to AADL: there is currently no mature composability theory in the field of real-time systems (but promising work is on the way, for instance the BIP framework [BAS 06] developed at the Verimag laboratory, or the work on contract-based scheduling carried out through the FRESCOR (`http://www.frescor.org`) European IST project).

A second limitation concerns the specification of "how should a reusable part be used", the "component contract". This limitation could be fixed by including in the future developments of the standard, some kind of support for component-level contracts. Even though the theoretical limitations presented above still apply to real-time properties, there are some interesting proposals for other properties (see, for instance, [JÉZ 05]).

11.7. Conclusion

The primary objective of this chapter was to introduce, without being exhaustive, the capabilities provided by the ADLs for the real-time domain. Consequently, this presentation must be considered as preliminary to a deeper study. Currently, by providing extensive support both in terms of architectural modeling and analysis tools, these languages can play an important role in the software process associated with increasingly complex real-time systems. Not only can the use of an architectural description (with an ADL) not be ignored for their architectural design, but it must also become the backbone of the subsequent development steps if the heterogenity of the different models used to achieve each of these steps is to be mastered.

We begin by addressing the notion of architecture with its issues, and introducing the key points of the general ADLs. Even though their basic concepts and their corresponding abstractions must obviously be kept, we have highlighted those specificities that are essential to the real-time domain and that require some extensions or adaptations. Among the multiple proposals concerning ADLs, the AADL language has now emerged thanks to its SAE standard. Thus, it seemed natural for us to favor it for the rest of our presentation. We have chosen a simple example to present the main features of this language, some of its expressiveness capabilities as well as the way it guides design and validation. The present core language can now be considered as a stable answer to the description requirements of embedded real-time applications. There are, however, still some deficiencies or even ambiguities that it would be interesting to correct, for instance:

– to complement and expand the behavioral modeling of the execution platform components as well as of the system environment;

– to enhance the component reusability by some means, like a component contract;

– to improve the expression of time at the operational mode level (enabling for example the specification of a deadline on a mode switch, or a watchdog on an event occurrence).

At the time of writing this chapter, the specification of the AADL 2.0 standard was underway. Let us hope that it will offer some solutions to the problems described above.

Let us look at the verification techniques and tools put forward in the other chapters of this book. Undoubtedly, they are able to solve a number of V&V requirements for real-time applications. In particular, for timed automata (see Chapter 4) as well as for timed Petri nets (see Chapter 1), the corresponding dense time model checking tools are powerful and mature enough to be used in a model-driven development process. Nowadays, however, bridging the gap between design model and validation model is still a difficult problem. Some reasons for this are as follows:

– the architecture description languages are semi-formal ones (in the sense that a formal semantics is usually defined only for a subset of the language). Consequently, the "design -> validation" transformation techniques need to include some subjective interpretation. It is not easy to establish what kinds of properties are preserved by such a transformation and what the scope of the analysis results is;

– in addition, how to interpret the analysis results, i.e. the definition of "validation -> design" transformation techniques, is a problem (certainly more difficult than the previous one). How do we map onto an architecture design the execution trace produced by a model checker for proving that a property is not met? How do we adequately refactor this architecture so as to solve this problem?

– as UML shows, the success of a design language (or notation) relies in part on its flexibility and its capability to be extended and/or specialized. Such extensions are often introduced with the aim of using some specific verification techniques. The syntactic compatibility is fairly preserved but it is a pity that, at the semantic level, such extensions are thought about independently. It is then very difficult to use many of them jointly while guaranteeing that the global semantics remain consistent.

To finish, we wish to point out that real-time software architecture design is now receiving a great deal of attention from industry, which aims to make its large-scale developments safer, as well as from researchers who want to transfer their theoretical knowledge. Thus, AADL is at the core of a number of studies, as shown by the current AADL user list (see the AADL official webpage `http://www.aadl.info/`). All these activities concerning the language should give responses to the previous points in the relatively near future.

11.8. Bibliography

[BAR 93] BARBACCI M.*et al.*, "Durra: A Structure Description Language for Developing Distributed Applications", *Software Engineering Journal*, vol. 8, num. 2, p. 83–94, 1993.

[BAS 06] BASU A., BOZGA M., SIFAKIS J., "Modeling Heterogeneous Real-time Systems in BIP", *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, IEEE Computer Society, 2006.

[BIN 96] BINNS P., ENGLEHART M., JACKSON M., VESTAL S., "Domain-Specific Software Architectures for Guidance, Navigation and Control", *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, num. 2, 1996.

[DEB 05] DEBRUYNE V., SIMONOT-LION F., TRINQUET Y., "EAST-ADL: An Architecture Description Language. Validation and Verification Aspects", *Architecture Description Languages*, vol. 176 of *IFIP*, Springer, p. 181–196, 2005.

[DUR 98] DURAND E., "Description et vérification d'architecture temps réel: CLARA et les réseaux de Petri temporels", PhD thesis, University of Nantes, Nantes, 1998.

[FAR 03] FARINES J. *et al.*, "The COTRE Project: Vigorous Software Development for Real-time Systems in Avionics", *27th IFAC/IFIP/IEEE Workshop on Real-Time Programming*, 2003.

[FAU 02] FAUCOU S., "Description et construction d'architectures opérationnelles validées temporellement", PhD thesis, University of Nantes, Nantes, 2002.

[FAU 05] FAUCOU S., DÉPLANCHE A., TRINQUET Y., "An ADL-centric Approach for the Formal Design of Real-time Systems", *Architecture Description Languages*, vol. 176 of *IFIP*, Springer, p. 67–82, 2005.

[FEI 04] FEILER P., GLUCH D., HUDAK J., LEWIS B., "Embedded System Architecture Analysis Using SAE AADL", Report num. CMU/SEI-2004-TN-005, Software Engineering Institute, 2004.

[FEI 06] FEILER P., GLUCH D., HUDAK J., "The Architecture Analysis & Design Language: An Introduction", Report num. CMU/SEI-2006-TN-11, Software Engineering Institute, 2006.

[GAR 93] GARLAN D., SHAW M., "An Introduction to Software Architecture", AMBRIOLA V., TORTORA G., Eds., *Advances in Software Engineering and Knowledge Engineering*, Singapore, World Scientific Publishing Company, p. 1–39, 1993.

[GAR 97] GARLAN D., MONROE R., WILE D., "ACME: An Architecture Description Interchange Language", *CASCON'97*, p. 169–183, 1997.

[HAN 96] HANSSON H., LAWSON H., STRÖMBERG M., "BASEMENT: a Distributed Real-Time Architecture for Vehicle Applications", *Real-Time Systems*, vol. 11, num. 3, p. 223–244, Kluwer, 1996.

[HUD 05] HUDAK J., FEILER P., "Developing AADL Models for Control Systems: A Practitioner's Guide", Report num. CMU/SEI-2005-TR-022, Software Engineering Institute, 2005.

[JÉZ 05] JÉZÉQUEL J.-M., "Real Time Components and Contracts", *Model Driven Engineering for Distributed Real-time Embedded Systems*, Hermes Science, 2005.

[KRA 89] KRAMER J., MAGEE J., SLOMAN M., "Constructing Distributed Systems in Conic", *IEEE Transactions on Software Engineering*, vol. 15, num. 6, p. 663–675, 1989.

[KRU 95] KRUCHTEN P., "Architectural Blueprints: The '4+1' view model of software architecture", *IEEE Software*, vol. 12, num. 6, 1995.

[MAG 95] MAGEE J., DULAY N., EISENBACH S., KRAMER J., "Specifying Distributed Software Architectures", *European Software Engineering Conference*, vol. 989 of *LNCS*, Springer, p. 137–153, 1995.

[MAG 96] MAGEE J., KRAMER J., "Dynamic Structure in Software Architectures", *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, p. 3–14, 1996.

[MCC 98] MCCONNELL D., LEWIS B., GRAY L., "Reengineering a Single Threaded Embedded Missile Application onto a Parallel Processing Platform Using MetaH", *Real-Time Systems*, vol. 14, num. 1, p. 7–20, Kluwer, 1998.

[MED 00] MEDVIDOVIC N., TAYLOR R., "A Classification and Comparison Framework for Software Architecture Description Language", *IEEE Transactions on Software Engineering*, vol. 26, num. 1, p. 70–93, 2000.

[OMG 04] OMG – Object Management Group, "UML 2.0 Superstructure Specification", 2004.

[PRI 86] PRIETO-DIAZ R., NEIGHBORS J., "Module Interconnection Languages", *The Journal of Systems and Software*, vol. 6, num. 4, p. 307–334, 1986.

[REM 76] DE REMER F., KRON H., "Programming-in-the-large versus Programming-in-the-small", *IEEE Transactions on Software Engineering*, vol. 2, num. 2, 1976.

[SAE 04] SAE – Society of Automotive Engineers, "SAE Standard AS5506: Architecture Analysis and Design Language (AADL)", 2004.

[SAE 06] SAE – Society of Automotive Engineers, "SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex E: Error Model Annex", 2006.

[SEI 06] SEI – Software Engineering Institute, "An Extensible Open Source AADL Tool Environment (OSATE) – Release 1.3.0", 2006.

[SIN 05] SINGHOFF F., LEGRAND J., NANA L., MARCÉ L., "Scheduling and Memory Requirements Analysis with AADL", *ACM SIGAda International Conference*, 2005.

[SOK 06] SOKOLSKY O., LEE I., CLARKE D., "Schedulability Analysis of AADL Models", *International Parallel and Distributed Processing Symposium*, 2006.

[TÖR 98] TÖRNGREN M., "Fundamentals of Implementing Real-Time Control Applications in Distributed Computer Systems", *Real-Time Systems*, vol. 14, num. 3, p. 219–250, Springer, 1998.

[VES 98] VESTAL S., "MetaH User's Manual – Version 1.27", Honeywell Technology Center, Minneapolis, MN, USA, 1998.

[WAL 03] WALL A., "Architectural Modeling and Analysis of Complex Real-time Systems", PhD thesis, Department of Computer Science and Engineering, Mälardalen University, 2003.

[ZEL 96] ZELESNIK G., "The UniCon Language Reference Manual", School of Computer Science, Carnegie Mellon University, 1996.

This page intentionally left blank

List of Authors

Bernard Berthomieu
LAAS – CNRS
Toulouse
France

Patricia Bouyer
Laboratoire Spécification et Vérification
École Normale Supérieure de Cachan
France

Marius Bozga
VERIMAG
Gières
France

Paul Caspi
VERIMAG
Gières
France

Camille Constant
IRISA – INRIA Rennes
France

Anne-Marie Déplanche
Institut de Recherche en Communications et Cybernétique de Nantes
France

Sébastien Faucou
Institut de Recherche en Communications et Cybernétique de Nantes
France

Susanne Graf
VERIMAG
Gières
France

Serge Haddad
LAMSADE
Université Paris-Dauphine
Paris
France

Grégoire Hamon
Chalmers University of Technology
Institutionen för Datavetenskap
Gothenburg
Sweden

Thierry Jéron
IRISA – INRIA Rennes
France

Marta Kwiatkowska
Oxford University Computing Laboratory
Oxford
United Kingdom

François Laroussinie
Laboratoire Spécification et Vérification
École Normale Supérieure de Cachan
France

Hervé Marchand
IRISA – INRIA Rennes
France

Radu Mateescu
INRIA Rhône-Alpes
Montbonnot
France

Stephan Merz
LORIA – INRIA Nancy Grand Est
Vandoeuvre-lès-Nancy
France

Patrice Moreaux
LISTIC
University of Savoy
Annecy
France

Laurent Mounier
VERIMAG
Gières
France

Nicolas Navet
LORIA – INRIA Nancy Grand Est
Vandoeuvre-lès-Nancy
France

Gethin Norman
Oxford University Computing Laboratory
Oxford
United Kingdom

Iulian Ober
Laboratoire GRIMM
IUT de Blagnac
France

David Parker
Oxford University Computing Laboratory
Oxford
United Kingdom

Florent Peres
LAAS – CNRS
Toulouse
France

Marc Pouzet
Laboratoire de Recherche en Informatique
Paris-Sud University
Orsay
France

Pascal Raymond
VERIMAG
Gières
France

Vlad Rusu
IRISA – INRIA Rennes
France

Jeremy Sproston
Dipartimento di Informatica
Università di Torino
Turin
Italy

François Vernadat
LAAS – CNRS
Toulouse
France

Index