

## **INFORMATION TO USERS**

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning  
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA  
800-521-0600

**UMI<sup>®</sup>**



**University of Alberta**

**SIMULATION AND PROBABILISTIC VALIDATION  
OF COMMUNICATION PROTOCOLS**

by

**Theodore Ono-Tesfaye**



**A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctor of Philosophy.**

**Department of Computing Science**

**Edmonton, Alberta**

**Spring 2000**



**National Library  
of Canada**

**Acquisitions and  
Bibliographic Services**

**395 Wellington Street  
Ottawa ON K1A 0N4  
Canada**

**Bibliothèque nationale  
du Canada**

**Acquisitions et  
services bibliographiques**

**395, rue Wellington  
Ottawa ON K1A 0N4  
Canada**

*Your file* *Voire référence*

*Our file* *Notre référence*

**The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.**

**The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.**

**L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.**

**L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.**

**0-612-60010-6**

**Canada**

**University of Alberta**

**Library Release Form**

**Name of Author:** Theodore Ono-Tesfaye

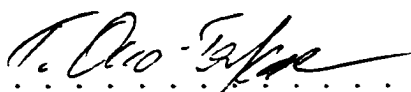
**Title of Thesis:** Simulation and Probabilistic Validation of Communication Protocols

**Degree:** Doctor of Philosophy

**Year this Degree Granted:** 2000

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.


  
.....  
Theodore Ono-Tesfaye  
#2005, 8210 111th Street  
Edmonton, AB  
Canada, T6G2C7

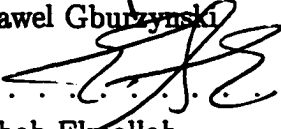
**Date:** April 17, 2000

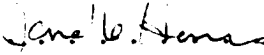
**University of Alberta**

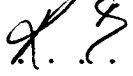
**Faculty of Graduate Studies and Research**

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Simulation and Probabilistic Validation of Communication Protocols** submitted by Theodore Ono-Tesfaye in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**


  
.....  
Pawel Gburzynski

  
.....  
Ehab Elmallah

  
.....  
Janelle Harms

  
.....  
Dipak Ghosal

  
.....  
Jack Tuszynski

  
.....  
Ioanis Nikolaidis

**Date:** *April 17, 2000*

# Abstract

*Protocol Validation* is the process of showing that a (suitably specified) protocol has certain (suitably specified) properties, for example, that it is free from deadlocks. In the past, the acceptance of automated tools for the validation of protocols has been hindered by three main problems: (1) most validation tools are lacking in the level of realism they support: they are either unable to deal with the timing aspects (e.g., delays between events) or the probabilistic aspects (e.g., probability of loss) of protocols; (2) unlike most performance evaluation tools which are based on common programming languages like C or C++, validation tools often employ special-purpose languages (e.g., PROMELA in SPIN, KRONOS); and (3) protocol validation is computationally difficult due to the *state space explosion problem*—the fact that the state space of a protocol is exponential in the number of variables and processes. Validation algorithms published in the literature often have unrealistic resource requirements.

In this thesis, we propose a validation tool that addresses these problems. Our tool is based on the well-known *discrete event simulation* (DES) paradigm and checks whether a property expressed in a simple linear-time logic (*event logic*, (EL)) is satisfied by a protocol. The tool is based on DES and thus supports time and probabilities (problem 1 above). Problem (2) is addressed by implementing the tool as a C++ library. To better deal with the state explosion problem, we propose using state transition probabilities as a heuristic to guide the state space search, and present some memory-efficient algorithms for the checking of EL formulas. An added advantage of our approach is that the same protocol specification can be used for both performance evaluation by simulation and for validation. A number of experiments described in the thesis demonstrate the feasibility of our validation tool.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Survey of Previous Work</b>	<b>6</b>
2.1	Overview . . . . .	6
2.2	Non-Probabilistic Validation . . . . .	7
2.2.1	Validation of Untimed Models . . . . .	7
2.2.2	Validation of Timed Models . . . . .	19
2.3	Probabilistic Validation . . . . .	25
2.3.1	Specification of Models . . . . .	25
2.3.2	Specification of Properties . . . . .	29
2.3.3	Algorithms and Tools . . . . .	30
2.4	Summary . . . . .	31
<b>3</b>	<b>A Probabilistic Protocol Validation Model</b>	<b>32</b>
3.1	Overview . . . . .	32
3.2	Low-level Model: Timed Probabilistic Transition System . . . . .	33
3.2.1	Timed Probabilistic Transition System . . . . .	33
3.2.2	Probabilities . . . . .	34
3.2.3	Event Logic . . . . .	35
3.3	High-Level Model: Probabilistic Protocols . . . . .	37
3.3.1	Events and Processes . . . . .	38
3.3.2	Probabilistic Protocols . . . . .	39
3.3.3	Model Restrictions . . . . .	41
3.4	Observers . . . . .	43
3.5	Comparison With Other Models . . . . .	45



3.6	Summary . . . . .	48
<b>4</b>	<b>Algorithms</b>	<b>49</b>
4.1	Overview . . . . .	49
4.2	State Space Search Algorithm without Model-Checking . . . . .	49
4.2.1	Unlimited Search . . . . .	50
4.2.2	Truncated Search . . . . .	51
4.3	State Space Search with Model-checking of Event Logic Formulas . .	52
4.3.1	Inner Depth-First Search . . . . .	53
4.3.2	Outer Depth-First Search . . . . .	55
4.3.3	Example . . . . .	57
4.3.4	Proof of Model-Checking Algorithm . . . . .	59
4.3.5	Implementation . . . . .	64
4.3.6	Optimized Handling of Cycles . . . . .	65
4.3.7	Combining Model-Checking with Truncated Searches . . . . .	66
4.4	Approximate Algorithm: Time Intervals . . . . .	69
4.5	Summary . . . . .	76
<b>5</b>	<b>The C++ Library</b>	<b>78</b>
5.1	Overview . . . . .	78
5.2	User Classes . . . . .	79
5.2.1	val_time . . . . .	79
5.2.2	val_event . . . . .	80
5.2.3	Kernel Classes . . . . .	81
5.2.4	val_process . . . . .	83
5.2.5	Random Variable Support . . . . .	85
5.3	User Functions . . . . .	86
5.3.1	The root Function . . . . .	86
5.3.2	The report Function . . . . .	86
5.4	The EL Formula Compiler e12cc . . . . .	86
5.5	Running the Model . . . . .	87
5.6	Summary . . . . .	89

<b>6</b>	<b>Examples</b>	<b>90</b>
6.1	Overview . . . . .	90
6.2	Multiprocessor System . . . . .	91
6.3	Broadcast Channel Protocols . . . . .	97
6.3.1	Collision Avoidance Protocol . . . . .	97
6.3.2	CSMA/CD . . . . .	100
6.3.3	GARP Multicast Registration Protocol (GMRP) . . . . .	105
6.4	Steam Boiler Control Program . . . . .	110
6.4.1	Description of Model . . . . .	110
6.4.2	Experimental Results . . . . .	118
6.4.3	Comparison with other Tools . . . . .	120
6.5	Queueing System . . . . .	121
6.6	Multimedia System . . . . .	124
6.7	Alternating Bit Protocol . . . . .	127
6.8	Sliding Window Protocol . . . . .	129
6.9	Summary . . . . .	135
<b>7</b>	<b>Conclusion and Future Research</b>	<b>137</b>
	<b>Bibliography</b>	<b>139</b>

# Chapter 1

## Introduction

Our lives are increasingly dependent on all kinds of hardware and software systems, and there is great interest in ensuring that these systems operate flawlessly. *Validation* is the process of showing that a (suitably specified) system has certain (suitably specified) properties, for example, that it is free from deadlocks. In this thesis, we also use the terms *verification* and *model-checking* with the same meaning. Our primary application domain is the validation of *communication protocols*.

Due to the distributed nature of communication protocols and their resulting complexity, the use of automated tools for the task of validation is highly desirable. In order to use automated tools, it is necessary for protocol designers to suitably model their protocol in a form that is accessible to a validation tool. A quick survey of popular communication conferences (INFOCOM, GLOBECOM) indicates that modelling is already widely performed — but not for the purpose of validation: protocols are routinely modeled to obtain performance measures. The tool of choice for performance evaluation is *discrete event simulation* (DES).

In contrast, and despite the existence of tools and some recent progress [37, 66, 55, 85], the *use* of automated tools for the validation of protocols is much less common. Instead of using automated tools, protocol designers frequently validate their protocols using methods such as rigorous testing, inspection, and peer review. While these methods have yielded remarkably stable and robust protocols in the past (such as Ethernet and IP), they are time-consuming, expensive and require a great deal of expertise on the part of the protocol designer. The use of automated validation tools for protocol validation software development promises to be more economical

and efficient.

There are several reasons why the use of validation tools has lagged behind the use of performance evaluation tools like DES:

- Most validation tools support a limited level of realism. All validation tools can validate properties related to *qualitative* time, i.e., properties related to the ordering of events. To realistically capture the properties of a protocol, however, it is often desirable to be able to specify its timing aspects using *quantitative* time, i.e., references to time that include the delays between events expressed in some measuring unit. For example it is not good enough for the alarm system in a nuclear power plant to start “eventually” — one would rather expect it to function within an absolute time period of  $x$  seconds following a malfunction.
- In addition to time, it is often useful to model the probabilistic aspects of a protocol. For example, one may want to specify the probability of a packet loss by a lower-layer protocol or the failure probability of a hardware component. Note that the most widely known validation tool, SPIN [54, 55], supports qualitative time, but neither quantitative time nor probabilities.
- Unlike most DES tools (OPNET, SMURPH [43]) that are based on common programming languages, like C or C++, validation tools often employ special-purpose languages (e.g., PROMELA in SPIN, KRONOS [37]). The use of these languages is rooted in the desire to achieve a level of mathematical precision that cannot be easily represented by using C or C++.
- Protocol validation is computationally difficult due to the *state space explosion problem*—the fact that the state space of a protocol is exponential in the number of variables and processes. Validation algorithms published in the literature often have unrealistic resource requirements. For example, the probabilistic validation algorithm proposed in [16] requires the manipulation of a Markov chain representing the entire state space of the protocol.

One of the contributions of this thesis is to address these problems by extending the DES paradigm to perform the functions of a protocol validator. This approach has several advantages: protocols can be modeled realistically, including all timing and

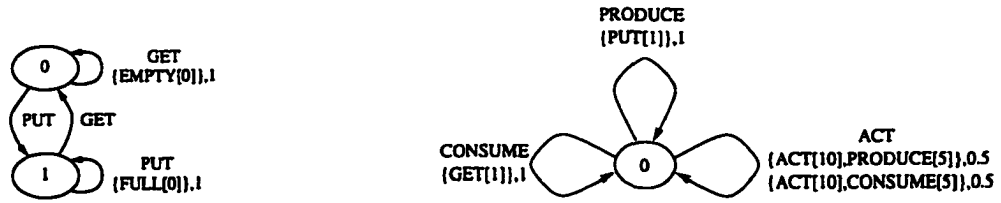


Figure 1.1: Buffer Process (left) and Producer/Consumer Process (right)

probabilistic aspects; a single model can be used both for the performance evaluation via simulation and the validation of protocols.

Of course, discrete event simulators have been used for validation (debugging) of protocols in the past, but this has generally happened in an ad-hoc fashion by running a large number of simulations with different random seeds, initial states, etc. Our approach is more systematic and combines the DES paradigm with methods already used by validation tools. The task is simplified by the inherent similarity between DES and protocol validation: in both cases, it is necessary to first specify the protocol (typically as a set of processes) and then execute it in some fashion. Throughout our work, the emphasis is on practical utility.

Our high-level protocol model is that of finite state machines (*processes*) that exchange messages. Figure 1.1 shows a simple protocol consisting of a data buffer process and a producer/consumer process. The semantics of the protocol is given in terms of our low-level model which is a transition system whose transitions are labeled with times, probabilities and event types. Figure 1.2 shows the low-level model of the protocol in 1.1.

We use state transition probabilities for two purposes: first, as a heuristic to guide the state space search to more probable regions of the state space. This is used in those cases where, due to resource limitations, an exhaustive state space search is not feasible. The second application of transition probabilities is to express and verify properties such as “the probability of observing a buffer overflow within the first 1000 time units is 0.01”. An algorithm that calculates the probability that such properties will be satisfied with respect to a model is a contribution of this thesis. In our model, transition probabilities need not be given explicitly, but can also be given *implicitly*. For example, in figure 1.3, from the system state  $g$  (at time 0), there are two pending events (transitions): event A must take place between  $t_1 = 0.5$  and

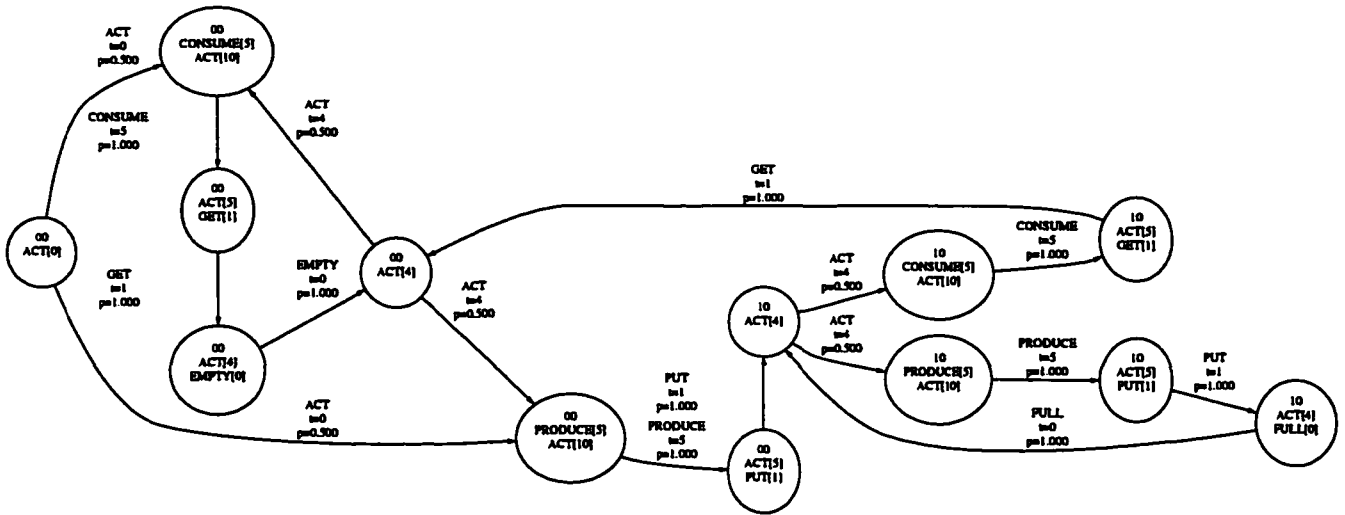


Figure 1.2: TPTS for Data Buffer Protocol



Figure 1.3: Overlapping Timed Events (left) and Global States

$t_2 = 1.5$ , and event B must take place between  $t_3 = 1.25$  and  $t_4 = 2.5$ . There are two possible interleavings of the events A and B: “A then B” or “B then A”. Assuming that A and B are equally likely to occur at any point during their respective intervals, then the relative probabilities of the transition sequences “A then B” and “B then A” are implicitly given by the areas in figure 1.4.

The thesis is organized as follows. Chapter 2 is a survey of previous work in the area. Chapter 3 describes our protocol specification method and a logic for specifying properties of protocols. Our verification algorithms are presented in chapter 4. The results of this chapter are the foundations for our prototype validation tool which is described in chapter 5. Examples and experimental results from a variety of applications are shown in chapter 6. Finally, our conclusions and some directions for future research are in chapter 7.

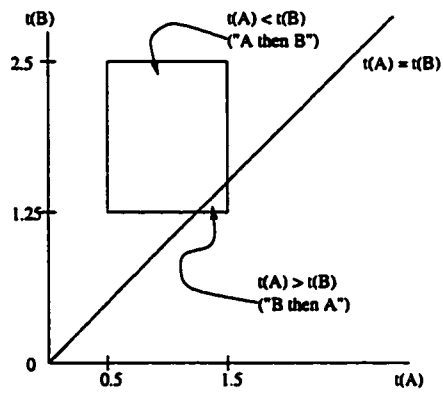


Figure 1.4: Relative Transition Sequence Probabilities

# Chapter 2

## Survey of Previous Work

### 2.1 Overview

This chapter is a survey of automatic validation methods. The application domain we will focus on is the validation of communication protocols. We classify the methods in three types: the validation of *untimed*, *timed* and *probabilistic* models. Probabilistic systems, in turn, can be subdivided into timed and untimed models.

Many interesting properties of systems can be expressed based solely on the ordering of events, without using quantitative time — this is the domain of untimed validation methods, which are reviewed in section 2.2.1. Other applications, like the validation of real-time and multimedia systems, require that properties related to absolute time (e.g., “the nuclear reactor will shut down within 10 minutes of event A”) be checked, and there has therefore been a significant amount of research in the development of timed validation methods. These methods are reviewed in section 2.2.2.

The motivation for probabilistic validation methods is based on two observations:

- Due to the state space explosion problem, a generally useful validation method remains elusive. Indeed, a validation algorithm is considered to be “efficient” if it is polynomial in the size of the state space [16], so it is unlikely that such a solution will ever be found. Therefore, it is usually sufficient to show that a protocol’s failure probability is smaller than that of the underlying hardware; the general idea is to only analyze states whose probability of occurring is greater than some very small threshold, e.g.  $10^{-12}$ .
- Most real-life systems exhibit probabilistic behaviours (e.g., a communication



channel may introduce errors) which need to be modeled.

Approaches to probabilistic validation are surveyed in section 2.3.

All validation methods have to address three fundamental issues: (i) how to specify the system, (ii) how to specify its desired properties, and (iii) how to check that the system satisfies these properties. For this reason, the sections in this chapter have three parts dealing with each issue in turn.

## 2.2 Non-Probabilistic Validation

### 2.2.1 Validation of Untimed Models

In this section, we survey validation methods that are restricted to untimed systems, i.e., systems that do not include an explicit notion of time. Thus, they can prove properties related to the temporal *ordering* of events (e.g., “if A happens before B, the system will deadlock”), but not properties related to the absolute delay between events. Such untimed validation methods are now widely used in hardware applications like the design of integrated circuits. Excellent overviews of untimed validation methods are [54], which focuses on communication protocols, and [60], which focuses on hardware applications.

In the first two parts of this section, we examine methodologies for specifying such untimed systems and their properties, respectively. The last part of this section is an overview of untimed validation algorithms and tools that check properties against system specifications.

#### Specification of Systems

Two popular ways of specifying untimed systems of concurrent processes are *finite state machines* and *process algebras*.

*Finite State Machines.* Early approaches to system specification with finite state machines (FSMs) or automata are described by Bochmann and Sunshine in [21]. A finite state machine consists of a set of states  $S$  and a set  $T \subseteq S \times S$  of transitions between states. Often, the transitions are labeled with one or more symbols indicating the event(s) that either cause the machine to change from one state to another or result from such a change. We will frequently refer to FSMs as *labeled transition*



Figure 2.1: Two States of a simple Petri Net

*systems* (LTS). Specifying nontrivial systems as single finite state machines is rather inconvenient, and therefore, many extensions and variations of the basic FSM model are used for validation purposes: *Extended Finite State Machines (EFSMs)* allow the declaration and manipulation of state variables. Using *Communicating Finite State Machines (CFSMs)* [23], a protocol can be modeled as a system of communicating sub-processes, thereby allowing modular design.

A different kind of state machine are *Petri nets* (see, e.g., [68] for an overview), usually depicted as a graph. In the most basic type of Petri net ([68] describes some variants), the nodes of the net are partitioned into two subsets - places and transitions. Edges can only lead from places to transitions or vice versa. A place can contain zero or more *tokens*; a transition is *enabled* if all places with edges leading to it (*incoming places*) have at least one token. An enabled transition can *fire*, which results in one token being removed from all incoming places, and one token being added to all places that the transition leads to (*its output places*). A state of the Petri net is defined by the number of tokens in each place and is also called a *marking*. Figure 2.1 shows a simple Petri net with 4 places (shown as circles) and 3 transitions (shown as lines). The initial marking (tokens are shown as black dots) is on the left. Transition a can fire in this state, leading to the marking on the right, in which transitions b and c can both fire.

Petri nets are popular because they can often model systems more compactly and intuitively than simpler state transition systems like finite state machines. However, the semantics of a Petri net is given in terms of a simple labeled transition system, i.e., a directed graph whose nodes are the states of the Petri net and whose edges are its transitions. From each state, there is an LTS transition to each state reachable by the firing of a transition in the Petri net. The LTS obtained in this fashion is often called the *reachability graph* of a Petri net. Figure 2.2 shows the reachability graph

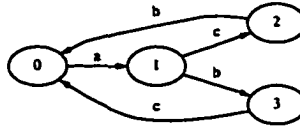


Figure 2.2: Reachability Graph of the Petri Net in Fig. 2.1

of the Petri net in figure 2.1. Note that in general, the reachability graph of a Petri net is not necessarily finite: for example, there can be places in which the number of tokens increases without ever decreasing.

Finite state machines can also be specified by textual means, of course. Several languages for describing systems of CFSMs have been proposed, including PROMELA (used in the tool SPIN). The Pascal-based language *Estelle* and the graphical language *SDL* are standardized formal description techniques (FDTs). They are primarily designed for the unambiguous definition of communication protocols in standard documents, and not for automated validation. However, both Estelle and SDL specifications can be readily converted into FSM-type representations [66] that are accessible to FSM validation methods.

*Process Algebras.* Process algebras are special-purpose languages for describing and reasoning about communicating processes. Examples of process algebras are Milner's Calculus of Communicating Systems (CCS) [67] and Hoare's Communicating Sequential Processes (CSP) [53]. A fundamental introduction to process algebras in general is in [48]. LOTOS [22] is a standardized protocol description language based on CCS and CSP. [69] describes a general for a method for converting a LOTOS specification into an FSM-type representation. We describe some of the features of process algebras using a CSP-like process algebra as an example. Like any language, process algebras have two components, *syntax* and *semantics*. A simple process specified in CSP syntax is is

$$A = a.A$$

The semantics of this specification is process A takes part in event a and then behaves like process A. In other words, all process A does is repeatedly engage in event a. A slightly more complicated specification is

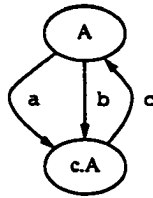


Figure 2.3: Derivation Graph of a CCS process

$$A = a.c.A + b.c.A$$

This process takes part in event  $a$  or  $b$ , then in event  $c$ , and starts all over again. Communication in CSP is *asynchronous*, which means that a process can engage in an event  $a$  even if no other process is simultaneously prepared to engage in the event  $a$ . If several processes are prepared to engage in the same event, they do so simultaneously. Formally, the semantics of CSP processes is given in terms of axioms and rules that associate a labeled transition system (LTS) with every CSP expression. The states of this LTS are CSP expressions, and the transitions are labeled with CSP events. In the example specification above, the expression  $a.c.A + b.c.A$  can change to  $c.A$  if the process engages in event  $a$  or  $b$ . Thus, there are transitions  $A \xrightarrow{a} c.A$  and  $A \xrightarrow{b} c.A$  in the LTS. The LTS obtained in this fashion is called the *derivation graph* of the process. The derivation graph for the specification above is in figure 2.3.

### Specification of Properties

Once a system has been specified, it is necessary to somehow express its desired properties. We review three methods of specifying properties: automata-based methods, algebraic methods and temporal logic.

*Finite State Machines.* Automata are finite-state machines that accept or reject words over some alphabet, depending on whether or not the machine is in one of a set of designated *accepting* states after the word has been read. Such an automaton specifies a set of accepted words which is called the *language* of the automaton, and since each word can be viewed as a finite computation path, the automaton specifies a set of accepted finite computation paths.

In general, however, the computation paths of a concurrent system are infinite, and we therefore need automata that can specify sets of desired infinite paths. *Büchi automata* are such automata that accept words of infinite length. Like standard

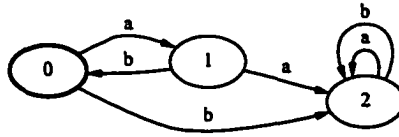


Figure 2.4: Buchi Automaton

automata, a Büchi automaton has a set of designated accepting states. The difference between a Büchi automaton and a standard automaton is in the way that accepted words are defined: the Büchi automaton accepts an infinite word if accepting states are visited infinitely often while reading the infinite path. Figure 2.4 shows a Büchi automaton over the alphabet  $\{a, b\}$  that accepts exactly those infinite words have the form  $abababababab\dots$ . In the figure, state 0 is both the start state and the accepting state.

*Process Algebras.* In a process algebra like CSP, actions among the processes that make up a system can be hidden to the outside world — in other words, they can be *internalized*. As an example, consider the CSP process

$$A = a.c.A + b.c.A$$

described above. It can be composed with another CSP process

$$C = c.C$$

to yield a system  $D$  by writing  $D = A\{c\}|C$  which forces process  $A$  and  $C$  so synchronize on action  $c$ . By further writing  $D = A\{c\}|C\backslash c$ , we specify that action  $c$  is hidden and thus not visible to an outside observer.  $D$  behaves like the process specified by

$$D = a.D + b.D$$

i.e., at any time,  $D$  can engage in events  $a$  or  $b$ . Using these concepts of internal (hidden) and external behaviours, it is possible to define a system of communicating processes at different levels of abstraction. The specification of  $D$  as  $a.D + b.D$  is more abstract than  $D = A\{c\}|C\backslash c$  because it abstracts away from the internal action. The verification problem, then, is equivalent to checking if the more detailed process



Figure 2.5: Simple Labelled Transition System (LTS)

specification (the implementation) has the same external behaviour as the simpler process specification.

*Temporal Logics.* Non-temporal propositional logic can be used to express properties of a system that are true or false at a given moment in time, e.g., “the system’s buffers are full”. Temporal logics, on the other hand, can express properties relating to a system’s evolution over time, e.g., “if the system is in state A, then eventually the system’s buffers will be full”. Temporal logics are interpreted in the context of a *model* or *structure* that describes the system’s evolution over time. The labeled transition systems that define the semantics of both process algebras and FSM-representations can serve as structures over which temporal logic formulas are interpreted. For instance, the LTS of figure 2.5 can be seen to have the properties “event a is always followed by event b”. There are several flavours of temporal logic, the most prominent of which are *linear-time logics* and *branching-time logics* [9].

Branching time logics view the future as a tree of possible computation paths. At each point in time, therefore, one can make statements such as “it is possible that proposition  $p$  is true at some time in the future”. Such a statement is true in a state  $s$  if  $p$  is true in at least one state on at least one path starting at  $s$ . A well-known branching time logic is *computation tree logic (CTL)* [29]. The syntax of a CTL formula  $\phi$  is

$$\phi := p \mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \mid \exists \bigcirc \phi_1 \mid \exists \square \phi_1 \mid \exists \phi_1 U \phi_2$$

where  $p$  is an atomic proposition. The formula  $\exists \bigcirc \phi$  is true in a state  $s$  if there is a direct successor state of  $s$  in which formula  $\phi$  is true.  $\exists \square \phi$  is true in  $s$  if there is path starting at  $s$  on which  $\phi$  is always true.  $\exists \phi_1 U \phi_2$  is true in a state  $s$  if there is a path starting at  $s$  on which  $\phi_1$  remains continually true until  $\phi_2$  becomes true. Widely used abbreviations are  $\forall \diamond \phi := \neg \exists \square \neg \phi$  ( $\phi$  is eventually true on all paths) and  $\exists \diamond \phi := \exists (\text{TRUE} U \phi)$  (there is a path on which  $\phi$  is eventually true).

Linear time logics (LTL) view the future as a single computation path. At each

point in time, one can make statements like “eventually, proposition  $p$  will always be true”. Such a statement is true in state  $s$  and an (*infinite*) path  $r$  if there is some state  $s'$  following  $s$  on  $r$ , such that  $p$  is true in  $s'$  and all other states that follow  $s'$  on  $r$ . Note that in contrast to CTL propositions, a proposition in LTL is not meaningful in a single state — it is only meaningful in the context of a particular path. The syntax of an LTL formula  $\phi$  is (here, we use the LTL flavour used in SPIN)

$$\phi := p \mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \mid \diamond\phi_1 \mid \square\phi_1 \mid \phi_1 U \phi_2$$

where  $p$  is again an atomic proposition. The formula  $\diamond\phi$  is true in a state  $s$  on a path  $r$  if there is at least one state after  $s$  on  $r$  in which  $\phi$  is true ( $\phi$  will eventually be true). The formula  $\square\phi$  is true in a state  $s$  on a path  $r$  if  $\phi$  is true in all states following  $s$  on  $r$ . A path satisfies a formula  $f$  if every state on the path satisfies  $f$ , and an LTL formula can be interpreted as the set of paths that satisfy it. An interesting and very useful property of LTL is that every LTL formula  $f$  can be converted into an equivalent Büchi automaton such that the Büchi automaton accepts exactly those paths that satisfy  $f$  [31]. This equivalence between Büchi automata and LTL formulas is used in the tool SPIN.

## Algorithms and Tools

*Algebraic Methods.* Depending on the interpretation of a process-algebraic specification (its semantics), it is possible to define some notion of equivalence. One such equivalence is used by CCS and is called *bisimulation*. Roughly speaking, two systems are deemed equivalent if their black box representations (i.e., the representation where only external events are visible but not the events among the processes that make up each system) are indistinguishable by experiment. Based on this notion of equivalence, one can derive equational laws like

$$P + Q = Q + P$$

One application of these laws is to check for equivalence of an “implementation” with its specification (the difference between implementation and specification is that an implementation is more detailed). Algebraic transformation rules can also be used to prove process properties expressed in temporal logic. Theorem proving systems like

HOL [45] and PVS [35] can be used to partially automate this process, see [71] for an implementation of CCS in HOL. A thorough survey of automated theorem proving is beyond the scope of this chapter, however, and the interested reader is referred to [45] and [35].

*Enumerative Model-Checking.* An enumerative model-checking algorithm (figure 2.6) was presented by Clarke et al. in 1983 [29, 30]. The algorithm checks whether a system specified as a finite-state machine (the “structure”) satisfies a property expressed as a CTL formula  $f$ . The method works as follows. Each state  $s$  is labeled with the set  $L(s)$  of subformulas of  $\phi$  that it satisfies: first with atomic formulas and then with successively larger subformulas. For instance, a state  $s$  satisfies the subformula  $\phi_1 \wedge \phi_2$  if both  $\phi_1$  and  $\phi_2$  are already in  $L(s)$ . Checking temporal “until”-formulas like  $\exists\phi_1 U \phi_2$  is more complex: from states in which  $\phi_2$  is true, a backwards depth-first search of the graph along paths in which  $\phi_2$  is true can be used to find states in which  $\exists\phi_1 U \phi_2$  is true. Checking a formula like  $\exists\Box\phi_1$  involves detecting cycles in the model with the property that  $\phi_1$  is true in every state of the cycle. This enumerative algorithm has the distinction of being one of the first model-checking algorithms. Unfortunately, it is not practical for verifying non-trivial systems because the complete state transition graph of the system needs to be constructed and stored. A slightly modified version of the algorithm, however, has gained widespread acceptance in the symbolic model-checkers described below.

*Automata-theoretic Methods (On-the-fly Model Checking).* When both the system under test and its desired properties are expressed as automata — yielding languages  $L_S$  and  $L_P$ , respectively — the verification problem is equivalent to checking whether the language  $L_S \cap L_P^c$  is empty or not. The tool SPIN [54, 55] uses the algorithms from [82, 31] to check whether a system specified in the language PROMELA satisfies a property expressed in LTL. The PROMELA specification represents a finite-state machine, and the LTL property is negated and converted into a Büchi automaton. The synchronous product of these two automata is again a Büchi automaton. Because the LTL property is negated, the model checking problem is equivalent to checking whether the language accepted by the composite automaton is empty (in which case the negated property is false and the original property is true) or not (in which case the negated property is true and the original property has been shown to be false).



```

// Algorithm enumerative()
// Labels the states of the model with the subformulas of  $f$  that they
// satisfy.
// Assumes that  $FS$  is a stack containing the subformulas of  $f$ 
// such that  $f$  is at the bottom of the stack, atomic propositions
// are at the top, and the subformulas  $\phi_1, \phi_2, \dots$  that make
// up a subformula  $\phi$  are always above  $\phi$  in the stack.
enumerative() {
    while stack not empty
        pop  $\phi$  from  $FS$ 
        if  $\phi = \neg\phi_1$ 
            label all states that are not labeled with  $\phi_1$ 
        else if  $\phi = \phi_1 \wedge \phi_2$ 
            label all states that are labeled with  $\phi_1$  and  $\phi_2$ 
        else if  $\phi = \exists \bigcirc \phi_1$ 
            label all states that are predecessors of states
            labeled with  $\phi_1$ 
        else if  $\phi = \exists \square \phi_1$ 
            [see main text]
        else if  $\phi = \exists \phi_1 U \phi_2$ 
            [see main text]
}

```

Figure 2.6: Enumerative Model-Checking for CTL (Skeleton)

SPIN checks the emptiness of the automaton using a nested depth-first search algorithm of the state graph and avoids constructing the entire state graph in memory. A large hash table of bits is used to keep track of states already visited: whenever a state is encountered, the state is converted into a hash value which serves as an index into the hash table, and the bit at that index is set to 1. Recent extensions of SPIN include a PROMELA-to-C compiler [18] and number of optimizations like hash compaction [78, 79] and partial-order reduction techniques.

*Symbolic Model Checking.* Figure 2.7 displays an algorithm similar to the enumerative algorithm above. Instead of associating sets of subformulas with states, it associates sets of states with subformulas: starting from atomic propositions, each subformula  $\phi$  of  $f$  is labeled with the set of states  $L(\phi)$  that satisfy it. For instance, a formula  $\phi_1 \wedge \phi_2$  is labeled with those states that are in both  $L(\phi_1)$  and  $L(\phi_2)$ , i.e.,  $L(\phi_1 \wedge \phi_2) = L(\phi_1) \cap L(\phi_2)$ .

Labels for temporal formulas like  $\exists \diamond \phi$  can be obtained by characterizing them as *fixpoints* of simpler formulas. To obtain the set of states  $L(\exists \diamond \phi)$  that satisfy  $\exists \diamond \phi$ , we start with the empty set. Then the algorithm iterates as follows: at each iteration, a new set  $L(\exists \diamond \phi)$  is obtained by including all states already in the set, and by adding all states from which a state in  $L(\exists \diamond \phi)$  can be reached in *one* step. This continues until the set stops growing (since our model is finite, and the set is monotonically increasing, this must happen), at which point all states that satisfy  $\exists \diamond \phi$  have been found. In the language of fixpoint theory,  $\exists \diamond \phi$  is the least fixpoint of the functional  $\lambda y.(p \vee \exists \bigcirc y)$ . Similar fixpoint characterizations can be obtained for the CTL formulas  $\exists \phi_1 U \phi_2$  and  $\exists \square \phi$ .

Compared to the enumerative model-checking algorithm, the fundamental difference of this algorithm is that it manipulates *sets* of states rather than individual states — thus, the computational cost of having to visit every state at least once can be potentially avoided. The idea behind McMillan’s *symbolic model checking* [25, 65] algorithm is to exploit this by representing state sets and transition relations *symbolically* using *binary decision diagrams* (BDDs) [24].

A BDD is a way of representing a subset  $S$  of the set  $\{0, 1, \dots, 2^k - 1\}$  as a directed graph with two kinds of nodes: *internal nodes* and *end nodes*. One special node is designated as the *root* of the graph. End nodes and edges are labeled with 0 or 1.

```

// Algorithm symbolic()
// Labels each subformula of  $f$  with the set of states of the model
// that satisfies it.
// Assumes that  $FS$  is a stack containing the subformulas of  $f$ 
// such that  $f$  is at the bottom of the stack, atomic propositions
// are at the top, and the subformulas  $\phi_1, \phi_2, \dots$  that make
// up a subformula  $\phi$  are always above  $\phi$  in the stack.
enumerative() {
    while stack not empty
        pop  $\phi$  from  $FS$ 
        if  $\phi = \neg\phi_1$ 
            label  $\phi$  with  $S \setminus L(\phi_1)$ 
        else if  $\phi = \phi_1 \wedge \phi_2$ 
            label  $\phi$  with  $L(\phi_1) \cap L(\phi_2)$ 
        else if  $\phi = \exists \bigcirc \phi_1$ 
            label  $\phi$  with
            {states from which  $L(\phi)$  can be reached in one step}
        else if  $\phi = \exists \square \phi_1$ 
            [see main text]
        else if  $\phi = \exists \phi_1 U \phi_2$ 
            [see main text]
}

```

Figure 2.7: Symbolic Model-Checking for CTL (Skeleton)

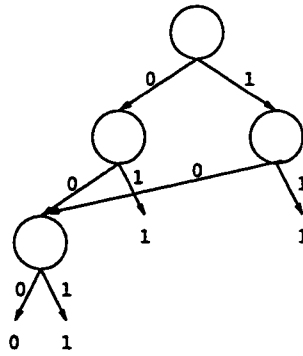


Figure 2.8: Binary Decision Diagram

Figure 2.8 shows an example BDD. To see whether an integer  $n \in \{0, 1, \dots, 2^k - 1\}$  is in the set  $S$ ,  $n$  is represented as a binary number with digits  $d_1 d_2 \dots d_k$ . Starting at the root of the BDD, the graph is traversed with the binary digits of  $n$  indicating the path in the graph, until an end node is reached.  $n$  is in  $S$  exactly if the end node is labeled 1. Using  $k = 3$  and the BDD of figure 2.8, we have that  $7 \in S$  since the path given by the binary representation of 7 — 111 — leads to an end node labeled 1, but  $4 \notin S$  because 100 leads to an end node labeled 0.

There are efficient algorithms for computing various operations on BDDs (union, intersection, negation). If a model has  $k$  atomic propositions  $p_1, \dots, p_k$ , the states of a model can be identified with integers  $\{0, 1, \dots, 2^k - 1\}$  by setting the binary digit  $d_i$  to 1 if a state satisfies proposition  $p_i$ . Moreover, the transition relation of a model can also be written as a BDD, and the model-checking algorithm can be represented in terms of manipulations of BDDs. The first tool utilize BDDs for model-checking was McMillan's SMV [25, 65]; BDDs and their variants have since then been used in numerous other model-checking tools.

A key problem with BDD's is that not all state sets admit an efficient BDD representation — in fact, it is easy to construct sets whose BDD representation is as large as the sets themselves — and finding an efficient BDD in those cases where one exists is difficult. McMillan gives an algorithm for obtaining a BDD from a given subset of  $\{p_1, \dots, p_k\}$ . The size of the resulting BDD representation of a set of states depends on the ordering of the propositions  $p_1, \dots, p_k$  and a 'bad' ordering can be exponentially worse than a 'good' ordering. See [74] for a discussion of BDD performance problems. Despite these drawbacks, BDD-based symbolic model checking

has achieved practical successes in verification, especially for hardware systems with a regular structure (integrated circuits). Recent developments in the field include the application of partial-order reduction techniques to symbolic model-checking [4] and the combination of symbolic model-checking with simulation [85].

### 2.2.2 Validation of Timed Models

Although the success of untimed validation systems like SPIN and SMV shows that many protocol properties can be validated without the notion of time, properties related to absolute time are either impossible or very difficult to validate without explicitly considering time. Some examples are:

- *Real-time applications.* Real-time applications have to be shown to respond to external events within a specified maximum delay.
- *Medium Access Control (MAC) protocols.* Many MAC protocols are designed in such a way that they only function as long as the absolute values of some delay parameters are within certain limits. CSMA/CD-type protocols, for instance, do not work properly if the duration of a packet transmission is less than the maximum propagation delay in the shared medium.
- *Application-layer protocols.* Multimedia applications like video conferencing require that corresponding voice and video packets arrive within a maximum delay tolerance [75, 10]. Even simple applications like a mouse click/double-click recognizer [77] can not be validated without expressing quantitative time.

In some cases, including time can be a natural way of reducing the state space for reachability analysis. For instance, if 2 events, a and b, are executable on 2 processes, P1 and P2, an untimed validator has to execute both possible interleavings of a and b. By annotating the events with time values — say time of a is 4, and time of b is 5 — it is possible to give the validator additional information which makes one of the interleavings redundant.

In this section, we survey verification methods for timed systems. These methods are usually extensions to untimed validation methods, and the organization of the section is parallel to that of the previous section on untimed systems: we first discuss

ways of specifying timed models and then go on to describe ways of specifying properties of such timed models. Timed verification tools and algorithms are described in the last part of the section.

## Specification of Models

*Finite State Machines.* Any LTS can of course be interpreted as a timed system by defining the number of steps between two states as the time that elapses between the states. Here, we will focus on methods of modelling time more conveniently. One such simple, yet useful approach is to attach additional labels to the transitions of an LTS indicating the amount of time that elapses between the states. We will call these LTS *time-labeled* LTS. The LTS derived from timed Petri nets and timed process algebras (see below) are such LTS. When modeled by such an LTS, the delay between two states can only have one of a finite number of possible values, i.e., the delay values are *discrete*.

More sophisticated models like Alur's *timed graphs* [5, 8] are needed to model continuous time. A timed graph is the depiction of a timed transition system, which has a number of *clocks* that can be reset and queried. An edge of the transition system can have timing constraints associated with it; these constraints indicate the clock values that are required in order for the transition to be enabled. The edges can also be associated with the set of clocks that are reset when the transition is taken.

As in the case of untimed systems, there are variants of the basic timed automaton that allow more convenient modelling of protocols through the use of extensions and modularity. Cacciari and Rafiq [26] propose the notion of *temporal communicating machines (TCM)* which are essentially a synthesis of the CFSM model with timed automata. Communication channel delays in the TCM are specified as time *intervals* rather than fixed (deterministic) delays. The model allows the definition of a protocol as a *temporal communicating system (TCS)* consisting of a system of TCMs. Since every infinitesimal increase in time takes the TCS to a new global state, the graph of reachable states is locally infinite. Cacciari and Rafiq show, however, that by grouping global states that differ only in the timer values but are otherwise identical into *regions*, the graph becomes locally finite, and a reachability analysis can be performed.

Lin and Liu describe another extension of the CFSM with time [62]. Like in the TCM, delays in their model — the *integrated time transmission grammar model (ITTG)* — are expressed as time intervals. In [56], Huang and Lee use a variant of Lin and Liu’s model (called *timed communicating finite state machine*) for an Estelle-based timed protocol verification system. A combination of Lin and Liu’s model with the probabilistic approach of [64] is proposed in [57].

*Timed Petri Nets* are Petri nets that have delays associated with each transition. As in a basic untimed Petri net, a transition in such a timed Petri net is enabled if all places with edges leading to it have at least one token. The transition may then fire, resulting in tokens being removed from the incoming places. However, new tokens are not added to the output places of the transition until the delay associated with the transitions has elapsed. The semantics of timed Petri nets are given in terms of an LTS whose transitions are labeled with delays.

The timed Petri net model we described here is only one way of extending the Petri net formalism with time; numerous others have been proposed in the literature. For instance, one could associate delays with places instead of with transitions [68], or delays in the model can be stochastic instead of deterministic — stochastic delays lead us to the stochastic Petri nets which are described in the next section.

*Timed Process Algebras.* Process algebras can be extended to allow special actions that model the passing of time [72]. For example, the passing of time could be indicated by the construct  $d.A$  (where  $d$  is taken from some time domain like the nonnegative integers or real numbers), specifying a process that lets time  $d$  pass and then behaves like process  $A$ . As with untimed process algebras, the semantics of timed process algebras are given by a set of rules and axioms that associate an LTS with each expression of the timed process algebra. Some labels of this LTS are therefore values indicating the passing of time. As an example, consider the processes

$$A = 3.a.A$$

and

$$B = b.3.B$$

Their composition is  $C = A \parallel B$ . An important rule that most timed process algebras have is that if the system consists of more than one process, then time can only pass

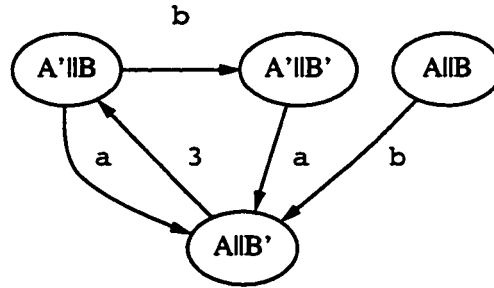


Figure 2.9: Derivation Graph of a Timed PA Term

if all processes agree to do so. This rule is necessary because it is consistent with our intuitive notion of global time. Using this rule, and by rewriting the processes as

$$A = 3.A'; A' = a.A$$

and

$$B = b.B'; B' = 3.B$$

the LTS for process  $C$  can be constructed as in figure 2.9. The first action of the joint process is  $b$ , which is followed by a delay of 3 time units. Then, there are two possible sequences of actions:  $a, b$  and  $b, a$ . Each of these sequences is followed by a delay of 3 time units, and so on.

### Specification of Properties

*Timed Automata.* Alur's *timed automata* [5, 8, 3] are timed graphs with accepting states. A timed automaton operates on *timed words* of infinite length, i.e., infinite sequences of time-stamped symbols. A timed word is *accepted* if it causes the automaton to visit accepting states infinitely often. An example (taken from [8]) of a timed automaton is in figure 2.10. The automaton has one clock  $x$  and accepts all infinite words  $ababab\dots$  provided the delay between a symbol  $a$  and the following  $b$  is any real value less than 2.

The theory of timed automata is analogous to that of conventional (untimed) automata that operate on infinite words: just as the *language* of an untimed automaton is a set of words, the language of a timed automaton is a set of timed words. Thus, the timed automaton can be interpreted to be specifying the acceptable computation



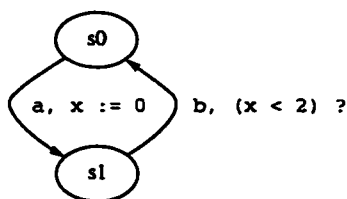


Figure 2.10: Simple Timed Automaton

paths and delays of a system. Since timed automata describe sets of timed computation paths, both the desired properties and implementation of a real-time system can be given in terms of timed automata. The verification problem is then reduced to that of determining whether two automata accept the same language, an approach that is described in [8]. In [7], the properties of a probabilistic system (defined as a Markov process) are given as a timed automaton.

*Timed Process Algebras.* The use of timed process algebras to specify system properties is analogous to the untimed case: because systems can be specified at different levels of abstraction (through the mechanisms of composition and hiding), both the properties of a system and its implementation can be specified in a process algebra. Algebraic transformations can then be employed to determine whether the two are equivalent or not.

*Temporal Logics.* Untimed temporal logics are also called *qualitative* temporal logics because they express properties related to the temporal ordering of events, but can not be used to specify *quantitative* time. To make quantitative temporal statements about structures that include some form of quantitative timing information about a system's evolution — the time-labeled LTS described above are such structures — we must use quantitative temporal logics. Quantitative temporal logics (also called *real-time* logics) can be obtained from untimed temporal logics by annotating temporal operators with time bounds. Both linear time and branching time logics have been extended in this fashion [9]. For example, the untimed until operator  $U$  can be modified to  $U^{\leq t}$ , so that the qualitative CTL formula

$$\exists \phi_1 U^{\leq t} \phi_2$$

means that there is at least one computation path on which  $\phi_1$  is true and will remain true until  $\phi_2$  becomes true within  $t$  time units. The satisfaction relation for real-time

logics is defined analogously to their untimed counterparts. A single state in a time-labeled LTS can satisfy a timed CTL formula or not; the entire LTS satisfies the formula if every state satisfies it. In contrast, the satisfaction of timed LTL formulas in a state is defined in the context of a particular timed path.

Unlike the untimed case, where every LTL formula can be converted into an equivalent Büchi automaton, we know of no similar equivalence for timed LTL formulas.

### Algorithms and Tools

In [5], Alur et al. describe an algorithm for checking whether a system specified as a timed graph satisfies a timed CTL formula. The difficulty lies in the fact that the state space of a timed graph is infinite — a state consists of the location in the graph and the clock valuations, which have real values. The authors show how to construct a finite quotient from this infinite graph by grouping states into *regions* such that the states of a region share the integral parts of the clock valuations and the *ordering* of the fractional parts. The integral parts are needed to decide which transition constraints are satisfied; the ordering of the fractional parts is needed to decide which clock will change its integral value first. Once the finite graph has been constructed, the model-checking algorithm is similar to the enumerative model-checking algorithm for untimed systems (figure 2.6). A symbolic algorithm based on the same finite quotient idea is described in [49] and implemented by the tool KRONOS [36, 37].

VERUS [34] is a language for describing communicating processes with timing constraints such as delays. The associated tool — also called VERUS — converts the description to a finite state machine. The resulting finite state machine is untimed, however, and uses the number of transitions between states as a proxy for the elapsed time between states.

A timed extension of SPIN and its language PROMELA is presented in [81]. The real-time PROMELA specification is converted to a timed graph, which is composed with the never claim (given as a timed automaton) to yield another timed automaton. As in untimed SPIN, the verification problem then consists of checking whether the language of the resulting automaton is empty or not. In a similar approach to the one taken in [5] and [49], the state space is made finite by grouping states into equivalence classes.

## 2.3 Probabilistic Validation

Research in probabilistic validation is motivated by two factors. First, verification in general is intractable due to the state space explosion problem and it is therefore practical to settle for a result like “the system is correct with high probability”, rather than no result at all. From this point of view, any validation algorithm based on a state space search can be considered to be probabilistic if resource limitations restrict the search to part of the state space. The bitstate hashing algorithm of SPIN [54, 55] and its variants [78, 79] are examples of this. In West’s *random state exploration* [84], the validator takes a random walk over the graph of reachable states, rather than doing a systematic search. Unlike these approaches, which randomly limit the state space, Maxemchuk’s probabilistic search algorithm [64] uses state transition probabilities as a heuristic to guide the state space search to more probable regions. The idea here is to annotate each transition of the protocol with its probability of occurrence. Using these annotations, the validator can keep track of the probabilities of all the states that it visits, and avoid searching states whose probability of occurring is below a given threshold.

Another reason for probabilistic validation — and the focus of this section — is that many systems and properties of interest are inherently probabilistic. For example, two processes might communicate via a channel that can lose messages with a given probability, and the message transfer delay may have some probabilistic distribution. The organization of this section is analogous to sections 2.2.1 and 2.2.2: We begin by surveying methods of specifying probabilistic systems and their properties. Then, we describe some probabilistic validation algorithms and tools.

### 2.3.1 Specification of Models

*Finite State Machines.* The simplest and best-known probabilistic state machines are discrete-time and continuous-time Markov chains. Discrete-time Markov chains (DTMC) are finite state machines whose transitions are labeled with probabilities; each state transition is assumed to take one time unit. In continuous-time Markov chains (CTMC), transition *rates* are associated with each pair of states; the time that the system spends in a state is assumed to be exponentially distributed. Other

models extend the basic Markov chain in various directions:

- *Modularity*. As with non-probabilistic finite state machines, it is impractical to specify a large probabilistic system in terms of a single monolithic transition system. A number of methods for the modular or compositional modelling of systems have therefore been proposed.
- *Time*. As in the non-probabilistic case, there are untimed verification methods that abstract away from time, and models that support the modelling of time.
- *Nondeterminism*. Several proposed approaches make a distinction between *nondeterministic* transitions (also called pure nondeterminism) and probabilistic transitions (also called probabilistic nondeterminism). The difference between these two forms of nondeterminism is that in probabilistic nondeterminism, a choice is made according to a given probability distribution, whereas in pure nondeterminism, no assumptions regarding the probabilities of the possible choices can be made. Instead, it is assumed that nondeterministic choices are made (resolved) by some unknown *scheduler* (also called *adversary* or *policy*). A consequence of pure nondeterminism in a model is the notion of *fairness*: in order to make useful statements about a system, it is necessary to assume that the scheduler is *fair* in some sense. For instance, if two nondeterministic choices are available in a given state that is visited infinitely often, it seems unfair if the scheduler always chooses one over the other. A number of definitions of fairness have been proposed. In this example, one possible fairness requirement is that the scheduler makes each choice infinitely often.

We now briefly review how the issues of time, modularity and nondeterminism are handled in some proposed verification methods.

Hansson and Jonsson [46], Iyer and Narasimha's *probabilistic lossy channel systems* [58], Aziz et al. [11], Courcoubetis and Yannakakis [32, 33], Baier et al. [15, 17, 14, 16] use discrete-time Markov chains as structures over which temporal logic formulas are interpreted. These approaches are essentially untimed, but the elapsed time between two system states can be given by the number of state transitions of the Markov chain, which is inconvenient if large delays need to be modeled. Models whose transitions

are labeled with values indicating the time delay between two states are used by de Alfaro [39, 38, 40], Segala [76] and Alur et al. [7, 6]. Alur's *real-time probabilistic processes* are continuous-time Markov processes whose states consist of the system state, a set of pending events, and a set of real-valued countdown clocks measuring the time until the triggering of each event.

The modelling of nondeterminism is addressed in [39, 38, 40, 15, 32, 73, 76] using models similar to *Markov decision processes* (MDP). Markov decision processes are extensions of discrete-time Markov chains that include nondeterminism and can be extended to include time. In an ordinary Markov chain, a probability distribution that is used to select the successor state is associated with each state. In an MDP, a *set* of such probability distributions is associated with each state. The selection of a probability distribution is nondeterministic. Like Markov chains, MDPs can be extended for modelling time. A discussion of the interaction of nondeterminism and probability is in [47].

The issue of modularity is supported by de Alfaro's *stochastic transition systems* (STS) and *timed probabilistic transition systems* (TPS) which combine time, modularity and nondeterminism in a probabilistic framework [39, 40]. The high-level models are specified modularly as STS and transitions can have exponential or unspecified delays. The semantics of these systems is given in terms of the low-level TPS models which are MDPs with costs (delays) attached to each transition.

*Stochastic Petri Nets* are Petri nets that have exponentially distributed delays associated with each transition: the delay between the removal of markers from a transition's incoming places and the addition of markers to a transition's output places is exponentially distributed. In *Generalized stochastic Petri nets* (GSPNs) [28, 2], some transitions may have deterministic delay 0. GSPNs have been widely used for the modelling of stochastic systems like queueing systems. The reachability graph of a GSPN is a CTMC and performance measures of a GSPN model can be obtained using CTMC solution methods.

*Process Algebras*. Probabilistic extensions to process algebras without time are discussed in [83]. In traditional process algebras, the choice of action by a process that has two or more enabled actions is nondeterministic. The authors of [83] propose instead to annotate actions with probabilities. So, if a process is specified as

$$A = (0.8)a.B + (0.2)b.C$$

and actions  $a$  and  $b$  are both enabled, then action  $a$  occurs with probability 0.8, and action  $b$  with probability 0.2. If only one of the actions is enabled, then it occurs with probability 1. Process algebras extended with probabilistic time — *stochastic process algebras* (SPAs)— have been proposed for the specification of probabilistic systems. We explain the features of SPAs using Performance Evaluation Process Algebra (PEPA) [52] as an example. Other SPAs like TIPP [51, 50] and EMPA [19] are similar. PEPA is based on the untimed process algebras CSS and CSP. Time is introduced by associating an exponentially distributed random variable with each action. This additional information makes it possible to extract performance measures from the model. So, whereas the untimed specification

$$A = a.B$$

means that the process  $A$  engages in action  $a$  and then behaves like process  $A$  again, the stochastic specification

$$A = (a, \lambda).A$$

has the additional meaning that process  $A$  can engage in event  $a$  at the rate of  $\lambda$ . If two processes engage in an action and have different transition rates, then the rate of the joint action is given by the *slowest* process. As in untimed process algebras, the semantics of a PEPA specification is given in terms of a labeled transition system, its derivation graph. The derivation graph for the specification

$$A = (a, r_1).(b, r_2).A + (c, r_3).(d, r_4).A$$

is shown in figure 2.11. This graph is in fact a 3-state CTMC and steady state probabilities for the nodes of the derivation graph — which are also the states of the CTMC — can be obtained using standard (numerical) CTMC solution techniques. Then, performance measures can be calculated by associating *rewards* with PEPA actions and associating with state of the CTMC the rewards of the actions enabled in it. When used for performance evaluation, both SPAs and GSPNs are thus simply methods of describing CTMCs. Some papers that compare the merits of the two methods are [41] and [42].

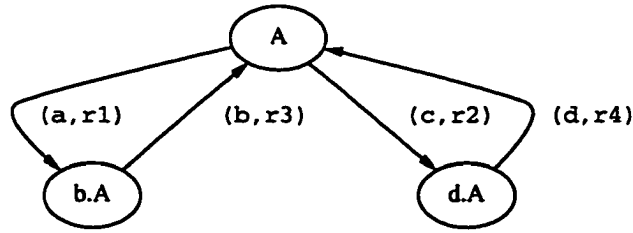


Figure 2.11: Derivation Graph of a PEPA process

### 2.3.2 Specification of Properties

*Process Algebras.* Stochastic process algebras can be used for the specification of properties in the same way as untimed process algebras. The implementation of a system can be described by including low-level details, while the high-level specification of the system properties is more abstract and hides these details. The verification problem, then, is to check whether the two descriptions are equivalent.

*Probabilistic Logics.* Branching time logics have been extended so that they can express probabilistic properties of discrete-time Markov chains. A probability measure can be defined on the set of paths of a DTMC, and this measure is then used to define the probabilistic satisfaction relation. PCTL [46, 17, 14] is a probabilistic version of the branching time logic CTL. In PCTL, probabilities are expressed using the constructs

$$(\bigcirc\phi)_{\geq p} \text{ and } (\phi_1 U \phi_2)_{\geq p}$$

with the meaning that the formula is satisfied in a state  $s$  if the measure of paths starting at  $s$  and satisfying  $(\bigcirc\phi)$  and  $(\phi_1 U \phi_2)$ , respectively, is at least  $p$ . Linear time logics can be interpreted in a probabilistic way without new constructs [16, 33, 73] — the probability of a DTMC satisfying an LTL formula  $\phi$  is the measure of paths that satisfy  $\phi$ . The probabilistic extensions can be readily combined with the real-time extensions described in section 2.2.2; for instance, [46] uses the construct

$$(\phi_1 U \phi_2)_{\geq p}^{\leq t}$$

to express the property that  $\phi_2$  will be true within  $t$  time units with probability  $\geq p$ , and that  $\phi_1$  will remain true until then.

### 2.3.3 Algorithms and Tools

The continuous-time Markov chains (CTMC) that underlie stochastic process algebras and Petri nets can be evaluated using standard numerical methods. Like their non-probabilistic counterparts, stochastic process algebras are accessible to verification by algebraic means.

A number of algorithms for model-checking temporal logic formulas with respect to systems specified as discrete-time Markov chains have been proposed, although few implementations as tools appear to exist. The simplest probabilistic model-checking algorithms [73, 6] are virtually identical to their non-probabilistic counterparts because they check if the formula (LTL in [73] and timed CTL in [6]) is satisfied with probability 1 — as a result, the actual transition probabilities need not be considered.

The algorithm in [46] checks if a probabilistic timed CTL formula is satisfied by a DTMC or not. The algorithm is based on the enumerative algorithm shown in figure 2.6: states are labeled first with the atomic formulas that they satisfy, and then with successively larger subformulas of the main formula. The number of transitions between two states is taken to indicate the elapsed time between the states. [14] proposes a symbolic version of the algorithm using multi-terminal binary decision diagrams (MTBDD) to represent the Markov chain. An implementation of the algorithm based on the tool VERUS is in progress [13].

Courcoubetis and Yannakakis [32, 33] describe an algorithm for checking untimed DTMC against LTL formulas. Their algorithm, which involves solving a system of linear equations the size of the DTMC, calculates the exact measure of paths in the DTMC that satisfy the formula. A less expensive approximate algorithm is presented in [16]. Both papers address the issue of nondeterminism.

The models of Alur [6] and de Alfaro [38, 20] view time as a continuous value, and as a result, the state space is infinite and not directly suitable for model-checking. As in the non-probabilistic case [5], the authors propose algorithms for grouping the system states into a finite number of regions; the resulting finite-state system can then be verified using algorithms similar to the one in figure 2.6.



## 2.4 Summary

Methodologies for protocol validation can be broadly separated into untimed, timed and probabilistic approaches.

*Untimed* methods abstract away from the notion of qualitative time and allow the specification of properties related to the ordering of events, but not to the quantitative delay between events. A common basic model is that of the *labeled transition system* (LTS), which serves as a structure for interpreting temporal logic formulas. Other models like Petri nets and process algebras can be translated into LTS. The main model-checking algorithms are the *enumerative* approach (which labels each state of the LTS with successively larger subformulas), the *symbolic* approach (which associates with each successively larger subformula the set of states that satisfy it), and the *automata-theoretic* approach (which uses the fact that linear temporal logic formulas can be converted to automata).

*Timed* methods extend untimed approaches with the notion of qualitative time. Discrete-time models can be obtained by attaching delay values to the transitions of a LTS or by interpreting the number of transitions between states as the elapsed time between them. Such timed LTS are used as structures over which *qualitative* temporal logic formulas can be interpreted. Continuous-time models like *timed automata* have infinite state spaces and need to be converted into finite projections before performing model-checking. Aside from this conversion process, the model-checking algorithms are based on their untimed counterparts.

*Probabilistic* methods are for the most part based on *continuous-time Markov chains* (CTMC) or *discrete-time Markov chains* (DTMC). While both types of Markov chains can be tackled by numerical means, DTMC are more accessible as structures over which *probabilistic temporal logic formulas* can be interpreted. There are untimed and timed probabilistic models; the model-checking algorithms are based on the non-probabilistic versions.

# Chapter 3

## A Probabilistic Protocol Validation Model

### 3.1 Overview

In this chapter, we describe the validation model that is the foundation of our validation tool. As is apparent from the survey in the preceding chapter, the design of a validation system has two components: a method for modelling systems and a method for expressing properties of the model.

Sections 3.2 and 3.3 describe our validation model and our logic. Following, e.g., [38], we define two modelling methods: a low-level model and a high-level model. The low-level model is mathematically simple but inconvenient for the description of complex systems. Our logic and our algorithms operate at this level. The high-level model allows the modular specification of systems as a set of communicating processes. The semantics of the high-level model are given in terms of the low-level model. Given the plethora of verification algorithms described in the survey chapter, it is natural to ask, “Why do we need yet another validation model?”. Section 3.5 attempts to answer the question by comparing our model with others proposed in the literature and pointing out its advantages and disadvantages.

The results of the chapter are summarized in section 3.6.

## 3.2 Low-level Model: Timed Probabilistic Transition System

This section describes our low-level model, which is essentially a discrete-time Markov chain with additional labels denoting the event type and delay on every transition. We show how to assign a probability measure to the execution paths of the model, and how to express its properties using the linear-time logic EL.

### 3.2.1 Timed Probabilistic Transition System

A Timed Probabilistic Transition System (TPTS) is a discrete-time Markov chain with additional labels  $e$  (event type) and  $t$  (delay) on every transition. The delay of a transition indicates the number of time units that the system spends in the origin state before making the (instantaneous) transition to the destination state. A special state is designated as the *initial state* of the system. More formally, we have the following <sup>1</sup>

**Definition 1 (Timed Probabilistic Transition System (TPTS))** A timed probabilistic transition system (TPTS) has the following components:

- $G$  is a set of states
- $g_0 \in G$  is an initial state
- $\mathcal{E} : G \times G \rightarrow \mathbf{E}$  is a function that assigns an event  $e$  to each transition between states.
- $\mathcal{T} : G \times G \rightarrow \mathbf{N}^0$  is a function that assigns a delay  $t$  to every transition between states
- $\mathcal{P} : G \times G \rightarrow [0, 1]$  is a function that assigns a transition probability  $p$  to each pair of states such that  $\sum_{g_i \in G} P(g, g_i) = 1$  for all  $g \in G$ .

A TPTS can be written as a quintuple  $\langle G, g_0, \mathcal{P}, \mathcal{T}, \mathcal{E} \rangle$ . It can be depicted as a directed graph whose nodes are states and whose edges are the transitions with non-zero probabilities. These transitions are written as  $g \xrightarrow{e, t, p} g'$ . Figure 3.1 shows an

---

<sup>1</sup>In this and the following definitions,  $\mathbf{N}^0$  is the set of the natural numbers including zero, and  $\mathbf{E}$  is some ordered set of symbols, the *events*

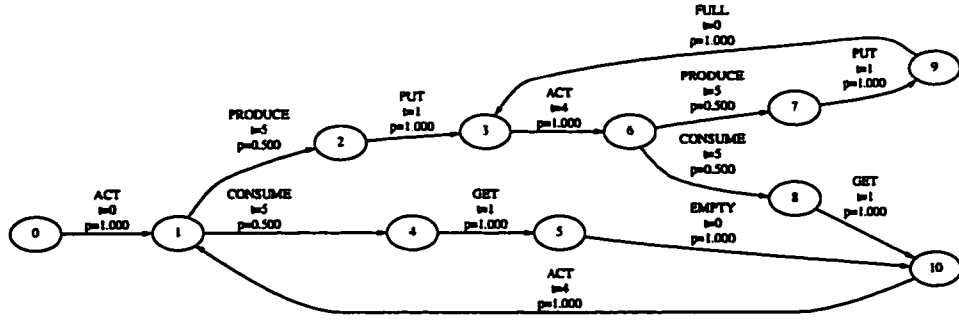


Figure 3.1: Example TPTS

example TPTS with 11 states. It represents a system where an activity (event ACT) results either in the production of data (event PRODUCE) or in the consumption of data (event CONSUME) after 5 time units. When data is produced, it is put in the buffer (PUT) with a delay of 1 time unit, and when it is consumed, it is obtained from the buffer (GET) after 1 time unit. The buffer has a capacity of 1 and an attempt to PUT into a full buffer is immediately followed by a FULL event. Similarly, an attempt to retrieve data from an empty buffer is followed by an EMPTY event.

### 3.2.2 Probabilities

The probability  $\mathcal{P}_M$  of a path  $g_0, g_1, \dots, g_n$  in a TPTS  $M$  is defined by:

$$\mathcal{P}_M(g_0, g_1, \dots, g_n) := \mathcal{P}_M(g_0, g_1) \mathcal{P}_M(g_1, g_2) \cdots \mathcal{P}_M(g_{n-1}, g_n)$$

For a set  $Q = \{p_1, \dots, p_r\}$  of paths, we define  $\mathcal{P}_M(Q) := \mathcal{P}_M(p_1) + \dots + \mathcal{P}_M(p_r)$ . Let  $\Omega_{g,n}$  be the set of all paths of length  $n$  that start at global state  $g$ .  $\mathcal{P}_M$  is a probability measure on  $\Omega_{g,n}$ . It is easy to see that the usual axioms apply,

1.  $\mathcal{P}_M(\Omega_{g,n}) = 1$
2.  $0 \leq \mathcal{P}_M(Q) \leq 1$  for all  $Q \subseteq \Omega_{g,n}$
3.  $\mathcal{P}_M(Q_1 \cup Q_2) = \mathcal{P}_M(Q_1) + \mathcal{P}_M(Q_2)$  for  $Q_1, Q_2 \subseteq \Omega_{g,n}$  and  $Q_1 \cap Q_2 = \emptyset$

The probability of an infinite path  $g_0, g_1, g_2, \dots$ , is  $\mathcal{P}_M(g_0, g_1) \mathcal{P}_M(g_1, g_2) \cdots$ , which is typically zero unless there is a cycle in  $M$  where each transition has probability 1. The measure of the set of *all* infinite paths with a common prefix  $g_0, g_1, \dots, g_n$ , is

$\mathcal{P}_M(g_0, g_1)\mathcal{P}_M(g_1, g_2) \cdots \mathcal{P}_M(g_{n-1}, g_n)$ . We associate the subtree of all paths starting at  $g_n$  with this set. The subtree is said to *contain* any infinite path that goes through  $g_n$ .

### 3.2.3 Event Logic

We consider events and their delays to be the only externally observable results of the execution of a TPTS. It is therefore useful to define the notion of *event sequence*: a list of events and delays  $e = e_0 \xrightarrow{t_1} e_1 \xrightarrow{t_2} e_2 \cdots \xrightarrow{t_n} e_n$  is an event sequence starting at  $g_0$  if there is a path  $g_0 \xrightarrow{e_0, t_0, p_0} g_1 \xrightarrow{e_1, t_1, p_1} \cdots \xrightarrow{e_n, t_n, p_n} g_n$ . More than one path in  $M$  may have the same event sequence — we write that each event sequence  $e$  represents a set of paths  $[e]$ . The probability of an event sequence,  $\mathcal{P}_M(e)$ , is defined as  $\mathcal{P}_M([e])$ .

We present a simple temporal logic — event logic (EL) — for expressing properties of a TPTS  $M$ . EL is based on the linear-time logic implemented by SPIN and borrows probabilistic real-time extensions similar to those used by the logic PCTL [46]. PCTL itself is an extension of the branching time logic CTL. EL formulas contain event types, the usual boolean connectives, and the additional temporal “until” operator  $U$ . The meaning of a formula  $f_1 U^{\leq t} f_2$  is that  $f_2$  becomes true within  $t$  time units and that  $f_1$  will be true until  $f_2$  becomes true. Some examples of EL formulas and their meanings are

- “event A is followed by event B within 100 time units”

$$A \rightarrow \text{TRUE} U^{\leq 100} B$$

- “event A is followed by event B within 100 time units with probability  $\geq 0.50$ ”.

$$P(A \rightarrow \text{TRUE} U^{\leq 100} B) \geq 0.5$$

(such formulas involving a probability threshold are *probabilistic* EL formulas)

- “event E never occurs”

$$\neg E$$

More formally, the grammar for EL formulas is as follows:

- the constants TRUE and FALSE and the event types are atomic EL formulas

- if  $f_1, f_2$  are EL formulas, then  $\neg f_1, f_1 \wedge f_2$  and  $f_1 \vee f_2$  are EL formulas
- if  $f_1, f_2$  are EL formulas, then  $f_1 U^{\leq t} f_2$  for  $t \in N$  is an EL formula
- if  $f$  is an EL formula, then  $P(f) \geq p$  for  $p \in [0, 1]$  is a probabilistic EL formula

A common abbreviation is  $A \rightarrow B$  for  $\neg A \vee B$ . The typical non-probabilistic EL formula is  $A \rightarrow \text{TRUE} U^{\leq t} B$  which is also called a *bounded response* property.

The satisfaction relation  $\models$  of non-probabilistic EL formulas is defined on events that are part of infinite event sequences. An infinite event sequence satisfies a formula if the formula is satisfied by all events of the sequence. For an event type  $e_i$ , an event sequence  $e = e_0 \xrightarrow{t_1} e_1 \cdots e_i \cdots \xrightarrow{t_{i+1}} e_{i+1}$  and a non-probabilistic formula  $f$  we define

- if  $f$  is atomic, then  $e_i \models_e f$  if  $f = \text{TRUE}$  or  $f = e_i$ , otherwise  $e_i \not\models_e f$
- if  $f = f_1 \vee f_2$ , then  $e_i \models_e f$  if  $e_i \models_e f_1$  or  $e_i \models_e f_2$
- if  $f = f_1 \wedge f_2$ , then  $e_i \models_e f$  if  $e_i \models_e f_1$  and  $e_i \models_e f_2$
- if  $f = \neg f_1$ , then  $e_i \models_e f$  if  $e_i \not\models_e f_1$
- if  $f = f_1 U^{\leq t} f_2$ , then  $e_i \models_e f$  if  $e_i \models_e f_2$  or ( $e_i \models_e f_1$  and ( $t_{i+1} \leq t$  and  $e_{i+1} \models_e f_1 U^{\leq t-t_{i+1}} f_2$ ))
- $e \models f$  if for all  $i \in \{1, 2, \dots\}$ :  $e_i \models_e f$ .

A path  $g_0 \xrightarrow{e_0, t_0, p_0} g_1 \xrightarrow{e_1, t_1, p_1} \dots$  in a TPTS satisfies  $f$  if the corresponding event sequence  $e_0 \xrightarrow{t_1} e_1 \cdots$  satisfies  $f$ . A probabilistic EL formula  $P(f) \geq p$  is satisfied by  $M$  if the measure of satisfying event sequences of  $M$   $\mathcal{P}(\{e : e \models f\})$  is greater than or equal to  $p$ . An algorithm that checks this is described in section 4.3.

Although the satisfaction of an EL formula  $f$  is defined with respect to infinite event sequences, only a finite subsequence of the infinite sequence needs to be examined in order to determine whether an event  $e_i$  of the sequence satisfies  $f$  with respect to the infinite sequence. This is essentially a consequence of the fact that the until-operator  $U$  always has a time bound, and can be shown with a straightforward inductive argument. The *time-length* of a sequence is the sum of its time delays: for example, the time-length of the sequence  $e_0 \xrightarrow{t_1} e_1 \cdots \xrightarrow{t_n} e_n$  is  $\sum_{i=1}^n t_i$ . We call

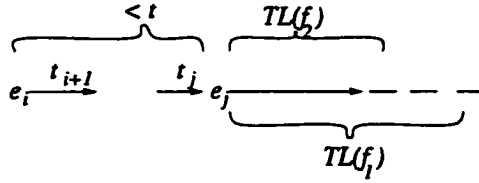


Figure 3.2: Checking for Satisfaction of  $f_1 U^{\leq t} f_2$

the time-length of the maximum required subsequence the time-length of  $f$  and write  $TL(f)$ . For atomic formulas, only a single event of the sequence —  $e_i$  itself — needs to be examined to determine if  $e_i \models_e f$ , and therefore,  $TL(f) = 0$  if  $f$  is atomic. So, to show that a non-atomic formula  $f$  also has finite time-length, we can assume inductively that its subformulas have finite time-length. If  $f = \neg f_1$ ,  $f = f_1 \wedge f_2$ , or  $f = f_1 \vee f_2$ , then  $TL(f)$  is clearly finite if  $TL(f_1)$  and  $TL(f_2)$  are. If  $f = f_1 U^{\leq t} f_2$ ,  $e_i \models_e f$  if  $f_2$  comes true within  $t$  time units and  $f_1$  is true until then. The worst case is as follows:  $f_2$  remains false until the last possible moment, i.e., until  $t$  time units have passed, and  $f_1$  remains true until then. In this case (figure 3.2), a subsequence of time-length  $TL(f_1)$  extending beyond  $e_j$  is required to determine that  $e_j \models_e f_1$ , and a subsequence of length  $TL(f_2)$  is needed to determine that  $e_j \models_e f_2$ . Thus,  $TL(f_1 U^{\leq t} f_2) = t + \max(TL(f_1), TL(f_2))$  and finite. Note that we have used throughout this argument the requirement that our TPTS is stutter-free (next section), and that therefore a finite time-length implies an event sequence of finite length.

### 3.3 High-Level Model: Probabilistic Protocols

This section describes our high-level protocol modelling methodology, which conveniently and modularly represents protocols as a set of finite state machines — *processes* — that exchange *events*. Processes and events are defined in 3.3.1. Subsection 3.3.2 shows how processes are combined to form protocols, and how this representation maps to the low-level TPTS model presented in the previous section. Subsection 3.3.3 points out some restrictions on the general process model that are necessary in order to make protocols accessible to our algorithms.

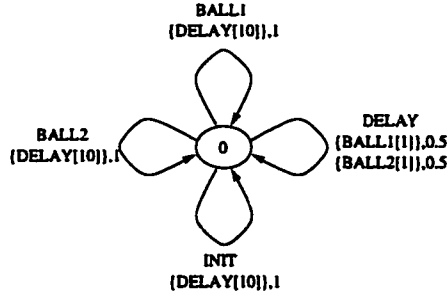


Figure 3.3: Juggler Process

### 3.3.1 Events and Processes

Processes are finite state machines that respond to events. When a process receives an event it may change its state and may generate a set of new timed events. For example, a process representing a receiver may generate an acknowledgement event in response to an event representing the arrival of data at the receiver. The probability of generating a particular set of output events is given by the process definition.

An example of a simple “juggler” process is in figure 3.3. Following the INIT event, the process sends a DELAY event to itself. When it receives the DELAY event, the process sends one of  $n$  different balls ( $BALL_1, \dots, BALL_n$ ) to itself. Upon receiving a BALL event, it again sends a DELAY event, and the cycle continues. In figure 3.3,  $n = 2$ , and the delays for the DELAY and  $BALL_i$  events are 10 and 1, respectively. Formally,

**Definition 2 (Event, Process)** Let  $Y := E \times N^0$ . A timed event  $y \in Y$  is a pair  $(e, t)$  with  $e \in E$  (the event type), and  $t \in N^0$  (the event’s scheduled time). A process has the following components:

- $S$  is the set of process states.
- $s_0 \in S$  is the initial state.
- $T \subseteq S \times E$  is the transition relation of the process. For  $(s, e) \in T$ ,  $s$  is the original state and  $e$  is the input event type.
- $N : T \rightarrow S$  is the next-state function.
- $O : T \rightarrow 2^{2^Y}$  assigns a set of timed output event sets to each transition.



- $P$  is a probability function  $T \times 2^Y \rightarrow [0, 1]$  such that for  $t \in T$ ,  $\sum_{A \in O(t)} P(t, A) = 1$ .

A process can be written as a 5-tuple  $\langle S, s_0, T, N, O, P \rangle$ . When a process is in state  $s$  and it receives an event of type  $e$  with  $t = (s, e) \in T$ , it changes to state  $s' = N(t)$  and generates a set of new timed events  $A \in O(t)$ . The probability of generating a particular set of output events is given by the function  $P$ . We write  $s \xrightarrow{y/A/P(t,A)} s'$ . The union of all input event types of a process is called its *input set*; the union of all output event sets of a process is called its *output set*.

### 3.3.2 Probabilistic Protocols

A set of communicating processes form a protocol. For example, the juggler process from the preceding section forms a protocol consisting of only one process. Another simple protocol with two processes is in figure 3.4: a data buffer with capacity 1 and a producer/consumer that utilizes the buffer (this example was briefly presented in the chapter 1). The buffer is modeled by the process in figure 3.4 (left). It has two states – 0 (empty) and 1 (full) and receives PUT and GET events that make it move between these states. If the buffer process receives a PUT event while it is in the full state, it generates a FULL event to itself (exposing error conditions as events in this way makes it possible to specify EL formulas to reason about them). Conversely, if it receives a GET event while in the empty state, it generates an event of type EMPTY.

The producer/consumer is modeled by the process in figure 3.4 (right). When it receives an activity event ACT, it generates the next activity event with a delay of 10 time units and either a PRODUCE or CONSUME event to itself with a delay of 5 time units. PRODUCE or CONSUME are each chosen with a probability of 0.5. When the producer-consumer receives a PRODUCE event, it sends a PUT event with delay 1 to the buffer process; when it receives a CONSUME event, it sends a GET event to the buffer process. Note that the producer/consumer process has only one state. A probabilistic protocol can be composed of these two processes and the initial event type ACT.

The following definition ensures that the processes of a protocol are compatible in the sense that each event that occurs during the execution of the protocol is the

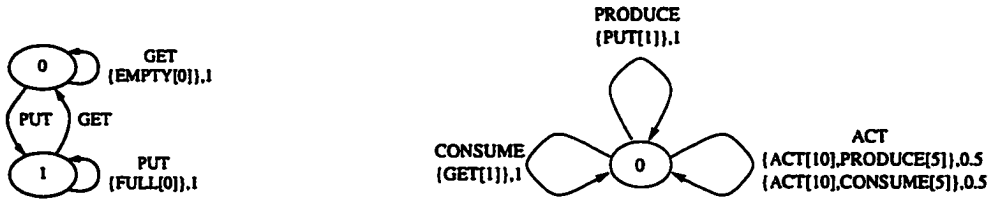


Figure 3.4: Buffer Process (left) and Producer/Consumer Process (right)

input of exactly one process, its *destination*.

**Definition 3 (Probabilistic Protocol)** A probabilistic protocol is a pair  $\langle M, Z \rangle$  consisting of a set of processes  $M = \{M_1, \dots, M_n\}$ ,  $M_i = \langle S_i, s_{0,i}, T_i, N_i, O_i, P_i \rangle$ , and a set of event types  $Z = \{z_1, \dots, z_m\}$  (the initial events), such that the  $z_1, \dots, z_m$  and each output event type of each processes is an input event type for exactly one of the processes  $M_i$ .

The protocol works by keeping a list of pending events (the *event list*) and repeatedly *executing* the earliest event in the list. In the initial global state, all processes are in their initial states, and the event list consists of the initial events scheduled at time 0. When an event is executed, the event's time is subtracted from the times of the remaining events and the event's destination process state changes according to the process transition relation. New events are generated by choosing one set of new events  $A_i (i = 1, \dots, L)$  according to the probability distribution of the transition and appending it to the event list. Thus, the times of output events denote the delay between the current time and the new event, not the absolute event times. More formally, we describe the semantics of a protocol in terms of a TPTS  $(G, g_0, \mathcal{P}, \mathcal{T}, \mathcal{E})$ :

- the states  $g \in G$  of the TPTS are protocol global states consisting of the individual process states  $s_i \in S_i$  and a set  $E$  of events (the event list):  $g = \langle s_1, \dots, s_n, \{e_1, \dots, e_r\} \rangle$ .
- the initial state  $g_0$  is  $\langle s_{0,1}, \dots, s_{0,n}, \{\langle z_1, 0 \rangle, \dots, \langle z_m, 0 \rangle\} \rangle$ : all processes are in their initial states and the event list consists of the initial event types scheduled at time 0.
- $\mathcal{P}, \mathcal{T}, \mathcal{E}$  are obtained as follows. The event  $e \in E$  with the lowest time is *occurable*; if several events have the same lowest time then the event with the highest

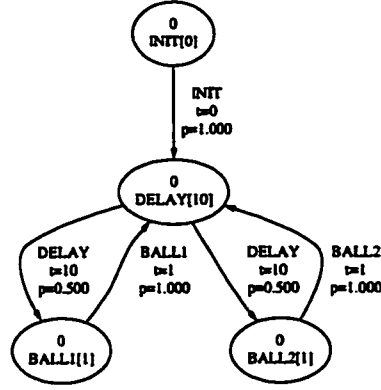


Figure 3.5: TPTS for Juggler Process

priority is occurable. Let

- $e_i = \langle y_i, t_i \rangle$  be the occurable event of a global state  $g = \langle s_1, \dots, s_n, \{e_1 = \langle y_1, t_1 \rangle, \dots, e_r = \langle y_r, t_r \rangle\} \rangle$ ,
- $M_j = \langle S_j, s_{0,j}, T_j, N_j, O_j, P_j \rangle$  the destination process of  $e_i$ ,
- $t = (s_j, y_i) \in T_j$  [ $y_i$  is an occurable event type in state  $s_j$ ],  $N(t) = s'_j$  [ $s'_j$  is the destination state], and  $O_j(t) = \{A_1, \dots, A_l\}$  [the output event sets].

Then, there are transitions from  $g$  to  $l$  different global states  $g'_k = \langle s_1, \dots, s'_j, \dots, s_n, \{\langle y_1, t_1 - t_i \rangle, \dots, \langle y_{i-1}, t_{i-1} - t_i \rangle, \langle y_{i+1}, t_{i+1} - t_i \rangle, \dots, \langle y_r, t_r - t_i \rangle\} \cup A_k \rangle$  where  $k \in \{1, \dots, l\}$ . The transition  $(g, g'_k)$  is labeled with  $\mathcal{P}(g, g'_k) = P_j(t, A_k)$ ;  $\mathcal{T}(g, g') = t_i$ ;  $\mathcal{E}(g, g'_k) = y_i$ .

The event sets  $A_i$  are used to express probabilistic non-deterministic choices. Every successor state contains exactly one  $A_i$ , so the number of global states reachable by executing the event is  $l$ . Figures 3.5 and 3.6 show the TPTS obtained for the juggler protocol and producer/consumer protocol examples, respectively.

### 3.3.3 Model Restrictions

Because processes can generate more than one event in response to receiving an event, it is possible for a probabilistic protocol to result in an infinite TPTS. Also, processes can generate events with zero delay, and it is therefore possible for the resulting TPTS to have an infinite loop in which time never progresses — it *stutters*. In the rest of

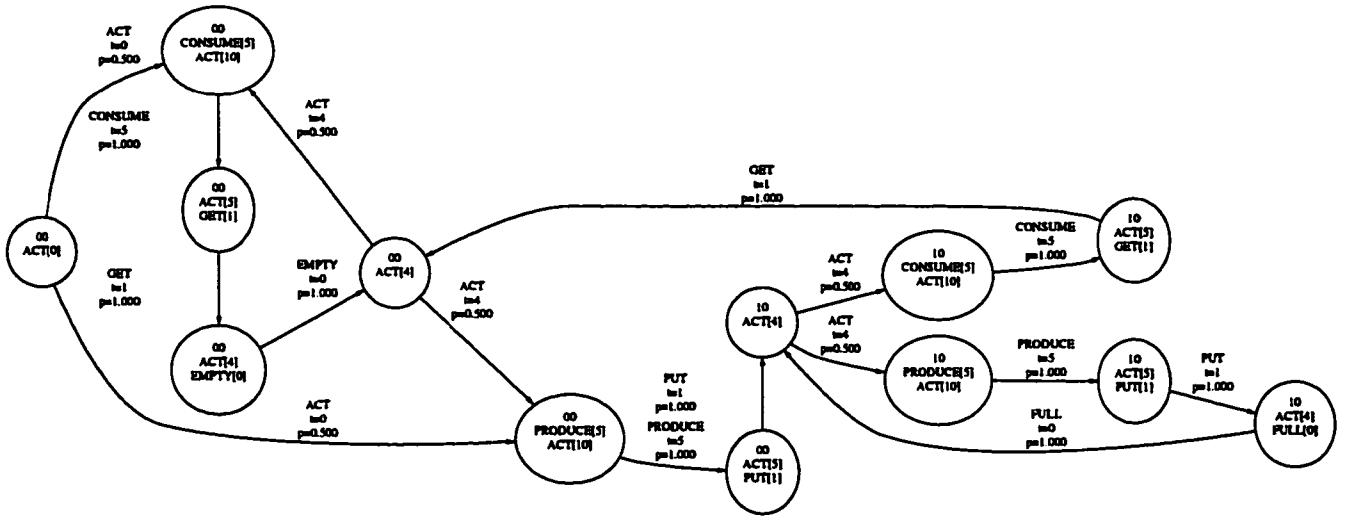


Figure 3.6: TPTS for Data Buffer Protocol

this paper, we will require that protocols are well-behaved in the sense that their TPTS representation

- is finite
- is stutter-free (i.e., has no zero cycles)
- has no sink states (i.e., states with no successors)

In addition to these requirements, the event logic model-checking algorithm in section 4.3 makes the following assumption about the *strongly connected components* (a strongly connected component is a maximal subset of states such that there is a path in the TPTS between any two states of the set) of the TPTS:

- every strongly connected component is a **BOTTOM STRONGLY CONNECTED COMPONENT**

This means that it is not possible to reach one strongly connected component from another, or, in other words, every infinite path in the TPTS reaches exactly one strongly connected component. Such a condition is not unusual in the world of Markov chains: many performance evaluation methods for Markov chains and probabilistic validation tools based on Markov chains [16, 52], assume that the Markov chain is *ergodic*, which essentially means that the entire Markov chain forms a single bottom

strongly connected component. Our assumption is more general than this ergodicity requirement.

### 3.4 Observers

This section describes *observers*, which are a special kind of process that can be used as an alternative to EL as a way of specifying protocol properties. The event logic EL is a convenient way of expressing many qualitative and quantitative properties of a TPTS, but some more complex properties can not be easily formulated as EL formulas. For example, the properties

“If there are three events of type A with 100 time units, then event B must occur within 10 time units following the third event A.”

and

“If the delay between event A and event B is less than 100 time units, then event C must occur within 10 time units following the event B.”

can not be easily written as EL formulas. Such complex protocol properties can better be verified using observers similar to those in [43]. Observers are like processes except that

- they cannot send or receive events of their own
- they see all executed events in the entire system

An observer sees a sequential list of events and must decide whether the observed sequence is acceptable or not. Like processes, observers see only one possible event path. To check properties, therefore, observers must be written as error detectors, i.e., they falsify properties rather than proving them. Since they can not send or receive events of their own, observers do not interfere in the operation of a protocol. They do have a state, however, and therefore increase the state space of the protocol.

Both observers and EL formulas can be associated with sets of event sequences: an observer can be associated with the set of event sequences that result in an error being detected, and an EL formula can be associated with the set of event sequences

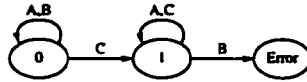


Figure 3.7: Observer Example

that do not satisfy it<sup>2</sup>. An interesting question is whether observers are strictly more powerful than EL formulas in the sense that for every formula  $f$ , an observer can be written to recognize the set of sequences associated with  $f$ , but that not every observer can be replaced by an EL formula in this manner. While a precise treatment of this question is beyond the scope of this thesis, we can offer some observations that indicate that observers are indeed more powerful than EL formulas.

First, we show that not every observer can be replaced by an equivalent EL formula. Figure 3.7 shows the state transition diagram for an observer for the event types A, B, and C. It recognizes event sequences that contain a subsequence of the form  $C(A,C)^*B$  as errors. Assume that there is an EL formula  $f$  recognizing such sequences, and let  $e = C \xrightarrow{1} A \cdots \xrightarrow{1} A \xrightarrow{1} B \cdots$  be an event sequence with  $n$  consecutive A's. Then,  $C \not\models_e f$ . We know that checking  $C \models_e f$  can be done by examining a subsequence of  $e$  that has finite length  $TL(f)$  (section 3.2.3). By setting  $n > TL(f)$ , we have a contradiction: the first  $TL(f)$  events of  $e$  contain no B's, and it is impossible to determine whether  $e$  has a subsequence of the form  $C(A,C)^*B$  by only looking at the first  $TL(f)$  events of  $e$ . Therefore, our assumption is wrong and there can be no formula  $f$  that represents sequences that contain subsequences of the form  $C(A,C)^*B$ .

Second, we show that for every EL formula  $f$ , one can construct an observer that recognizes the same set of event sequences. The formula  $f$  defines a set  $S$  of finite event sequences of length  $\leq TL(f)$ , namely, the event sequences whose first event does not satisfy  $f$ . This set  $S$  itself is finite. Now, if  $S$  has only one element  $s$ , then the problem of recognizing whether a path  $e$  contains  $s$  or not is equivalent to the well-known string matching problem where a (short) substring has to be found in a (much longer) text —  $s$  corresponds to the short substring, and  $e$  to the text. Linear-time string matching algorithms like the Knuth-Morris-Pratt algorithm [12]

<sup>2</sup>Of course, a formula can also be associated with the set of sequences that satisfy it, but the exposition in this part is simplified if we consider the sequences that are *not* satisfying

can be readily implemented as an observer. If  $S$  has two elements  $s_1$  and  $s_2$ , then two observers can be constructed (one to recognize  $s_1$  and one to recognize  $s_2$ ). These two observers can be combined in an observer whose state space is the cross product of their state spaces; this new observer recognizes both  $s_1$  and  $s_2$ . This construction can be extended for arbitrary (but finite)  $S$ .

### 3.5 Comparison With Other Models

In this section, we compare our model to other validation models and explain some of the rationale behind our design. Our main goal was to design a validation model that can serve as a foundation for a *practical* validation tool of realistic models — something which, we believe, has not always been a focus of past efforts.

One way in which we aim to increase the practicality of the model is by designing a high-level model resembling that of discrete event simulators which are familiar to protocol designers. The choice of a linear temporal logic (LTL) over a branching time logic is also motivated by the desire to make the notion of “probability of satisfaction” of a formula intuitive with respect to DES: the measure of paths that satisfy an EL formula is equivalent to the probability of observing a satisfying path when running a simulator. In other words, if we run a simulator many times with different random seeds, then the relative frequency of a given event sequence will approach its probability.

Neither linear time nor branching time logics are perfect for our needs, however. For example, expressing a property like “95% of all packets will be delivered within some time limit  $t$  from their generation” can not be adequately expressed in either logic. If we formulate it in EL, our flavour of LTL, then an infinite path in the model satisfies it if every state on the path satisfies it. Unfortunately, in most stochastic models, *every* infinite path will at some point violate the formula, and therefore, the measure of satisfying paths would be zero even if the actual proportion of packets that are not delivered within the time limit is arbitrarily small. Branching time logics like CTL pose a similar problem. If the property is expressed in CTL, then a state satisfies the formula if on 95% of paths starting here, the delivery will be within  $t$  time units. Since a model is required to satisfy the formula in every state, even a

Model	Prob-abilities?	Time?	Non-det?	Spec. of Properties	Language	Algorithm
DTMC [46, 14, 13]	Yes	no. of steps	No	PCTL	VERUS [13]	enumerative [46] symbolic [14, 13]
Probabilistic Real-Time Systems [7, 6]	Yes	Yes (cont)	No	Automata [7] TCTL [6]	—	enumerative, requires entire MC
DTMC [32, 33]	Yes	No	Yes	LTL	—	tableau-based, requires entire MC
Real-Time SPIN [81]	No	Yes (cont)	Yes	Automata	RT-PROMELA	on-the-fly check for emptiness
DTMC [16]	Yes	No	Yes	LTL	—	calculates approx. satisfaction prob., requires entire MC
Our Model	Yes	Yes (discr)	No	EL (LTL)	C++	calculates exact. prob., on-the-fly constr. of MC

Table 3.1: Comparison of some Validation Models

single non-satisfying state with arbitrarily low probability of occurrence would make the system non-satisfying. One way to avoid this problem with our model is to use the model as a DES tool and make statistical statements about the satisfaction of the property.

Our approach is most similar to those described in [46, 14, 13], [7, 6], [32, 33],[81], [16]. These methods, along with our own, are summarized in table 3.1. Other approaches, like stochastic process algebras and stochastic Petri nets, have their advantages. Most notably, they allow the concise and elegant description of some queueing systems, but it is questionable if large-scale systems can be programmed by non-experts.

None of the approaches listed in the table feature a high-level model based on DES. Only the approaches in [46, 14, 13] and [7, 6] support both time and probabilities, two modelling parameters that we feel are essential for the realistic modelling of protocols. Their model-checking algorithms, however, require the manipulation of the entire underlying Markov chain, which is not realistic. The timed extension to SPIN described in [81] uses the same efficient bitstate hashing algorithm as SPIN, but does not support the modelling of probabilistic systems. A number of the listed validation systems address the issue of non-determinism. We chose to omit this in the



interest of simplicity and because the modelling of non-determinism is not normally needed in a performance evaluation tool.

PROMELA, the modelling language of SPIN, includes special keywords *accept* and *progress* that are used to detect *livelocks* and *non-progress cycles*, respectively.

- *Livelocks* are cycles that contain particular undesirable states labeled with the keyword *accept*, i.e., labeling a state with *accept* indicates that it can not occur infinitely often.
- *Non-progress cycles* are cycles that contain particular desirable states labeled with the keyword *progress*. Thus, non-progress cycles are the dual of acceptance cycles in that they specify that “something good must eventually happen”, whereas acceptance cycles specify that “something bad cannot happen infinitely often”.

Both concepts are not directly supported by our TPTS model; in part to keep the model as simple as possible, but also because they are not compatible with the fact that in our model all temporal statements have a time horizon expressed with the “until”-operator. The “until”-operator, however, can be used to express time-limited versions of the livelock and non-progress properties.

In addition to *accept* and *progress*, SPIN uses the keyword *end* to identify “proper” end states (states without successors) of processes. If all processes are in such end states, the global state is not considered to be a deadlock even if there are no successor states. In contrast, our model assumes that all global states without successors are deadlocks. An end-state as in SPIN can be easily emulated, however, by letting a process cycle infinitely between two end states.

Unlike the logics employed by the approaches in table 3.1, which can express properties related to system *states* our logic EL can only be used to make statements about properties explicitly exposed as *events*. Combined with the model requirement that a process’ destination state can depend only on its own state and the received event (an not on the states of other processes), this has the advantage that all internal actions of processes are *atomic* by default, resulting in fewer transitions, and thus in a smaller state space.

## 3.6 Summary

Our approach allows the modelling of probabilistic systems at two levels: a low-level and a high-level. The low-level model — *timed probabilistic transition system* (TPTS) — is based on discrete-time Markov chains; our logic and our algorithms operate at this level. The high-level model — *probabilistic protocol* — is based on a variant of discrete event simulation used in parallel simulation tools and allows the modular specification of systems as a set of communicating processes. The semantics of the high-level model are given in terms of the low-level model.

Protocol properties can be expressed using special processes called *observers* or with *EL*, a linear time logic that can express properties of event sequences.

# Chapter 4

## Algorithms

### 4.1 Overview

This chapter describes our algorithms. The algorithms make use of bitstate hash tables similar to Holzmann's algorithm in SPIN, because the research literature [78] and the relative popularity of SPIN indicate that this approach is most effective for validation communication protocols (in contrast to hardware verification, where symbolic methods relying on BDDs are widely used).

There are three main algorithms described in this chapter. Section 4.2 describes the basic state space search algorithm that does not perform model-checking of EL formulas. Section 4.3 presents an algorithm for the model-checking of a TPTS with respect to an EL formula. Finally, in section 4.4, we show how to obtain an *approximate* TPTS from a protocol specification in order to deal with time intervals more efficiently. The algorithms in both section 4.2 and section 4.3 can be applied to the approximate TPTS.

### 4.2 State Space Search Algorithm without Model-Checking

This section describes the basic state space search algorithm without EL formulas. This basic algorithm can be used in cases when it is not necessary to check the validity of an EL formula. We first briefly describe Holzmann's algorithm, which is applicable when no limits are placed on the search depths. Then, we describe how to extend the algorithm to cases where the search is limited.

### 4.2.1 Unlimited Search

The algorithm for unlimited searches is a depth-first search of the protocol state space and is similar to the approach described by Holzmann [54, 55] in that the state space is constructed on-the-fly and that a hash table of bits is used to detect previously visited states.

- The *input* to the algorithm is a probabilistic protocol specified as a set of communicating processes
- The *output* of the algorithm is either
  - *OK* if no errors are detected during the search; or
  - a trace describing the sequence of events from the initial state up to the error state if an error is detected

The global state space is constructed on-the-fly using the probabilistic protocol-to-TPTS mapping described in section 3.3.2. Error states that can be detected in this fashion are those flagged by individual processes (including observers) or global error states such as deadlocks (states with no pending events).

A state search algorithm needs to detect when it encounters a state that has already been visited. When the state space is too large to be stored in memory, Holzmann's bitstate hashing algorithm can be used. The algorithm uses a large hash table of bits to mark visited states: whenever a state is visited, a hash value of the state is calculated. This value is used as an index to the hash table, and the bit at the indexed location is set to 1. Thus, the validation algorithm can detect previously visited states by checking if the corresponding bit in the hash table has been set. This algorithm greatly reduces the memory requirements of the state space search, but there is a possibility that two different states hash to the same value. No collision detection is performed and therefore, there is a small probability of mis-identifying a new state with one that was already visited. If the load factor of the hash-table is low, i.e., if the table is much larger than the number of states, the probability of missing a significant number of states is very low. Holzmann's algorithm uses two independent hash functions and two hash tables to further reduce the probability that

two different states map to the same hash values. The memory requirement of this algorithm is 2 bits per hash table entry.

### 4.2.2 Truncated Search

The bitstate hashing method is very efficient in that it requires only a few bits of storage per state. Nevertheless, the state space explosion problem means that industrial-sized protocols can often not be exhaustively validated, and the state space needs to be restricted in some way. When using the bitstate hashing method, the state space is naturally limited by the size of the hash table — the state graph is randomly truncated. Other ways to limit the state graph are to set a maximum depth threshold, a maximum time threshold, or a minimum probability threshold, which results in a truncated search.

- The *input* to the truncated search algorithm is a probabilistic protocol specified as a set of communicating processes, and a search threshold, given either as a maximum depth, a maximum time, or a minimum probability.
- The *output* of the algorithm is either
  - *OK* if no errors are detected during the search; or
  - a trace describing the sequence of events from the initial state up to the error state if an error is detected

We first discuss a state space search with a probability threshold. When using such a probability threshold, path probabilities are used as a heuristic to guide the state space search towards more probable regions of the state graph. The principle is simple: we set a small probability threshold (e.g.,  $10^{-10}$ ) and truncate a search path when the current path probability becomes smaller than the threshold.

The basic bitstate hashing method does not work for a probabilistic search. When the validation algorithm searches the event graph, it keeps track of the probability of the path from the root until the current node; we call this probability the *current probability*. A state can be visited several times on event paths with different current probabilities. For instance, assume that a state is first visited with a probability  $p$ , and then with probability  $q$ . Then, the second path through the state should not

be aborted if  $q > p$ , because the second path may extend deeper into the search graph before reaching the probability threshold.

One possible solution for this is to make the probability part of the state whose hash value is calculated. This would result in states always being classified as different if they are on event paths with different probabilities. In our model, we use a different solution: instead of using just one bit in the hash table to indicate that a state has been visited, we use  $n$  bits. The  $n$  bits indicate the largest current probability for a state that hashes to that hash table entry. Our implementation uses  $n = 8$  and stores the order of magnitude of the probability, without the minus sign, e.g.,  $p = 1.3 \times 10^{-22}$  becomes 22. Probabilities  $< 10^{-255}$  are stored as 255. A search path is aborted if the validator finds that the same state has already been visited with a higher current probability.

Time-limited and depth-limited searches can be handled in a similar manner. In both cases, two states can not always be classified as being identical if they occur at different times and depths, respectively. If a state is re-visited with a larger time (more time has elapsed since the initial state, i.e., there is less time left until the time threshold), it is safe to stop the current search path. Similarly, it is safe to abort a search path in a depth-limited search if it is revisited at a greater depth. The solutions we discussed for probabilistic searches directly apply to these cases.

### 4.3 State Space Search with Model-checking of Event Logic Formulas

We now describe an algorithm for calculating the measure of the event sequences of a TPTS  $M$  that satisfy an EL formula  $f$ . The basic approach is to search the tree of all paths of  $M$  and to remove paths that do *not* satisfy  $f$ . The measure of the removed paths is added to some global variable  $NS$  which is the output of the algorithm when it terminates. Like the model-checking algorithm in SPIN, we employ a nested depth-first search (DFS): an outer DFS that searches the global state transition tree and an inner DFS that is called by the outer DFS from each state that it visits.

This section is organized as follows. The inner DFS is described in subsection 4.3.1; the outer DFS in 4.3.2. We illustrate the combination of inner and outer DFS with

an example in subsection 4.3.3. A proof of the algorithm is in subsection 4.3.4. Some implementational details and a method for the optimized handling of cycles are discussed in 4.3.5 and 4.3.6, respectively. Finally, subsection 4.3.7 points out some issues related to the combination of the model-checking algorithm with truncated searches.

### 4.3.1 Inner Depth-First Search

The inner DFS is called by the outer DFS at every node of the global search tree. The inner DFS removes paths from the global search tree if they do not satisfy the EL formula  $f$ .

- The *input* to the inner DFS is a TPTS, a formula  $f$ , and a state  $g$  of the TPTS.
- The *output* of the algorithm is the measure of paths starting at  $g$  that were removed by it. The inner DFS modifies the global search tree in such a way that the removed paths are not revisited by the outer DFS.

From every state  $g$  that is visited by the outer DFS, the inner DFS determines for each infinite event sequence  $e = e_0 \xrightarrow{t_1} e_1 \xrightarrow{t_2} e_2 \dots$  starting at  $g$ , whether *the first event*  $e_0$  of  $e$  satisfies the (non-probabilistic) formula  $f$  or not. Note that since EL formulas are finite and since every “until” operator  $U$  has a finite time horizon, we can determine if  $e_0$  does not satisfy  $f$  by examining finite subsequences  $e'$  of  $e$  (section 3.2.3). If  $e_0$  does not satisfy  $f$  in the context of  $e'$ , the subtree rooted at the node of  $e'$  that follows the last (i.e., farthest from the root) branching point of  $e'$  is removed from the search tree.

Figure 4.1 shows an example of this subtree removal: node  $h$  is the last branching point of  $e'$ , so  $e'$  is removed starting at  $k$ , the node of  $e'$  that follows  $h$ . We can remove the entire subtree rooted in  $k$  because every infinite path that goes through  $k$  must contain the entire finite subpath  $e'$  and is therefore non-satisfying. The measure of the removed infinite paths is exactly the probability of the finite path from the root of the TPTS up to  $k$ , which can be easily calculated as the product of the transition probabilities of the path.

The inner DFS procedure returns the measure of all infinite paths that were removed in this fashion.

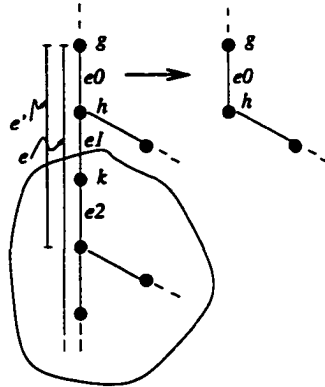


Figure 4.1: Example of Inner DFS

```

// Algorithm checkEL (inner DFS)
// global constants: TPTS  $M$ , formula  $f$ 
// returns the measure of all event sequences starting at  $g$ 
// whose first event does not satisfy  $f$ .
inner_dfs( $g$ ) {
     $n := 0$ ;
    for all event sequences  $e = e_0e_1e_2 \dots$  starting at  $g$ 
         $e' :=$  required finite subpath of  $e$ ;
        if  $e_0 \not\models_e f$ 
             $k :=$  successor of last branching point of  $e'$ ;
             $n+ = \mathcal{P}(k)$ ;
            remove subtree rooted in  $k$ ;
    return  $n$ ;
}

```

Figure 4.2: Algorithm for Checking EL Formulas, Inner DFS



### 4.3.2 Outer Depth-First Search

We now describe the outer DFS, which is our main model-checking algorithm. This part of the algorithm must ensure that every non-satisfying path in the tree is measured exactly once. For every state  $g$  in the search tree, we define the  $N(g)$  as the measure of infinite paths starting at  $g$  that do not satisfy  $f$ . The task, therefore, is to compute  $N(g_0)$ . In other words,

- The *input* to the outer DFS is a TPTS and formula  $f$ .
- The *output* of the algorithm is the measure of paths starting at the initial state  $g_0$  of the TPTS that do not satisfy  $f$ .

$N(g)$  is a real value  $\in [0, 1]$  and storing  $N(g)$  for all states is too expensive. Because of this, the algorithm stores only limited information about what is known about  $N(g)$  for a state  $g$ . This information can have one of four values:

- 0:  $N(g)$  is 0, i.e., all path starting at  $g$  satisfy  $f$
- 1:  $N(g)$  is 1, i.e., none of the paths starting at  $g$  satisfies  $f$
- 0/1: either all paths starting at  $g$  satisfy  $f$  or none of the paths starting at  $g$  satisfy  $f$
- UNKNOWN: it is not known whether any of the three cases above apply or not — this is the default situation for unexamined states

Storing this information requires 2 bits per visited state, we will refer to it as  $I(g)$ . In addition to this limited information about the  $N(g)$  of all states, the algorithm also stores the exact  $N(g)$  of the nodes on the current path from the root of the search tree.  $N(g)$  is initialized to 0 when  $g$  is first visited by the outer DFS.

The inner DFS procedure is started at each node that the outer DFS visits.  $N(g)$  is set to the value that the inner DFS procedure returns for a node  $g$  — the measure of paths that were removed by it. Then,  $N(g)$  is propagated backwards along the current path by updating  $N()$  for the nodes on the path (figure 4.3 left). If the path from the root  $g_0$  to  $g$  is  $g_0 \xrightarrow{e_0, t_0, p_0} g_1 \xrightarrow{e_1, t_1, p_1} \dots g_{n-1} \xrightarrow{e_n, t_n, p_n} g$ , then the value  $p_n N(g)$  is added to  $N(g_{n-1})$ , the value  $p_{n-1} p_n N(g)$  is added to  $N(g_{n-2})$ , and so on.

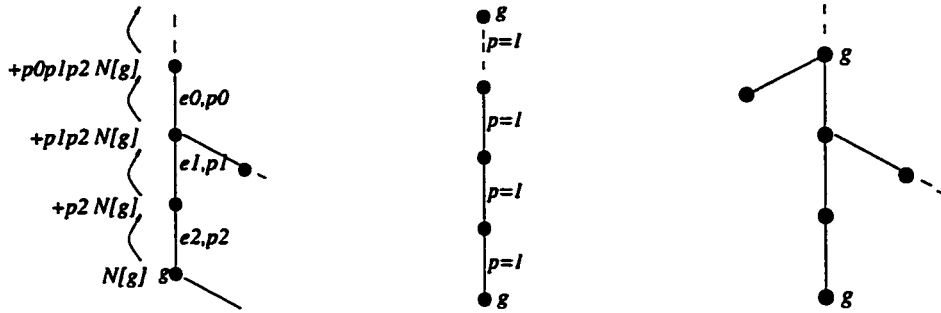


Figure 4.3: Propagation of  $N$ -values (left), Satisfying Cycle (middle), 0/1-Cycle (right)

This propagation of  $N()$ -values along the current path is necessary because any non-satisfying infinite path detected by the inner DFS that goes through  $g$  must also go through all other nodes on the current path, but its measure has not yet been added to their  $N()$ -values. To see why the adjustment factors are needed, consider that the measure of removed paths as returned by the inner DFS refers to the measure *relative to  $g$* . If, for example, the entire subtree rooted in  $g$  is removed by the inner DFS, the inner DFS would return 1, and  $N(g)$  would be set to 1. Relative to  $g_{n-1}$ , however, the measure of the removed subtree is not 1, but  $p_n$ , so the value  $p_n$  must be added to  $N(g_{n-1})$ .

A search path in the outer DFS is aborted if

1. the measure of non-satisfying paths starting at the current state  $g$  is known, i.e.,  $I(g) = 0$  or  $I(g) = 1$  (the first two cases above), or
2. the current state  $g$  has already been visited on the current path from the root.

In case 1, the value is propagated upwards (figure 4.3 left) just as if it had been computed again. Case 2 implies that a cycle has been detected. We must consider two subcases: (i) all transitions between the two identical states have probability 1 (figure 4.3 middle), and (ii) there is at least one transition between the two states whose probability is not equal to 1 (figure 4.3 right). Scenario (i) means that  $N(g)$  is 0, since no non-satisfying path can be reached from  $g$ , and  $I(g)$  is set to 0. In scenario (ii),  $N(g)$  can only be 0 (if no non-satisfying path can be reached from  $g$ ) or 1 (if at least one non-satisfying path can be reached from  $g$ , this means that each infinite path starting at  $g$  is non-satisfying with probability 1).  $I(g)$  is thus set to 0/1.



Figure 4.4: Outer DFS Scenario

If  $I(g)$  is 0/1 and  $N(g) > 0$  when the outer DFS has finished searching the subtree rooted in a state  $g$ , we can conclude that the final value of  $N(g)$  must be 1. The difference  $1 - N(g)$  must be propagated backwards along the path. It is important to note that  $I(g) = 0/1$  and  $N(g) = 0$  on the other hand does not imply that the final value of  $N(g) = 0$ . Figure 4.4 shows a situation where  $N(g) = 0$ ,  $I(g) = 0/1$  after the subtree rooted in  $g$  has been searched, but the final value of  $N(g)$  is not necessarily 0. This value can be either 0 if no non-satisfying path can be reached from state  $h$ , or 1 if at least one non-satisfying path can be reached from  $h$ . Note that the algorithm does not “miss” any non-satisfying paths — they will be detected during the search of the subtree rooted in  $h$ , and  $N(h)$  will be correctly set to 1.

The algorithm stops when the entire tree rooted in  $g_0$  has been searched.  $N(g_0)$  is then the measure of paths that do not satisfy  $f$ , and conversely,  $1 - N(g_0)$  is the probability that  $f$  is satisfied.

### 4.3.3 Example

We illustrate the algorithm with the simple 1-process juggler example from section 3.3.1. Figure 4.6 shows part of the depth-first search tree for the juggler example with  $n = 3$ . The property we want to check is  $f = \neg \text{BALL}_1$ , i.e., we claim that  $\text{BALL}_1$  is never sent. The probability of this being satisfied is, of course, 0. Figure 4.7 shows how the algorithm operates on this search tree.

In part (a), the outer DFS starts at the root node (named DFS1 in the figure) and descends to node DFS2 and then to node DFS3. From node DFS3, the inner DFS determines that the occurable event  $\text{BALL}_1$  does not satisfy  $f$ . The subtree rooted at DFS4 is removed by the inner DFS, and its measure — 1 — is propagated up the

```

// Algorithm checkEL (outer DFS)
// global constants: TPTS  $M$ , formula  $f$ 
// call as: outer_dfs( $g_0$ ,  $g_0$ );
//  $N[g_0]$  will contain the measure of paths that do not satisfy  $f$ .
outer_dfs( $g$ ,  $path$ ) {
    if  $I(g) = 0$  or  $I(g) = 1$  then
        propagate  $I(g)$ ;
        return;
    else if  $g$  visited earlier in  $path$  then
        if all probabilities = 1 then  $I(g) := 0$ ;
        else  $I(g) := 0/1$ ;
        return;
     $N[g] :=$  inner_dfs( $g$ );
    propagate  $N[g]$ ;
    for each state  $g'$  in  $M$  with  $p' = \mathcal{P}(g, g') > 0$ 
        outer_dfs( $g'$ ,  $path \xrightarrow{p'} g'$ )
    if  $I(g) = 0/1$  and  $N(g) > 0$  then
        propagate  $1 - N[g]$ ;
         $I(g) := 1$ ;
        return;
}

```

Figure 4.5: Algorithm for Checking EL Formulas, Outer DFS

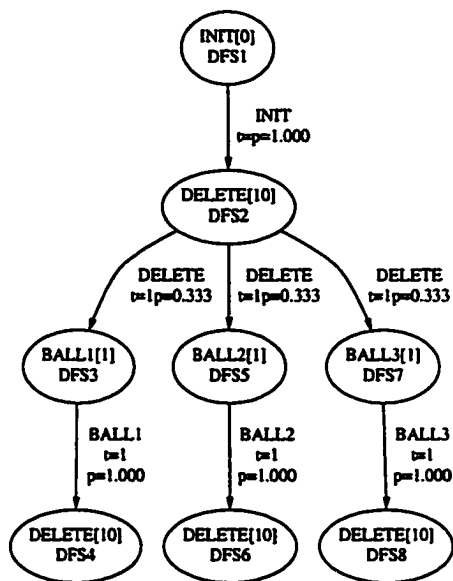


Figure 4.6: DFS Tree for Juggler Example, up to depth 3

path by the outer DFS. As a result,  $N(\text{DFS2}) = N(\text{DFS2}) := 0.333\dots$ . Since node DFS3 now has no more descendants, the outer DFS returns to DFS2 and descends to node DFS5 and node DFS6 (part (b)). At this point, the outer DFS detects that states DFS6 and DFS2 are identical. Not all transition probabilities between these states are 1, so  $I(\text{DFS2})$  is set to 0/1. The outer DFS aborts searching this path, backs up to node DFS2 and descends to nodes DFS7 and DFS8 (part (c)). As in part (b), it detects that DFS8 is identical to DFS2 and sets  $I(\text{DFS2})$  to 0/1 (which in this case is redundant). When the outer DFS returns to state DFS2 (part (d)), the entire tree rooted in DFS2 has been searched. Since  $N(\text{DFS2})$  is 0.333 and  $I(\text{DFS2})$ , the final value of  $N(\text{DFS2})$  must be 1. Therefore,  $N(\text{DFS2})$  is set to 1 and the difference between the old and new values (0.666..) is propagated up the path, resulting in  $N(\text{DFS1})$  being set to 1.

#### 4.3.4 Proof of Model-Checking Algorithm

Our algorithm works by successively removing sets of infinite paths (i.e., subtrees) from the search tree and adding the measure of each removed set to  $N(g_0)$ . It is not obvious that the combination of outer DFS and inner DFS correctly calculates the measure of *all* paths that do not satisfy  $f$ . Consider the scenario in figure 4.8, for instance: there is a non-satisfying path reachable from both  $g$  and  $h$ , but the ODFS

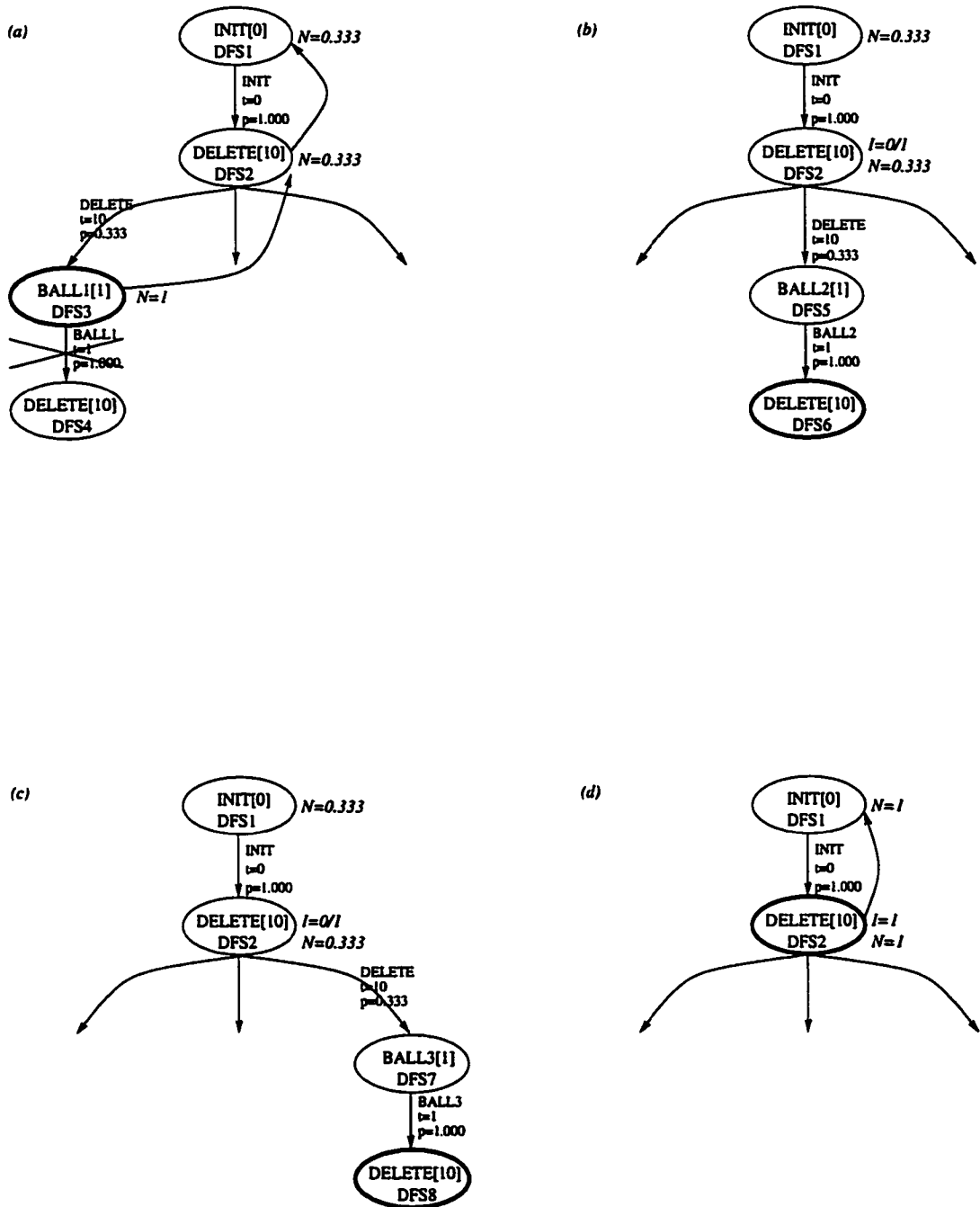


Figure 4.7: EL Model-Checking Example

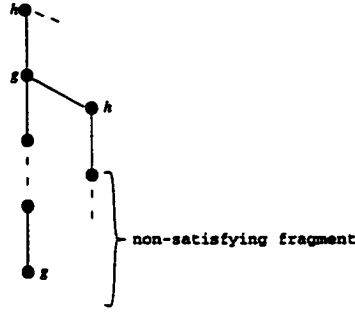


Figure 4.8: Algorithm Scenario

is aborted first at the lower  $g$  and then at the lower  $h$ . We need to show that the measure of the non-satisfying paths is somehow added to  $N(g_0)$ . More precisely, our aim is to prove the following

**Theorem 1** *For a TPTS  $M$  and an EL formula  $f$ , the model-checking algorithm removes a path  $p = g_0 \xrightarrow{e_0, t_0} g_1 \xrightarrow{e_1, t_1} \dots$  from the DFS search tree of  $M$  if and only if there are  $i, j$  such that  $e_i \not\models_{p_{i,j}} f$ , where  $p_{i,j} = g_i \xrightarrow{e_i, t_i} \dots \xrightarrow{e_{j-1}, t_{j-1}} g_j$  is a finite subpath of  $p$ .*

This section has three parts:

- first, we make the notion of “removing paths” more precise
- we then prove a simple lemma about the correctness of the  $I()$ -values as computed by the algorithm
- finally, we proceed with the actual proof of the theorem

*Note that throughout this section, we ignore any errors that may be introduced due to hash collisions, i.e., we assume that no hash collisions occur because the hash function is perfect and there is ample available memory.*

To make the notion of “removing paths” more precise, consider an infinite path  $p$  that is non-satisfying. By definition, there is  $i$  and  $j$  such that

$$p = g_0 \xrightarrow{e_0, t_0} g_1 \xrightarrow{e_1, t_1} \dots g_i \xrightarrow{e_i, t_i} \dots g_j \xrightarrow{e_j, t_j} \dots$$

and

$$p_{i,j} = g_i \xrightarrow{e_i, t_i} \dots \xrightarrow{e_{j-1}, t_{j-1}} g_j$$

is a finite subpath of  $p_{i,j}$  with  $e_i \not\vdash_{p_{i,j}} f$ . Only the finite fragment  $p_{i,j}$  is needed to determine that  $p \not\vdash f$ . Recall that a subtree of the global search tree *contains* an infinite path if the entire path except for a finite prefix is in the subtree. When we write “ $p_{i,j}$  is removed”, we mean that a subtree containing the set of infinite paths that contain  $p_{i,j}$  is removed from the global search tree, and its measure is added to  $N(g_0)$ . Similarly, when we write that an infinite path  $p$  is removed, we mean that a subtree containing  $p$  is removed. In our algorithm, a subtree rooted in a node  $g$  can be removed either by the IDFS (which does so by marking  $g$ ) or by the ODFS (which can set  $I(g) = 1$ ).

There is nothing to prove for the IDFS, the algorithm is straightforward and generates all paths starting at a given node  $g$  to find the ones on which the first event does not satisfy  $f$ . To show the correctness of our complete algorithm we first show that the  $I()$ -values calculated by the model are correct.

**Lemma 1** *If  $I(g) = 0$ , then all infinite paths starting at  $g$  satisfy  $f$ . If  $I(g) = 1$ , then the measure of infinite paths that start at  $g$  and do not satisfy  $f$  is 1.*

*Proof.* An inspection of the inner and outer DFS algorithms shows that  $I(g) = \text{UNKNOWN}$  initially for all  $g$ , and that  $I(g)$  can only be changed to 0 or 1 by the ODFS.

$I(g)$  can be set to 0 only if a satisfying cycle starting at  $g$  is detected such that all transitions of the cycle have probability 1. This means that an infinite path reaching  $g$  can not have a non-satisfying fragment, and  $I(g)$  must therefore be 0.

$I(g)$  can be set to 1 only if a cycle starting at  $g$  is detected and at least one non-satisfying fragment is reachable from  $g$ . The cycle is part of a strongly connected component (SCC), and because all SCCs are required to be bottom SCCs (BSCC, section 3.3), part of a BSCC. Thus, the non-satisfying fragment is part of the BSCC as well, and every infinite path reaching  $g$  reaches the non-satisfying fragment with probability 1. This means that all infinite paths going through  $g$  are non-satisfying.

□

We are now ready to proceed with the proof of theorem 1.

*Proof.* (i) We show that if there is a path  $p_{i,j} = g_i \xrightarrow{e_i, t_i} \dots g_{j-1} \xrightarrow{e_{j-1}, t_{j-1}} g_j$ , such that  $e_i \not\vdash_{p_{i,j}} f$ , then  $p_{i,j}$  (and thus  $p$ ) is removed by the algorithm. Our proof is by induction on the length  $i$  of the path from the initial state  $g_0$  to  $g_i$ . The base case



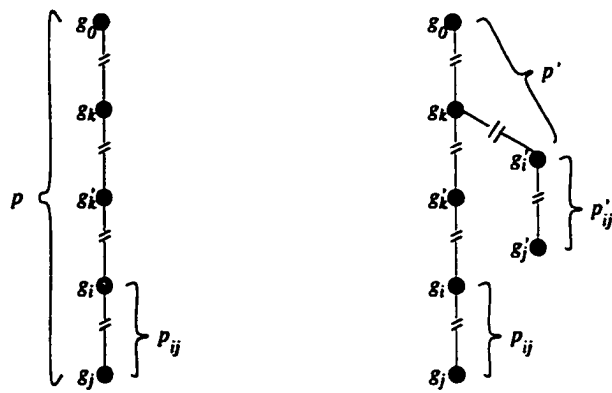


Figure 4.9: Proof of Algorithm, Part 1

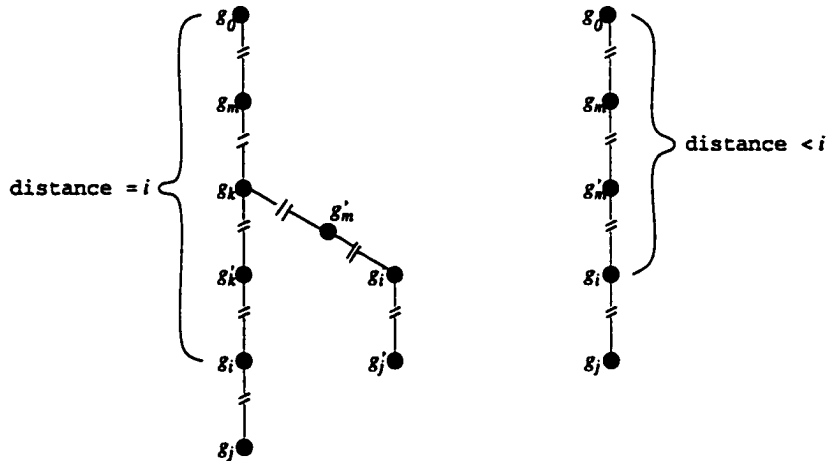


Figure 4.10: Proof of Algorithm, Part 2

$i = 0$  (i.e.,  $g_0 = g_i$ ) is obvious because the IDFS, if started at  $g_0$ , will detect that  $e_i \notin p_{i,j}$  and remove  $p_{i,j}$ .

*Induction Step.* If the ODFS visits  $g_i$ , then  $p_{i,j}$  will be removed by the IDFS. If the ODFS does not visit  $g_i$ , this can be for three reasons:

1. a subtree that contains  $p_{i,j}$  was already removed by the IDFS
2. a subtree that contains  $p_{i,j}$  was already removed by the ODFS
3. a cycle was detected by the ODFS in the path from  $g_0$  to  $g_i$ .

There is nothing to show in the first two cases. Case (3) is depicted in figure 4.9 left. The states  $g_k$  and  $g'_k$  are identical, indicating a cycle and preventing the ODFS from continuing the search at  $g'_k$ . Since  $g_i$  is reachable from  $g'_k$ , it is also reachable from

$g_k$  (shown as  $g'_i$  in figure 4.9 right) on another path  $p'$ . Now, if the ODFS visits  $g'_i$  on the other path, there is nothing left to show. Suppose that  $g'_i$  is *not* visited by the ODFS. As before, there are three possible reasons why the ODFS was aborted between  $g_k$  and  $g'_i$ :

1. a subtree that contains  $p'_{i,j}$  was already removed by the IDFS
2. a subtree that contains  $p'_{i,j}$  was already removed by the ODFS
3. a cycle was detected by the ODFS in the path from  $g_0$  to  $g'_i$ .

Again, there is nothing left to show in the first two cases. In case (3), we have the situation shown in figure 4.10 left and — slightly redrawn — in figure 4.10 right. This is the same situation as in figure 4.10 left, except that the distance from the root  $g_0$  to  $g'_i$  is less than  $i$ . By the induction assumption we know that a subtree containing  $p'_{i,j}$  is removed. The root  $g_m$  of this subtree is on the path from  $g_0$  to  $g'_i$ . If the  $g_m$  is below  $g_k$ , then  $I(g_k)$  will be set to 1 when the ODFS returns to  $g_k$ , and  $p_{i,j}$  will be removed. If, on the other hand,  $g_m$  is above  $g_k$ , then  $p_{i,j}$  is in  $g_m$ 's subtree and therefore removed.

(ii) We now show the other direction, i.e., if a path  $p$  is removed by the algorithm, then  $p$  has a finite subpath  $p_{i,j} = g_i \xrightarrow{e_i, t_i} \dots g_{j-1} \xrightarrow{e_{j-1}, t_{j-1}} g_j$ , such that  $e_i \notin_{p_{i,j}} f$ . We observe that paths can be removed either by the IDFS or by the ODFS. The IDFS is straightforward. As for the ODFS, it can remove  $p$  by setting  $I(g_k) = 1$  for some node  $g_k$  on  $p$ . By Lemma 1, this implies that all paths going through  $g_k$  — including  $p$  — are non-satisfying.  $\square$

### 4.3.5 Implementation

Like our basic search algorithm, the implementation of our EL model-checker makes heavy use of Holzmann's bitstate hash tables without collision detection. Two tables are used: one that stores the  $I()$ -values of all states, and another to mark the subtrees that were removed by the IDFS. Storing the  $I()$ -values takes two bits per state and is basically not different from the standard bitstate hashing method — a state's hash value is used to index a two-bit entry in the hash table.

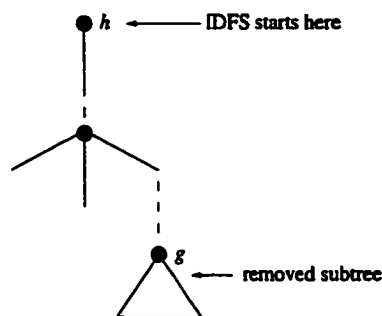


Figure 4.11: Removal of a Subtree

Using a hash table to mark removed subtrees requires a little more thought. An important realization is that when a subtree rooted in a state  $g$  is removed by the inner DFS, then it is only removed from *that particular location* in the global search tree. Figure 4.11 illustrates why this is the case. In the figure, the IDFS starts at node  $h$  and removes the subtree rooted in  $g$ . The reason for this removal is that any path that goes through  $h$  and  $g$  is non-satisfying. However, if the state  $g$  re-occurs elsewhere in the search tree there is no reason to believe that it is preceded by a state  $h$  in the same way it is at this location in the search tree, and the subtree rooted in that  $g$  must still be searched.

We solve this problem by computing the hash values for the roots of removed subtrees using the entire path from the global root to the root of the removed subtree — this ensures that identical global states will hash to different locations (with a residual probability of error). The bit corresponding to the hash value is set to 1 indicating that the subtree has been removed and should not be searched. A simple trick can make this removal of subtrees much more reliable: when the outer DFS encounters a marked subtree and backs off, the subtree is *un-marked*. This is possible because the marker is no longer needed - the outer DFS will never be in this part of the search tree again. However, removing unnecessary markers like this greatly reduces the probability of an un-marked subtree being recognized as a marked subtree.

### 4.3.6 Optimized Handling of Cycles

The model-checking algorithm in the previous section is designed to make efficient use of memory by storing only a minimum amount of information per visited state. The cost of this is that states often need to be visited several times, thereby increasing the

search time. This increased search time is especially significant in the case of cycles. Consider for example the TPTS sketched in figure 4.12 (left). States  $g, h, k$  of the TPTS are on a cycle, and there are two paths  $p_1, p_2$  from  $h$  to  $k$ . The corresponding search tree is in figure 4.12 (right). The DFS starting at  $g$  first takes path  $p_1$  from  $h$  to  $k$  and goes on to reach state  $g$ , which results in the cycle being detected, and  $I(g)$  is set to 0/1. Then, the DFS takes path  $p_2$  from state  $h$  and reaches state  $k$ . In its basic form, the model-checking algorithm above would continue searching the path from  $k$  to  $g$  for a second time because the search path can only be aborted at a state  $k$  if  $N(k)$  is known or if  $k$  has already been visited on the current path from the root  $g$ .

We propose a simple optimization that reduces the search time in such cases at the expense of increased memory use. The idea is based on the observation that once a cycle has been detected, the states that are part of the cycle do not need to be visited again *as long as the outer DFS is still in the subtree that contains the cycle*. In figure 4.12, this means that any state that is on the path between the two  $g$ 's need not be visited again while the outer DFS is in  $g$ 's subtree: any such visit would again either lead to another state  $g$  or stop before reaching another state  $g$ . To implement this optimization, we simply mark all states between the two  $g$ 's by setting their  $I()$ -values to 0. Once  $g$ 's subtree has been searched, the  $I()$ -values of the states of the cycle are either set to 1 (if  $I(g) = 1$ ) or reset to UNKNOWN. The marking of the cycle with  $I() = 0$  can be done by traversing the current path backwards; in order to be able to unmark the cycle after searching the subtree rooted in  $g$ , however, we need to save it in some data structure. This results in increased memory use, especially when the cycles are very long or when there are many cycles in the same subtree. In our experiments (chapter 6), the benefits of the optimization generally outweighed the costs by a large factor.

### 4.3.7 Combining Model-Checking with Truncated Searches

Section 4.2.2 describes how the basic state space search (without model-checking) can be limited by depth, time or probability. To combine model-checking with such truncated searches, we first need to examine what the “measure of satisfying paths” means in the context of truncated searches. All paths in a truncated search are finite and have a probability which is the product of all transition probabilities along the

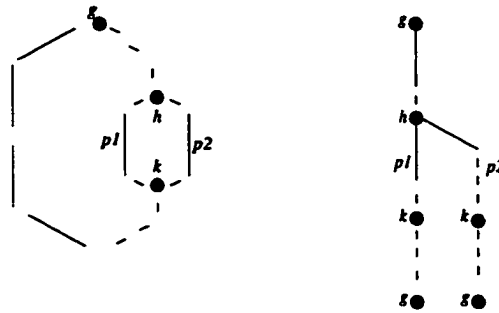


Figure 4.12: Cycle Handling Scenario

path. The measure of satisfying paths is the sum of the probabilities of all paths that satisfy an EL formula  $f$ . So, in the case of a time-limited search with threshold  $t$ , the measure of satisfying paths is the measure of all paths with time-length  $\leq t$  that satisfy  $f$ .

Using our model-checking algorithm in the limited search case presents us with problems because the algorithm assumes that all paths are infinite — this assumption is used in the way that cycles are handled. Therefore, when performing a truncated state space search, the model-checking algorithm must disable cycle-detection and search paths until the limit (given by depth, time, or probability) is reached.

We now consider the time-limited search as an example; the probability-limited and depth limited searches are analogous. Since the search tree a protocol is typically exponential in  $t$ , simply searching the entire tree is inefficient, and we want to avoid re-visiting states as much as possible. In the time-limited search without model-checking, an array, indexed by the hash value of a state, indicates the smallest  $t$  for which a subtree rooted in that state has already been searched for errors. If the state is re-visited with a greater  $t$  than that given by the array (indicating that the current state is lower in the search tree), the state need not be re-visited, because the subtree rooted in the current state is included in the subtree that was already searched. If, on the other hand, the state is re-visited with a smaller  $t$  than that given by the array (indicating that the current state is higher in the search tree), then the state must be re-visited because the search tree rooted in the current state is larger than the one searched before.

To extend this approach to model-checking a formula  $f$ , we must consider how the depth of a state affects its  $I(\cdot)$ -value. Recall that the  $I(g)$  for a state  $g$  is either 0

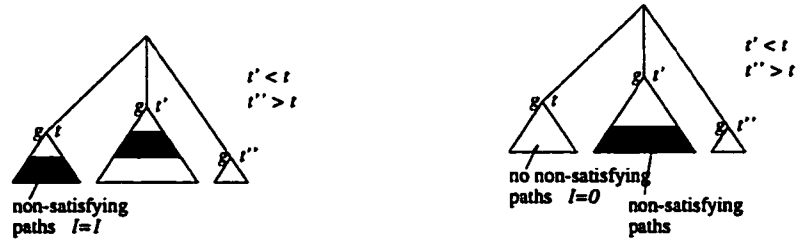


Figure 4.13: Re-visiting States in Limited Searches

	time-limited	probability-limited	depth-limited
$I = 0$	smallest $t$	largest $p$	smallest $d$
$I = 1$	largest $t$	smallest $p$	largest $d$

Table 4.1: Hash Array Contents for Limited Search

(meaning that all paths starting at  $g$  satisfy  $f$ ), 1 (meaning that none of the paths starting at  $g$  satisfy  $f$ ), 0/1 (meaning that  $g$  is part of a cycle), or UNKNOWN (the default).

Now, in a time-limited search, assume that the subtree rooted in  $g$  is searched, all paths starting at  $g$  are found to be non-satisfying, and  $I(g)$  is set to 1 as a result. The situation then is similar to the case without model-checking: if the state  $g$  is re-visited with a lower  $t$  (higher in the tree), then the subtree need not be searched again because all paths starting at this  $g$  will inevitably be found to be non-satisfying as well. If the state  $g$  is re-visited with a higher  $t$  (lower in the tree), the subtree needs to be searched again because paths starting at the new  $g$  are shorter and are not necessarily non-satisfying. Figure 4.13 (left) illustrates this reasoning.

This observation needs to be reversed for the case  $I(g) = 0$ : if  $g$  is re-visited with a lower  $t$  (higher in the search tree), then the subtree in  $g$  needs to be searched again; and if  $g$  is re-visited with a higher  $t$  (lower in the search tree), then the subtree need not be searched again (figure 4.13 right). Thus, our hash array needs to indicate for each  $g$  the “best” position for which a subtree rooted in  $g$  has been searched already; this can be either the smallest  $t$  (if  $I(g) = 0$ ) or the largest  $t$  (if  $I(g) = 1$ ). Table 4.1 summarizes what the array’s contents for the different types of limited searches.

## 4.4 Approximate Algorithm: Time Intervals

Recall that in our process model, nondeterminism is expressed by the output function  $O : T \rightarrow 2^{2^Y}$  of a process which assigns a set of timed output event sets to each transition  $t \in T$ , and by the probability function  $P : T \times 2^Y \rightarrow [0, 1]$  which assigns a probability to each output event set of a transition.

A very common form of nondeterminism in protocol specifications is that of time nondeterminism: an event is specified to occur not with a deterministic delay but with lower and upper delay bounds. In terms of process transitions, this type of nondeterminism is specified with output event sets containing single events that have identical event types and whose delay values form an interval of  $\mathbb{N}$ . We call such an event set  $\{e = (y, t) \mid t \in \mathbb{N}^0 \wedge t_{min} \leq t \leq t_{max}\}$  an *event interval*. It can be compactly represented as a pair  $E = (y, [t_{min}, t_{max}])$ , meaning that an event of type  $y$  will occur with a delay of between  $t_{min}$  and  $t_{max}$ , inclusive. We define  $\text{type}(E) := y$ ,  $\text{start}(E) := t_{min}$ ,  $\text{end}(E) := t_{max}$ .

When the event intervals are large relative to the indivisible time units, the TPTS obtained by directly applying the definition in section 3.3 becomes very large, and model-checking becomes very expensive. In the basic algorithm, the number of transitions from a state is equal to the number of time units in the event interval. In this section, we discuss a method of dealing with event intervals more efficiently. The description is closely based on the TCSM model from [56, 62], except for our method of assigning probabilities to transitions. For a protocol  $(M, Z)$ , we describe a way of obtaining a TPTS that can be smaller than that obtained by the applying the definition in section 3.3. Let  $M = \{M_1, \dots, M_n\}$ ,  $M_i = \langle S_i, s_{0,i}, T_i, N_i, O_i, P_i \rangle$ , be a set of processes, and  $Z = \{z_1, \dots, z_m\}$  be the initial events. First, we define the notion of a *global state interval*:

**Definition 4 (Global State Interval)** A Global State Interval (GSI)  $g$  consists of the individual process states  $s_i \in S_i$  and a set  $E$  of event intervals (the event interval list):  $g = \langle s_1, \dots, s_n, \{E_1, \dots, E_r\} \rangle$ .

The idea is to construct a TPTS whose nodes are global state intervals (as opposed to global states in the basic algorithm). Instead of executing events, we execute

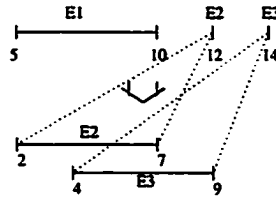


Figure 4.14: Recalculation of Event Delays

event intervals and subtract the time of the executed interval  $T'$  from the times of each remaining interval  $T$  using the interval subtraction operator  $\ominus$ :  $T \ominus T' := [\max(t'_{min} - t_{max}, 0), \max(t'_{max} - t_{min}, 0)]$ . The difference between the two intervals represents the minimum and maximum possible delay between the executed event and the remaining event. If the output events resulting from the execution of the event form an event interval, the new event interval is added to the list of event intervals; if not, the algorithm proceeds like the basic state space search.

The list of event intervals alone is not sufficient to preserve the timing constraints between events, as figure 4.14 illustrates. In this example, an event  $E_1$  is scheduled to occur with a delay of between 5 and 10 time units. Events  $E_2$  and  $E_3$  are scheduled to occur with delays of 12 and 14 time units, respectively. When event  $E_1$  is executed, the delays of events  $E_2$  and  $E_3$  are recalculated to be  $12 \ominus [5, 10] = [2, 7]$  and  $14 \ominus [5, 10] = [4, 9]$ , respectively. Based on these intervals, it would be possible for event  $E_3$  to occur *before* event  $E_2$ , something that was not possible before event  $E_2$  was executed. To preserve timing constraints between events, a *timing constraint matrix* (TCM) is used in [56, 62]. The timing constraint matrix contains the timing constraints between any two events in the event list: entry  $(i, j)$  of the matrix is  $\text{start}(E_j) - \text{end}(E_i)$ . Thus, entry  $(i, j)$  can be interpreted as the minimum number of time units that must elapse between events  $i$  and  $j$ , and entry  $-(j, i) = -(\text{start}(E_i) - \text{end}(E_j)) = (\text{end}(E_j) - \text{start}(E_i))$  can be interpreted as the maximum number of time units that can elapse between events  $i$  and  $j$ . An event  $E_j$  can not occur as long as there is an  $i$  such that entry  $(i, j)$  is positive (event  $E_j$  is constrained from occurring).

The *mature event* of an event list is an event interval with the earliest end time. Two event intervals *overlap* if either event can occur first. The *occurable event intervals* are the event intervals that overlap with the mature event and are not constrained by the TCM. We now construct a TPTS  $(G, G_0, \mathcal{P}, \mathcal{T}, \mathcal{E})$  that uses these concepts. This



TPTS departs from definition 1 in that transitions can be labeled with time intervals instead of single time values. In terms of the model-checking algorithms, this is not a significant change because a transition labeled with a time interval can be treated as a set of transitions, each labeled with one element of the time interval.

The states  $(g, c) \in G$  of the TPTS consist of a GSI  $g$  and a TCM  $c$ . The initial GSI  $g_0$  is  $\langle s_{0,1}, \dots, s_{0,n}, \{\langle z_1, 0 \rangle, \dots, \langle z_m, 0 \rangle\} \rangle$ : all processes are in their initial states and the event interval list is the list of the initial events scheduled at time 0. The initial TCM  $c_0$  is an  $m \times m$  matrix filled with zeros (since all events are scheduled at time zero, the minimum and maximum time between any two events is 0). We show how  $\mathcal{T}$  and  $\mathcal{E}$  are obtained. Let

- $(g, c)$ ,  $g = \langle s_1, \dots, s_n, \{E_1, \dots, E_m\} \rangle$ ,  $c = (c)_{ij}$  be a state of the TPTS,
- $E_i = (y_i, [t_{i,min}, t_{i,max}])$  be an occurable event interval in  $(g, c)$ ,
- $M_j = \langle S_j, s_{0,j}, T_j, N_j, O_j, P_j \rangle$  be the destination process of  $E_i$ ,
- $t = (s_j, y_i) \in T_j$  ( $y_i$  be an occurable event type in state  $s_j$  of process  $M_j$ ),  $N(t) = s'_j$  ( $s'_j$  is the destination state), and  $O_j(t) = \{A_1, \dots, A_l\}$  (the output event sets).

Then, a successor GSI  $g'$  and a successor TCM  $c'$  are constructed as follows:

1. Truncate the time of the executed event  $E_i$  to the end time of the mature event if the priority of  $E_i$  greater or equal to the priority of the mature event, and truncate the end time of  $E_i$  to one time unit before the end time of the mature event if the priority of  $E_i$  less than the priority of the mature event. This is because the time of the executed event can not be later than the end time of the mature event. If the priority of the executed event  $E_i$  is less than that of the mature event, then it can not occur at the end time of the mature event.
2. Recalculate the time intervals of the remaining events. To calculate the minimum time  $s$  between the executed event  $E_i$  and a remaining event  $E_j$ , consider that

- $s \geq 0$  if the priority of  $E_i$  is greater than the priority of  $E_j$  and  $s \geq 1$  if the priority of  $E_i$  is less than the priority of  $E_j$  (if  $E_i$  has a lower priority than  $E_j$  and  $E_i$  occurs first, then  $E_j$  can not occur at the same time as  $E_i$ ).
- $s \geq \text{start}(E_j) - \text{end}(E_i)$  if  $\text{start}(E_j) > \text{end}(E_i)$ .
- $s \geq c[i, j]$ ;

Thus,  $s = \max(0, \text{start}(E_j) - \text{end}(E_i), c[i, j])$  if the priority of  $E_i$  is greater than the priority of  $E_j$  and  $s = \max(1, \text{start}(E_j) - \text{end}(E_i), c[i, j])$  if the priority of  $E_i$  is less than the priority of  $E_j$ . To calculate the maximum time  $e$  between the executed event  $E_i$  and a remaining event  $E_j$ , consider that

- $e \geq s$
- $e \leq \text{end}(E_j) - \text{start}(E_i)$
- $e \leq -c[j, i]$

Thus,  $e = \max(s, \min(\text{end}(E_j) - \text{start}(E_i), -c[j, i]))$ .

3. Remove  $E_i$  from the event list, and remove row  $i$  and column  $i$  from the TCM  $c$ . If the set of output events  $O_j(t) = \{A_1, \dots, A_l\}$  is an event interval  $A$ , add the new event interval to the event list and calculate the new TCM entries. If  $O_j(t)$  is not an event interval there are  $l$  reachable GSIs, each containing the events of one  $A_k$ ,  $k \in 1, \dots, l$ , and a TCM entry for each new event  $\in A_k$ .
4. Replace  $s_j$  with  $s'_j$

We set  $\mathcal{T}((g, c), (g', c')) = [t_{i, \min}, t_{i, \max}]$ , and  $\mathcal{E}((g, c), (g', c')) = y_i$ . To complete the construction of the TPTS, we need to define a probability distribution function  $\mathcal{P}$ , i.e., we need to assign a probability  $\mathcal{P}((g, c), (g', c'))$  to each transition of the TPTS. Since each node  $(g, c)$  of the TPTS represents a set of protocol global states, it is reasonable to use the following general definition:

$$P(E_i) = \frac{|\{\text{no. of global states } \in (g, c) \text{ in which } E_i \text{ occurs first}\}|}{|(g, c)|} \quad (4.1)$$

In other words, the probability of an event interval of type  $y$  being executed from a node  $(g, c)$  is proportional to the number of global states in  $(g, c)$  in which an event

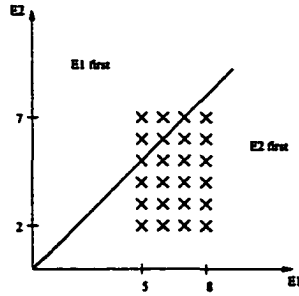


Figure 4.15: Event Lists and Event Intervals

of type  $y$  occurs first. Figure 4.15 shows an example event interval list with two event intervals  $E_1$  and  $E_2$ . This event interval list represents  $4 \times 6 = 24$  event lists (marked with 'x' in the figure) with two events each, and if we assume that the events in the event intervals have equal probability, then all 24 event lists have the same probability. The event of type  $type(E_1)$  occurs first in the event lists above the diagonal, and the event of type  $type(E_2)$  occurs first in the event lists below the diagonal. The sequence of events for the event lists on the diagonal depends on the relative priorities of  $E_1$  and  $E_2$ ; here, we assume that  $E_1$  has a higher priority than  $E_2$ , and therefore  $E_1$  occurs first in the event lists on the diagonal. The probability of  $E_1$  occurring first, therefore, is  $\frac{6}{24} = \frac{1}{4}$ .

If only one node  $(g', c')$  is reachable by executing  $E_i$ , then

$$\mathcal{P}((g, c), (g', c')) = P(E_i)$$

If the set of output event sets  $O_j(i) = A_1, \dots, A_l$  obtained by executing  $E_i$  is not an event interval, then  $l$  nodes  $(g'_k, c'_k)$ ,  $k \in \{1, \dots, l\}$ , are reachable by executing  $E_i$ . The probability of reaching node  $(g'_k, c'_k)$  that contains the new events from event set  $A_k$  is

$$\mathcal{P}((g, c), (g'_k, c'_k)) = P(E_i) \times P_j(t, A_k)$$

In assigning probabilities to transitions, we make several assumptions that simplify the computation at the expense of accuracy. Our basic simplifying premise is that all events in an event interval have the same probability, i.e., the probability distribution in an interval is uniform. This is only an approximation, even if the probability distribution in event intervals is always uniform when they are generated by processes: if we generate a successor GSI by executing the event interval  $E_1$  in the GSI of

figure 4.15, the successor GSI must contain those event lists in which  $E_1$  occurs first, and the times of event  $E_2$  in the successor GSI range from 5 to 7. However, the event probabilities in the interval  $[5, 7]$  are no longer uniformly distributed because there are, for instance, 3 event lists in which  $E_2$  occurs with delay 7 and only 1 event list in which  $E_2$  occurs with delay 5.

To further simplify the calculation of transition probabilities, we ignore the effect that the TCM  $c$  has on the set of global states represented by  $(g, c)$ . Thus, we have

$$P(E_i) \approx \frac{|\{\text{no. of event lists} \in E_1 \times E_2 \times \dots E_m \text{ in which } E_i \text{ occurs first}\}|}{|E_1 \times E_2 \times \dots E_m|}$$

Assume, without loss of generality, that the event intervals are indexed in decreasing order of priority, and set  $E_{i,l} = 1$  if  $start(E_i) \leq l \leq end(E_i)$  and 0 otherwise. If the executed event  $E_i$  occurs at time  $k$ , then the number of possible delays that a remaining event  $E_j$  can have is

$$\sum_{l=k+1}^{end(E_j)} E_{j,l}$$

if  $j < i$ , i.e., the priority of  $E_j$  is greater than that of  $E_i$ , and

$$\sum_{l=k}^{end(E_j)} E_{j,l}$$

if  $j > i$ , i.e., the priority of  $E_j$  is less than that of  $E_i$ . Thus the number of global states in which  $E_i$  occurs first and occurs at time  $k$  is (ignoring the effect of the TCM  $c$ )

$$\prod_{j=1}^{i-1} \left( \sum_{l=k+1}^{end(E_j)} E_{j,l} \right) \prod_{j=i+1}^m \left( \sum_{l=k}^{end(E_j)} E_{j,l} \right)$$

Because  $k$  ranges from  $start(E_i)$  to  $end(E_i)$ , the probability of event  $E_i$  occurring first is approximately

$$P(E_i) \approx \frac{\sum_{k=start(E_i)}^{end(E_i)} \prod_{j=1}^{i-1} \left( \sum_{l=k+1}^{end(E_j)} E_{j,l} \right) \prod_{j=i+1}^m \left( \sum_{l=k}^{end(E_j)} E_{j,l} \right)}{|E_1 \times E_2 \times \dots E_m|}$$

We use a simple lossy channel example to illustrate the event interval algorithm. The protocol has three processes, a *sender*, a *channel* and a *receiver*. Following an  $INIT_S$  event, the sender transmits a message  $MSG_C$  to the channel with a delay of 8 to 12 time units. The channel has two states: “good” and “bad”. It starts in the good state and alternates between the two states, spending from 5 to 10 time units in each state;

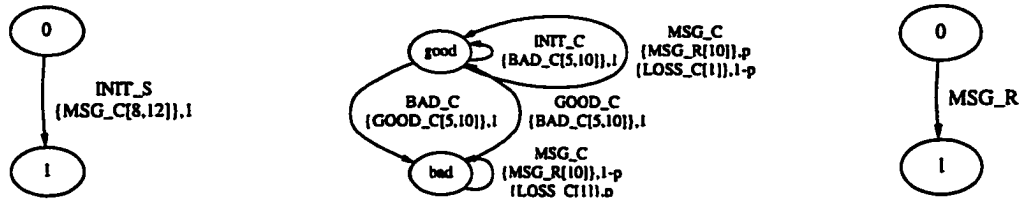


Figure 4.16: Sender (left), Channel (middle), Receiver (right) Processes

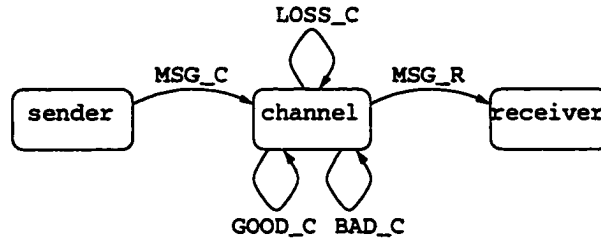


Figure 4.17: Event Exchange Diagram for Lossy Channel Protocol

events  $GOOD_C$  and  $BAD_C$  trigger the state changes. When the channel receives a  $MSG_C$  event while in the good state, it sends a  $MSG_R$  event to the receiver with delay 10 and probability 0.9 (indicating a successful transmission), and a  $LOSS_C$  event to itself with delay 1 and probability 0.1 (indicating a message loss). These probabilities are reversed when the channel is in a bad state — a message is lost with probability 0.9 and successfully transmitted with probability 0.1. The state transition and event exchange diagrams for the protocol are in figures 4.16 and 4.17, respectively.

After the execution of the two initial events, the event interval list contains the two events  $MSG_C$  and  $BAD_C$  scheduled with delays [8,12] and [10,15], respectively, and the channel is in its good state. The events and the timing constraint matrix are displayed in figure 4.18. In this figure, we use shortcuts to compactly represent the process states and the TCM. The states of the three processes are represented as a three-digit number indicating the state of the sender, channel and receiver. The rows and columns of the TCM are labeled using the event priorities which are assigned as follows:  $INIT_S$  — 0,  $INIT_C$  — 1,  $MSG_C$  — 2,  $MSG_R$  — 3,  $GOOD_C$  — 4,  $BAD_C$  — 5,  $LOSS_C$  — 7.

Both event intervals in figure 4.18 are occurable. The two event intervals represent  $(12-8+1) \times (15-10+1) = 30$  possible event lists (in fact, these event lists are exactly those displayed in figure 4.15). Among these event lists, there are 6 in which event  $BAD_C$  occurs first, and 24 in which  $MSG_C$  occurs first. Thus,  $P(BAD_C) = \frac{6}{30} = 0.2$

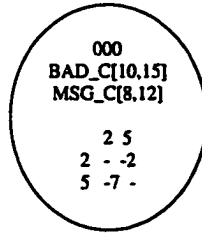


Figure 4.18: A Global State of the Lossy Channel Protocol

and  $P(\text{MSG}_C) = \frac{24}{30} = 0.8$ . When event  $\text{MSG}_C$  is received by the channel, it can generate one of two events:  $\text{MSG}_R$  with probability 0.9 and  $\text{LOSS}_C$  with probability 0.1. There are therefore three reachable GSIs. The entire TPTS for the protocol is displayed in figure 4.19.

## 4.5 Summary

Our approach allows the modelling of probabilistic systems at two levels: a low-level and a high-level. The low-level model — *timed probabilistic transition system* (TPTS) — is based on discrete-time Markov chains; the algorithms described in this section operate at this level.

The three main algorithms are a plain state space search similar to Holzmann’s algorithm in SPIN, a novel algorithm for calculating the satisfaction probability of EL formulas with respect to a TPTS, and an approximate algorithm for efficiently dealing with large time intervals. The algorithms make heavy use of hash tables.

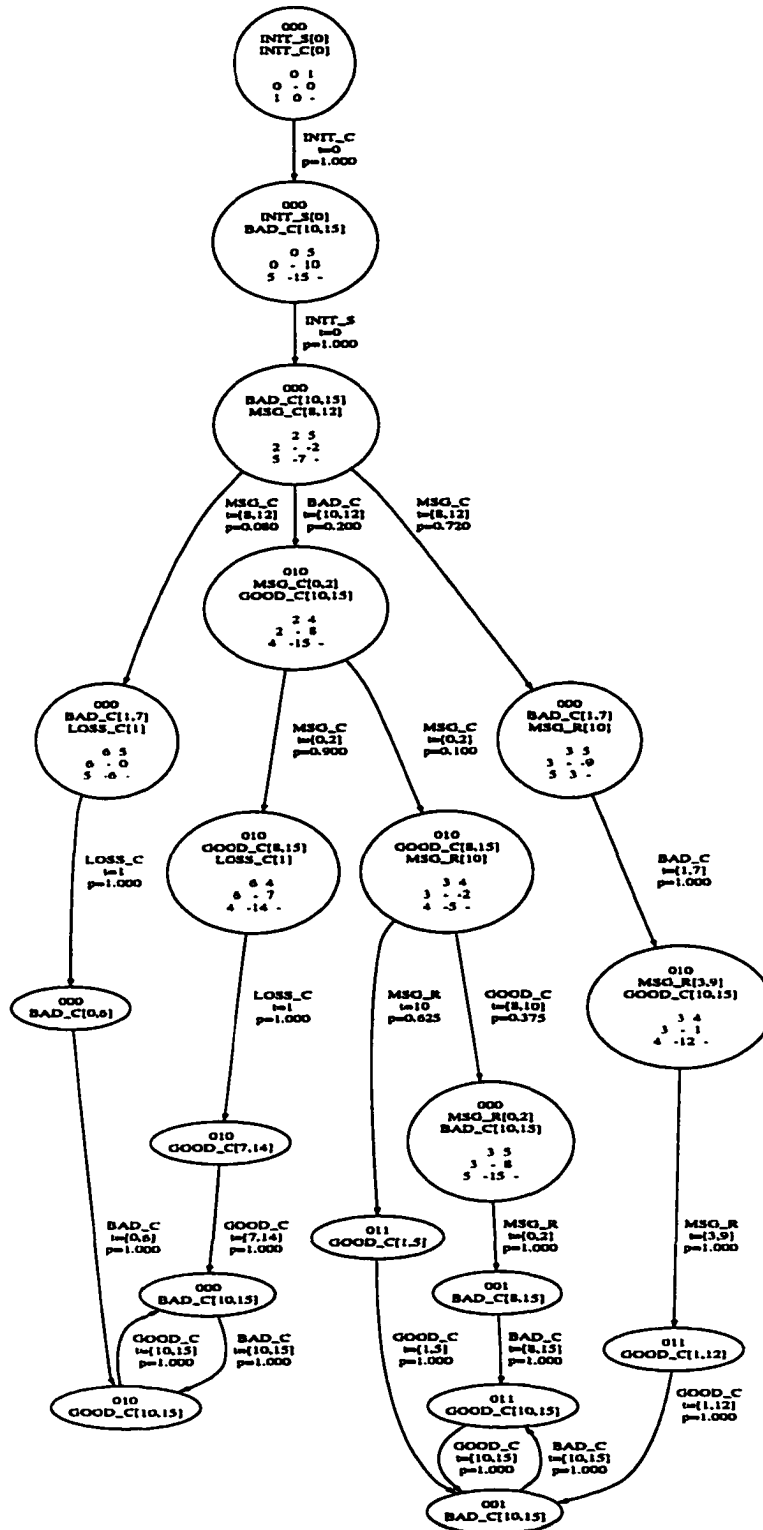


Figure 4.19: TPTS for Lossy Channel Protocol

# Chapter 5

## The C++ Library

### 5.1 Overview

We have implemented a prototype validator/simulator as a C++ library similar to SimKit [44] and SMURPH. The basic steps that protocol designers need perform in order to write a protocol specification using our library are

- define and implement appropriate subclasses of `val_process`. An important part of this is the definition of a `perform` function for each process type — this function implements the successor relation for the process
- write a global `root()` function that instantiates the protocol components and creates the initial events
- write a global `report()` function that prints out a final report for simulation runs
- write an EL formula and compile it into C++ using the EL-to-C++ compiler `e12cc`

The resulting C++ source files are compiled and linked with the validator/simulator library `libvalis.a` to yield the validator executable (figure 5.1). This executable can be run in validation mode or in simulation mode, depending on the command line parameters. This section is organized as follows: section 5.2 describes the C++ class interfaces that a protocol designer needs to understand in order to write specifications using the library. Section 5.3 explains the two user functions `root` and `report`. The EL-to-C++ compiler and its use are described in section 5.4. Finally, section 5.5 lists



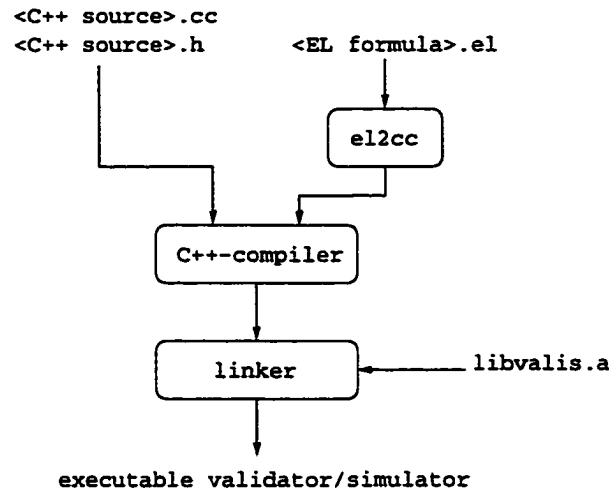


Figure 5.1: Building a Validator

and explains the command line parameters that influence the run-time behaviour of the validator executable.

## 5.2 User Classes

### 5.2.1 val\_time

The `val_time` class is used to represent time and time intervals in the model. The main methods of this class are

- `val_time()` is the constructor; there are three overloaded versions of this method:
  - `val_time(void)`, which instantiates an object with both start and end time 0;
  - `val_time(int t)`, which instantiates an object with start and end time `t`; and
  - `val_time(int s, int e)` which instantiates an object with start time `s` and end time `e`.
- `int start()` returns the start time of the time interval.
- `int end()` returns the end time of the time interval.

The stream operators `<<` and `>>` can be used to output time intervals. The operators `+` and `-` can be used to perform addition and subtraction of time intervals, respectively.

## 5.2.2 val\_event

This is the class that represents events in the model. The main methods of this class are

- `val_event(int type, val_process *dest, val_time time, int dist=VAL_UNIFORM)` is the constructor. The are parameters are
  - `int type`: the event type
  - `val_process *dest`: the destination process of the event
  - `val_time time`: the delay of the event
  - `int dist`: this parameter indicates the probability distribution of the values in the interval. Possible values are `VAL_UNIFORM` (the default) and `VAL_EXP`.
- `int type()` returns the event type.
- `val_process *dest()` returns the destination process.
- `int dist()` returns the probability distribution.

Event objects are created either at setup time (in the `root()` function or by processes in response to other events. They are then passed to the kernel using the `add_event()` and `newevents()` functions, respectively. The kernel then may treat the event as more than one event if:

- The event time is a time interval and the approximate event interval processing algorithm (section 4.4) is not enabled. In this case, the only supported distribution is `VAL_UNIFORM`, and the event is treated internally as  $n$  different events (where  $n$  is the number of values in the interval), each with probability  $\frac{1}{n}$ .
- The event time is a single value, and the probability distribution is `VAL_EXP`. In this case, the given time value is taken to be the mean  $\frac{1}{\lambda}$  of the exponential random variable, and the exponential distribution  $f(t) = \lambda e^{-\lambda t}$  is approximated by a finite number  $m$  of support points. If  $m = 1$  ( $m$  is given by the command line parameter `-expapprox`), then there is only one support point at  $t = \frac{1}{\lambda}$ , and

the probability of this point is 1. If  $m > 1$ , then the time interval  $[0, \infty]$  is partitioned as follows:

- the  $m$ -th interval is  $[t_c, \infty]$ , where  $t_c$  is chosen so that  $f(t_c) = y$  ( $y$  is given by the command line parameter `-maxexp` and defaults to 0.01).
- the interval  $[0, t_c]$  is divided into  $m - 1$  equal segments to obtain the first  $m - 1$  partitions.

Figure 5.2 illustrates the procedure. For a finite partition  $[t_s, t_e]$ , we calculate the probability  $p$  of obtaining a time within that interval with

$$p = \int_{t_s}^{t_e} \lambda e^{-\lambda t} dt$$

and the conditional mean  $t$  of the interval with

$$t = \frac{1}{p} \int_{t_s}^{t_e} \lambda t e^{-\lambda t} dt$$

For the final infinite partition  $[t_c, \infty]$ , we use

$$p = 1 - \int_0^{t_c} \lambda e^{-\lambda t} dt$$

and

$$t = \frac{1}{p} (1 - \int_0^{t_c} \lambda t e^{-\lambda t} dt)$$

All these integrals can of course be obtained by evaluating a simple exponential function. The result of these calculations is a set of  $m$  events with different times and probabilities, but whose mean matches that of the exponential function. When running a model in validation mode, the size of the state space depends critically on  $m$ , and it is often advisable to use a small value. On the other hand, high-fidelity simulation results can be still obtained by choosing a large  $m$  (and small  $y$ ) for simulation runs.

### 5.2.3 Kernel Classes

The kernel classes represent protocol global states and manage the execution of events in a validator or simulator. There are three subclasses of the abstract `val_gsi` class: `val_sgs`, `val_agsi` and `val_sim`. Which class is instantiated depends on the command

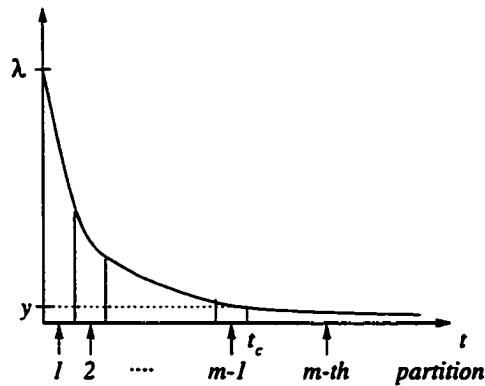


Figure 5.2: Approximation of Exponential Distribution

line parameters. The classes `val_sgs` and `val_agssi` run validations using the exact algorithm and the interval approximation algorithms, respectively. If the model is run in simulation mode, an object of the class `val_sim` is instantiated. The kernel object is available to users as the global variable `val_kernel`. The main methods of the `val_gsi` class are

- `void add_event(val_event ev);`

Adds the new event `ev` to the initial global state. This function can only be called from the `root()` function.

- `void newevents();`

This function is called from the `perform()` method of the process class and is used to add event sets to the current global state. In effect, it implements the output function  $O : T \rightarrow 2^{2^Y}$  and the probability function  $P : T \times 2^Y \rightarrow [0, 1]$  of the process (see definition 2). Thus, by the time control returns from `perform()`, `newevents()` will have been called in order to define the set of output event sets and the probability of each set that are generated in response to the the input event. There are four overloaded versions of this function:

- `void newevents(int n, val_event* e1, double p1, ...):`

Adds `n` event sets containing one event each; the probability of event set  $e_i$  is  $p_i$ .

- `void newevents(int n, val_event* e1, ...)`

Adds a single event set containing `n` events. The probability of this set is

1.

– `void newevents(val_event_list nl)`

Adds a single event set. The probability of this set is 1.

– `void newevents(int n, double p, val_event* e1, ...)`

Adds a single event set containing `n` events. The probability of this set is `p`. If this function is used, it must be called repeatedly so that the sum of all event set probabilities is 1.

• `void delete_event(int type);`

Removes events of the given type from the event list. This is a convenience function because, strictly speaking, the process model from definition 2 does not allow the removal of events. However, processes are free to ignore events of any type, and thus, deleting events from the event list is equivalent to setting a flag in the process indicating that events of the given type should be ignored.

#### 5.2.4 `val_process`

An object of the `val_process` class represents a process in the probabilistic protocol. The class is abstract and does not do anything useful by itself — protocol designers must define subclasses of `val_process` that represent the specific process types that occur in the protocol. The main methods of `val_process` are

• `void perform(int type)`

This is the event handler of the process and is called from the main validation or simulation engine. It is the most important method of the class because it defines the transition relation of the process. The method receives the event whose type is given by the parameter `type`, and can change the process state and generate a set of new event sets as a response to the event. The kernel function `newevents()` is used to pass new event sets to the kernel.

An important restriction in writing event handlers for processes is that the new process state can *only* depend on the input event and on the old process state. This means that random number generators and other global variables can not be used by the `perform()` method.

If an error is detected by the `perform()` method, the variable `Error` can be set to a non-zero value — the kernel will then process the error (e.g., by stopping the state space search and printing a trace) once control returns from the function.

- `void save_state(void *location)`

This function saves the process state at the given memory location. The easiest way to implement it is to simply `memcpy` the entire object; for example, if the process class is `sender`, then one could use `memcpy(location, (char *)this, sizeof(sender));`. Alternatively, a more sophisticated state saving function could be used if, for example, part of the object never changes.

- `void get_state(void *location)`

This function retrieves the process state state from the given memory location. Like the `state_save()` function, it can be implemented by a simple `memcpy` or by using more efficient methods.

- `int size()`

Returns the size of the process data in bytes.

- `void label(ostringstream & buff)`

Prints a short label identifying the current state. This is used for debugging purposes.

- `void dump()`

Dumps process data to standard output.

- `unsigned int hash(unsigned int n)`

This function returns a hashcode  $\in [0 \dots n-1]$ . It is important that this function be implemented carefully so that the probability of unequal states hashing to the same value is minimized. The hash values returned by the individual processes are combined by the validation engine to yield a hash of the global state.

- `int observer();`

This function returns 1 if the process is an observer, 0 if not.

## 5.2.5 Random Variable Support

The prototype library offers some basic support for computing statistical properties of random variables. There is a class for discrete random variables — `val_rand_disc` — and a class for continuous random variables — `val_rand_cont`. The difference between the two classes is that for continuous random variables, each value is assumed to be associated with a *duration*. The main methods of the `val_rand_disc` class are

- `val_randvar_disc()`  
This is the constructor. `hist_max` and `hist_size` are used to specify
- `void update(double value)`  
Adds a new sample.
- `double average()`  
Returns the current arithmetic average (expectation) of the random variable.
- `double variance()`  
Returns the variance (2nd central moment) of the random variable. The standard deviation is the square root of the variance.

The methods of the `val_rand_cont` class are identical except for the `update()` function:

- `void update()`  
Adds a new sample. There are two overloaded versions of this function:
  - `void update(double value, double duration)`  
Adds a new sample with its duration.
  - `void update(double value);` Adds a new sample. The duration of the sample is assumed to be the simulation time difference between the previous update and this update.

## 5.3 User Functions

### 5.3.1 The root Function

This function is called at the start of a validation or simulation run and is the system initialization function analogous to the `main()` function in C. Typically, the root function accomplishes four tasks:

- Parse command line parameters.
- Create processes.
- Connect processes to each other.
- Schedule initial event(s).

At least one process and one initial event must be created in order for a protocol to be meaningful.

### 5.3.2 The report Function

This function is called at the end of a simulation run. It is not called after a validation, and therefore, the user may supply an empty function if no simulations are expected to be run. The main purpose if this function is to print out the results of the simulation.

## 5.4 The EL Formula Compiler `e12cc`

Event logic formulas can be written in a plain text file; the program `e12cc` takes the text file and produces a C++ function that creates the formula in an internal representation suitable for the validator. This file must be compiled and linked to the executable model.

`e12cc` accepts the following key words corresponding to EL operators: AND ( $\wedge$ ), OR ( $\vee$ ), NOT ( $\neg$ ), X ( $\circ$ ) and UNTIL  $[t]$  ( $U^{\leq t}$ ). Atomic symbols (event types) must begin with a letter and may contain numbers. These atomic symbols are copied directly into the generated C++ code — they therefore need to be `#defined` somewhere in the model source code. Event types are often indexed with one or more numbers, and it is then convenient to use parts of the 32-bit integer event type for indexing



purposes. The C++ operator `<<` can be used for this purpose. For example, a file containing

```
(NOT (OVER<<12)+1) AND (NOT (OVER<<12)+2) AND (NOT (OVER<<12)+3)
```

corresponds to the EL formula  $\neg\text{OVER}_1 \wedge \neg\text{OVER}_2 \wedge \neg\text{OVER}_3$ .

## 5.5 Running the Model

Once the model has been built and linked, it can be run in either validation mode or in simulation mode. The list of supported command line parameters is:

- **-approx**  
Runs the validator using the interval approximation (section 4.4). This is the default and can be overridden by the **-single** option.
- **-bits  $n$**   
Specifies the size of the bitstate hash table.  $n$  is the length of the hashes in bits, i.e., the size of the hash table is  $2^n$  bits.
- **-depth  $n$**   
Runs the validator in depth-limited search mode.  $n$  is the maximum search depth.
- **-dot *filename***  
Produces a graphical output of the state space in a format compatible with the graph visualization package `dot`. This option is useful for debugging very small protocols, but should be used with caution because the amount of output can easily become unmanageable.
- **-dump**  
Produces diagnostic output: states and transitions.
- **-expapprox  $n$**   
Sets the number of intervals  $n$  used to approximate the exponential distribution (section 5.2.2).

- **-maxexp  $y$**   
Sets the size of the interval used to approximate the exponential distribution (section 5.2.2). The right edge of the interval  $r$  is chosen so that  $f(r) = y$ .
- **-formula  $p$**   
Runs the validator in model-checking mode.  $p$  is the required measure of satisfying paths; the algorithm stops as soon as the measure of non-satisfying paths exceeds  $1 - p$ . To check whether *all* paths of a protocol satisfy the formula,  $p$  should be set to 1. To calculate the measure of non-satisfying paths,  $p$  should be set to 0. If this parameter is omitted, the model is run as a plain state space search without model-checking.
- **-ppaths**  
Produces diagnostic output: paths.
- **-prob  $p$**   
Runs the validator in probability-threshold mode.  $p$  is the minimum probability threshold.
- **-progress**  
Produces diagnostic output every 100 visited states.
- **-rand  $n$**   
Specifies the integer random seed for the simulation mode.
- **-sim**  
Runs the model in simulation mode.
- **-single**  
Overrides the interval approximation algorithm (section 4.4), i.e., each node of the constructed TPTS corresponds to exactly one global state of the protocol.
- **-time  $t$**   
Specifies a time-limited search with threshold  $t$  (in validation mode) or the simulation end time (in simulation mode).

- `-unique`  
(Only used with the `dot` option.) Forces all visited states to be unique in the search graph. This means that the graph represents the search tree.

## 5.6 Summary

The library `libvalis.a` is a C++ library that enables protocol designers to implement a model of their protocol in C++. The model can then be run as a validator or as a simulator. In validation mode, the program checks whether the model satisfies a property expressed in the logic EL. In simulation mode, the program can be used to obtain performance characteristics.

# Chapter 6

## Examples

### 6.1 Overview

In this section, we describe a number of examples that we implemented using our prototype validator/simulator. We present these examples with three main objectives in mind.

First, we use the examples to study the behaviour of our validator under different scenarios. Some of the questions we examine are

- How effective is the probabilistic search heuristic in guiding the search?
- How does the interval-based approximate algorithm (section 4.4) compare with the standard algorithm?
- When the property to be model-checked consists of subformulas and conjunctions ( $f = f_1 \wedge f_2 \cdots f_n$ ), is it more efficient to check  $f$ , or is it better to check each subformula  $f_i$  separately?
- How does the size of the state space vary when the number of intervals used to approximate the exponential distribution is changed?
- How good is the ability of the system to detect errors?

One of our claims in this thesis is that the same model can be used for both validation and performance evaluation by simulation. Therefore, our second objective is to obtain performance results for the systems in our examples.

Our third objective in this section is to show by example how simulation/validation models can be implemented using our prototype library. Two types of graphs will be used to illustrate our models: state transition graphs for each process, and an event exchange diagram indicating which events are sent and received by which processes. In some cases, we will also show some fragments of code.

We will not attempt to reach all these objectives for each of the subsequent examples; rather, each example will be used to illustrate those aspects of our validator/simulator that it seems best suited to illuminate.

## 6.2 Multiprocessor System

We describe a multiprocessor system in which several processors compete for access to shared memory. A similar example was used in [52] to demonstrate the stochastic process algebra PEPA.

Each processor in the system alternates between a state in which it is “thinking” and a state in which it accesses memory. A memory access can be either local or to the globally shared memory. Local memory accesses present no problem. Accesses to shared memory, on the other hand, need to be coordinated. A simple protocol ensures that only one processor can access the shared memory at any one time: before accessing the shared memory, a processor needs to send a request and receive a reply. Once the access is completed, the processor must explicitly release the shared memory.

The shared memory alternates between idle and busy states. If it receives a request while in the idle state, it changes to the busy state and immediately sends a positive reply to the requesting processor. It returns to the idle state when the memory is released. Access requests that are received while the memory is in the busy state are queued and processed in first-in-first-out order.

It is natural to use two types of processes to model the system: one process type for modelling a processor and one process type for modelling the shared memory. Figures 6.1 and 6.2 show the state transition diagrams for the two process types. We only show the shared memory process for a 2-processor system, because this process has more than  $2^{n-1}$  states if there are  $n$  processors (this is a result of queueing

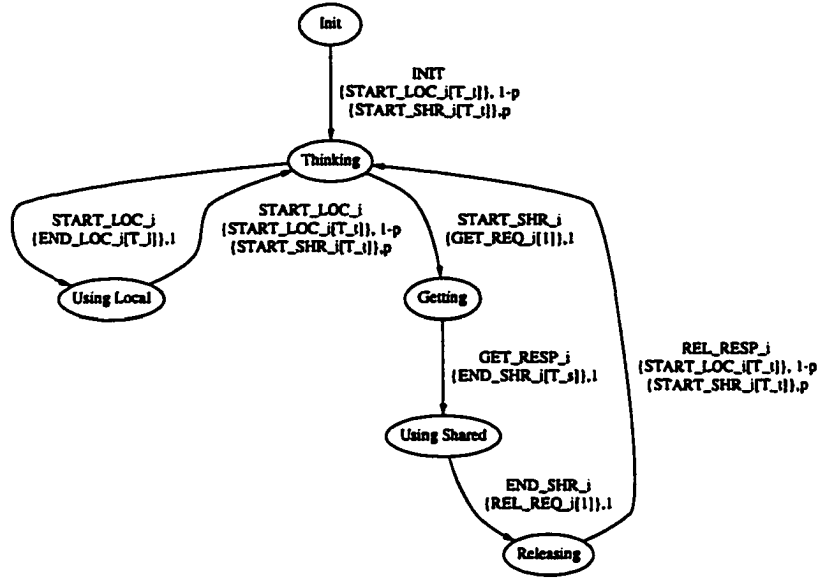


Figure 6.1: Process for Processor  $i$

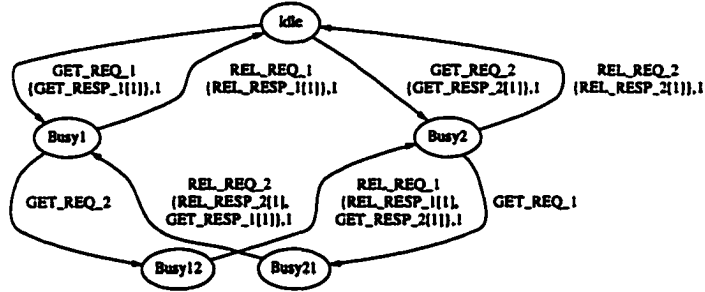


Figure 6.2: Shared Memory Process

access requests: each possible combination of requests is one state of the process). Local memory access by processor  $i$  is modeled with the event types  $START\_LOC_i$  and  $END\_LOC_i$ . Global memory access is modeled using the event types  $START\_SHR_i$ ,  $GET\_REQ_i$ ,  $GET\_RESP_i$ ,  $END\_SHR_i$ ,  $REL\_REQ_i$ ,  $REL\_RESP_i$ . Communication between the shared memory and any processor is assumed to take 1 time unit. Figure 6.3 shows the message exchange diagram for this example.

Our model has 5 parameters:  $n$ , the number of processors;  $T_t, T_l, T_s$ , the times spent in the thinking, local access and shared memory access states, respectively; and  $p_s$ , the probability that a memory access is to the shared memory. The time parameters  $T_l$  and  $T_s$  may be constant, exponentially distributed with the given mean, or may be given as intervals, in which case they are taken to be uniformly distributed.

We first conduct some plain state space search experiments to ensure that the pro-

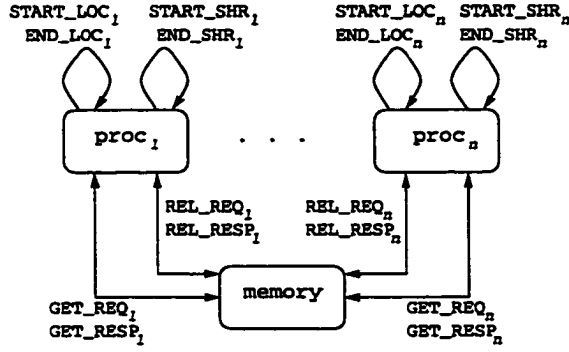


Figure 6.3: Event Exchange Diagram for Multiprocessor System

protocol does not have errors like deadlocks and unspecified event receptions. Table 6.1 shows the size of the state space for various combinations of the parameters. The state space increases with the number of processors  $n$ , time nondeterminism (setting delays to  $[5,10]$  instead of 10) and the magnitude of  $T_i$ . Of these, the value of  $n$  appears to have the most significant effect. No errors were detected in these experiments.

A basic temporal property that we want to check is that only one processor can access the shared memory at one time. A processor  $i$  may access the shared memory from the moment it receives a  $\text{GET\_RESP}_i$  event until it receives an  $\text{END\_SHR}_i$  event (note that due to the delay incurred when communicating between memory and processor, there is a delay between the time a processor receives an  $\text{END\_SHR}_i$  event and the time it receives the  $\text{REL\_RESP}_i$  from the memory — we assume that a processor will not access the shared memory during this delay). The EL formula

$$f_i := \text{GET\_RESP}_i \rightarrow \left( \bigwedge_{\substack{j=1 \\ j \neq i}}^n \neg \text{GET\_RESP}_j \right) U^{\leq T_i, \text{max}} \text{END\_SHR}_i$$

specifies that once processor  $i$  is granted access to the shared memory, no other processor is granted access until processor  $i$  stops using the shared memory. The specification for all processors, then, is

$$f := \bigwedge_{i=1}^n f_i$$

We can either check for  $f$ , for each subformula  $f_i$  or, arguing that the system is symmetric with respect to the processors, check only for one of the  $f_i$  (say  $f_1$ ). Table 6.2 shows the behaviour of the validator for these cases and two sets of parameters. We see that — compared to table 6.1 — the state space is significantly larger due to

$n$	$T_t$	$T_l$	$T_s$	$p_s$	no. of states
2	10	[5,10]	[5,10]	0.5	956
2	20	[5,10]	[5,10]	0.5	1444
2	40	[5,10]	[5,10]	0.5	2378
3	10	10	[5,10]	0.5	10545
3	20	10	[5,10]	0.5	14870
3	40	10	[5,10]	0.5	22732
3	10	[5,10]	10	0.5	11187
3	20	[5,10]	10	0.5	16327
3	40	[5,10]	10	0.5	22394
3	10	[5,10]	[5,10]	0.5	12801
3	20	[5,10]	[5,10]	0.5	20475
3	40	[5,10]	[5,10]	0.5	27538
4	10	10	[5,10]	0.5	86603
4	20	10	[5,10]	0.5	116008
4	40	10	[5,10]	0.5	140545

Table 6.1: Multiprocessor System State Space

$n$	$T_t$	$T_l$	$T_s$	$p_s$	$f_1$	$f_1 \wedge f_2$
2	20	10	10	0.5	189321	203952
2	40	10	10	0.5	1399597	1491703

Table 6.2: Multiprocessor System Model-Checking

the model-checking of the formula. The table also shows that although checking for  $f_1 \wedge f_2$  is more expensive than checking for just  $f_1$ , the difference is not significant.

Our current implementation of the multiprocessor system satisfies  $f$  with probability 1. It is interesting to note, however, that our first implementation contained a bug that was detected by the validation system. The error was in the way the shared memory process responded to REL\_REQ events. If the buffer was not empty, two events were sent: one GET\_RESP event to the first waiting request in the queue, and a REL\_RESP to the source of the REL\_REQ event. The buggy code fragment was

```

val_kernel->newevents(2,
    new val_event((MP_GET_RESP<<12)+src, Id2Proc[Buffer[0]], 1),
    new val_event((MP_REL_RESP<<12)+src, Id2Proc[src], 1));

```



...

which resulted in the GET\_RESP event being indexed with the wrong value *src*. As a consequence, this particular GET\_RESP<sub>src</sub> was not followed by END\_SHARED<sub>src</sub> as specified by *f*, and the sequence starting with GET\_RESP<sub>src</sub> was reported by the validation tool. The corrected fragment is

```
val_kernel->newevents(2,  
    new val_event((MP_GET_RESP<<12)+Buffer[0], Id2Proc[Buffer[0]], 1),  
    new val_event((MP_REL_RESP<<12)+src, Id2Proc[src], 1));
```

...

We now use our model to analyze the performance of the multiprocessor system. The *load l* on the shared memory caused by a processor *i* is the amount of time that the processor would spend accessing shared memory if it did not have to share it with other processors. We can calculate *l* as a function of the model parameters by observing that the mean length of a cycle from the start of a thinking state to the start of the next thinking state is

$$T_i + p_s T_s + (1 - p_s) T_l$$

(we ignore the communication overhead here). Of this time, the mean time spent accessing shared memory is  $p_s T_s$ . Therefore,

$$l = \frac{p_s T_s}{T_i + p_s T_s + (1 - p_s) T_l}$$

The cumulative load from all processors is  $n \times l$ . Note that this is only an upper bound on the actual load, because the time that each processor spends waiting for access to shared memory reduces the load. As a performance metric for the system, we choose the ratio *w* of the time spent waiting for shared memory access to the “working time”, i.e., time spent thinking and accessing local or shared memory [52]. Some values of *w* (averaged over all processors) for different scenarios are in table 6.3. In figure 6.4, where we plot the system performance as a function of the load for  $n = 3$  and  $n = 5$ .  $T_i$  and  $T_l$  are fixed at 100,  $T_s$  is fixed at 80, and  $p_s$  is set to achieve the desired load. The graph shows that *w*, although growing non-linearly, remains reasonably small at 0.12 and 0.18 for  $n = 3$  and  $n = 5$ , respectively, at a load of 0.6.

$n$	$T_t$	$T_l$	$T_s$	$p_s$	$w$
2	100	110	150	0.2	0.0475556
3	100	100	100	0.1	0.0270736
3	100	100	50	0.8	0.271662
3	100	100	100	0.5	0.216942
4	100	50	150	0.2	0.181092

Table 6.3: Performance of Multiprocessor System

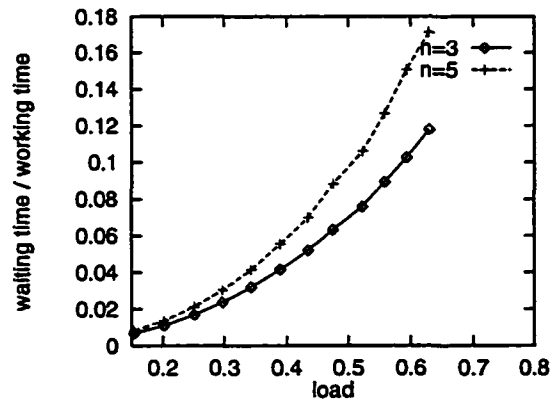


Figure 6.4: Performance of Multiprocessor System

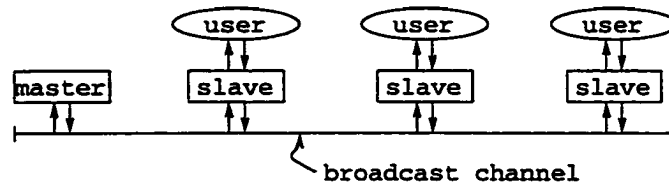


Figure 6.5: Collision Avoidance Protocol with 3 Users

## 6.3 Broadcast Channel Protocols

In this section, we examine two broadcast channel protocols: a collision avoidance protocol [59] and CSMA/CD. We model a generic broadcast channel similar to the one provided by SMURPH. The channel is implemented as a single process. During the initialization of the channel, a delay matrix giving the propagation delays between each pair of attached is provided. The channel operates by receiving events from any one attached station and sending a copy of that event to all attached stations (including the sending one), using the values given in the delay matrix. To properly model packet transmission and collision, attached processes must send two events per packet: one that indicates the start of a packet (SOP) and one that indicates the end of a packet (EOP). Then, a collision of packets at a receiving process occurs if (and only if) two SOP events are received without an EOP event in-between. Note that this channel model allows packets to pass through each other and still be received correctly. In practice, more than two event types are required to model a broadcast channel protocol. Both protocols in this section use two indexes to distinguish different start-of-packet and end-of-packet events. The indexes  $i, j$  of an event type  $SOP_{i,j}$  or  $EOP_{i,j}$  have the following meaning:

- $i$ : the intended destination station index.
- $j$ : the destination process. This is not the same as  $i$  because the broadcast channel propagates events to all attached processes, not just the process representing the intended destination.

### 6.3.1 Collision Avoidance Protocol

The first broadcast channel protocol we consider is a collision avoidance protocol described in[59]. The operation of the protocol is straightforward: a dedicated station

(the *master*) periodically polls the other stations (the *slaves*) in round-robin order. If a slave has data in its buffer, it transmits it as soon as it receives the polling signal. If a slave has no data to transmit, it ignores the polling signal. The master station starts a timer whenever it polls a station. It polls the next slave in round-robin order if it either determines that the current slave has finished transmitting, or if the timeout expires indicating that the slave has nothing to send. For simplicity, the stations are assumed to be equidistantly placed on a bus-shaped channel (figure 6.5), and the buffer size at each slave is set to 15 packets.

The protocol is modeled using four process types: a master process,  $n$  slave processes,  $n$  user processes and the broadcast channel process. The master process has index 0, the slave and user processes have indexes  $1, \dots, n$ , and the broadcast channel process has index  $n + 1$ . The user process periodically generates a data packet and sends it to the attached slave process. The destination of a data packet is chosen among the other user processes in the network with uniform probability. Figure 6.6 shows the process types and the events exchanged between them. The generation of new packets by users  $i$  is controlled by event type  $NEW_i$ . In response to a  $NEW_i$  event, user  $i$  sends a  $SEND_{i,j}$  event to its attached slave process, where  $j$  is the index of any of the other user processes. When slave  $i$  receives a data packet correctly, it sends an event of type  $RCV_i$  to its user.

The transmission of data packets over the broadcast channel is modeled with two classes of events,  $SOD_{i,j}$  and  $EOD_{i,j}$ , that indicate the start and the end of a data packet, intended for station  $i$  and received at station  $j$ , respectively. Similarly, polling packets are modeled with the two events  $SOP_{i,j}$  and  $EOP_{i,j}$ . Our implementation has 5 parameters, see table 6.4. All time, packet size and distance values are expressed in indivisible time units. The packet interarrival time at each user is exponentially distributed with mean  $a$ .

We first check for collisions and buffer overflows. The slave processes are programmed so that they set an error flag if they detect a collision, therefore this experiment is a plain state space search and does not require checking an EL formula. Table 6.5 shows the results for some combinations of  $a$ ,  $n$  and  $t$ . The packet size  $l$  and the distance  $d$  are fixed at 30 and 20, respectively. It is easy to see that collisions can occur if the timeout value is less than the maximum round-trip delay in

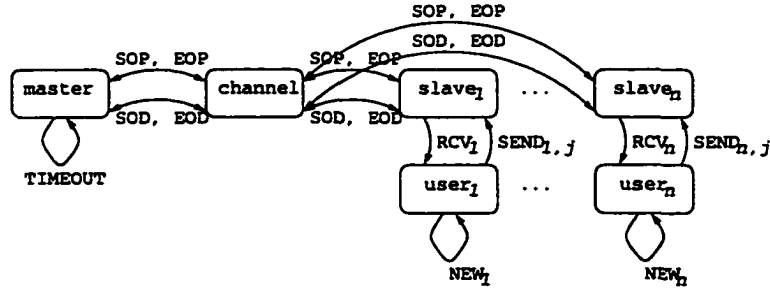


Figure 6.6: Event Exchange Diagram for Collision Avoidance Protocol

Parameter	Meaning
$n$	no. of users
$a$	mean packet generation delay (per user)
$l$	size of packets
$d$	distance between two adjacent stations
$t$	master station timeout

Table 6.4: Collision Avoidance Protocol Parameters

the network. The service rate (the maximum throughput) of the protocol depends on the number of stations, the distance between stations and the master station timeout value. Buffer overflows occur if the cumulative arrival rate is greater than the service rate of the network.

Another property we want to check is that every generated packet is delivered at its destination within some time limit  $t$ . This can be expressed as

$$k_t := \bigwedge_{i=1}^n \bigwedge_{j=1}^n \text{SEND}_{i,j} \rightarrow \text{TRUE } U^{\leq t} \text{RCV}_j$$

The corresponding probabilistic formula is  $P(k_t) \geq 1$ , because we want property  $k_t$  to be true on all paths. Table 6.6 shows the size of the state space of this experiment for the two error-free cases of table 6.5 and three values of  $t$ . As expected,  $k_t$  is false when  $t$  is small, and true when  $t$  is large. Note that determining that  $k_t$  is false is typically much faster than determining that it's true — this is because in the former case, the search is aborted as soon as a single non-satisfying path is discovered.

The protocol is simple enough so that performance measures like mean response time can be easily calculated. Nevertheless, we run our tool in simulation mode to demonstrate its capabilities. We want to determine the capacity of the protocol in

$n$	$a$	$t$	no. of states	result
2	200	50	78	collision
2	200	100	1009	ok
3	200	140	1445	buffer overflow
3	300	140	7753	buffer overflow
3	400	140	9711	ok

Table 6.5: Collision Avoidance Protocol: Plain Search

$n$	$a$	$t$	plain	$k_{200}$	$k_{400}$	$k_{500}$
2	200	100	1009	FALSE 336	TRUE 3029	TRUE 3029
3	400	140	7808	FALSE 375	FALSE 451	TRUE 98264

Table 6.6: Collision Avoidance Protocol: formula verification

terms of the maximum cumulative packet arrival rate. To do this, we vary the packet generation delays for the error-free cases of table 6.5 with the packet size set to 100. For a system with  $n$  slaves and packet generation delay  $a$ , the cumulative packet arrival rate at all slaves is  $\frac{n}{a}$  per time unit. Figure 6.7 plots the mean queue length (again, this is the cumulative queue length at all  $n$  sources) against the cumulative arrival rate. The graph exhibits a characteristic “knee” shape: the mean queue length is very small until it suddenly grows very steeply. The capacity of the system is the arrival rate just before this steep growth. Note that for this collision avoidance protocol, the capacity decreases as the number of stations increases — this is due to the overhead caused by polling.

### 6.3.2 CSMA/CD

The second broadcast channel protocol we consider is a variant of the CSMA/CD protocol (see, e.g., [43, 80]). The basic principle of such a protocol is that stations can start transmitting data packets whenever they sense that the broadcast channel is idle. If this results in a collision, those stations that are involved in the collision wait for a randomized *back-off* period and retransmit their packets.

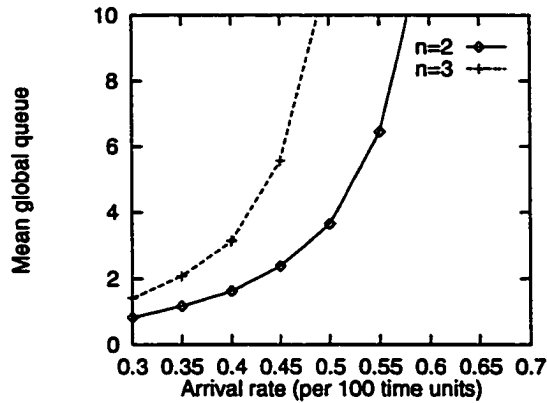


Figure 6.7: Performance of Collision Avoidance Protocol

Our stations use the following back-off procedure: when a station detects that a packet that it is transmitting has suffered a collision, it continues to transmit the rest of the packet, and also sets a back-off timer. When the timer expires at the end of the back-off period,

- if the channel is idle, the station either retransmits the packet (with probability  $p$ ) or waits for another back-off period (with probability  $(1 - p)$ )
- if the channel is busy, the station waits for another back-off period

At the end of the second back-off period, it again either retransmits the packet or waits for another back-off period. The process continues until the packet is transmitted successfully. This back-off procedure is known as  $p$ -persistent; some types of CSMA/CD protocols like Ethernet use slightly different procedures.

The protocol is modeled using three process types:  $n$  station processes,  $n$  user processes and the broadcast channel process. Figure 6.9 shows the event exchange diagram for the protocol. The user and broadcast processes are identical to the ones we use to model the collision avoidance protocol in the previous section: the user process periodically generates a data packet and sends it to the attached station (event type  $SEND_i$ ). Each station has one packet buffer, if a data packet is received from the user while the buffer is still occupied, the station sends an  $OVER_i$  to the user process. The transmission of data packets by the station processes is modeled with two classes of events,  $SOD_{i,j}$  and  $EOD_{i,j}$ , that indicate the start and the end of a data packet intended for station  $i$  and received at station  $j$ , respectively. The back-off

Parameter	Meaning
$n$	no. of users
$a$	packet generation delay (per user)
$b$	length of back-off period
$l$	size of packets

Table 6.7: CSMA/CD Parameters

procedure is modeled using the two event types  $EOB_i$  (end of back-off) and  $BOA_i$  (back-off again). Following a collision, a station sends an  $COLL_i$  event to itself and either an  $EOB_i$  event (with probability  $p$ ) or a  $BOA_i$  event (with probability  $(1 - p)$ ) to itself. When it receives an  $EOB_i$  event, the station retransmits the collided packet; when it receives a  $BOA_i$  event, it again sends either a  $EOB_i$  or  $BOA_i$  event to itself. If the packet buffer in the station overflows (this is always possible because the protocol does not provide service guarantees), it generates an  $OVER_i$  event.

Our implementation has 4 parameters, see table 6.7. All time, packet size and distance values are expressed in indivisible time units. For simplicity, the stations are assumed to be equidistantly placed on a bus-shaped channel (figure 6.8) with a distance  $d = 20$  between them, the persistence parameter  $p$  is fixed at  $\frac{1}{2}$ , and all packets have the same size  $l$ . The packet generation delay is

Table 6.8 shows the results of a state space search for some values of  $n$ ,  $a$  and  $l$ . In those cases where an exhaustive search is not possible due to resource limitations, we perform probabilistic searches with thresholds  $p = 10^{-15}$ ,  $10^{-20}$  instead. It is clear that larger  $n$  must lead to an increase in the state space. However, larger values of  $a$  and  $l$  also generally lead to larger state spaces because they increase the number of possible events in the event list. For example, if  $a = 100$ , then the event  $NEW_i$  can appear in the event list with delays  $0, \dots, 100$ ; and if  $a = 200$ , it can appear with delays  $0, \dots, 200$ .

The CSMA/CD protocol does not work properly if the packet length  $l$  is less than twice the propagation delay between two stations. Consider two stations A and B that are  $d$  time units apart. If A sends a packet of length  $l$  to B, and B starts sending a packet to A just before A's packet starts arriving at B, then B's packet will start



$n$	$a$	$b$	$l$	$p$	no. of states
2	100	60	30		74680
2	100	60	50		150685
2	200	60	30		121275
2	200	60	50	$10^{-15}$	460755
2	200	60	50	$10^{-20}$	953942
3	100	60	30	$10^{-15}$	482940
3	200	60	100	$10^{-15}$	185903

Table 6.8: CSMA/CD: Plain Search

arriving at A almost  $2d$  time units after A's start of transmission. In this case, station A would not detect the collision if  $l < 2d$ . The result is that from A's point of view, the packet was successfully transmitted, although it didn't arrive correctly at station B — the packet is effectively lost. Some of the scenarios in table 6.8 exhibit this error, but it is not detected by a simple state space search. To formulate an EL specification to detect this kind of error, consider that once a station  $i$  starts transmitting a packet to station  $j$  (this is indicated by a sending and event  $SOD_{j,0}$  to the channel), it must either be followed by the successful reception of the packet at station  $j$  (indicated by  $RECV_j$ ) or else it must result in the detection of a collision by station  $i$  ( $COLL_i$ ),

$$f_{i,j} := SOD_{j,0} \rightarrow \text{TRUE } U^{\leq t} (RECV_j \vee COLL_i)$$

For all source/destination pairs, the specification is thus  $\bigwedge_{i=1}^n \bigwedge_{j=1, j \neq i}^n f_{i,j}$ . However, this protocol is essentially symmetric (all sources and destinations are equal) and for a protocol like this, it is usually sufficient and more effective to just check the specification for just one source/destination pair. The time limit  $t$  is set to be slightly larger than the sum of the packet length and the propagation delay between stations  $i$  and  $j$ . In table 6.8, we show the results of checking formula  $f_{1,2}$  for four scenarios of table 6.8.

Finally, we use our model to obtain some performance measures for the CSMA/CD protocol. As in our analysis of the collision avoidance protocol, we want to determine the capacity of the protocol in terms of the maximum cumulative packet arrival rate. To do this, we vary the packet generation delays for the error-free cases of table 6.5 with the packet size set to 100. For a system with  $n$  slaves and packet generation delay

$n$	$a$	$b$	$l$	$p$	$f_{1,2}$
2	100	60	30		FALSE
2	100	60	50		TRUE
3	100	60	30	$10^{-15}$	FALSE
3	200	60	100	$10^{-15}$	TRUE

Table 6.9: CSMA/CD: Results for Property  $f_{i,j}$

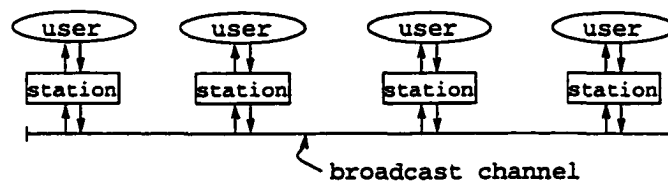


Figure 6.8: CSMA/CD with 4 users

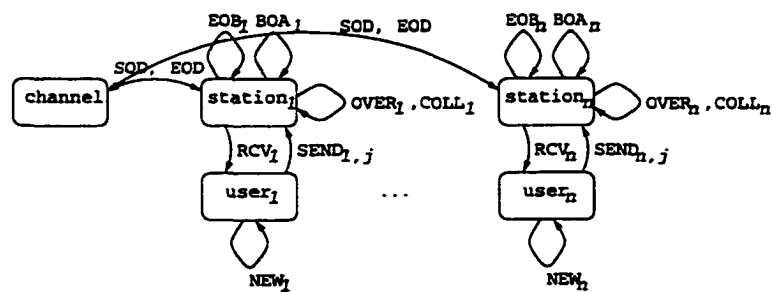


Figure 6.9: Event Exchange Diagram for CSMA/CD

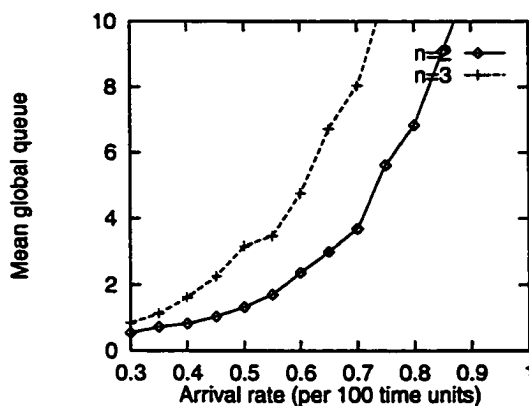


Figure 6.10: Performance of CSMA/CD

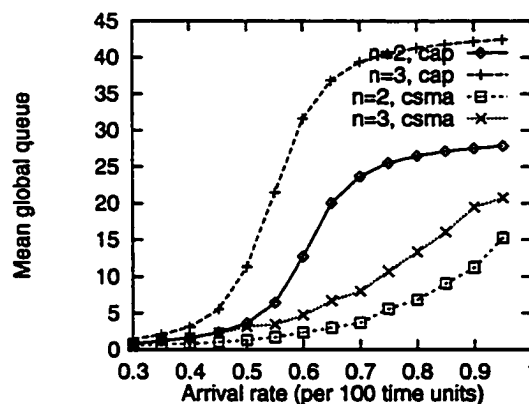


Figure 6.11: Comparison of Collision Avoidance Protocol and CSMA/CD

$\alpha$ , the cumulative packet arrival rate at all slaves is  $\frac{n}{a}$  per time unit. Figure 6.11 plots the mean cumulative queue length against the cumulative arrival rate. Figure 6.11 compares the results for the CSMA/CD protocol with those of the collision avoidance protocol. The performance of CSMA/CD is clearly superior across all arrival rates (note that the buffer size at each station is limited to 15 — this means that the maximum mean cumulative queue length is 30 for  $n = 2$  and 45 for  $n = 3$  and explains the flattening of the collision avoidance protocol curves at higher arrival rates).

### 6.3.3 GARP Multicast Registration Protocol (GMRP)

The third broadcast channel protocol we study is the GARP Multicast Registration Protocol (GMRP), which is a version of GARP, the Generic Attributes Registration Protocol. GARP and GMRP are part of the IEEE 802.1p standard. An untimed

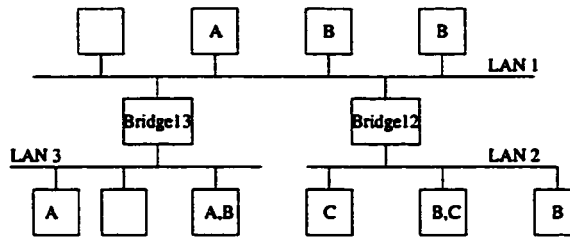


Figure 6.12: Bridged LAN with 3 Multicast Groups (A, B, C)

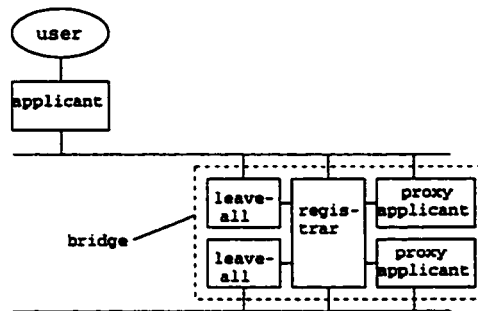


Figure 6.13: GMRP Entities

version of the protocol is validated using SPIN in [70]. The problem that GMRP is designed to solve is that of excessive flooding of multicast packets across a bridged LAN. Figure 6.12 shows a bridged network with three subnets (LAN 1, LAN 2, LAN 3) and three multicast groups (A, B, C). The stations on the network are labeled with the multicast groups that they are part of. Rather than simply flooding all multicast packets across all LANs, the aim is to use filters in the bridges to forward packets only to those LANs that have multicast group members in them. Thus, multicast packets with address A need not be forwarded to LAN 3, and multicast packets with address C only need to be forwarded to LAN 3.

The basic idea of the protocol is that stations register with the bridges whenever they wish to join or leave a multicast group. Based on this information, the bridges update a filtering database which they use to decide which packets need to be forwarded to which port. The entities of the protocol are as follows (figure 6.13):

- *User*. The user indicates to the applicant whenever it wants to join or leave a multicast group.
- *Applicant*. This is the entity that handles the user side of the address registration protocol. When a user wants to join multicast group, the applicant

broadcasts a *join* request indicating the requested address and starts a timer with timeout  $t_j$ . At this point, the applicant considers itself part of the multicast group and forwards any packets for that address to the user. If the applicant perceives another join request for the same group (from another applicant) before the timeout, it does nothing further; if on the other hand, no other join request is received, it re-sends the same join request after the timeout. This re-sending of the join request is not strictly necessary but increases the robustness of the protocol. When a user wants to leave a multicast group, the applicant broadcasts a *leave* message indicating the multicast group. Any applicant that is still part of that multicast group must re-register their membership by sending a corresponding *join* message.

- *Bridge*. The bridge has three GMRP components: a *registrant*, a *proxy applicant* for each port and a *leave-all* process for each port. The registrant is responsible for most of the bridge-side handling of GMRP. Whenever it receives a join request on a port, it updates its filtering database if necessary. When it receives a leave request on a port, the registrant starts a timer with timeout  $t_l$ . If no contradicting join request is received during the wait period, the address is removed from the database. When a new address is added to the database for one of its ports, the registrar sends a message to the proxy applicant of the other port, which registers the multicast address with the bridges on the other broadcast LAN. The proxy applicant functions exactly like an applicant, except that it receives requests from the registrar and not a user. Finally, the leave-all process is responsible for garbage collection: it periodically generates a leave request for all the multicast addresses registered at a port — this forces applicant that are still members of the multicast group to confirm their membership with a join message. Stale addresses that are no longer needed are removed in this way.

We model the protocol with 5 process types corresponding to the protocol entities user, applicant, registrant, leave-all and the broadcast channel. In addition to these protocol entities, we use a source process to model a multicast data source. The source periodically generates multicast packets; the probability of a multicast packet

event	meaning
CHANGE <sub><i>i</i></sub>	causes user <i>i</i> to add/delete group memberships
ADD <sub><i>i,a</i></sub>	causes applicant <i>i</i> to initiate joining of multicast group <i>a</i>
DEL <sub><i>i,a</i></sub>	causes applicant <i>i</i> to initiate leaving of multicast group <i>a</i>
MP <sub><i>a</i></sub>	multicast packet with address <i>a</i>
MD <sub><i>a</i></sub>	multicast data with address <i>a</i>
JOIN <sub><i>i,a</i></sub>	GMRP <i>join</i> request for multicast group <i>a</i> by applicant <i>i</i>
LEAVE <sub><i>i,a</i></sub>	GMRP <i>leave</i> request for multicast group <i>a</i> by applicant <i>i</i>
JOIN_TO <sub><i>i,a</i></sub>	GMRP <i>join</i> timeout at applicant <i>i</i>
LEAVE_TO <sub><i>i,a</i></sub>	GMRP <i>leave</i> timeout at bridge <i>i</i>
GARBAGE <sub><i>i</i></sub>	causes GMRP <i>leaveall</i> process to start at bridge <i>i</i>

Table 6.10: Events for GMRP Model

having address  $i$  is  $\frac{1}{m}$ , where  $m$  is the number of multicast addresses. To simplify the model, we use only single events to model packet transmission on the broadcast LAN, instead of two events as in the previous two sections. We also assume that the LAN transmits all packets in constant time (one time unit). Table 6.10 and figure 6.14 show the events and the event exchange diagram of our model, respectively.

The topology of the bridged LAN is star-shaped, i.e., there is a central LAN to which all others are attached with bridges. Our model has 8 parameters:  $n_l$ , the number of LANs;  $n_u$ , the number of users per LAN;  $m$ , the number of multicast addresses;  $t_j$ , the join timeout period;  $t_l$ , the leave timeout period;  $t_g$ , the garbage collection period (used by the *leaveall* process);  $t_c$ , the period with which users change their multicast group memberships and  $t_p$ , the packet generation period (used by the source).

We first do some plain state space searches to check for errors like deadlocks and unspecified receptions. The user processes are programmed to detect incorrectly addressed multicast packets and set an error flag if they do. The system passes this test for some parameter values (table 6.11), but as soon as the timing values are randomized just a little, an error is quickly detected. The error is essentially caused by a race

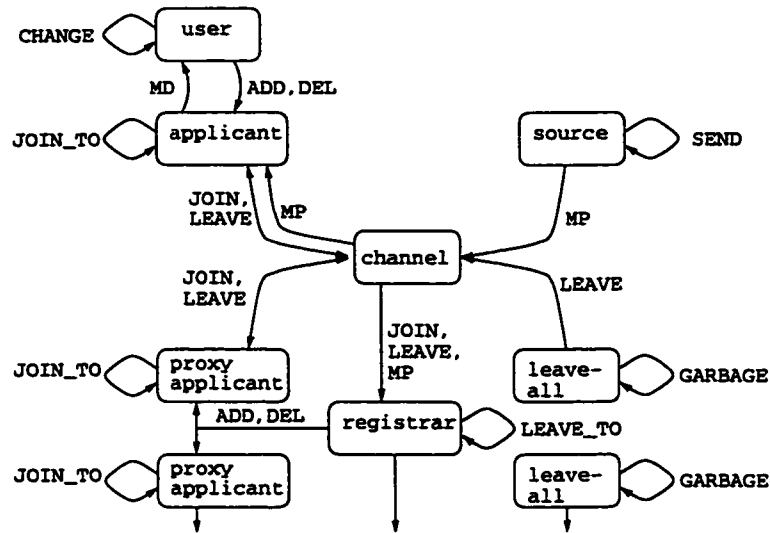


Figure 6.14: Event Exchange Diagram for GMRP

$n_l$	$n_u$	$m$	$t_j$	$t_l$	$t_g$	$t_c$	$t_p$	result
1	1	2	10	10	10	10	20	ok, 52 states
2	1	2	10	10	10	10	20	ok, 347
2	2	2	10	10	10	10	20	ok, 726
2	2	2	10	10	10	[9,10]	20	error (see text)
1	1	2	10	10	10	[14,15]	20	error (see text)
1	2	2	10	10	10	[9,10]	20	ok, 251445
2	1	2	10	10	10	[9,10]	20	ok, 202366 $p = 10^{-20}$

Table 6.11: GMRP State Space Search

condition: a user considers itself to be removed from a multicast group when it sends an DEL event to the attached applicant. It is possible, however, that a previously sent multicast packet reaches the applicant *before* the DEL message, and the multicast packet is therefore forwarded to the user, resulting in an error. The error was not detected with the first few time parameters because they resulted in deterministic delays between events and avoided the race condition. We can reasonably relax the requirement so that *one* unwanted multicast packet is allowed — the last two rows of table 6.11 show that no errors are detected in this case. This experiment shows the importance of avoiding overly deterministic models and of using the ability of the validator to deal with time intervals.

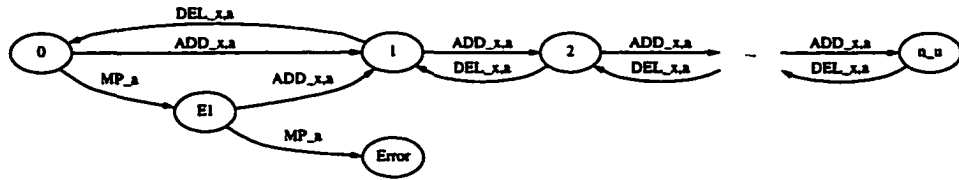


Figure 6.15: Observer for GMRP

The next requirement we want to check is that multicast packets are forwarded to a LAN only if there are multicast group members on that LAN. As before, we would like to accommodate race conditions by relaxing this requirement to mean that a small number (say, 1) of packets is allowed. Instead of attempting to write an EL formula expressing this, we write an observer process that checks for this property. The observer works by keeping track of the number of members of one multicast group in one of the LANs (by monitoring DEL and ADD events), and checking if any multicast packets are forwarded to the LAN if this number is zero. The observer flags an error if more than one multicast packet is forwarded to the LAN after all members have left the group. The model including the observer has two additional parameter, namely, the multicast address  $a$  to be monitored and an index  $l$  indicating which LAN should be observed. The central LAN should not be observed in this way, since the source is attached to it and therefore, all packets originate there.

Figure 6.15 shows the state transition diagram for the observer process. In the figure, an event  $ADD_{x,a}$  indicates an ADD event for address  $a$  generated by any user process on LAN  $l$ . For our experiments, we set  $t_j = 10$ ,  $t_l = 10$ ,  $t_g = 10$ ,  $t_c = [5, 10]$ ,  $t_p = 20$  and check for several combinations of  $n_l$ ,  $n_u$ , and  $m$ . Because of the size of the state space, we ran a probabilistic validation with threshold  $10^{-20}$ . Table 6.12 shows the validation results: no errors were detected. The last two columns of the table compare the protocol state space with and without observer.

## 6.4 Steam Boiler Control Program

### 6.4.1 Description of Model

We consider the task of validating a steam boiler control program. The problem is based on a real-life scenario and has previously been analyzed using other validation



$n_l$	$n_u$	$m$	result without observer	result with observer
1	2	2	808144	792325
2	1	2	3040748	2994586
2	2	2	$21 \times 10^6$	$21 \times 10^6$

Table 6.12: GMRP State Space Search with Observer

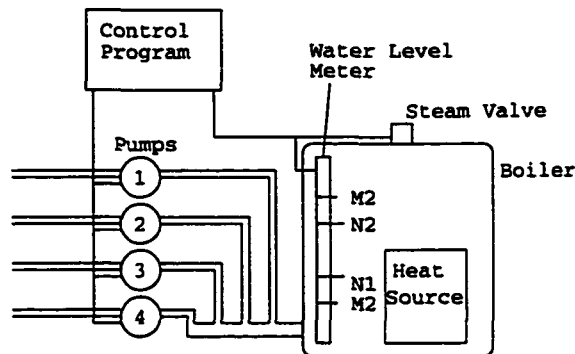


Figure 6.16: Steam Boiler System

systems, including SPIN [27, 63]. Figure 6.16 shows the components of a steam boiler system. It consists of a *steam boiler* containing a variable amount of water; a *heat source* that causes some of the water to turn into steam and escape through the steam valve; four *pumps* that can be turned on and off; and a *control program*. Two measuring devices, a *water level meter* and a *steam meter*, periodically send information to the control program.

One or more of the pumps may fail, in which case they are unable to pump water. The measuring devices may fail as well, in which case they stop sending information to the control program (we assume that they do not fail maliciously and send false data to the control program — it is easy to see that the system will fail if both measuring devices somehow collude to fool the controller). Whenever something fails, it may be repaired after an unspecified period of time.

The task of the control program is to maintain the water level in the boiler between the levels  $N_1$  and  $N_2$  (in normal mode), or at least between  $M_1$  and  $M_2$  (in “degraded” mode, when one or more components fail). The only way that the controller can influence the water level is by switching pumps on and off. The exact nature of the

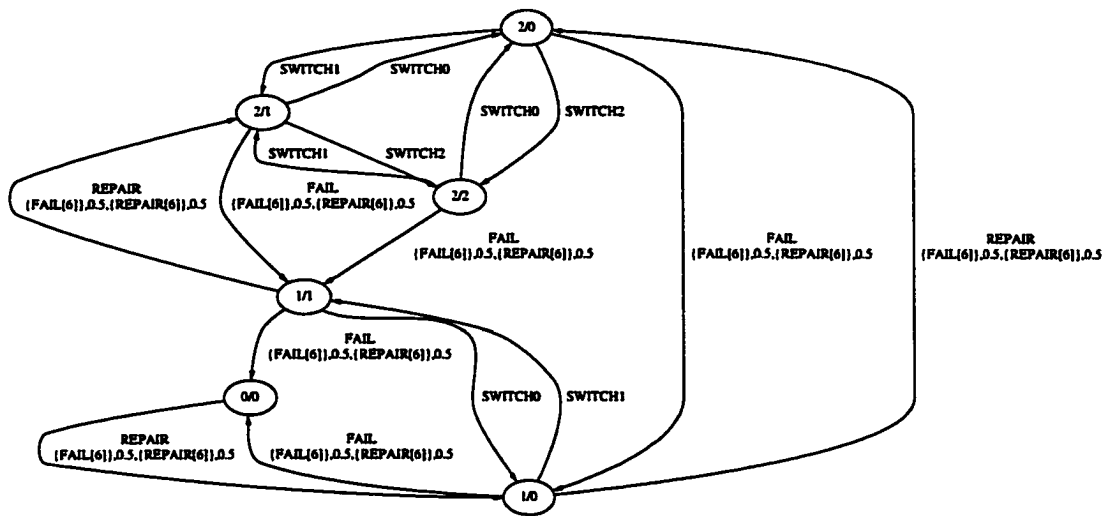


Figure 6.17: Pump Process (for 2 pumps)

heat source is left unspecified, but three parameters define the range of its possible behaviours:

- $W$ , the maximum steam generation rate (in litres per time unit)
- $U_1$ , the maximum gradient of increase of  $W$  (in litres per squared time units). So, for example, if the steam generation rate is 10 at time  $t$ , then after a delay  $\Delta t$ , the steam generation rate can not be greater than  $10 + \Delta t \times U_1$ .
- $U_2$ , the maximum gradient of decrease of  $W$  (in litres per squared time units). This is analogous to  $U_1$ : if the steam generation rate is 10 at time  $t$ , then after a delay  $\Delta t$ , the steam generation rate must be at least  $10 - \Delta t \times U_2$ .

We model the system using three processes corresponding to the pumps, the steam boiler, and the control program.

- *Pumps*. The pump process models all four pumps. The process accepts SWITCH<sub>*i*</sub> events from the controller which indicate that  $i$  of the four pumps should be switched on, and the remaining  $4 - i$  pumps should be off. Every time unit, the process sends a WATER event to the boiler process indicating how much water (in litres) is being pumped in that time unit. Also periodically (every 5 time units), the pump send an P\_INF event to the controller process, indicating how many of the four pumps are functioning and how many are on. The periodic

transmissions are triggered by WATER\_UPD and INF\_UPD events. Occasionally, one or more pumps may fail and be repaired after some time. The failure and restoration of pumps is modeled with the events P\_FAIL and P\_REPAIR, respectively, as follows: every  $t_p$  time units, the process generates either a P\_FAIL event — with probability 0.05 — or a P\_REPAIR event — with probability 0.95. A P\_FAIL event results in the number of working pumps being decreased by 1; a P\_REPAIR event results in the number of working pumps being increased by 1. Both events cause the rescheduling of P\_FAIL or P\_REPAIR events with a delay of  $t_p$ . Figure 6.17 shows the state transition diagram for the process. Due to the large number of transitions, we only show the diagram for a 2-pump system. Also, we omit the periodic update events. Each state in the figure is labeled with two values: the first indicates the number of operational pumps, and the second indicates the number of pumps that are on.

- *Boiler.* The boiler process models the steam boiler including the heat source and the the two measuring devices. It is responsible for simulating the water level in the boiler. The boiler periodically — in response to an S\_UPDATE event which is scheduled every 5 time units — calculates the current water level based on the amount of water coming in from the pumps, the previous water level, the current steam generation rate and the parameters  $W$ ,  $U_1$  and  $U_2$ .

To model the random behaviour of the heat source, the boiler process needs to choose a value for the change in the steam generation rate is from the interval  $[-U_2, +U_1]$ . This is done by sending  $n_s$  S\_UPDATE events, each indexed with one value from the interval, and each with probability  $\frac{1}{n_s}$ . The parameter  $n_s$  (the number of steps) is given on the command line.

Also in response to the S\_UPDATE event, the measured water level and the steam generation rate are sent to the controller if the measuring units are functional; if the units are not functional, an illegal value is sent. The events W\_INF<sub>*i*</sub> and S\_INF<sub>*i*</sub> represent the water level and steam measurements, respectively. The events W\_FAIL and W\_REPAIR represent the failure and repair of the water level meter. A W\_FAIL event results in the scheduling of a W\_REPAIR event after  $t_{wr}$  time units (the repair time for the water meter). A W\_REPAIR event re-

sults in the scheduling of an  $W\_FAIL$  (probability 0.05) or an  $W\_REPAIR$  event (probability 0.95) with a delay of  $t_f$  (the boiler fail-cycle). The events  $S\_FAIL$ , and  $S\_REPAIR$  have the analogous meanings for the steam rate meter.

- *Control*. This is the most complex process and the component we are interested in validating. Following [1, 27], our controller has 4 modes: *normal*, *degraded*, *rescue*, and *emergency stop* (actually, the specification in [1] has an additional *initialization mode* which we omit for brevity). The four modes have these functions:

- *Normal Mode*: In this mode, both measuring devices are functioning correctly; the controller tries to maintain the water level between  $N_1$  and  $N_2$ .
- *Degraded Mode*: The steam rate meter has failed, but the water level measuring unit is still functional.
- *Rescue Mode*: The water level measuring device has failed, but the steam measuring device is still functional.
- *Emergency Stop Mode*: Both measuring devices have failed, or the water level threatens to exceed  $M_2$  or go below  $M_1$ . We consider this mode to be an *external* mode, where operators external to the system are responsible for a safe shutdown.

In the *normal*, *degraded* and *rescue* modes, the controller works by periodically — every 5 seconds — calculating the low  $E_l$  and high  $E_h$  estimates of possible water levels after 5 time units. Since this range depends on the number of pumps that are switched on, the calculation is done for all possible (given the number of operational pumps) numbers of pumps that are on. Among the obtained ranges, the controller chooses the range which satisfies the constraints  $E_l \geq M_1$ ,  $E_h \leq M_2$  and whose midpoint  $\frac{E_l + E_h}{2}$  is closest to the midpoint of the range  $N_1, N_2$ .

The estimates  $E_l$  and  $E_h$  depend on the current water level and the current steam generation rate. If one of the measuring devices is defective, the controller

can obtain a simple estimate for the missing value as follows. Let the current water level and steam generation rate be  $w$  and  $s$ , respectively, and the previous values, i.e., 5 seconds ago,  $w'$  and  $s'$ , respectively. Let  $n$  be the number of pumps that are on. If the water level measurement device is defective, the current water level can be estimated as

$$w = w' - s \times 5 \text{ sec} + n \times r_p \times 5 \text{ sec}$$

Similarly, if the steam rate measurement device is broken, the current steam generation rate can be estimated as

$$s = \frac{w' - w + n \times r_p \times 5 \text{ sec}}{5 \text{ sec}}$$

These estimates can then be used to calculate values for  $E_l$  and  $E_h$ . The relevant part of the perform function of the controller process is shown in figures 6.18 and 6.19.

Our validation system does not allow global states without successors because such states are flagged as deadlocks. To implement an emergency stop mechanism, we therefore had to resort to a trick: when the controller decides that an emergency stop is needed, it sends a STOP message to the other processes — causing them to cease all activity — and also to itself. When it receives the STOP message, it re-sends it to itself. Thus, the global emergency stop state has one successor (itself), and is not recognized as a deadlock.

Figure 6.20 shows the event exchange diagram for our model. We fix some of the model parameters as follows:

- $N_1 = 40\%$ ,  $N_2 = 60\%$ ,  $M_1 = 30\%$ ,  $M_2 = 70\%$  of the boiler capacity  $C$
- $U_1 = U_2 = 1 \text{ litre} / \text{sec}^2$
- the controller update cycle  $t_c$  is 5 seconds
- the boiler fail cycle is  $t_f = 6$  seconds (i.e., every 6 seconds, the boiler water meter and/or steam rate meter may fail with probability 0.05)
- the boiler repair time is  $t_b = 24$  seconds

```

void st_control::perform(int type) {

    int value = type & 255;
    type >>= 12;

    switch(type) {
        case C_UPDATE:
            // determine our mode depending on what's working and what's not
            Mode = NORMAL;
            if (!Water_OK && Steam_OK)
                Mode = RESCUE;
            else if (Water_OK && !Steam_OK)
                Mode = DEGRADED;
            else if (!Water_OK && !Steam_OK)
                Mode = EMERGENCY;

            switch (Mode) {
                case NORMAL:
                    // find best number of pumps to switch on
                    int best_p = calc_best_p();
                    if (best_p >= 0) {
                        // send message to pumps process, and an update
                        // reminder to ourselves
                        val_kernel->newevents(2,
                            new val_event((C_UPDATE<<12), this, Cycle),
                            new val_event((PUMPS<<12)+best_p, Pumps, 0));
                    } else {
                        // we're in trouble - we didn't find anything that
                        // guarantees the system to remain in [M1,M2], so
                        // generate emergency stop messages
                        val_kernel->newevents(3,
                            new val_event((STOP<<12), this, 0),
                            new val_event((STOP<<12), Pumps, 0),
                            new val_event((STOP<<12), Boiler, 0));
                    }
                    Prev_Water_Level = Water_Level;
                    Prev_Steam_Rate = Steam_Rate;
                    break;
                case DEGRADED:
                    // estimate the steam generation rate, then proceed as in
                    // normal mode
                    Steam_Rate = (Prev_Water_Level - Water_Level +
                        Pumps_ON * Pump_rate * Cycle) / Cycle;
                    [as in NORMAL mode]
                    break;
                case RESCUE:
                    // estimate water level, then proceed as in normal mode
                    Water_Level = (Prev_Water_Level - Steam_Rate * Cycle +
                        Pumps_ON * Pump_rate * Cycle);
                    [as in NORMAL mode]
                    break;
            }
    }
}

```

Figure 6.18: Perform Function of Controller Process (part 1)

```

        case EMERGENCY:
            // send an emergency stop message to all devices and to
            // ourselves
            val_kernel->newevents(3,
                new val_event((STOP<<12), this, 0),
                new val_event((STOP<<12), Pumps, 0),
                new val_event((STOP<<12), Boiler, 0));
            break;
        }
    break;
    case P_INF:
        // got data from pump
        Pumps_ON = value & 15;
        Pumps_OK = (value>>4) & 15;
    break;
    case W_INF:
        // got value from water level measuring device
        if (value >= 0 && value <= 100) {
            Water_Level = value;
            Water_OK = 1;
        } else {
            // garbage value, device is out of order
            Water_OK = 0;
        }
    break;
    case S_INF:
        // got value from steam rate measuring device
        if (value >= 0 && value <= 100) {
            Steam_Rate = value;
            Steam_OK = 1;
        } else {
            // garbage value, device is out of order
            Steam_OK = 0;
        }
    break;
    case STOP:
        // re-send stop message to ourselves
        val_kernel->newevents(1,
            new val_event((STOP<<12), this, 0));
    break;
    default:
        cerr << "*** Control Process: unknown event type error.\n";
}
};

```

Figure 6.19: Perform Function of Controller Process (part 2)

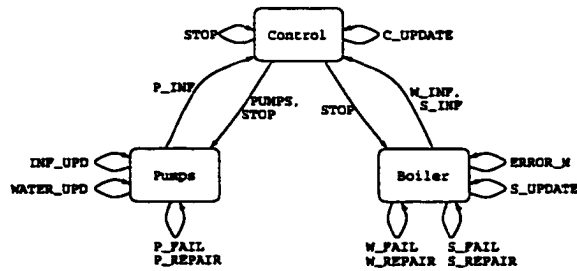


Figure 6.20: Event Exchange Diagram for Steam Boiler Model

$C$	the capacity of the boiler in litres
$r_p$	pump rate per pump (in litres/sec)
$W$	maximum steam generation rate
$n_s$	number of possible new values for the change in the steam generation rate

Table 6.13: Parameters for Steam Boiler Model

- the pump fail cycle is  $t_p = 6$  seconds

The remaining three parameters are variable and summarized in table 6.13. In its initial state, the boiler is 50% full, all devices are functional, and the steam generation rate is zero, and all pumps are off.

## 6.4.2 Experimental Results

Figures 6.20 and 6.21 show the traces for two simulation runs. The graphs plot the water level vs. the simulation time. They show that in the first scenario ( $C = 1000, r_p = 15, W = 35, n_s = 5$ ), the water level remains well within the safe zones, while in the second scenario ( $1000, r_p = 10, W = 35, n_s = 5$ ), the water level quickly threatens to go below  $M_1$  and the controller has to initiate an emergency stop.

Our control program is supposed to ensure that the water level remains between  $M_1$  and  $M_2$  under all circumstances. The boiler process detects violations of this requirement and sets the error flag. Thus, we can validate the program with a state space search (without model-checking). We first attempt to use the same parameters we used in the simulation runs. This is not successful, however, because the validator runs out of stack memory after reaching a search depth of about 9000. To reduce the state space, we use two state space reduction strategies:



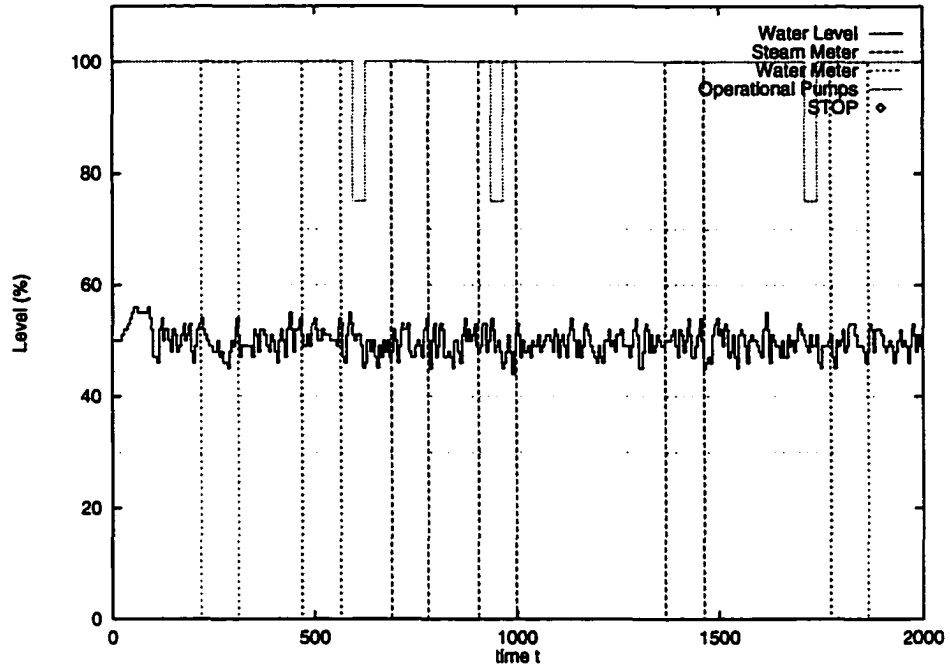


Figure 6.21: Simulation Run, pump rate 15,  $W = 35$

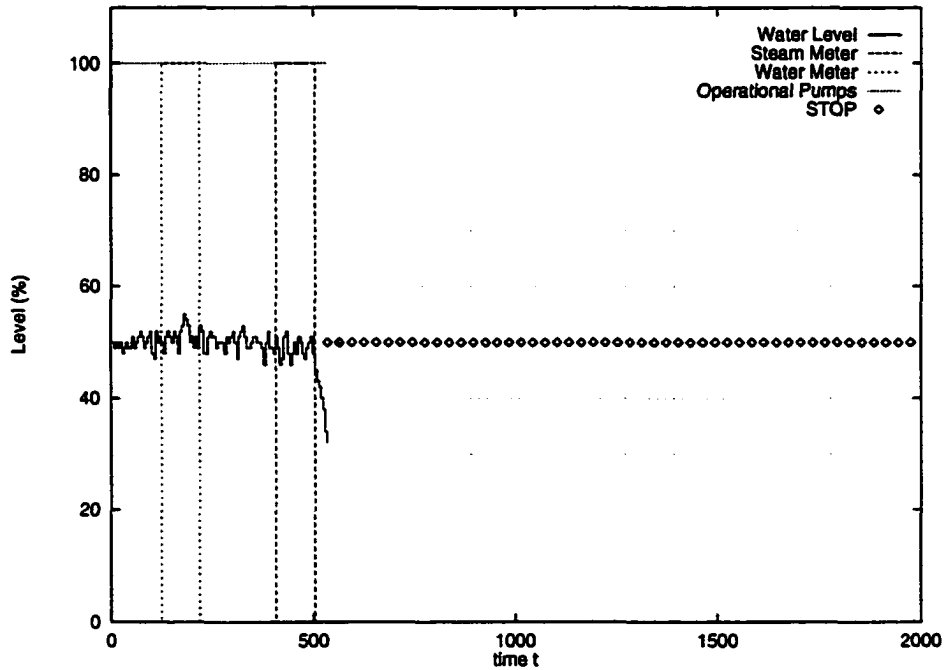


Figure 6.22: Simulation Run, pump rate 10,  $W = 35$

- We reduce the resolution of the model by setting  $C = 50$ . As a result, the boiler and controller state spaces are reduced by a factor of 20. The values for  $U_1, U_2, W$  and  $r_p$  are scaled appropriately and set to 0.25, 0.25, 4 and 1, respectively.
- We reduce  $n_s$  from 5 to 2, meaning that only the minimum and maximum possible changes in the steam generation rate can occur, resulting in a smaller state space. This change is reasonable, since any problems with the controller program are likely to occur in these extreme situations.

With these changes, we can validate the model; the size of the state space is approximately 790000 states.

### 6.4.3 Comparison with other Tools

The steam boiler control program example has been examined with other validation tools, for example SPIN[27, 63], FOCUS and CIP [1]. These tools have in common that they provide special-purpose languages for the specification of systems: PROMELA in SPIN, LUSTRE in FOCUS, and the process algebra CIP. Compared to our C++ model, these languages permit a highly abstract and more concise description of the system. They do not support quantitative time and probabilities; instead, much of the system's behaviour (e.g., failure of components) can be left unspecified. While this has the advantage that it is often convenient to leave unknown parameters unspecified, it makes it difficult to obtain performance measures from the model. For example, to obtain performance measures from the LUSTRE model described in [1], the authors have to essentially construct a probabilistic model "by hand" and solve it using techniques from probability theory.

All three tools let the designers check the property that the water level never exceeds  $M_2$  and never goes below  $M_1$ . However, verifying the control program in a model with unspecified behaviours says little about its usefulness: it is possible to design a simple control program that always goes into the emergency stop mode. Such a program would be proven "correct" by the validation tool, but is of little use in practice. Our approach, on the other hand, lets us examine the detailed performance

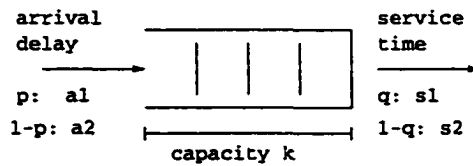


Figure 6.23: Simple Queue

$k$	$p$	$a_1$	$a_2$	$q$	$s_1$	$s_2$	$negOVER$
5	0.1	5	10	0.8	5	10	FALSE
15	0.1	5	10	0.8	5	10	FALSE
5	0.1	5	20	0.5	7	8	FALSE
5	0.1	10	20	0.5	7	8	TRUE
15	0.1	5	10	0.8	1	2	TRUE

Table 6.14: Model-checking Results for  $\neg OVER$

of the control program under a variety of conditions. The price of this convenience is that we have to make decisions about the qualitative parameters of the system.

## 6.5 Queueing System

In this section, we study a simple queueing system (figure 6.23). It consists of a packet queue with capacity  $k$  and a server. The interarrival time between packets is either  $a_1$  (with probability  $p$ ) or  $a_2$  (with probability  $1 - p$ ). Similarly, the service time for a packet is either  $s_1$  (with probability  $q$ ) or  $s_2$  (with probability  $1 - q$ ). The system is modeled using three event types: INIT (initialization), ARRIVAL (packet arrival), EOS (end-of-service). An additional event type — OVER — is generated whenever a buffer overflow occurs.

We can check if the queueing system is free from overflows, i.e., we can check for the EL formula  $\neg OVER$ . This is not very meaningful, however, as a simple analysis shows that the system is free from overflows if and only if both possible service times  $s_1$  and  $s_2$  are less than the shortest interarrival delay. If there is an  $s_i$  that is greater than one of the interarrival times  $a_j$ , then there always a non-zero probability that enough successive service times have duration  $s_i$  and enough successive interarrival times have duration  $a_j$ , eventually filling up the buffer. The model-checking results in table 6.14 confirm this view.

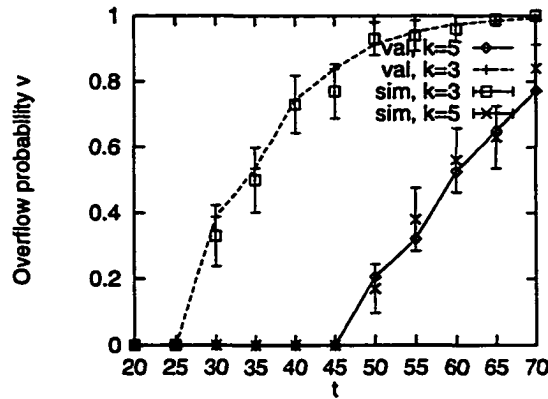


Figure 6.24: Overflow Probability, Simulation and Validation

Therefore, in the case of a queueing system like ours, it is more meaningful to look at probabilistic performance measures. Certain questions regarding the performance of a protocol (e.g., the average delay of a queue) can be answered both by simulation and by checking for a property like “the probability of the queue length being greater than  $x$  is less than  $y$ ”. We want to compute the probability  $v$  of observing at least one buffer overflow within  $t$  time units of starting the system. This value can be quickly estimated by running a few simulations, or obtained exactly by using the validator to calculate the measure of paths with lengths  $< t$  that satisfy the property  $\neg\text{OVER}$ .

We set  $p = 0.9, a_1 = 5, a_2 = 10, q = 0.1, s_1 = 5, s_2 = 10$  and obtain the overflow probability for different values of  $t$ . Figure 6.24 shows the results obtained by validation and by simulation. The simulation results were obtained for 100 simulation runs; the figure shows the 95% confidence interval. In all cases, the (exact) overflow probabilities calculated by the validator are within the simulation confidence intervals.

Both approaches give quick results for small values of  $t$ . As  $t$  grows larger, the state space of the validator explodes (figure 6.25, the y-axis is log-scaled) and it becomes more feasible to estimate  $v$  via simulation. The reason for the inefficiency of the validator lies in our time-limited validation procedure (section 4.3.7) which works by assigning values 0 or 1 to states that are roots of subtrees that are satisfying or nonsatisfying, respectively. In other words, for a state to be assigned  $I = 1$ , it must *inevitably* lead to an overflow event before the time limit  $t$ , and for a state to be assigned  $I = 0$ , it must be impossible for a state to lead to an overflow before the

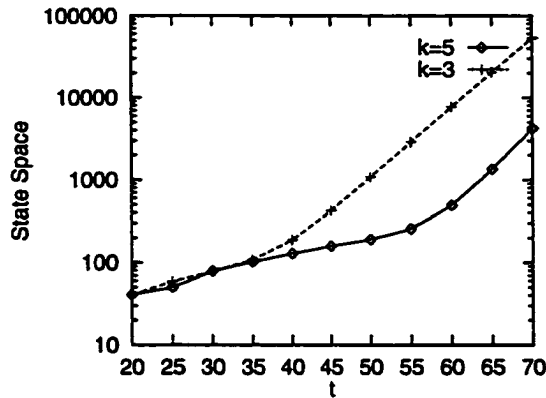


Figure 6.25: Size of State Space for Time-Limited Search

time limit. In the queueing system, such states are very rare — in most cases, it is possible but not inevitable that an overflow occurs, and the validator is forced to search a search tree whose size is exponential in  $t$ . Note that this reasoning does not extend to unlimited searches — such searches are generally much quicker: in the case  $k = 5$ , for example, the validator obtains the result that all paths are non-satisfying after searching only 35 states!

Finally, we change the arrival and service times to be exponentially distributed. The resulting queue is the well-known  $M/M/1/k$ -queue. We want to test to what extent the approximation of the exponential function with a finite number of support points affects the accuracy of performance evaluation results. The mean occupancy of the  $M/M/1/k$ -queue can be easily calculated using results from queueing theory (see, e.g., [61]). Figure 6.26 plots the mean occupancy of the queue for  $k = 10$  over the system load defined as  $\lambda/\mu$ , where  $\lambda$  is the arrival rate and  $\mu$  is the service rate. There are four curves: one for the exact analytical result and three curves obtained by simulations in which the exponential functions were approximated by  $ns = 2$ ,  $ns = 5$ , and  $ns = 10$  support points. The results indicate that while  $ns = 2$  does not yield a useful approximation, the plots for  $ns = 5$  and  $ns = 10$  are nearly indistinguishable from the exact analytical curve. Thus, this experiment, although by no means exhaustive, does indicate that reasonable performance evaluation results can be obtained by approximating the exponential function with as few as 5 support points.

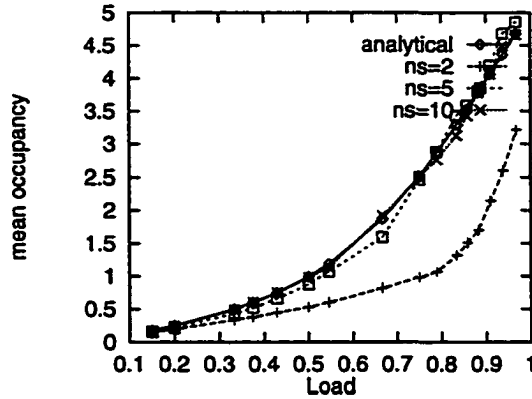


Figure 6.26: Accuracy of Finite Approximation for M/M/1/k

## 6.6 Multimedia System

We examine a simple multimedia transmission/presentation system (figure 6.27). A generic multimedia system consists on  $n$  streams of periodic data, each of which may have different periods, processing delays and transmission channels. These streams are merged at the presentation device. In our model, we have  $n = 2$  streams: sound and video data. The periods of the two streams are  $p_s$  and  $p_v$ , respectively. If  $g$  is the greatest common divisor (gcd) of the two periods, then the joint period of the two streams is  $(p_s p_v)/g$ , because  $(p_s p_v)/g$  is divisible by both  $p_s$  and  $p_v$ . In order to check the synchronization of the two streams, they are tagged with sequence numbers as follows: the stream with the shorter period — sound in this case — is the reference stream. Its packets are tagged with numbers  $0, 1, \dots$  up to the end of the joint period, and each time period of length  $p_s$  is associated with the tag of the packet sent at the beginning of the period. The packets of the stream with the longer period are tagged with the tags associated with the periods that are within  $p_s/2$  of the packet time. Figure 6.29 illustrates this for the case of  $p_s = 12$  and  $p_v = 40$ . The joint period is then 120.

Using the sequence numbers, synchronization errors of the video packets relative to the reference sound stream can be detected at a simple presentation device without using timers. A video packet is considered to be out of sync if none of its tags correspond to the tag of the sound packet preceding it.

The implementation of this example is straightforward and uses three process

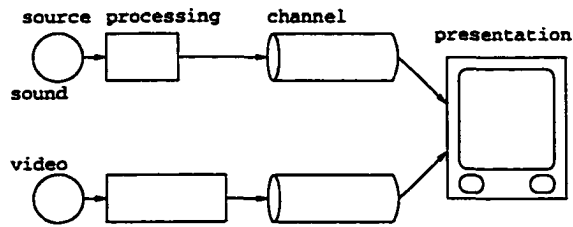


Figure 6.27: Multimedia Stream Model

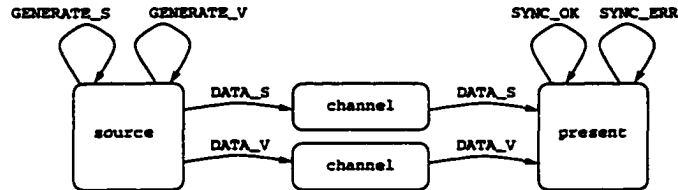


Figure 6.28: Event Exchange Diagram for Multimedia Stream Model

types: the source, the presentation device and a channel process. Figure 6.28 shows the processes and the events exchanged between them. The presentation device process generates `SYNC_OK` and `SYNC_ERR` events to itself in response to video packet receptions. Because EL formulas are expressed in terms of event types, exposing synchronization errors as events makes it possible to write EL specifications about them.

First, we do a state space search for different delay values to check for non-temporal errors like deadlocks and unspecified receptions. No EL formula is needed for this test. Given the simplicity of the system, it is not surprising that no such errors are detected. Table 6.15 summarizes the results for different delay values. Next, we check for synchronization errors. We want to specify that all video packets are in sync, i.e., that they result in the generation of a `SYNC_OK` event within some time limit  $t$ . In EL, this is expressed as a formula

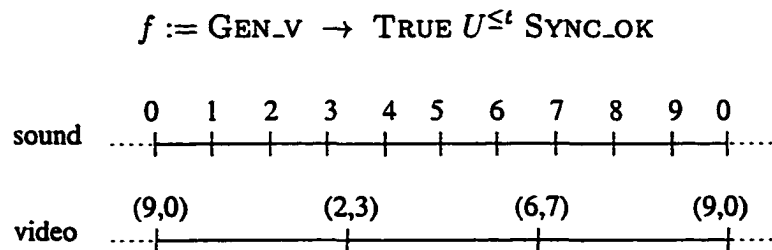


Figure 6.29: Packet Tags for Two Streams

Sound Del	Video Del	Channel Del	No. States	$f$
10	10	60	55	True
10	10	[50,60]	1079	True
10	[8,12]	[50,60]	32467	False
10	[7,13]	[50,60]	84838	False

Table 6.15: State Space and Synchronization, without Synchronizer

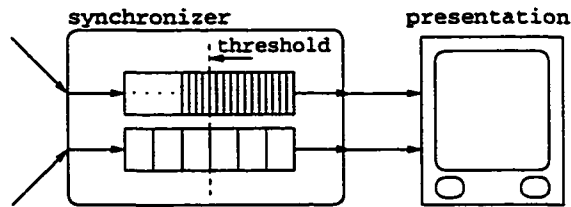


Figure 6.30: Synchronizer

where  $t$  is set to be greater than the sum of the maximal processing and channel delays. Since out-of-sync packets result in the generation of an `SYNC_ERR`, we can equivalently (and more efficiently) check for the formula

$$f := \neg \text{SYNC\_ERR}$$

The results for this test are in the last column of table 6.15.

In addition to the basic system, we consider a multimedia system with a simple synchronization device (figure 6.30). The synchronizer works by buffering packets of both streams until both buffers are at least half full. Once both buffers have exceeded this threshold, it retrieves packets with the same period as the sources and continues buffering any incoming packets. The buffers are dimensioned to be twice the size of the joint period of the two streams ( $= 2 \times 120 = 240$  in this case), so the buffer sizes for the sound and video streams are 20 and 6, respectively.

Running the same tests as above, we find that the EL formula is satisfied in all cases. Note that the addition of the synchronizer results in a considerably larger state space. Due to memory constraints, the fourth experiment (final row in table 6.16) could not be completed as an exhaustive search – we performed a probabilistic search with threshold  $10^{-15}$  instead.



Sound Del	Video Del	Channel Del	No. States	$f$
10	10	60	124	TRUE
10	10	[50,60]	206852	TRUE
10	[8,12]	[50,60]	1139013	TRUE
10	[7,13]	[50,60]	1104382	TRUE

Table 6.16: State Space and Synchronization, with Synchronizer

## 6.7 Alternating Bit Protocol

The alternating bit protocol [43] is very simple, yet powerful enough to — under certain conditions — correctly transmit data over lossy channels. Due to its simplicity, the protocol is commonly used as an example in the protocol verification literature.

We model the protocol with three process types, *sender*, *receiver* and *channel* that work as follows:

- The *sender* receives data packets from the higher-level protocol entity and tags them with a control bit. The first packet is tagged with bit 0, the second with bit 1, the third again with bit 0, and so on. After tagging the packet, the sender transmits it via the channel to the receiver and starts a timer. No other packets are transmitted until either the packet is acknowledged by the receiver or a timeout occurs. If the packet is acknowledged (as indicated by the control bit of the ACK message), then the timer is reset, and another packet is transmitted. If a timeout occurs, the packet is re-transmitted.
- The *receiver* receives packets from the channel and submits them to the higher-level entity if they have the expected control bit: the first expected control bit is 0, then 1, then 0, and so on. Any packet, whether it is expected or not, is acknowledged with an ACK packet bearing the same control bit as the packet.
- The *channel* is a simple process modelling a unidirectional lossy channel (thus, the protocol needs two such processes). Packets and ACKs are either passed on correctly or lost with a given loss probability.

In addition to these alternating bit protocol entities, we require two processes repre-

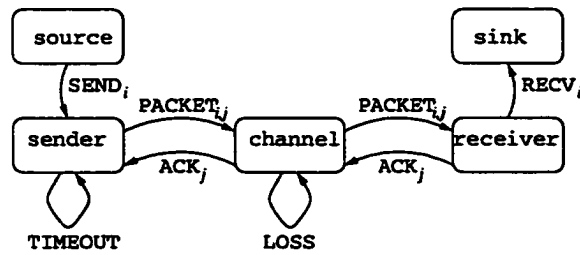


Figure 6.31: Event Exchange Diagram for the Alternating Bit Protocol

sending the packet source and sink that model the higher-level entity. The alternating bit protocol provides a service to these entities. To see the reason for having these two processes in our model, consider the problem of verifying that the alternating bit protocol correctly transfers packets and preserves their sequence. From the protocol point of view, there are only two kinds of packets — those with control bit 0 and those with control bit 1. As long as successive packets differ in the control bit, the receiver process accepts them as correct data. To verify that packets are sent in the correct order, we need processes that are *outside* the core protocol. The source and sink processes check for out-of-sequence packets by attaching sequence numbers to the data that is generated and received. These sequence numbers are invisible to the alternating bit protocol. For our case, we limit the protocol state space by using sequence numbers module 4, i.e., the numbers range from 0 to 3.

Figure 6.31 shows the event exchange diagram for the protocol including the source and the sink. The generation of data by the source and the submission of data to the sink are modeled with  $\text{SEND}_i$  and  $\text{RECV}_i$  events, respectively, where  $i$  is the sequence number. The packets and ACKs are modeled with  $\text{PACKET}_{i,j}$  and  $\text{ACK}_j$  events, where  $i$  is the sequence number and  $j$  is the control bit. Timeouts are modeled with  $\text{TIMEOUT}$  events.

Our implementation of the protocol has three parameters: the timeout delay  $t_o$ , the channel delay  $d$  and the channel loss probability  $p_l$ . To keep the state space small, we do not introduce any time non-determinism. Figure 6.32 shows the complete 49-state TPTS for the parameters  $t_o = 30$ ,  $d = 10$  and  $p_l = 0.5$ . In this diagram, the protocol global states are displayed as 4 letters “ABCD” where A is the state of the source (indicating the next sequence number to be sent), B is the state of the sink (indicating the next expected sequence number), C is the state of the sender (indi-

Channel	Timeout	$P(f) \geq 0.9$
5	10	TRUE
5	20	TRUE
5	50	FALSE
10	20	TRUE
10	30	TRUE
10	50	FALSE
20	40	FALSE
20	50	FALSE
20	60	FALSE

Table 6.17: Alternating Bit Protocol and an EL Formula

cating the next control bit to be sent), and D is the state of the receiver (indicating the next expected control bit).

First, we run a plain state space search to check for deadlocks, out-of-sequence packets and similar errors. No errors are detected. Next, we want to check whether the protocol delivers packets within a given time limit  $t$  with at least some probability  $p_s$ . In EL, this translates to the non-probabilistic formula

$$f := \text{SEND}_i \rightarrow \text{TRUE } U^{\leq t} \text{RCV}_i$$

where  $i$  is one of the possible sequence numbers. The probabilistic requirement, then, is  $P(f) \geq p_s$ . On the unmodified protocol, however, the probability of  $f$  being satisfied in a lossy system is zero, because, with probability 1, any infinite path in the protocol contains enough successive packet losses to violate the time limit  $t$ . Therefore, we modify the protocol to send only *one* packet and check whether this packet is received within the time limit. Table 6.17 shows the results for different values of the channel delay and timeout parameters. The channel loss rate  $p_l$  is set at 0.5, and  $p_s$  to 0.9. In order for the formula to be satisfied, the channel delay must be small, and the timeout bound tight.

## 6.8 Sliding Window Protocol

In this section, we use a unidirectional sliding window protocol (SWP) [80] to test the behaviour of probability-threshold based searches. SWP can be viewed as a

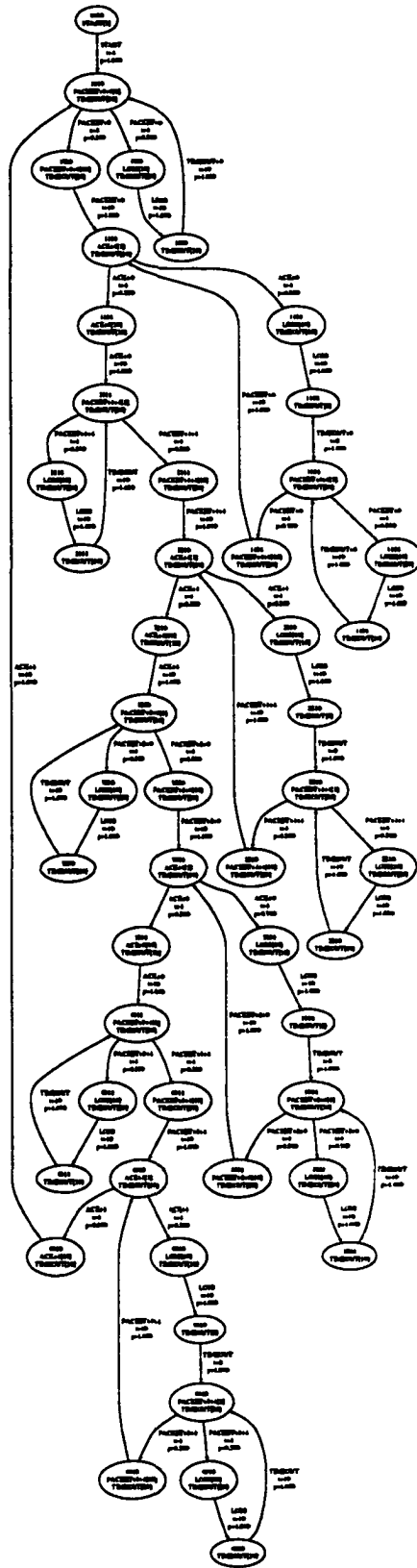


Figure 6.32: TPTS for Alternating Bit Protocol

generalization of the alternating bit protocol, and variations of it are widely used in transport layer protocols such as TCP [80].

Like our implementation of the alternating bit protocol, the implementation of SWP has 6 processes: *source*, *sink*, *sender*, *receiver*, and 2 *channels* (one for each direction). The sliding window protocol is implemented in the sender and receiver processes. The sender can activate and de-activate the source. Briefly, the sender and receiver processes work as follows (see, e.g., [80] for a more detailed description).

- *Sender*. A basic sender parameter is the *window size*  $n$ : the sender sends up to  $n$  packets without receiving an acknowledgment, buffers the packets and starts a timer associated with each packet. These packets are given successive sequence numbers modulo  $n+1$  (note that there are thus  $n+1$  different sequence numbers :  $0 \dots n$ , but only  $n$  packets may be unacknowledged).

If an acknowledgment arrives with the sequence number of one of the buffered unacknowledged packets, it means that all buffered packets up to and including the one whose sequence number is being acknowledged have been received correctly. The sender deletes the timers associated with the packets, discards the packets from the buffer, and sends more packets. If a timer for a packet expires, all packets in the buffer are retransmitted; and the timers are re-started.

The data that the sender transmits originates from the source process. To implement a flow control mechanism, the sender has the ability to enable and disable the source, depending on whether the packet buffer in the sender is full or not.

- *Receiver*. The receiver process keeps track of the next expected sequence number, starting at 0. If a packet arrives with the correct sequence number, it is submitted to the sink process. All arriving packets — regardless of whether the sequence number is the expected one or not — are acknowledged with a packet bearing the last correctly received sequence number, i.e., the sequence number before the expected one.

Figure 6.33 displays the event exchange diagram for our model. Data generated by the source and submitted to the sink are represented by the event types `DATA_S` and

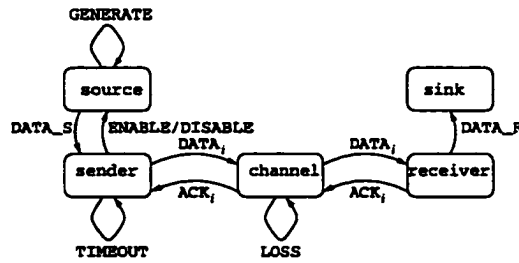


Figure 6.33: Event Exchange Diagram for the Sliding Window Protocol

DATA<sub>R</sub>, respectively. The generation of data happens in response to GENERATE events and can be started and stopped by the sender with ENABLE and DISABLE events. Within the protocol layer, data packets are modeled with DATA events, acknowledgments with ACK events, and timeouts with TIMEOUT events.

The model has three parameters: the window size  $n$ , timeout  $t_o$ , and the channel delay  $t_c$ . Our validator allows the search to be limited by a probability threshold, by a depth threshold, or both. In order to assess the effectiveness of these approaches, we compare the behaviour three searches for the parameter values  $n = 3$ ,  $t_o = 100$  and  $t_c = [18, 22]$ .

- a depth-threshold search using the approximate time interval algorithm
- a probability-threshold search using the approximate time interval algorithm
- a probability-threshold search using the exact algorithm

The probability thresholds are set to  $2 \times 10^{-10}$  for the exact algorithm,  $10^{-9}$  for the approximate algorithm, and the depth threshold is set to 64. These values were chosen to result in the same state space for all searches — about 70000 states. Figure 6.34 compares the search behaviour: it plots the number of visited validator states over the search depth.

Comparing the approximate depth-threshold and probability threshold searches, we observe that the maximum search depth for the depth-threshold search is 64, with many states being searched around that depth, while in the probability-threshold search many states around that depth are bypassed in favour of higher-probability states at deeper levels. The maximum search depth is more than 100 in this case. A comparison of the two probability-threshold searches shows that the approximate

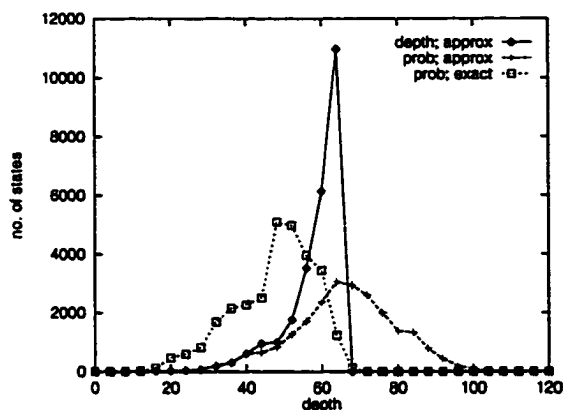


Figure 6.34: Depth Threshold vs. Probability Threshold

algorithm alleviates the state space explosion somewhat, resulting in a right-shift of the graph.

The sliding window protocol does not work always correctly if packets can be re-ordered by the lower layers (IP [80], for instance, may re-order packets). Figure 6.35 shows two kinds of errors that can occur in this case. In both cases, the result is that a delayed duplicate packet is wrongly accepted by the receiver and forwarded to the sink (thick line in the figures). To detect errors like these, we need to add an observer to our protocol. Our observer counts the data packets `DATA_S` sent from the source and the number of packets `DATA_R` sent to the sink and makes sure that the latter does not exceed the former. The observer code fragment for this is very simple:

```

if (type == DATA_S)
    Count++;
else if (type == DATA_R) {
    if (Count)
        Count--;
    else {
        flag_error();
    }
}

```

The first error occurs when the timeout value is smaller than the maximum round-trip delay. Figure 6.35 (left) shows this error for  $n = 1$ . For various combinations of

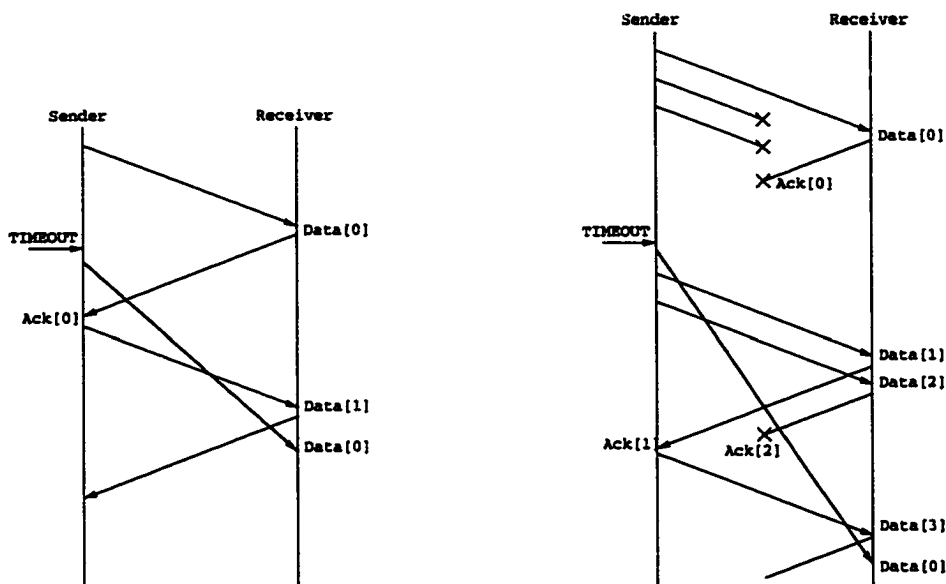


Figure 6.35: Some SWP Errors

$n, p$	$10^{-3}$	$10^{-6}$	$10^{-7}$	$10^{-8}$	$10^{-10}$
1	yes 200				
2	no 1k	yes 32k			
3	no 1k	no 112k	no 500k	yes 878k	
4	no	no 112k		no 2.1M	no 11.2M

Table 6.18: Probability Thresholds and Error Detection

probability thresholds and SWP window sizes, table 6.18 lists whether the error was detected or not and how many states were searched by the validator.

The second kind of error is similar to the first but can happen *even if the timer values are set correctly*. Figure 6.35 (right) shows an example of this error for  $n = 3$ . However, the probability of this kind of error is very small because a number of unlikely events need to occur in succession (it almost seems as if the network would have to be particularly malicious). It can be argued that in practical networks with large window sizes, the probability of such an error is zero. For a SWP with maximum sequence number 3, our validator detected the this error when the probability



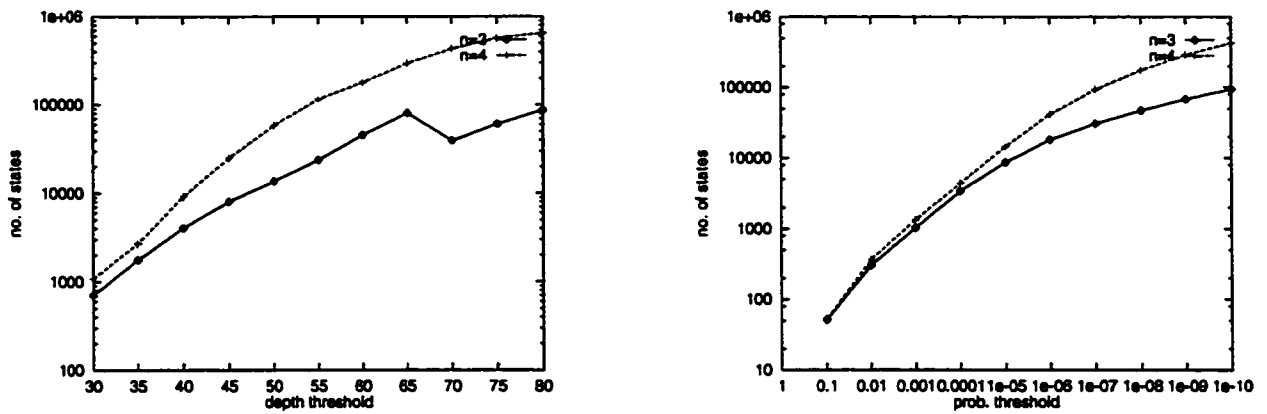


Figure 6.36: State Space vs. Thresholds

threshold was set to  $10^{-10}$ .

These examples show that the choice of probability threshold can influence whether an error in a protocol is detected or not. It is of course impossible to know the “right” probability threshold before actually finding the error. In practice, the threshold will be dictated by external factors such as the available computing power and the available amount of time. For depth-based searches, the state space is initially exponential in the depth threshold until the growth slows because the state space is ultimately finite (figure 6.36 left). Probability-threshold searches (figure 6.36 right) exhibit a similar behaviour. Protocol designers will try to set the threshold as low as possible while still achieving an acceptable response time.

## 6.9 Summary

A number of examples from different application domains were used to demonstrate the prototype implementation of the validation system. In general, the validator appears to be a useful tool for detecting errors in system specifications, and for obtaining performance measures.

In some cases, an exhaustive state space search was not possible due to resource limitations. Some strategies to cope with the state space explosion problem are

- threshold searches based on time, depth of probability
- reduced-complexity models that can be obtained by reducing the time non-determinism (i.e., using smaller time intervals) or by using fewer support points

for the exponential approximation

As a rule of thumb, checking for EL formulas is significantly more expensive than a raw state space search — programmers should therefore try to incorporate correctness checks into the process models if possible.

# Chapter 7

## Conclusion and Future Research

In this thesis, we have presented a model for the probabilistic validation of communication protocols and a tool based on the model. Compared to other validation systems, our tool has several advantages:

- protocols can be modeled realistically, including their timing constraints and their probabilistic behaviour
- the high-level model is based on the well-known DES paradigm
- the tool uses C++ as its specification language, thereby eliminating the learning curve associated with special-purpose languages
- due to the use of a DES-like high-level model, the same model can be used both for validation and for performance evaluation

Combined, these advantages result in a more PRACTICAL validation tool than was previously available. Two main contributions of this thesis are algorithms for the validation of probabilistic claims (expressed in the logic EL) and the integration of simulation and validation in one framework.

The important question, of course, is “Does it work?”. The answer depends on the point of view. On the one hand, we were only able to exhaustively search the state space for greatly simplified protocols — thus, we can not convincingly claim that our tool *proves* the correctness of protocols. Moreover, even in the cases where we could perform an exhaustive search, the result is not a conclusive proof of correctness due to the probabilistic nature of our algorithm, and due to the fact that the validator

itself has of course never been proven to work correctly. From the point of view of correctness proofs, the answer must therefore be, “No”.

On the other hand, our numerous experiments from different application domains show that our tool is without doubt useful for the modelling, validation and performance analysis of protocols. During our experiments, the validator detected numerous errors — many unintentional ones, and some so arcane that they would have been extremely difficult to detect without the aid of our tool. Therefore, from the point of view of detecting errors in protocols, the answer to the question is “Yes”. We believe that this aspect of the system is particularly useful because it comes essentially for “free” since performance evaluation by DES is routinely performed anyway.

Our tool is only a prototype and therefore has a number of shortcomings that can be addressed in future research. Improvements are needed at the C++ level, at the algorithmic level, and at the user interface level. At the C++ level, many obvious optimizations were omitted in order to code for correctness rather than speed — for example, the depth-first-search relies on inefficient recursive calls to the validation procedure. More significant are the possible improvements at the algorithmic level. Here, we believe that we can take advantage of the many improvements that continue to be made to SPIN, because the tools have many similarities. Some of these advances deal with partial state reduction techniques and hash compaction; more improvements are regularly presented at the annual SPIN workshops. Finally, improvements are of course needed at the user, i.e., programmer, interface level. In particular, our support for statistics is rudimentary and should be brought nearer to the high level utility available in simulation tools like SMURPH.

Due to its inherent difficulty, protocol validation will remain a fertile research field for the foreseeable future. It is our expectation that the advantages of our approach will lead to increased use of automated validation tools like ours and, ultimately, to better protocols.

# Bibliography

- [1] J-R. Abrial, E. Börger, and H Langmaack. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control (LNCS 1165)*. Springer-Verlag, 1996.
- [2] M. Ajmone Marsan, G. Balno, G. Chiola, G. Conte, S. Donatelli, and G. Franceschinis. An introduction to generalized stochastic Petri nets. *Microelectron. Reliab.*, 31:699–725, 1991.
- [3] R. Alur. Timed automata. *NATO-ASI summer school on verification of digital and hybrid systems*, 1998.
- [4] R. Alur, R. K. Brayton, T. A. Henzinger, A. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. *Computer Aided Verification 97*, pages 340–363, 1997.
- [5] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. *Proceedings, Logic in Computer Science*, pages 414–425, 1990.
- [6] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for probabilistic real-time systems. *Proceedings of the 18th ICALP, LNCS 510*, 1991.
- [7] R. Alur, C. Courcoubetis, and D. Dill. Verifying automata specifications of probabilistic real-time systems. *Real-Time: Theory in Practice, LNCS 600*, pages 28–44, 1991.
- [8] R. Alur and D. Dill. The theory of timed automata. *Real-Time: Theory in Practice, LNCS 600*, pages 45–73, 1991.
- [9] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. *Real-Time: Theory in Practice, LNCS 600*, pages 74–106, 1991.

- [10] A. F. Ates, Bilgic M., S. Saito, and B. Sarikaya. Using timed CSP for specification, verification and simulation of multimedia synchronization. *IEEE Journal on Selected Areas in Communications*, 14:126–136, 1996.
- [11] A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. *Computer Aided Verification 95*, pages 155–165, 1995.
- [12] S. Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 1987.
- [13] C. Baier, E. M. Clarke, and V. Hartonas-Garmhausen. Probabilistic VERUS: Semantic foundations and practical results. *Proceedings of PROBMIV 98*, 1998.
- [14] C. Baier, E. M. Clarke, M. Kwiatkowska, V. Hartonas-Garmhausen, and M. Ryan. Symbolic model checking for probabilistic processes. *Proceedings of ICALP 97*, 1997.
- [15] C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. Technical Report CSR-96-12, University of Birmingham, School of Computer Science, 1996.
- [16] C. Baier, M. Kwiatkowska, and G. Norman. Computing probability lower and upper bounds for LTL formulae over sequential and concurrent Markov chains. *Proceedings of PROBMIV 98*, 1998.
- [17] C. Baier, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. Technical Report CSR-97-2, University of Birmingham, School of Computer Science, 1997.
- [18] A. Basu, G. Morrisett, and T. von Eicken. Promela++: A language for constructing correct and efficient protocols. *IEEE INFOCOM '98*, 1998.
- [19] M. Bernardo and R. Gorrieri. Extended markovian process algebra. *Proceedings of CONCUR 96*, pages 315–330, 1996.

- [20] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. *Foundations of Software Technology and Theoretical Computer Science, LNCS 1026*, pages 499–513, 1995.
- [21] G. v. Bochmann and C. A. Sunshine. Formal methods in communication protocol design. *IEEE Transactions on Communications*, pages 624–631, 1982.
- [22] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, pages 25–59, 1987.
- [23] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30:323–342, 1983.
- [24] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
- [25] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking:  $10^{20}$  states and beyond. *Proceedings of 5th Annual Symposium on Logic in Computer Science*, pages 428–439, 1990.
- [26] L. Cacciari and O. Rafiq. A temporal reachability analysis. *Proceedings of Protocol Specification, Testing, and Verification XV*, pages 35–49, 1995.
- [27] T. Cattel and G. Duval. Specifying and verifying the steam boiler problem with SPIN. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control (LNCS 1165)*, 1996.
- [28] G. Chiola, M. Ajmone Marsan, G. Balbo, and G. Conte. Generalized stochastic Petri nets: A definition at the net level and its implications. *IEEE Transactions on Software Engineering*, 19:89–106, 1993.
- [29] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logics specification: A practical approach. *Proceedings, 10th PoPL*, pages 117–126, 1983.
- [30] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logics specifications. *ACM Transactions on Programming Languages and Systems*, pages 244–263, 1986.

- [31] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in Systems Design*, 1:275–288, 1992.
- [32] C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite-state probabilistic programs. *Proceedings, 29th Symposium on the Foundations of Computer Science*, 1988.
- [33] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42:857–907, 1995.
- [34] J-P. Courtiat and R. C. de Oliveira. About time nondeterminism and exception handling in a temporal extension of LOTOS. *Proceedings of the Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 70–78, 1995.
- [35] J. Crow, A. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to pvs. *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques*, 1995.
- [36] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. *Proceedings, DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.
- [37] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. *Proceedings, IEEE Real-Time Systems Symposium '95*, 1995.
- [38] L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997.
- [39] L. de Alfaro. Temporal logics for the specification of performance and reliability. *Proceedings of STACS'97*, pages 165–176, 1997.
- [40] L. de Alfaro. Stochastic transition systems. *Proceedings of CONCUR'98*, 1998.
- [41] S. Donatelli, H. Hermanns, J. Hillston, and M. Ribaud. GSPN and SPA compared in practice. *Qualitative Modelling in Parallel Systems*, 1995.



- [42] S. Donatelli, M. Ribaud, and J. Hillston. A comparison of performance evaluation process algebra and generalized stochastic petri nets. *Proceedings, 6th International Workshop on Petri Nets and Performance Models*, 1995.
- [43] P. Gburzynski. *Protocol Design for Local and Metropolitan Area Networks*. Prentice Hall, 1996.
- [44] F. Gomes, S. Franks, B. W. Unger, Z. Xiao, J. Cleary, and A. Covington. SimKit: A high performance logical process simulation class library in C++. *Proceedings of the 1995 Winter Simulation Conference*, 1995.
- [45] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
- [46] H. Hansson and B. Jonsson. A framework for reasoning about time and reliability. *Proceedings, IEEE Real-Time Systems Symposium '89*, pages 102–111, 1989.
- [47] J. I. Hartog and E. P. de Vink. Mixing up nondeterminism and probability: A preliminary report. *Proceedings of PROBMIV '98*, 1998.
- [48] M. Hennessy. *Algebraic Theory of Processes*. The MIT Press, 1988.
- [49] T. A. Henzinger, X. Nicollin, S. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.
- [50] H. Hermanns, U. Herzog, and V. Mertsotakis. Stochastic process algebras – between LOTOS and markov chains. *Computer Networks and ISDN Systems*, 30:901–924, 1998.
- [51] H. Hermanns, V. Mertsotakis, and M. Rettelbach. Performance analysis of distributed systems using TIPP – a case study. *Proc. of the 10th U.K. Performance Engineering Workshop for Computer and Telecommunication Systems, Edinburgh*, 1994.
- [52] J. Hillston and M. Ribaud. Stochastic process algebras: A new approach to performance modelling. *Modelling and Simulation of Advanced Computer Systems*, 1998.

- [53] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [54] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
- [55] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, pages 279–295, 1997.
- [56] C.-M. Huang and S.-W. Lee. Timed protocol verification for Estelle-specified protocols. *ACM Computer Communication Review*, pages 4–32, 1995.
- [57] C.-M. Huang, S.-W. Lee, and J.-M. Hsu. Probabilistic timed protocol verification for the extended state transition model. *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*, pages 432–437, 1994.
- [58] P. Iyer and M. Narasimha. Probabilistic lossy channel systems. *Proceedings of TAPSOFT 97*, pages 667–682, 1997.
- [59] H. E. Jensen, Larsenm K. G., and A. Skon. Modelling and analysis of a collision avoidance protocol using SPIN and UPPAAL. *Proceedings of 2nd SPIN Workshop*, 1996.
- [60] C. Kern and M. R. Greenstreet. Formal verification in hardware design: A survey. *ACM Computing Surveys*, 1998.
- [61] L. Kleinrock. *Queueing Systems, Volume I: Theory*. John Wiley & Sons, 1975.
- [62] F. J. Lin and M. T. Liu. An integrated approach to verification and performance analysis of communication protocols. *Proceedings of Protocol Specification, Testing, and Verification VIII*, pages 125–140, 1988.
- [63] S. Loeffler and A. Serhouchni. Creating a validated implementation of the steam boiler control. *Proceedings of 3rd SPIN Workshop*, 1997.
- [64] N. F. Maxemchuk and K. Sabnani. Probabilistic verification of communication protocols. *Proceedings of Protocol Specification, Testing, and Verification VII*, pages 307–320, 1987.

- [65] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [66] R. E. Miller and Y. Xue. Bridging the gap between formal specification and analysis of communication protocols. *Proceedings, 15th Annual Phoenix Conference on Computers and Communications*, pages 225–231, 1996.
- [67] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [68] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77:541–577, 1989.
- [69] K. Naik and B. Sarikaya. Testing communication protocols. *IEEE Software*, pages 27–37, 1992.
- [70] T. Nakatani. Verification of group address registration protocol using PROMELA and SPIN. *Proceedings of 3rd SPIN Workshop*, 1997.
- [71] M. Nesi. Value-passing CCS in HOL. *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and Its Applications — HUG'93*, pages 352–365, 1993.
- [72] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. *Real-Time: Theory in Practice, LNCS 600*, pages 526–547, 1991.
- [73] A. Pnueli and L. D. Zuck. Probabilistic verification. *Information and Computation*, 103:1–29, 1993.
- [74] R. Ranjan, J. Sanghavi, R. K. Brayon, and A. L. Sangiovanni-Vincentelli. High performance BDD package based on exploiting memory hierarchy. *Proceedings of the Design Automation Conference 96*, 1996.
- [75] T. Regan. Multimedia in temporal LOTOS: a lip-synchronization algorithm. *Proceedings of Protocol Specification, Testing, and Verification XIII*, pages 127–143, 1993.
- [76] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, MIT, 1995.

- [77] A. C. Shaw. Communicating real-time state machines. *IEEE Transactions on Software Engineering*, 18:805–816, 1992.
- [78] U. Stern and D. Dill. Improved probabilistic verification by hash compaction. *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 206–224, 1995.
- [79] U. Stern and D. Dill. A new scheme for memory-efficient probabilistic verification. *Formal Description Techniques IX*, 1996.
- [80] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 1996.
- [81] S. Tripakis and Courcoubetis. C. Extending Promela and SPIN for real time. *Tools and Algorithms for the Construction and Analysis of Systems TACAS'96*, pages 329–348, 1996.
- [82] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. *Proceedings, Logic in Computer Science*, pages 332–343, 1986.
- [83] R. von Glabbeek, S. A. Smolka, B. Steffen, and C. M. N. Tofts. Reactive, generative and stratified models of probabilistic processes. *Proceedings, Logic in Computer Science*, 1990.
- [84] C. H. West. Protocol validation by random state exploration. *Proceedings of Protocol Specification, Testing, and Verification VI*, pages 233–242, 1987.
- [85] J. Yuan, J. Shen, J. Abraham, and A. Aziz. On combining formal and informal verification. *Computer Aided Verification 97*, pages 376–387, 1997.