Analysing Industrial Protocols with Formal Methods

Copyright © 1999 J.M.T. Romijn.
ISBN 90-9013086-1
IPA Dissertation Series 1999-09

Typeset with LaTeX 2_ε.
Printed by Thela Thesis, Amsterdam.
Cover by Total Design, Amsterdam.

The research reported in this thesis was carried out as part of the project "Specification, Testing and Verification of Software for Technical Applications" at the Stichting Mathematisch Centrum for Philips Research Laboratories under Contract RWC-061-PS-950006-ps. The research has been carried out under the auspices of the Institute for Programming Research and Algorithmics (IPA).

ANALYSING INDUSTRIAL PROTOCOLS WITH FORMAL METHODS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr. F.A. van Vught,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 15 oktober 1999 te 15.00 uur.

door

Judi Maria Tirza Romijn

geboren op 13 september 1971

te Zaandam

Dit proefschrift is goedgekeurd door de promotoren

Prof.dr. H. Brinksma
Prof.dr. F.W. Vaandrager

Preface

In 1995, the Philips Research Laboratories in Eindhoven signed a contract with CWI to fund a PhD student at CWI. The contract was based on a research proposal by Ed Brinksma (University of Twente), Jan Friso Groote (University of Utrecht) and Frits Vaandrager (Centre for Mathematics and Computer Science, Amsterdam). Ron Koymans (Philips Research Laboratories, Eindhoven) became the project manager for the project at Philips, Frits Vaandrager was the project manager at CWI from August 1995 up to January 1996, subsequently Jan Friso Groote became project manager at CWI. Ed Brinksma became the promotor. I was hired as PhD student for four years at CWI.

At the end of these four years, I am glad to be able to express my thanks in the document one is supposed to deliver. Even though it is impossible to thank everybody, I make an effort here.

The first thanks go to the four captains on this ship: Ed, Frits, Jan Friso and Ron have made sure that there was work to be done, that the work gave rise to papers with scientific value, that these papers were finished and published, that a thesis was written, and last but not least, that I learned something from each case study.

Henk Apers, Hubert Garavel, Joost-Pieter Katoen and Thijs Krol are kindly thanked for their willingness to judge this thesis and take part in the committee. Hubert and Joost-Pieter have helped in improving the presentation with numerous useful comments.

Fortunately, I have not been working alone. Of the co-authors I especially want to thank Jan Springintveld (Chapters 4 and 5) and Frits Vaandrager (Chapter 2). Both have taught me a great deal about writing and science. Jean Moonen is thanked for a pleasant dual presentation in South Korea of the work presented in Chapter 4. The hard labour one faces when trying to build a tool environment, implement an algorithm or run an existing tool on extravagantly large examples, was relieved by the cooperation with/advice from Rudi Bloks, Hubert Garavel, Gerard Holzmann, Radu Mateescu, Jean Moonen, Jan Springintveld and Eddy Zondag.

At CWI, I have had four very good years, for which nice surroundings and colleagues are indispensable. My office mates Jan, Joost, David, Tobias, Hugo, Michel, and Mark have each contributed in their own way to the progress of my work and/or the improvement of my mood.

I would like to thank some other colleagues, friends and family with the following, very incomplete list: paranimfen Bas and Jan; Bert, Doeko, Izak, Jaco vdP, Jos, Marco, Sjouke, Thijs, Vincent, Yaroslav, Mieke & Marja, Jaco dB, and the other AP/SEN members and vis-

itors; the researchers working at related departments in KUN, TUE, UT, UU, UvA and VU; Frans, Jaap, Minnie, Simone, Walter, Debby, and the other PV-bestuurders; Joke, Barry, Henk, Jacques, and the other skaters; Annette, Dejuan, Lynda, Miriam, Natalia, Sylvia, and the other wimmin; de kantine; Carol, Dennis & Lieve & kleine Judi, Marieke, Miriam, Karin, Kathy, Saske, Suzanne, Arjan & Annette, Kor & Gwenn, Annemarie & Michael, Robbert & Ramya & Nina, Joop & Wil, Maria & Joost, Ruben, Job & Lonneke, Huib & Marthe, pap & mam, opa & oma Banga, and Otje.

Finally, I reserve the greatest thanks for Wan, who is always willing and able to understand, comfort, enjoy and love me.

Amsterdam, August 1999

Contents

Preface v

Contents vii

1 Introduction 1
1.1 Industrial scope 1
1.2 Formal methods 3
1.3 Protocols 6
1.4 The project 7
1.5 The formal methods and tools that were used 8
1.6 The case studies 11
1.6.1 Case 1: The RPC/Memory specification problem 12
1.6.2 Case 2: IEEE 1394 verification 13
1.6.3 Case 3: Automatic VHDL testing 14
1.6.4 Case 4: Symmetry reduction in test generation 15
1.6.5 Case 5: A software architecture 16
1.6.6 Case 6: HAVi DCM Management 16

2 A note on fairness in I/O automata 19
2.1 Introduction 19
2.2 Definitions 20
2.3 Main Result 21
2.4 Composition 24

3 The RPC-Memory specification problem 27
3.1 Introduction 27
3.1.1 Specification problem 28
3.1.2 Notes on the problem specification 30
3.1.3 Notes on the I/O automata model 31
3.2 Preliminaries 32
3.2.1 Fair I/O automata 32

3.2.2 Details on fair I/O automata . . . 32
3.3 Specifications and verifications for Problem 1 . . . 33
3.3.1 Problem 1(a): Specification of two memory components . . . 33
3.3.2 Problem 1(b): *RelMemory* implements *Memory* . . . 37
3.3.3 Problem 1(c): Nothing but $MEM_FAILURE_P$ actions? . . . 38
3.4 Specifications and verifications for Problem 2 . . . 39
3.4.1 Problem 2: Specification of the RPC component . . . 39
3.5 Specifications and verifications for Problem 3 . . . 41
3.5.1 Problem 3: Specification of the composition . . . 41
3.5.2 Set-up for the verification . . . 45
3.5.3 Problem 3: *MemoryImp* implements *Memory* . . . 45
3.6 Specifications for Problem 4 . . . 52
3.6.1 Problem 4: Specification of a lossy RPC . . . 52
3.7 Specifications and verifications for Problem 5 . . . 52
3.7.1 Problem 5(a): The RPC implementation *RPCImp* . . . 54
3.7.2 Problem 5(b): *RPCImp* implements *RPC* . . . 56

4 Automated conformance testing of VHDL designs 61
4.1 Introduction . . . 61
4.2 Global description of test environment and test process . . . 63
4.3 Stepwise through the testing process . . . 65
4.3.1 Generating tests with the Conformance Kit . . . 65
4.3.2 From abstract tests to executable tests . . . 66
4.3.3 Executing tests at the VHDL level . . . 68
4.4 Experiences . . . 71
4.5 Related work . . . 72
4.6 Later developments . . . 73

5 Exploiting symmetry in protocol testing 75
5.1 Introduction . . . 75
5.2 Finite state machines . . . 76
5.3 Symmetry . . . 77
5.4 Construction of a kernel . . . 79
5.5 Test derivation from symmetric Mealy machines . . . 84
5.6 Patterns . . . 88
5.7 Examples . . . 94
5.7.1 A chatbox service . . . 94
5.7.2 A cyclic train . . . 97
5.7.3 Implementing the algorithm Kernel . . . 99
5.8 Future work . . . 99

6 The HAVi leader election protocol 101
6.1 Introduction . . . 101
6.2 The DCM Manager leader election protocol . . . 102
6.2.1 HAVi components . . . 103
6.2.2 Protocol . . . 105

6.3 Languages and tools . . . 105
6.3.1 Spin and Promela . . . 106
6.3.2 Lotos, Cæsar/Aldébaran and Xtl . . . 106
6.4 Modelling decisions . . . 106
6.5 Model checking experiments . . . 109
6.5.1 Safety: At most one leader . . . 112
6.5.2 Safety: Best candidate becomes final leader . . . 113
6.5.3 Safety: All agree on the final leader . . . 114
6.5.4 Liveness: Eventually there will always be a final leader . . . 117
6.5.5 Is the HAVi protocol wrong? . . . 120
6.5.6 Statistics . . . 121
6.6 Conclusions . . . 124
6.6.1 Concerning Spin . . . 124
6.6.2 Concerning Cæsar/Aldébaran and Lotos . . . 127
6.6.3 Comparison of the tools . . . 128
6.6.4 Concerning this experiment . . . 130

7 The IEEE 1394 leader election protocol 131
7.1 Introduction . . . 131
7.2 The protocol . . . 133
7.2.1 The IEEE 1394 tree identify phase . . . 133
7.2.2 Other verifications of the protocol . . . 134
7.2.3 This verification . . . 136
7.3 IOA models . . . 138
7.4 Network preliminaries . . . 142
7.4.1 Networks . . . 142
7.4.2 Paths, cycles . . . 143
7.4.3 Connected networks . . . 144
7.5 Verification . . . 147
7.5.1 Invariants for TIP3 and TIP4 . . . 147
7.5.2 TIP4 implements TIP3 . . . 151
7.5.3 Liveness results for TIP4 . . . 152
7.5.4 Are the IEEE 1394 timing constants correct? . . . 156
7.6 Conclusions . . . 157

8 Conclusions 159
8.1 Project objectives . . . 159
8.2 Does the hypothesis hold? . . . 161
8.3 Future directions . . . 163

Bibliography 164

A I/O automata 177
A.1 Safe I/O automata . . . 177
A.2 Live I/O automata . . . 180
A.3 Timed I/O automata . . . 181
A.4 Fair Timed I/O automata . . . 184

Samenvatting 187

Chapter 1

Introduction

This thesis is about the application of formal methods. It lists and evaluates the results obtained in a project which was carried out at the Centre of Mathematics and Computer Science (CWI) in Amsterdam for the Philips Research Laboratories in Eindhoven. It is organised as follows. This chapter contains short introductions to formal methods, the project and its scope, the methods used, and the case studies performed. Chapters 2 to 7 contain the scientific papers written in the scope of this project. Chapter 8 presents an evaluation of the results and some directions for future research.

1.1 Industrial scope

The present In modern households, many devices are controlled by computer technology (embedded systems). The type of devices controlled by embedded systems varies from electronic devices such as audio equipment to something as simple as a coffee machine. By adding software and/or hardware, more and more intelligence and functionality is added to the devices. An example of this is a coffee machine with a clock that can be used to have the machine make coffee autonomously at a certain time of the day. The next step in the development of such embedded systems is to allow the embedded systems in different devices to communicate with each other. For instance, it is already possible to buy a multi-purpose remote control that is able to operate audio, video and other equipment from various vendors.

The future Let us consider a future household in which all embedded systems are able to communicate, which is shared by Jane and Ken. Jane is at work, and suddenly is inspired to prepare a certain dish for dinner. Since she is not sure of the contents of the refrigerator, she phones home to find out what is in stock. Her answering machine gets all the information from a camera installed in the refrigerator, including the expiry dates that Jane should take into account. On her way home she buys the groceries that she needs. When she enters her apartment, she is recognised by the sound of her voice and greeted with information on her favourite news items, on the people that phoned or visited the apartment, and on what the cat has been up to. She asks for a cd to be played, and automatically her profile is checked to find out which tracks to play in what order. While she goes into the kitchen, the music is redirected

to the kitchen speaker set. Now Ken comes home with a new cd player. He connects the device to a power outlet and to the video recorder, and puts in one of his favourite cds. The cd player says hello to the other devices in the network and obtains Ken's profile for the cd that it is supposed to play.

Should we classify this story as science fiction? It is clear that today's state of technology is not able to support the scenario sketched in a generic way. However, producers of consumer electronics are currently developing architectures that enable products of multiple vendors to communicate when combined in a network. These architectures are expected to be applied in products and have the above scenario become reality in just a few years. Two examples of such architectures are HAVi [GHM+98] and Jini [Sun99]. HAVi is a joint effort by several companies to solve audio/video interoperability in home networks, with IEEE 1394 FireWire [IEE96] as underlying medium. Jini is an initiative by Sun Microsystems, which is based on Java and supposed to connect arbitrary electronic devices.

How? As may be clear from our peep into the future, there are several advantages to such networks of devices controlled by intelligent embedded systems:

- Globally available user profiles. The preferences of each person need to be stored only once, and all devices in the household can afterwards access this information and operate according to the preferences.
- Plug and play with little user interaction. Whenever a device is added to the household, it is able to get acquainted with the other devices its new environment by itself, the user of the equipment only needs to connect the device to a power source and the network of devices already present. Removing a device leads to immediate notification of the parties that should be informed.
- Dynamic services from the network. When a user wants to operate a certain device, it may interact with another device that passes the message on to the actual destination. One could for instance ask the computer in the study to have the cd player in the living room skip the next track of the cd which is being played.
- Future proof. If the embedded systems are intelligent enough, they can learn about functionalities or services developed later, by communicating with devices with newer versions or finding information on the internet. Hence the need for the user to upgrade things vanishes.

While apparently desirable, the features described are quite hard to establish. We list some technical points that need to be addressed, and which are of crucial importance to the proper functioning of the technology:

1. Plug and play. It is important to know at all times which devices are present in the network. Whenever a device is introduced or taken away, the other devices should be informed about this, so they can act according to the new situation. It is not trivial to ensure that the information about the devices present is up to date when many changes follow each other in a short period of time.
2. Bandwidth requirements. A fair division of the bandwidth over the different applications, each with their own requirements, is complicated. Audio and video applications

need a large amount of bandwidth on the connecting medium, in order to guarantee a good service. Communication that concerns control can usually be done with less bandwidth, but requires acknowledgements and such.

3. Robustness. The products to be sold are destined for end users with possibly little technical knowledge. It is important that the operation of the network is transparent to and capable of handling errors by the user.

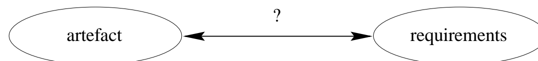
When solutions for these issues are devised, the question will remain whether the solutions work in all imaginable situations. This is where formal methods may help. The remainder of this chapter explains what formal methods are about and how they may be used, and gives an introduction to the research presented in this thesis.

1.2 Formal methods

Formal methods are the applied mathematics of computer system engineering, and are used to construct and/or make judgements about computer system artefacts. By artefacts we mean designs as well as implementations, and software as well as hardware. Examples of mathematical techniques used by formal methods are predicate calculus (first order logic), recursive function theory, lambda calculus, programming language semantics, and discrete mathematics (number theory, abstract algebra, etc).

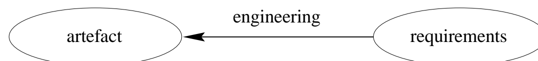
We now explain the basic terminology of formal methods by giving examples whole of what can be done with formal methods, inspired by [WM97, WM98]. Note that our interpretation of the terminology is one in a range of (subtly) different interpretations. Therefore this section also serves to fix the starting point and avoid confusion.

Suppose we start with a set of requirements, and an artefact, which should have some relation with the requirements:

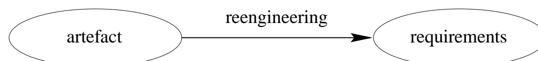


The requirements give the intended functionality and the artefact is the proposed recipe for obtaining that functionality. With formal methods we try to characterise the relationship between the requirements and the artefact.

Another possibility is to start with just requirements and derive an artefact from these that provides the required functionality, which is called *engineering*. When using formal methods to do this, we speak of *correctness by design*:

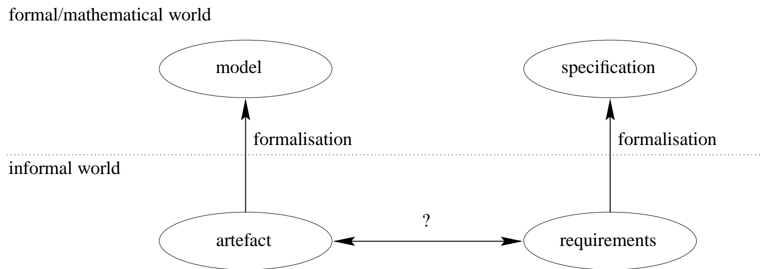


Finally, if we start with an artefact, and want to derive requirements from these, this is called *reengineering* or *reverse engineering*.



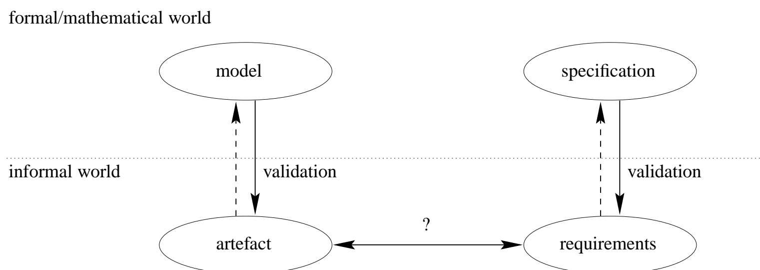
In this thesis, the research is restricted to the use of formal methods where the requirements and artefact are given.

Formalisation In practice, the requirements and artefact tend to be described in an informal and equivocal manner and often have (too) many details. Such descriptions are not suitable for using formal methods because they are not sufficiently precise or too complicated or too large. In order to obtain descriptions that are suitable for formal methods, we may *formalise* the requirements and the artefact:



Formalisation means translating a description in an informal language to a description in a formal language. A formal language is defined by a formal syntax, and often has associated *semantics*, which give precise meaning to expressions in the syntax. Different kinds of formal languages are suitable for expressing different kinds and different aspects of artefacts. In the formalisation decisions like the following often have to be made: details of the informal description are omitted (abstraction), assumptions are made and equivocal parts of the description are disambiguated. Even if a language is not completely formal, then it can still be possible to give very precise descriptions in that language. In such a case formalisation is easy. With the term *specification*, we refer to the formal representation of the requirements. With the term *model*, we refer to the formal representation of the artefact.

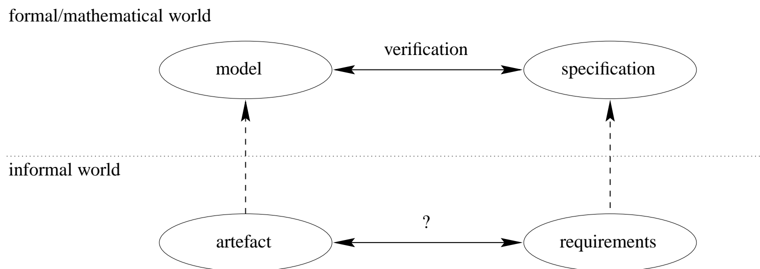
Validation The formalisation of requirements or artefact may not be correct, because of the abstractions, assumptions and/or interpretations made. The task of establishing whether the formalisation is correct is often referred to as *validation*. In the following figure, the dashed arrows depict the formalisation performed earlier.



In many cases, validation means that the formal and informal descriptions are compared by manual study, or that people responsible for the informal description are consulted. If there is tool support for the language of the formal descriptions, then features like syntax checking, type checking or simulation can support the validation.

Verification When we are interested in the correctness of an artefact with respect to requirements, we may establish a formal relation between the formal representations of these, i.e.

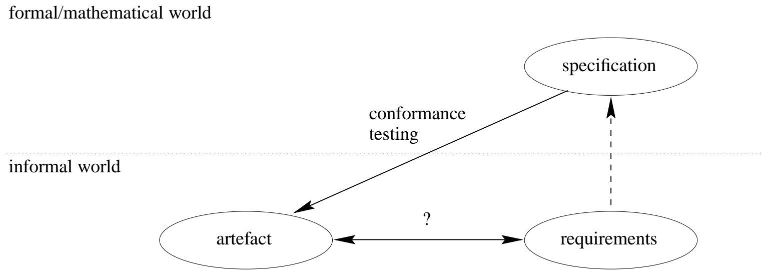
between the model and the specification. When this relation is given by mathematical proof, we speak of *verification*. In the following figure, the dashed arrows depict the formalisation performed earlier.



The relation between the specification and model says something about the behaviour that is allowed by the specification and the behaviour that the model exhibits. It may be that one is included in the other, or that they are equal. We distinguish two main approaches to establish the relation by verification, namely *theorem proving* and *model checking*. These are the exponents of a spectrum of mixed approaches. With theorem proving we denote the construction of a mathematical proof, which can involve all sorts of proof techniques such as proof by contradiction, by induction, etc. With model checking we denote the exhaustive exploration of all possible behaviours to show that the relation holds. Both kinds of verification may be supported by tools: model checking by model checkers, and theorem proving by proof checkers, proof assistants or theorem provers. Proof checkers take a complete proof as input and check whether all the steps in this proof are mathematically sound. Proof assistants are interactive proof checkers: the tool is able to check steps and provide suggestions, but the user has to direct the proof in an interactive way. Theorem provers attempt to find a proof without user guidance, with techniques like resolution. Proof assistants are expensive in *human effort*, in that it takes relatively much user interaction to construct a proof. Theorem provers and model checkers are expensive in *machine effort*, in that it it takes relatively much machine resources (memory/disk space, computing power) to construct and explore a proof tree, or to explore all possible behaviours of a model or specification.

When it has been established that a certain relation holds between specification and model, it remains to be seen whether this is also true at the informal level, between the requirements and the artefact. The validation of the formalisation is often used as a justification that indeed the relation in the formal world implies the relation in the informal world.

Conformance testing When a relation between any two of the entities artefact, requirements, model and specification is established by experiment, we speak of *testing*. Experiments are conducted by executing or simulating an artefact. Testing can support validation or substitute verification. There are many different kinds of testing, based on what the purpose of testing is. Some examples are functional, performance, reliability, and robustness testing. When testing is used to establish the relation between an artefact and requirements, by conducting experiments on the artefact, then we speak of *conformance testing*. When using formal methods for conformance testing, the relation is established between the artefact and the formal representation of the requirements, i.e. the specification. In the following figure, the dashed arrow depicts the formalisation performed earlier.



Formal methods for testing consist of methods for *generating* and *implementing* experiments and *evaluating* the outcomes. Test generation is done using the specification. In most cases, test evaluation just says whether the artefact passes or fails. The methods are almost always based on the *test hypothesis*, which states that the system to be tested can be modelled in the formal language of the test method.

Testing is influenced by many factors, e.g. whether one has access to the structure/contents of the artefact (white box testing) or not (black box testing), in what environment the artefact must be tested and how it can be observed.

In most cases, exhaustive tests of an artefact are not feasible since this generally requires a very large, or even infinite, number of test cases to be executed. Therefore, the actual test set is usually constructed under a number of additional assumptions regarding the occurrence of errors in the form of a fault model or test hypotheses. In such cases, complete test coverage can be obtained only relative to the assumptions made. Because of these restrictions, testing typically aims at the exposure of errors and increasing the confidence in the correctness of artefacts, but is generally too weak to guarantee absence of errors.

1.3 Protocols

The focus of this thesis is on artefacts which are *protocols*. A protocol is an agreed-upon method of communicating information between two or more entities, using an underlying service or medium. There are three basic ingredients for a protocol: (1) the messages, and their intended meaning, (2) the order in which messages should be exchanged, and (3) the way in which the underlying service or medium is used.

Many protocols start with a connecting phase, followed by a phase in which important information is passed, followed by a termination phase, each with their associated messages. For instance, consider a telephone conversation. The telephone medium indicates the conversation request by ringing. The phone is unhooked and the conversation is started with some opening message like ‘hello’, and both the caller and the person called give their name. After this initial phase, some information is exchanged until one of the two indicates that the conversation must end. At this point, some terminating message like ‘goodbye’ is used by both parties, after which the phones are put on hook again.

Note that a protocol does not necessarily define equipment or specific software.

An *open protocol* is a set of detailed, published rules for communications. Open protocols are available for public use and implementation. These protocols are often developed by the joint effort of a group of vendors and/or individuals.

A *standard* protocol is an open protocol which is in many cases globally accepted and

used. Standard protocols are often published by a standards organisation, such as IEEE, ISO and ITU.

1.4 The project

The general goal of the project can be found in the following quotation from the project proposal:

> The primary objective of the proposed project is to demonstrate and assess the effectiveness of using formal methods in the software development process within Philips. This goal is to be achieved by a consortium of three well-established research groups on formal methods with complementary expertise. Together these groups will validate (critical parts of) the software for a number of selected applications within Philips.

This is a rather ambitious goal for a project in which the manpower is planned as follows: one PhD student funded full-time, and four people supervising of which one promotor and two project managers, not funded by the project and geographically far apart. Given that the duration of the project was only four years, it is fair to have modest expectations of the outcome of the research and the degree to which the aforementioned demonstration and assessment can take place.

The central hypothesis We formulate a hypothesis which expresses the goal of the project proposal in a modest way. The research presented in this thesis is supposed to give evidence that supports or refutes this hypothesis.

> *Using formal methods to support the industrial software development process can be effective.*

There is a wide range of articles on the use of formal methods in the industrial software development process. Of these, we only refer to [BH95b, BH95a, CW96, Hal90, Rus95].

Concrete project objectives, case studies The general goal of the project does not lead to a straightforward research plan. The project proposal suggests some concrete objectives and a plan, as follows.

> Concretely the project will address the following more specific goals:
>
> 1. Development of heuristics about when formal methods should be applied.
> 2. Improvement of methods and tools so that bigger applications can be dealt with faster.
> 3. Integration of complementary approaches within formal methods research.
> 4. Improvement of technology transfer process from formal methods research to practice.

> In order to achieve the goals described, it is planned that during the three first years of the project six applications will be dealt with. The fourth year is reserved to write a PhD thesis.

Ideally, the selection of case studies would take the relation to the project objectives into account: a case study that is expected to contribute to one (or more) of the goals mentioned should get preference over case study from which no such contribution is expected. Such expectations are not easy to come up with, especially for Items 1 and 4. Moreover, it turned out that the selection of case studies was limited by the availability of suitable industrial development projects, the possibility of applying formal methods and the estimation of the effort required, to such a degree that preference with respect to the concrete objectives has not been taken into account in practice. Hence, it is only afterwards (at the time of writing of this thesis), that it is evaluated how these case studies do or do not contribute to the concrete objectives of the project. This is done in detail in Chapter 8. The case studies were proposed by Frits Vaandrager (Case 1, 4) and Ron Koymans (Case 2, 3, 5, 6).

1.5 The formal methods and tools that were used

This section shortly introduces the formal methods and tools that were used. The verifications involve models in the formal languages I/O automata, Promela and Lotos and the tools Spin and Cæsar/Aldébaran. The testing theory and experiments involve Mealy machine models and the tool KNP Conformance Kit.

The differences between I/O automata, Promela and Lotos are found mostly in the verification approaches. The expressivity of the languages is comparable, which is illustrated by the translation from I/O automata descriptions to Promela presented in [Jen99].

A mathematical notion that is often used for the interpretation or comparison of models in formal methods is the *labelled transition system*. A labelled transition system is a set of states with labelled transitions between them. Each transition is supposed to represent an indivisible or atomic event, and the label indicates what the event is. One or more states may be designated as initial state, meaning that behaviour may start from these states.

I/O automata The input/output automaton model [LT87, LT89], developed by Lynch and Tuttle, is a labelled transition system model for components in asynchronous concurrent systems. The actions of an I/O automaton are classified as input, output and internal actions, where input actions are required to be always enabled. The output action of one I/O automaton may be the input action of one or more other I/O automata, which can be used to enforce multi-way synchronisation. An I/O automaton has "tasks"; in a fair execution of an I/O automaton, an enabled task cannot be ignored indefinitely. The behavior of an I/O automaton is describable in terms of traces, or alternatively in terms of fair traces.

I/O automata are mostly described in a precondition/effect style with state variables, where a state is a valuation of the state variables. It is often straightforward to model behaviour in terms of state variables and actions.

Verification on I/O automata is done with both model and specification described as I/O automata, with theorem proving techniques. The verification relation between the I/O automata is based on the behaviour of the I/O automata: trace inclusion. Given the I/O automata descriptions in precondition/effect style, the common approach to establish this is by means of

simulation relations [LV95]. With a simulation relation one shows that any observable event from one I/O automaton can always be imitated with some activity by the other I/O automaton. The simulation relation is stronger than trace inclusion in the sense that the former implies the latter, but not vice versa.

It is possible to include timing requirements in I/O automata and in simulation relations to establish that the timed behaviour of one I/O automaton is included in the timed behaviour of another I/O automaton [LV96].

The presentation of I/O automata can be done in the IOA language [GLV97], which facilitates the precondition/effect style and manipulation of data. Tool support is currently worked on, in the form of a simulator and translations to Promela, Java and the input language for the Larch theorem prover [GH93].

In my experience, the examples I worked on could be easily modelled as (timed) I/O automata. The proof techniques could almost always be used in a straightforward manner, and if this was not the case, it was not hard to find a small extension of the theory that supported my needs. The IOA language allows for natural representation of I/O automata.

Promela, Spin Promela (a Process Meta Language) [Hol91] is a non-deterministic language, loosely based on Dijkstra's guarded command language notation and Hoare's language CSP, extended with other constructs. It is the input language to the tool Spin [Hol91, Hol97]. Models in Promela consist of definitions of process behaviour, with variable assignments, sequential and alternative composition, repetition and dynamic process creation. Communication between processes happens on synchronous or asynchronous channels. Synchronous communication always involves two processes. The support of data types is limited: basic types are booleans and naturals, from which arrays and record structures can be built.

The tool Spin facilitates simulation and verification of Promela models. Different simulation possibilities are random, guided and interactive. Simulating behaviour of Promela models helps in understanding what has been modelled.

Verification is supported in Spin through model checking. Here the model is encoded in Promela and the specification in linear temporal logic (LTL, [Pnu77, MP92]). The verification is done on the fly: the global state space is not constructed, but explored directly from an interpreted version of the Promela code. This means that for each new verification run, the effort of constructing the state space while exploring has to be done anew, but also that only that part of the behaviour has to be constructed and explored which influences the validity of the property checked.

In my experience, Promela is easy to use when one has experience with imperative programming languages. The Spin tool support is good and the interfaces are very user-friendly. However, it can be hard to decide how to model complicated behaviour. Different choices may have a great impact on the tractability of the model by Spin, and combinations of synchronous communication and other language constructs can cause rather obscure behaviour. When the complexity of the behaviour that is modelled increased, I had to consult the semantic definition of the language more often. When modelling a property to be checked, it can be hard to determine a proper formula in LTL that captures the property precisely if the property is a little complex.

Lotos, Cæsar/Aldébaran Lotos [ISO89] is a message passing process algebra with two parts: a process algebra based on CCS [Mil89] which is used to describe the flow of control in

a system, and an abstract data type language to describe the information manipulated by that system. Together the two parts are known as full Lotos. The data part is expressed in ACT-ONE, an algebraic formalism for abstract data types, and the behaviour part is expressed in process algebra with sequential, alternative and parallel composition, and recursion. Communication happens on synchronous gates and can involve more than two processes. Verification is mostly done by comparison of behaviour using (bi)simulation relations with theorem proving techniques or (when using tools) with model checking, or by checking properties expressed in temporal logic with model checking.

Cæsar/Aldébaran [FGK+96] is a tool set that supports simulation and model checking of Lotos specifications. The model is given in Lotos, the specification can be either a Lotos description or a property expressed in temporal logic. Cæsar/Aldébaran has tools for simulation, generation, minimisation, comparison and checking temporal properties on labelled transition systems. Cæsar accepts Lotos as input language and generates a labelled transition system and simulates both Lotos models and labelled transition systems. Aldébaran composes, compares and minimises labelled transition systems. Xtl checks properties expressed in a choice of several temporal logics on labelled transition systems.

In my experience, each given Lotos model has a clear meaning which is easy to grasp. The Cæsar/Aldébaran tool support is good and the interfaces are user-friendly. When starting to use Lotos, I needed a good description of the semantics. As soon as familiarity with all Lotos operators was obtained, the references were used less often. However, it is not always easy to express desired behaviour in Lotos since (1) there are no global variables, (2) the data types are rather restricted, (3) all activity must be modelled as communication, and (4) each processes that can communicate on a gate, *must* participate in any communication occurring on that gate (enforced synchronisation)[1]. When modelling a property to be checked, it can be hard to determine the proper formula in one of the temporal logics supported by Xtl if the property is a little complex. This is not just because of the nature of temporal logics, but also because in the formula, there is no way to access the value of parameters to a Lotos process, and since there is no notion of (global) state variables in Lotos.

Mealy machines, conformance test methods A Finite State Machine (FSM) is a finite labelled transition system. A Mealy machine is an FSM where each transition is labelled with an *input/output* pair. The idea of the Mealy machine model is that the behaviour of reactive systems can be modelled as follows: if the system is in a state s and input i is applied, then an output o may occur and the system may go to state t. This is reflected in the transition $s \stackrel{i/o}{\rightarrow} t$ in the corresponding Mealy machine. When a Mealy machine is *deterministic*, the next state is determined by the current state, the input and the output. When a Mealy machine is *input deterministic*, the next state is determined by only the current state and the input. When a Mealy machine is *input enabled*, then in each state there is an outgoing transition for each input in the input alphabet.

Black box conformance testing on Mealy machines can be done with the *automata theoretic method* [Cho78], also referred to as the *W-method* [Vas73]. This method is based on the comparison of two input deterministic, input enabled Mealy machines, i.e. it is assumed that the specification is an input deterministic, input enabled Mealy machine and that the artefact to

[1] This is illustrated by any design in which three identical processes want to have synchronous communication occur between each combination of two of them

be tested can be modelled as an input deterministic, input enabled Mealy machine. The latter assumption is often referred to as the *test hypothesis* [Tre92]. Note that with black box testing, the structure of the artefact to be tested is not known; it suffices to estimate the number of states of the artefact. From the specification, a set of test sequences is derived, which can be applied to the artefact. The test derivation is based on the notion of *characterising sets* which contain for each pair of states with different behaviour, a test sequence to distinguish these states. The other ingredient of the test derivation is the *transition cover* which contains for each state a test sequence that brings the system to that state from the start state.

The test method is complete in the sense that it has been proved in [Cho78] that if the number of states of the artefact is estimated correctly and the behaviour of the specification and artefact is equal for all test sequences, then the specification and artefact are equal.

The advantage of the test method over other state based test methods is that it is able to detect many kinds of errors such as output errors, next-state errors, and missing or extra states. Difficulties with the method are that it may be hard to estimate the number of states of the artefact, and that the number of test sequences produced with this method becomes unproportionally high when the number of states in specification and artefact increase.

A variant of the automata theoretic approach is the UIO method [ADLU91, SD88] which has the same ingredients except for the characterising set. Instead, a Unique Input Output (UIO) sequence is associated with each state in the specification, such that the behaviour of the state under this sequence is different from any other state in the Mealy machine. The advantage of this method is that the number of test sequences becomes smaller, since in each state that must be distinguished from another, only one sequence will be used instead of all the sequences in the characterising set. The difficulty with this method is that the UIO sequence does not always exist and that the error detecting power is smaller.

Tool support for test generation in this fashion exists in the form of the KPN Conformance Kit [KWKK91]. This tool supports the automatic test generation from input deterministic, input enabled Mealy machines for the UIO method.

1.6 The case studies

In Figure 1.1 a time schedule is shown of the actual work on each of the case studies within the project. One should interpret this figure as follows. The lines in the figure indicate research activity. If there is no line for a certain case at a certain time, this means that at that moment no activity was taking place for that case. If there are several lines at a certain time, this means that the time was divided over these cases. This division was not always equal. The moment on which the last activity line for a case ends means that the case was ended with a final version of a paper to be published (Cases 1, 3), the case was terminated unsuccessfully (Case 5), the last tool experiments took place (Case 4), or a paper was finished for which publication and finalising has yet to take place, which is expected not to take longer than two weeks (Case 2, 6). Time spent on holidays, conference visits, school/course participation and illness has not been included in the figure for sake of readability. Over the period displayed, the time not spent on the project adds up to a total of 6 months.

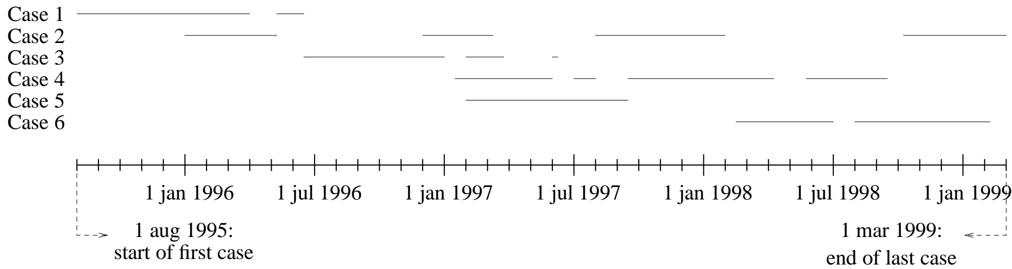


Figure 1.1: An overview of the work on the different case studies

1.6.1 Case 1: The RPC/Memory specification problem

problem A case study in distributed systems formalisation created by Leslie Lamport and Manfred Broy [BL96] for a workshop in Dagstuhl, Germany in September 1994. The problem contains formalisation and verification tasks and has failure, fairness and real time aspects.

goal To solve the problem with either I/O automata or μCRL theory, two methods not yet applied to this problem. To gain experience in protocol verification by doing the proofs manually.

method I/O automata for formalisation, (timed) invariants/simulations for proving safety aspects, fair trace inclusion for proving liveness

duration 10.5 months (total), 6.5 months (effectively)

findings All tasks in the problem statement have been solved completely. During the construction of the proof it turned out that the theory of I/O automata was not general enough if the problem statement was followed, because the cardinality of one of the parameters was not given. Also, the problem statement called for strong fairness restrictions. In this situation, the existing I/O automata theory did not guarantee the liveness of the models with respect to the desired fairness restrictions. However, Frits Vaandrager and I found sufficient conditions for liveness that were more general. It was easy to show that these conditions were true for the RPC/Memory models. This result led to the writing of a separate paper.
Two years after the paper with the solution was published, the proofs were checked in PVS [ORSH95] by Archer and Riccobene with the tool TAME [AH96]. Results appeared in [RAH98]. Some minor errors, one type inconsistency, and four proof errors were found which were repaired easily. One extra invariant was used, which could have been circumvented. The proof checking effort took about three weeks.
Note that the terminology in this case does not conform to our introduction in Section 1.2. For the term *specification* one should read *formalisation*.

papers (1) The solution to the problem with I/O automata [Rom96] which appeared in the LNCS volume with many different solutions for the case study, (2) The extension of I/O automata theory [RV96] which appeared in the journal Information Processing Letters.

in this thesis Chapter 3 is [Rom96] slightly adjusted: the errors found in [RAH98] were fixed. Chapter 2 is [RV96] with an addition: the proof for Theorem 2.

1.6.2 Case 2: IEEE 1394 verification

problem The IEEE 1394-1995 standard document describes an architecture for a high speed serial bus. The transaction, link and physical layer contain some protocols. Do these protocols behave correctly under the given assumptions?

goal To verify that (part of) one of the 1394 layers works correct.

method I/O automata for formalisation, IOA language for presentation, (timed) invariants/ simulations for proving safety aspects, fair trace inclusion for proving liveness

duration 3 years and 2 months (total), 8.5 months (effectively)

findings The research soon focused on the tree identify phase in the physical layer. The plan was to give a model that could be checked for syntactic correctness, and a manual verification (sketch).
The first attempt at specifying the tree identify phase resulted in a huge model which could not even be type checked completely in PVS because of the difficult conditions on the passage of time. Manual verification of this model was not feasible.
Late 1997, Frits Vaandrager suggested a layered verification, which could start with a very abstract version of the protocol, and then refine this stepwise to reach the amount of detail given in the standard document. The essence of the tree identify phase was extracted, specified at a very high level of abstraction and verified manually. The safety part of the proof was checked in PVS.
Following this verification effort, David Griffioen and Frits Vaandrager started refining the abstract model in order to prove correctness for behaviour with more details from the IEEE 1394 tree identify phase. They introduced a new type of simulation, useful for the verification of the refined tree identify phase model, and checked the proofs in PVS.
Following up on the work of David Griffioen and Frits Vaandrager, I refined the model of the tree identify phase even further to include timing information, and gave a manual proof of correctness. Timing is used to signal whether the network topology contains a cycle (which is an error) and to model the delay of messages in the network. Further refinement is necessary in order to obtain a correctness statement that includes all detail of the IEEE 1394 documentation.
Since the start of this case study, other researchers have studied (parts of) the IEEE tree identify phase as well. The relation between the different papers is explained in [Rom99b].

papers Two papers appeared of which I was (co-)author: (1) the formalisation and verification at a very abstract level of the tree identify phase [DGRV97], and (2) the formalisation and verification of the tree identify phase with more detail [Rom99b].

in this thesis Chapter 7 is [Rom99b].

1.6.3 Case 3: Automatic VHDL testing

problem In 1996, Olaf Sies graduated at the University of Eindhoven on the design and implementation of a test environment for conformance testing of VHDL designs [Sie96]. However, the test environment had not actually been used. Philips was interested in the conformance of some link layer code with respect to the IEEE 1394 standard.

goal To test the link layer code for conformance with respect to the IEEE 1394 standard in the test environment, and to show that testing in this manner is feasible.

method Test method: generation of abstract tests from an FSM model, translation of the abstract tests to the VHDL level, execution of the VHDL tests on a VHDL design.
Test strategy for link layer code: (1) construct an FSM model of the input/output behaviour of the link layer from the IEEE 1394 standard (a preliminary version by Sies existed), (2) make a translation from abstract input and output events as used in the FSM, to concrete VHDL input and output for the design to be tested (3) generate tests from the FSM model (4) translate the abstract tests to VHDL input and output (5) instantiate the VHDL test code with the link layer code (6) execute the VHDL tests (7) conclude whether the link layer code conformed to the IEEE 1394 standard

duration 1 year (total), 5.5 months (effectively)

findings The test environment consists of several tools of which only one was provided by a third party (the KPN Conformance Kit [KWKK91]). The fact that the test environment had not been used before (hence contained various errors) combined with the complexity of the interface behaviour of the link layer code and the lack of precision in the description of this behaviour in the IEEE 1394-1995 standard made it impossible to obtain meaningful test runs for the link layer code. Half-way through the planned time for the case, it was decided to switch to a different test case for which I constructed the FSM model and VHDL implementation. The insight in the VHDL code enabled the understanding of the several tools in the test environment and the components in the VHDL test software. The correction of several errors finally led to successful testing of the VHDL code, meaning that conformance was shown for a correct implementation, and errors were found for an incorrect implementation.
Following up on this research, other projects have worked with the test environment. In 1997, the test environment was used to test an MPEG2 decoder chip in the DIVA project [FMMW98]. In 1998, the test environment was used to test a 64 inch projection TV produced by Philips Consumer Electronics [Hol98a, TLH+99].

papers Three papers were written: (1) the presentation of the integrated test method and the results obtained [MRS+97a], (2) a manual on all of the tools and the VHDL code [MRS+97b], and (3) a manual for the demonstration workshop held at the Philips Research lab [MRS+96]

in this thesis Chapter 4 is [MRS+97a].

1.6.4 Case 4: Symmetry reduction in test generation

problem Building on the experiences of Case 3, Jan Springintveld and I felt that there is a huge need for reduction of tests that are generated when one is testing for conformance. In practice, one assumes that certain test scenarios are similar and will therefore skip some of these. On the other hand, conformance testing without a measure for coverage is not convincing enough. In model checking, partial orders and data and other symmetries are criterions used to reduce the part of the state space that is actually being explored. The question is whether the same can be done for conformance testing, that is, whether a formal basis can be given for not generating/executing some test scenarios and still having complete coverage with the smaller set of tests, when the criterion for reduction is based on some sort of *symmetry*.

goal To construct a formal basis for a notion of symmetry that makes it possible to test with a smaller test set and still have complete coverage.

method (1) Find a criterion under which behaviours of a FSM can be considered to be symmetric, (2) find a way to construct a kernel FSM for the FSM to be tested such that for each behaviour of the original FSM, a symmetric behaviour is in the kernel, (3) find a test generation method for the kernel FSM, (4) show the generated tests are exhaustive and complete for any implementation tested with them, and (5) show that the kernel is smaller than the original FSM and (hence) the generated test set is smaller than tests generated from the original FSM.

duration 1 year and 8 months (total), 8.5 months (effectively)

findings The starting points for the research were: black box conformance testing in the Chow/Vasilevskii [Cho78, Vas73] fashion, symmetry in terms of transactions, i.e. small patterns of actions, as few assumptions as possible on the implementation to be tested.
First a proper definition of symmetry had to be invented. This turned out to be a quite difficult task, since the kernel construction and test method depended heavily on the choices made in the basic definitions. The nature of the Chow test methods implied that some equivalence between states was needed, whereas in black box testing, too strong assumptions on the inner structure of the implementation are not desirable. Therefore we formulated symmetry in terms of equivalence of observable traces. The resulting symmetry definition enabled a straightforward algorithm for constructing a kernel FSM and a Chow-like test method, all of which have been proved correct. The assumption on the implementation implied only completeness of its observable behaviour under the symmetry equivalence, on the specification more assumptions were made.
Some experiments were done with the kernel construction algorithm, in the tool set Cæsar/Aldébaran [FGK+96]. This showed that for some toy examples, significant reductions in the kernel sizes and thus the test sets could be obtained.
Due to lack of time we have not yet experimented with real designs or programs, so test execution still remains to be explored.

papers One paper has been written in a report (full) and a conference (short) version [RS98].

in this thesis Chapter 5 is the full version of [RS98] without the code listings.

1.6.5 Case 5: A software architecture

problem In 1997, the Philips Multimedia Center in Palo Alto was working on the development of a communication architecture for supporting distributed applications. A team of four people at the Philips Research Lab in Eindhoven was to cooperate and help with the development of this architecture in several ways. My involvement in this project was to search protocols in the formal methods literature in order to guarantee a certain desired but not yet implemented functionality in the architecture.

goal To help in the development of a new product, with a protocol from literature, such that a correctly operating part of functionality can be guaranteed.

method Literature study, and if desired, fine tuning of the candidate protocol for the given situation and formal verification, possibly automated by the use of model checking tools

duration 7.5 months (total), 1.5 months (effectively)

findings The architecture development was in the hands of the Palo Alto team. They provided documentation about the architecture design. Of course the design changed a lot over the months, which made it hard to find functionality in the architecture for which a protocol should be found. Also, the desired functionality was not clear or stable. The distance between the Eindhoven and Palo Alto locations made communication hard. The focus of the Palo Alto team was most on getting a prototype to work before anything else, which really hampered the cooperation. This attitude was shown most explicitly when the opportunity arose to verify an algorithm for logical addressing/routing. The person working on this algorithm preferred to implement the algorithm before any formalisation or verification would take place.
In September 1997, it was decided that the focus of the two teams had diverged too much for further cooperation. The Eindhoven team joined the HAVi architecture development activities.

papers None

in this thesis None

1.6.6 Case 6: HAVi DCM Management

problem During 1997, the Philips Research Labs in Eindhoven became involved in the HAVi standard activity. The HAVi standard defines communication architecture for audio/video applications in the home environment. The involvement of this project was to formally prove that a leader election/resource allocation protocol in the HAVi architecture works correctly.

goal To guarantee that part of the HAVi standard operates correctly, by either finding and eliminating errors, or showing their absence, and to demonstrate the effectiveness of model checking and/or theorem proving tools.

method Formalisation in a suitable formal language, verification by model checking and/or theorem proving

duration 1 year (total), 6.5 months (effectively)

findings The first activity was to capture the leader election part of the protocol in a formal model. This activity was hampered by the lack of precision of the natural language description in the HAVi document. The first model was made in Promela [Hol91], the input language for the model checking tool Spin [Hol91, Hol97]. A second model was made in the language Lotos [ISO89]. Both models were to be checked for correctness with model checking tools. The Promela model was checked in Spin, the Lotos model with the tools Cæsar, Aldébaran and Xtl [MG98] in the Cæsar/Aldébaran tool set. Safety properties of the models were written down, in Promela this was very straightforward. For the Lotos model this had to be done in the temporal logic ACTL [DNV90], which meant that some requirements engineering was necessary. One liveness property was required, which could only be expressed in temporal logic. This was done in ACTL without too much trouble. In LTL (the only temporal logic accepted by Spin) the property could not be expressed. The property was checked by changing the models and looking for invalid end states. In the model checking process it turned out that only one of the pure safety properties as expressed in the Promela model was satisfied, and the liveness property was not satisfied. For the Lotos model, only some of the safety violations detected with Spin were found with model checking, but the liveness violation was found using the temporal property. All erroneous behaviour found with Spin could be reproduced in the corresponding Lotos models by simulation. The errors were caused by scenarios in which the protocol (as presented in the HAVi documentation) indeed behaved incorrect.

It has been acknowledged by Philips that the description of the protocol in the HAVi documentation was indeed not sufficiently precise to be error free, but it is expected that due to timing requirements (not expressed in the HAVi standard) the erroneous behaviour will not occur in implementations of the protocol. In the meantime, the HAVi specification has altered such that the initiative for communication in the leader election protocols now works the other way around. We believe that this alone is not enough to avoid the type of error that we found.

papers One paper has been written of which a short version has been submitted [Rom99a].

in this thesis Chapter 6 is [Rom99a] without the appendices and the detailed description of the Promela and Lotos models.

Chapter 2

A note on fairness in I/O automata

Summary

Notions of weak and strong fairness are studied in the setting of the I/O automaton model of Lynch & Tuttle. The concept of a *fair I/O automaton* is introduced and it is shown that a fair I/O automaton paired with the set of its fair executions is a live I/O automaton provided that (1) in each reachable state at most countably many fairness sets are enabled, and (2) input actions cannot disable strong fairness sets. This result, which generalises previous results known from the literature, was needed to solve a problem posed by Broy & Lamport for the Dagstuhl Workshop on Reactive Systems.

2.1 Introduction

Many specification formalisms for reactive systems incorporate notions of weak and strong fairness (see, for instance, [Jon94, Lam94a, LT87, MP92]). Informally, the requirement of weak fairness disallows executions in which certain sets of transitions are continually enabled but not taken beyond a certain point, whereas the requirement of strong fairness disallows executions in which certain sets of transitions are enabled infinitely often but taken only finitely many times. A natural criterion that any acceptable notion of fairness should satisfy is that it induces liveness properties in the sense of [AS85]: it should be possible to extend every finite execution to a fair one. Several authors have observed that weak and strong fairness induce liveness properties if the number of fairness sets (sets of transitions for which fairness is required) is countable [AL94, LT87]. If this number is uncountable then one does not obtain liveness properties in general: since in a transition system each execution contains at most a countable number of transitions, it is impossible to give fair turns to uncountably many fairness sets.

In most practical cases, the restriction to a countable number of fairness sets is unproblematic. However, there are classes of applications where this restriction cannot be made. A nice example here is the RPC-Memory specification problem proposed by Broy & Lamport [BL96] for the Dagstuhl Workshop on Reactive Systems. In this problem, there is a set of processes that can concurrently issue procedure calls to a memory component, which responds to these calls by issuing returns. Because there are no constraints on the number of processes and each call should eventually lead to a corresponding return, it is impossible to specify the required

liveness properties using only a bounded number of fairness sets. Essentially, the main result of this note is that liveness is also ensured if one does not impose a global constraint on the number of fairness sets, but instead assumes that in each reachable state only a countable number of fairness sets is enabled. The latter restriction applies to the Dagstuhl example since in each reachable state the number of outstanding calls is finite. The key argument in our proof is not difficult, but distinctly different from the arguments used in the proofs of [AL94, LT87].

We have stated our results in terms of the I/O automaton model [SGSL98, LT87], since it was needed for this I/O automata solution to the Dagstuhl problem (See Chapter 3). We propose a model of *fair I/O automata*, which is a generalisation of the original I/O automaton model of [LT87]. Our main result is that under certain assumptions fair I/O automata can be viewed as a special case of the *live I/O automata* of [SGSL98], another generalisation of the original model. Roughly speaking, this result says that each finite execution can be extended to a fair one independently of the inputs provided by the environment. The notion of a live I/O automaton is very general but its definition is complex and cumbersome to use: in order to prove that a certain structure is a live I/O automaton one has to exhibit a winning strategy in an infinite two-player game. Since it appears that all liveness properties that one needs in practice can be specified using weak and strong fairness properties only [Jon94, Lam94a, MP92] and since it is usually trivial to check that a structure is a fair I/O automaton, we think that there will be many situations where, after one has described a system as a fair I/O automaton, our result provides one with a live I/O automaton description almost for free.

The outline of this chapter is as follows. In Section 2.2, we introduce fair I/O automata. In Section 2.3 we prove that a fair I/O automaton paired with the set of its fair executions is a live I/O automaton provided that (1) in each reachable state at most countably many fairness sets are enabled, and (2) input actions cannot disable strong fairness sets. In Section 2.4, we define a composition operation on fair I/O automata and show that this operation is compatible with the composition operation on live I/O automata defined in [SGSL98].

2.2 Definitions

In this section we define the model of *fair I/O automata*, which is a generalisation of the original I/O automaton model of [LT87]: whereas the I/O automata of [LT87] only allow for weak fairness, fair I/O automata permit both weak and strong fairness. See Appendix A for definitions of safe I/O automata.

Fair I/O automata A *fair I/O automaton* A is a triple consisting of

- a safe I/O automaton *safe*(A), and
- sets *wfair*(A) and *sfair*(A) of subsets of *local*(*safe*(A)), called the *weak fairness sets* and *strong fairness sets*, respectively.

In the rest of this note we write *local*(A) for *local*(*safe*(A)), *steps*(A) for *steps*(*safe*(A)), etc. Also, we fix a fair I/O automaton A.

Enabling Let U be a set of actions of A. Then U is *enabled* in a state s if and only if an action from U is enabled in s. Set U is *input resistant* if and only if, for each pair of reachable

states s, s' and for each input action a,

$$s \text{ enables } U \wedge s \xrightarrow{a} s' \quad \Rightarrow \quad s' \text{ enables } U.$$

So once U is enabled, it can only be disabled by the occurrence of a locally controlled action.

Fair executions and traces An execution α of A is *weakly fair* iff the following conditions hold for each $W \in \mathit{wfair}(A)$:

1. If α is finite then W is not enabled in the last state of α.
2. If α is infinite then either α contains infinitely many occurrences of actions from W, or α contains infinitely many occurrences of states in which W is not enabled.

Execution α is *strongly fair* iff the following conditions hold for each $S \in \mathit{sfair}(A)$:

1. If α is finite then S is not enabled in the last state of α.
2. If α is infinite then either α contains infinitely many occurrences of actions from S, or α contains only finitely many occurrences of states in which S is enabled.

Execution α is *fair* iff it is both weakly and strongly fair. Finite executions are fair only if in the last state, no weak or strong fairness sets are enabled anymore. The intuition is that it would not be fair to stop execution otherwise. In an infinite fair execution, each weak fairness set gets turns if enabled continuously, and each strong fairness set gets turns if enabled infinitely many times. We write $\mathit{fairexecs}(A)$ for the set of fair executions of A. We write $\mathit{fairtraces}(A)$ for the set of traces of fair executions of a fair I/O automaton A.

Implementation relation Let A and B be fair I/O automata.
A *implements* B if $\mathit{fairtraces}(A) \subseteq \mathit{fairtraces}(B)$.

Fairness as a liveness condition We write $\mathit{live}(A)$ for the underlying safe I/O automaton of A paired with $\mathit{fairexecs}(A)$: $\mathit{live}(A) \triangleq (\mathit{safe}(A), \mathit{fairexecs}(A))$.

2.3 Main Result

In [SGSL98], live I/O automata are introduced as a generalisation of the I/O automata of [LT87] with general liveness properties (see also Appendix A). Our main result, stated below, says that, if fair I/O automata A satisfies two conditions then the pair $(\mathit{safe}(A), \mathit{fairexecs}(A))$ is a live I/O automaton. The first condition states that in each reachable state at most countably many weak and strong fairness sets are enabled. This cardinality assumption allows us to define, via a diagonalisation construction, a strategy for the I/O automaton that gives fair turns to each fairness set. The second condition states that all strong fairness sets are input resistant. This technical assumption excludes situations where the environment gives turns to the system only when some strong fairness set is not enabled. As an example, consider the fair I/O automaton of Figure 2.1. In this I/O automaton the strong fairness set $\{o\}$ is not input resistant. As a result the I/O automaton is not live: for each strategy ρ, the outcome $\mathcal{O}_\rho(s, \lambda\, i\, i\, \lambda\, i\, i\, \lambda \cdots)$ equals the unfair execution $s\, i\, s'\, i\, s\, i\, s' \cdots$. It seems that most applications with strong fairness aspects meet this requirement.

Figure 2.1: A fair I/O automaton that is not live.

Theorem 2.1 Suppose that fair I/O automaton A satisfies the following conditions: (1) each reachable state of A enables at most countably many sets in $wfair(A) \cup sfair(A)$, and (2) each set in $sfair(A)$ is input resistant. Then $live(A)$ is a live I/O automaton.

Proof With each finite execution α we associate an infinite two-dimensional array $\mathcal{A}_\alpha$ of weak and strong fairness sets. The array contains all the weak or strong fairness sets that are enabled at some point in execution α but from which no action has been executed in the subsequent part of α. We will use array $\mathcal{A}_\alpha$ to define a strategy that treats each fairness set in a fair manner and thus establishes that *live*(A) is a live I/O automaton. The array is defined by induction on the length of α:

- If α consists of a single state s, then $\mathcal{A}_\alpha$ is constructed by filling the first row with the sets in $wfair(A)$ and $sfair(A)$ that are enabled in s. While filling, the sets are alternatingly taken from $wfair(A)$ and $sfair(A)$. Remaining positions are filled with the symbol $\blacksquare$.
  If s enables 6 weak fairness sets and 2 strong fairness sets, then $\mathcal{A}_\alpha$ might look like this:

	1	2	3	4	5	6	7	8	9	$\cdots$
1	W_{1_1}	S_{1_1}	W_{1_2}	S_{1_2}	W_{1_3}	W_{1_4}	W_{1_5}	W_{1_6}	$\blacksquare$	$\cdots$
2	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

  Note that by Condition (1) we are able to squeeze all the enabled sets in a single row.

- If α contains $n > 1$ states and is of the form $\alpha' \, a \, s$, then $\mathcal{A}_\alpha$ is constructed from $\mathcal{A}_{\alpha'}$ by replacing each fairness set that contains action a by $\blacksquare$, and filling the n-th row with the sets in $wfair(A)$ and $sfair(A)$ that are enabled in s, as in the previous case.
  The array for an execution α with 4 states might look like this:

	1	2	3	4	5	6	7	8	$\cdots$
1	W_{1_1}	S_{1_1}	$\blacksquare$	S_{1_2}	$\blacksquare$	$\blacksquare$	W_{1_5}	$\blacksquare$	$\cdots$
2	$\blacksquare$	S_{2_1}	W_{2_2}	S_{2_2}	W_{2_3}	S_{2_3}	$\blacksquare$	$\blacksquare$	$\cdots$
3	S_{3_1}	$\blacksquare$	$\blacksquare$	S_{3_4}	S_{3_5}	S_{3_6}	$\blacksquare$	$\blacksquare$	$\cdots$
4	W_{4_1}	S_{4_1}	W_{4_2}	S_{4_2}	W_{4_3}	S_{4_3}	W_{4_4}	S_{4_4}	$\cdots$
5	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\blacksquare$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Let $\rho = (g, f)$ be any strategy defined on *safe*(A) that satisfies the following conditions:

1. If $f(\alpha) = \bot$ then the last state of α enables no set in *wfair*$(A) \cup$ *sfair*(A).
2. If $f(\alpha) = (a, s)$ then the last state of α enables a set in *wfair*$(A) \cup$ *sfair*(A), and a is member of the first set U that is enabled in the last state of α and that occurs in the sequence

$$
\begin{array}{rcl}
\Omega(\alpha) & \triangleq & \mathcal{A}_\alpha[1,1] \\
 & & \mathcal{A}_\alpha[1,2]\ \ \mathcal{A}_\alpha[2,1] \\
 & & \mathcal{A}_\alpha[1,3]\ \ \mathcal{A}_\alpha[2,2]\ \ \mathcal{A}_\alpha[3,1] \\
 & & \mathcal{A}_\alpha[1,4]\ \ \mathcal{A}_\alpha[2,3]\ \ \mathcal{A}_\alpha[3,2]\ \ \mathcal{A}_\alpha[4,1] \\
 & & \quad\vdots
\end{array}
$$

Note that a strategy ρ satisfying these properties exists since by construction the array $\mathcal{A}_\alpha$ contains at least all the weak and strong fairness sets that are enabled in the last state of α, and sequence $\Omega(\alpha)$ enumerates all elements of $\mathcal{A}_\alpha$.

We show that *live*(A) is a live I/O automaton by proving that the outcome $\alpha' = \mathcal{O}_\rho(\alpha, \mathcal{I})$ is fair for each finite execution α and each environment sequence $\mathcal{I}$.

Assume that α' is a finite execution. Then $\mathcal{I}$ contains only finitely many input actions and, for s the last state of α', $f(\alpha') = \bot$. Therefore, by the first assumption about strategy ρ, the last state of α' enables no set in *wfair*(A) or *sfair*(A). Hence α' is fair.

Thus we may assume that α' is infinite. We prove that α' is fair by contradiction. Suppose α' is not fair. We distinguish between two cases:

1. α' is not strongly fair.
   Then some strong fairness set S is enabled in an infinite number of states of α' and α' contains only finitely many occurrences of actions in S.

   Since S is input resistant, it is enabled in an infinite number of states in which a system move is allowed by $\mathcal{I}$. From the definition of strategy ρ it follows that S is enabled in an infinite number of states in which a locally controlled action occurs. Since α' contains only finitely many occurrences of actions in S, there is a state in α' after which no action in S occurs. Nevertheless, there is a subsequent state of α', say the i-th state, in which S is enabled. Therefore, there is a position $[i, j]$ such that, if α_k is the finite prefix of α' with k states, $\mathcal{A}_{\alpha_k}[i, j] = S$, for all $k \geq i$. Let $l = i + j - 1$. Then, for each $n \geq l$, each position preceding $[i, j]$ in the strategy's sequence that is filled with $\blacksquare$ in the array $\mathcal{A}_{\alpha_n}$, is also filled with $\blacksquare$ in any array $\mathcal{A}_{\alpha_m}$ with $m > n$. Each locally controlled action that occurs after the l-th state from a state that enables S causes a fairness set at a position preceding $[i, j]$ in the strategy's sequence to be replaced by $\blacksquare$ in the array. This happens infinitely many times. But this is a contradiction since the number of preceding positions is finite.
2. α' is not weakly fair.
   Then some weak fairness set W is enabled in all states of an infinite suffix of α' with only finitely many occurrences of actions from W.

By an argument that is almost identical to the one used in the previous case we arrive at a contradiction.

Hence α' is fair and we may conclude that *live*(A) is a live I/O automaton. ⊠

2.4 Composition

Building on the work of [SGSL98, LT87], there is an obvious way to define composition of fair I/O automata.

We say that two fair I/O automata A_1 and A_2 are *compatible* if *safe*(A_1) and *safe*(A_2) are compatible. Suppose that A_1 and A_2 are compatible fair I/O automata. Then the *composition* $A_1 \| A_2$ is the fair I/O automaton A given by

- $\mathit{safe}(A) = \mathit{safe}(A_1) \| \mathit{safe}(A_2)$,
- $\mathit{wfair}(A) = \mathit{wfair}(A_1) \cup \mathit{wfair}(A_2)$ and $\mathit{sfair}(A) = \mathit{sfair}(A_1) \cup \mathit{sfair}(A_2)$.

Thus we simply compose the underlying safe I/O automata and take the unions of the weak and strong fairness sets. The following theorem, which is easy to prove, states that the above composition operation for fair I/O automata is compatible with the composition operation for live I/O automata of [SGSL98].

Theorem 2.2 Suppose that A_1 and A_2 are compatible fair I/O automata. Then

$$\mathit{live}(A_1 \| A_2) \quad = \quad \mathit{live}(A_1) \| \mathit{live}(A_2).$$

Proof By definition, $\mathit{live}(A_1 \| A_2) = ((\mathit{safe}(A_1) \| \mathit{safe}(A_2)), \mathit{fairexecs}(A_1 \| A_2))$. By Definition 3.19 in [SGSL98], $\mathit{live}(A_1) \| \mathit{live}(A_2) = ((\mathit{safe}(A_1) \| \mathit{safe}(A_2)), \mathcal{F})$, where $\mathcal{F} = \{\alpha \in \mathit{execs}(\mathit{safe}(A_1) \| \mathit{safe}(A_2)) | (\alpha \lceil \mathit{safe}(A_1)) \in \mathit{fairexecs}(A_1) \wedge (\alpha \lceil \mathit{safe}(A_2)) \in \mathit{fairexecs}(A_2)\}$.
It remains to be proved that $\mathcal{F} = \mathit{fairexecs}(A_1 \| A_2)$.

- $\subseteq$ Suppose $\alpha \in \mathcal{F}$.
  Let $\alpha_1 = \alpha \lceil \mathit{safe}(A_1)$ and $\alpha_2 = \alpha \lceil \mathit{safe}(A_2)$. Then by definition of $\mathcal{F}$, $\alpha_1 \in \mathit{fairexecs}(A_1)$ and $\alpha_2 \in \mathit{fairexecs}(A_2)$.
  - Suppose α is finite.
    Then α_1 and α_2 are also finite. By definition, $\mathit{last}(\alpha_1) = \pi_1(\mathit{last}(\alpha))$ and $\mathit{last}(\alpha_2) = \pi_2(\mathit{last}(\alpha))$. Since $\alpha_1 \in \mathit{fairexecs}(A_1)$, $\mathit{last}(\alpha_1)$ does not enable W or S for any $W \in \mathit{wfair}(A_1)$ and any $S \in \mathit{sfair}(A_1)$. Likewise, $\alpha_2 \in \mathit{fairexecs}(A_2)$, so $\mathit{last}(\alpha_2)$ does not enable W or S for any $W \in \mathit{wfair}(A_2)$ and any $S \in \mathit{sfair}(A_2)$. We see that $\mathit{last}(\alpha)$ does not enable W or S for any $W \in (\mathit{wfair}(A_1) \cup \mathit{wfair}(A_2))$ and any $S \in (\mathit{sfair}(A_1) \cup \mathit{sfair}(A_2))$, hence $\alpha \in \mathit{fairexecs}(A_1 \| A_2)$.
  - Suppose α is infinite.
    - Suppose α_1 is finite.
      Then for an infinite suffix $s_0 a_1 s_1 a_2 s_2 \ldots$ of α, $\pi_1(s_i) = \pi_1(s_{i+1})$ with $i \geq 0$, that is, the state of A_1 remains the same in this suffix of α. Since $\alpha_1 \in \mathit{fairexecs}(A_1)$, none of the states $\pi_1(s_i), \pi_1(s_{i+1}), \ldots$ enable W or S for any $W \in \mathit{wfair}(A_1)$ and any $S \in \mathit{sfair}(A_1)$. We see that for each $W \in \mathit{wfair}(A_1)$,

α contains infinitely many states in which W is not enabled, and for each $S \in \mathit{sfair}(A_1)$, α contains only finitely many states in which S is enabled, hence $\alpha \in \mathit{fairexecs}(A_1 \| A_2)$.

* Suppose α_1 is infinite.
  Since $\alpha_1 \in \mathit{fairexecs}(A_1)$, for each $W \in \mathit{wfair}(A_1)$, α_1 contains infinitely many states in which W is not enabled, and for each $S \in \mathit{sfair}(A_1)$, α_1 contains only finitely many states in which S is enabled. We see that for each $W \in \mathit{wfair}(A_1)$, α contains infinitely many states in which W is not enabled. Since for each $S \in \mathit{sfair}(A_1)$, α_1 contains only finitely many states in which S is enabled, α_1 contains an infinite suffix in which each $S \in \mathit{sfair}(A_1)$ is permanently disabled. From the definition of α_1, we see that α must contain an infinite suffix in which each $S \in \mathit{sfair}(A_1)$ is permanently disabled. We see that for each $S \in \mathit{sfair}(A_1)$, α contains only finitely many states in which S is enabled, hence $\alpha \in \mathit{fairexecs}(A_1 \| A_2)$.

For α_2 we can reason likewise. We conclude that $\alpha \in \mathit{fairexecs}(A_1 \| A_2)$.

$\supseteq$ Suppose $\alpha \in \mathit{fairexecs}(A_1 \| A_2)$.
Let $\alpha_1 = \alpha \lceil \mathit{safe}(A_1)$ and $\alpha_2 = \alpha \lceil \mathit{safe}(A_2)$.

- Suppose α is finite.
  Then α_1 and α_2 are also finite. By definition, $\mathit{last}(\alpha_1) = \pi_1(\mathit{last}(\alpha))$ and $\mathit{last}(\alpha_2) = \pi_2(\mathit{last}(\alpha))$. Since $\alpha \in \mathit{fairexecs}(A_1 \| A_2)$, $\mathit{last}(\alpha)$ does not enable W or S for any $W \in (\mathit{wfair}(A_1) \cup \mathit{wfair}(A_2))$ and any $S \in (\mathit{sfair}(A_1) \cup \mathit{sfair}(A_2))$. We see that $\mathit{last}(\alpha_1)$ does not enable W or S for any $W \in \mathit{wfair}(A_1)$ and any $S \in \mathit{sfair}(A_1)$, hence $\alpha_1 \in \mathit{fairexecs}(A_1)$. Likewise, we see that $\mathit{last}(\alpha_2)$ does not enable W or S for any $W \in \mathit{wfair}(A_2)$ and any $S \in \mathit{sfair}(A_2)$, hence $\alpha_2 \in \mathit{fairexecs}(A_2)$.
- Suppose α is infinite. Since $\alpha \in \mathit{fairexecs}(A_1 \| A_2)$, for each $W \in (\mathit{wfair}(A_1) \cup \mathit{wfair}(A_2))$, α contains either infinitely many occurrences of W, or infinitely many states in which W is not enabled, and for each $S \in (\mathit{sfair}(A_1) \cup \mathit{sfair}(A_2))$, α contains either infinitely many occurrences of S, or only finitely many states in which S is enabled.
  * Suppose α contains infinitely many occurrences of W (S), with $W \in \mathit{wfair}(A_1)$ ($S \in \mathit{sfair}(A_1)$).
    Then α_1 is infinite and contains infinitely many occurrences of W (S), hence $\alpha_1 \in \mathit{fairexecs}(A_1)$.
  * Suppose α contains infinitely many states in which W is not enabled, with $W \in \mathit{wfair}(A_1)$.
    · Suppose α_1 contains infinitely many states in which W is not enabled.
      Then α_1 is infinite, hence $\alpha_1 \in \mathit{fairexecs}(A_1)$.
    · Suppose α_1 contains only finitely many states in which W is not enabled.
      It is easy to see that α_1 must be finite and that α must contain an infinite suffix $s_0 a_1 s_1 a_2 s_2 \ldots$ in which W is permanently disabled. Also, $\mathit{last}(\alpha_1)$ cannot enable W, otherwise this state would be different from $\pi_1(s_i)$ for each $i \geq 0$. So α_1 is finite, and $\mathit{last}(\alpha_1)$ does not enable W, hence $\alpha_1 \in \mathit{fairexecs}(A_1)$.

* Suppose α contains only finitely many states in which S is enabled, with $S \in$ *sfair*(A_1).
  Then α contains an infinite suffix $s_0 a_1 s_1 a_2 s_2 \ldots$ in which S is permanently disabled.
  · Suppose α_1 is finite.
    Then from some state in α onwards, the state for A_1 remains the same, so *last*$(\alpha_1) = \pi_1(s_i)$ for some $i \geq 0$, hence *last*(α_1) does not enable S, so $\alpha_1 \in$ *fairexecs*(A_1).
  · Suppose α_1 is infinite.
    Then α_1 contains only finitely many states in which S is enabled, hence $\alpha_1 \in$ *fairexecs*(A_1).

We conclude that $\alpha_1 \in$ *fairexecs*(A_1). For α_2 we can reason likewise.

⊠

Chapter 3

Tackling the RPC-Memory specification problem with I/O automata

Summary

An I/O automata solution to the problem posed in 1994 by Broy & Lamport at the Dagstuhl Workshop on Reactive Systems is presented. The problem calls for specification and verification of memory and remote procedure call components. The problem specification consists of an untimed and a timed part. In this chapter, both parts are solved completely.

3.1 Introduction

An example of an distributed system specification problem was stated at the Workshop on Reactive Systems, held in Dagstuhl, Germany in September 1994. The problem concerned the specification of a memory component and a remote procedure call (RPC) component, and the implementation of both.

The workshop's main intention was to compare different formalisms by applying them to this example, in order to understand the similarities and differences of the various approaches, as well as their strengths and weaknesses. The problem has been solved completely in [ALM96, Bro96, CBH96, Hoo96, KS96, LSW96, Stø96]. Other papers on this topic are [AR96, Bes96, BJ96b, Got96, Hun96, KNS96, UK96] which only solve the untimed part.

This chapter is the result of a successful attempt to model and verify the RPC-Memory problem with the I/O automata model [SGSL98, Lyn96, LT89, LV95, LV96, RV96]. It is organised as follows. The remainder of this section lists the problem statement, taken from [BL96], some notes on the problem statement and on the merits of I/O automata. Section 3.2 lists some preliminaries which are necessary for a good understanding of the specifications, as well as the proofs. Sections 3.3 to 3.7 solve parts 1 to 5 of the problem consecutively.

3.1.1 Specification problem

This section is quoted from [BL96] with permission from Springer-Verlag.

The procedure interface The problem calls for the specification and verification of a series of *components*. Components interact with one another using a procedure-calling interface. One component issues a *call* to another, and the second component responds by issuing a *return*. A call is an indivisible (atomic) action that communicates a procedure name and a list of *arguments* to the called component. A return is an atomic action issued in response to a call. There are two kinds of returns, *normal* and *exceptional*. A normal call returns a *value* (which could be a list). An exceptional return also returns a value, usually indicating some error condition. An exceptional return of a value e is called *raising exception e*. A return is issued only in response to a call. There may be "syntactic" restrictions on the types of arguments and return values.

A component may contain multiple *processes* that can concurrently issue procedure calls. More precisely, after one process issues a call, other processes can issue calls to the same component before the component issues a return from the first call. A return action communicates to the calling component the identity of the process that issued the corresponding call.

A memory component The component to be specified is a memory that maintains the contents of a set MemLocs of locations. The contents of a location is an element of a set MemVals. This component has two procedures, described informally below. Note that being an element of MemLocs or MemVals is a "semantic" restriction, and cannot be imposed solely by syntactic restrictions on the types of arguments.

Name	Read
Arguments	loc : an element of MemLocs
Return Value	an element of MemVals
Exceptions	BadArg : argument loc is not an element of MemLocs.
	MemFailure : the memory cannot be read.
Description	Returns the value stored in address loc.

Name	Write
Arguments	loc : an element of MemLocs
	val : an element of MemVals
Return Value	some fixed value
Exceptions	BadArg : argument loc is not an element of MemLocs, or argument val is not an element of MemVals.
	MemFailure : the write *might* not have succeeded.
Description	Stores the value val in address loc.

The memory must eventually issue a return for every Read and Write call.

Define an *operation* to consist of a procedure call and the corresponding return. The operation is said to be *successful* iff it has a normal (nonexceptional) return. The memory behaves as if it maintains an array of atomically read and written locations that initially all contain the value InitVal, such that:

- An operation that raises a BadArg exception has no effect on the memory.

- Each successful Read(l) operation performs a single atomic read to location l at some time between the call and return.
- Each successful Write(l, v) operation performs a sequence of one or more atomic writes of value v to location l at some time between the call and return.
- Each unsuccessful Write(l, v) operation performs a sequence of zero or more atomic writes of value v to location l at some time between the call and return.

A variant of the memory component is the reliable memory component.In this component, no MemFailure exceptions can be raised.

Problem 1 (a) Write a formal specification of the memory component and of the reliable memory component.

(b) Either prove that a reliable memory component is a correct implementation of a memory component, or explain why it should not be.

(c) If your specification of the memory component allows an implementation that does nothing but raise MemFailure exceptions, explain why this is reasonable.

Implementing the memory

The RPC component The RPC component interfaces with two environment components, a *sender* and a *receiver*. It relays procedure calls from the sender to the receiver, and relays the return values back to the sender. Parameters of the component are a set Procs of procedure names and a mapping ArgNum, where ArgNum(p) is the number of arguments of each procedure p. The RPC component contains a single procedure:

Name	RemoteCall
Arguments	proc : name of a procedure
	args : list of arguments
Return Value	any value that can be returned by a call to proc
Exceptions	RPCFailure : the call failed
	BadCall : proc is not a valid name or args is not a syntactically correct list of arguments for proc.
	Raises any exception raised by a call to proc
Description	Calls procedure proc with arguments args

A call of RemoteCall(proc, args) causes the RPC component to do one of the following:

- Raise a BadCall exception if args is not a list of ArgNum(proc) arguments.
- Issue one call to procedure proc with arguments args, wait for the corresponding return (which the RPC component assumes will occur) and either (a) return the value (normal or exceptional) returned by that call, or (b) raise the RPCFailure exception.
- Issue no procedure call, and raise the RPCFailure exception.

The component accepts concurrent calls of RemoteCall from the sender, and can have multiple outstanding calls to the receiver.

Problem 2 Write a formal specification of the RPC component.

The implementation A memory component is implemented by combining an RPC component with a reliable memory component as follows. A Read or Write call is forwarded to the reliable memory by issuing the appropriate call to the RPC component. If this call returns without raising an RPCFailure exception, the value returned is returned to the caller. (An exceptional return causes an exception to be raised.) If the call raises an RPCFailure exception, then the implementation may either reissue the call to the RPC component or raise a MemFailure exception. The RPC call can be retried arbitrarily many times because of RPCFailure exceptions, but a return from the Read or Write call must eventually be issued.

Problem 3 Write a formal specification of the implementation, and prove that it correctly implements the specification of the memory component of Problem 1.

Implementing the RPC component

A lossy RPC The Lossy RPC component is the same as the RPC component except for the following differences, where δ is a parameter.

- The RPCFailure exception is never raised. Instead,the RemoteCall procedure never returns.
- If a call to RemoteCall raises a BadCall exception, then that exception will be raised within δ seconds of the call.
- If a RemoteCall$(p,\ a)$ call results in a call of procedure p, then that call of p will occur within δ seconds of the call of RemoteCall.
- If a RemoteCall$(p,\ a)$ call returns other than by raising a BadCall exception, then that return will occur within δ seconds of the return from the call to procedure p.

Problem 4 Write a formal specification of the Lossy RPC component.

The RPC implementation The RPC component is implemented with a Lossy RPC component by passing the RemoteCall call through to the Lossy RPC, passing the return back to the caller, and raising an exception if the corresponding return has not been issued after $2\delta + \epsilon$ seconds.

Problem 5 (a) Write a formal specification of this implementation.

(b) Prove that, if every call to a procedure in Procs returns within ϵ seconds, then the implementation satisfies the specification of the RPC component in Problem 2.

3.1.2 Notes on the problem specification

Ambiguities The informal descriptions of the memory component in Problem 1 and the RPC component in Problem 2 are slightly ambiguous. It is not clear whether these components may issue a failure when a bad call is received. In both cases we have chosen to allow this, because it yields a more general specification. For the memory component this decision conforms with the implementation proposed in Problem 3.

Observable versus internal behaviour Problem 3 requires a proof that a composition of components implements the memory component. The memory component can perform at most one internal read action between call and return. The proposed implementation, however, can do this an arbitrary (but finite!) number of times. The proof for the implementation relation is simplified substantially if one assumes that the memory component can perform an arbitrary number of internal read actions instead of at most one. The solution of Abadi, Lamport & Merz [ALM96] uses such a more convenient memory component, and thus implicitly assumes that the two memory components are observationally equivalent. We prove formally that this assumption is correct, which requires a backward simulation proof of about four pages.

In the solution of Hooman [Hoo96] the correctness of this assumption is also proved, with seemingly much less effort. This is due to a difference in view on executions. Hooman introduces safety restrictions on the set of all possible executions. In this manner, unwanted behaviour is avoided. His approach also allows executions with an infinite number of internal actions between two external actions. Our executions are built in an operational manner by concatenating states and transitions. Hence safety restrictions are posed only on single actions, and not on executions. Besides, since each execution contains at most a countable number of actions, there is at most a finite number of actions between any two actions. We feel that the operational view is more natural and closer to any real-world implementation of this problem specification.

Fairness and real time In Problem 5, a timed implementation is compared with an untimed specification. The untimed behaviour is restricted by fairness, whereas the timed behaviour is completely determined by timing constraints. To be able to compare these behaviours, we defined the *fair timed I/O automaton*. This notion is explained in Appendix A.4.

3.1.3 Notes on the I/O automata model

Benefits I/O automata provide a natural way to describe processes with an input/output behaviour. Most distributed systems can be specified in this way. The specifications are highly readable, and can be explained without too much trouble to most non-experts.

In the untimed part of our solution, simulation relations provide the major part of proofs for implementation relations, the rest is taken care of by inclusion of fairness properties. All these are standard ingredients of verifications with I/O automata.

Real time aspects of specifications are also captured in I/O automata quite easily. When comparing timed specifications, simulation relations can be used to prove implementation relations in a straightforward way.

Imperfections When reasoning about an I/O automaton with more than five state variables and more than five locally controlled actions, proofs for safety properties involve an enormous amount of tedious detail, and are prone to typos and more serious errors. The amount of paper needed to get these proofs done in a semi-readable way is terrifying, whereas in general the properties being proved seem so trivial and intuitively correct. However, we are not aware of the existence of a similar formalism without this problem.

I/O automata theory lacks a proof system for fairness proofs. Many fairness proofs are constructed in an intuitive, ad-hoc manner and thus are error prone. The construction of a formal framework for this certainly qualifies as future research.

Another gap in current I/O automata theory is that it is not possible to impose restrictions on the behaviour of the environment. Especially when using timed I/O automata, one sometimes needs to assume that events controlled by the environment will occur within certain time bounds. This is another potential benefit deserving further investigation.

What we added to the classic model A desired property of any specification with fairness requirements is liveness (receptivity, machine closure). In the I/O automata model proposed by Lynch & Tuttle [LT87], liveness is guaranteed for any weak fairness restriction that holds a countable number of actions. However, the RPC-Memory problem requires strong fairness restrictions on the behaviour of the proposed implementation of the memory component in Problem 3. Secondly, this problem holds a parameter whose cardinality is unknown, namely the number of calling processes for a memory or RPC component. Well-known results for liveness with respect to fairness conditions deal with at most a countable number of fairness sets or actions, and cannot be applied to this problem.

The desire to establish liveness for any specification with uncountably many fairness sets has led to the invention of the *fair I/O automaton* [RV96]. This is a slight variant of the I/O automaton in [LT87], and a special case of the live I/O automaton in [SGSL98] provided that two conditions hold. These conditions require that each reachable state enables at most a countable number of fairness sets, and that input actions do not disturb the enabledness of these sets. In this chapter, each specification is proved to be a live I/O automaton by checking these two conditions. To our knowledge, no other solution to the RPC-Memory problem includes proofs of this kind.

Since endless listings of highly detailed proofs guarantee a boring story instead of a higher degree of understanding, we have omitted unnecessary detailed proofs and replaced some by sketches. The full formal proofs can be obtained by e-mail request.

3.2 Preliminaries

3.2.1 Fair I/O automata

The set-up of specification and verifications is as follows. All untimed specifications use the *fair I/O automata* model from [RV96], which is explained in Chapter 2. The model is a generalisation from the classic model by Lynch & Tuttle [LT87], and, under two restrictions, a special case of the live I/O automaton model by Gawlick et al. [SGSL98].

The timed specifications use the *fair timed I/O automata* model, which extends the timed I/O automata model of [LV96] with an ad hoc notion of fairness in the timed setting. The basics of this model are listed in Appendix A.4. Section 3.7 explains why we need to use fairness in the timed setting.

3.2.2 Details on fair I/O automata

Specification Each action is indexed with the process, for which this action is performed. Some of the state variables are also indexed with a process. The state space is roughly partitioned by the value of the *program counters*, the state variables pc_P. These variables keep track of what the automaton should be doing for process P. All automata initially wait for some action by the environment, and each pc_P has a value that expresses this waiting condition. As

soon as input is received for process P, pc_P changes accordingly, and each next input for P is discarded (the state is not changed), if pc_P does not satisfy the waiting condition. For each internal action, the precondition requires pc_P to have a specific value in order to ensure that the right actions are taken at the right moment. After the input for some process P has been handled, pc_P is set to the waiting condition again.

To give the values of each program counter the right meaning, we assume that the interpretation of the domain of each program counter is free, in the sense that different constant symbols are mapped to different elements in its domain ("no confusion"), and each element in the domain is denoted by some constant symbol ("no junk").

Presentation The following conventions are used.

- We omit the precondition of an input action (since this equals true by definition).
- In the effect part of transition types we omit assignments of the form $x := x$.
- We write if c then $[z_1 := f_1, \ldots, z_k := f_k]$ as an abbreviation for

$$
\begin{array}{lll}
z_1 & := & \text{if } c \text{ then } f_1 \text{ else } z_1 \\
 & \vdots & \\
z_k & := & \text{if } c \text{ then } f_k \text{ else } z_k
\end{array}
$$

- We write $x \in \{A, B, C\}$ for $x{=}A \vee x{=}B \vee x{=}C$, etc.
- To improve readability we often use Lamport's list notation for conjunction or disjunction. Thus we write

$$
\begin{array}{ll}
\wedge & b_1 \\
\wedge & b_2 \\
\vdots & \\
\wedge & b_n
\end{array}
$$

for $b_1 \wedge b_2 \wedge \cdots \wedge b_n$.

Proofs We prove an implementation relation between two fair I/O automata A and B by proving that $\mathit{fairtraces}(A) \subseteq \mathit{fairtraces}(B)$. To ease this proof, we mostly start out by proving inclusion on the ordinary and quiescent traces of A and B using refinements and simulations.

Since the only difference between the fair and classic I/O automata model lies in the fairness properties, all results in the latter that do not concern fairness carry over to the fair I/O automata model. This is used when proving ordinary and quiescent trace inclusion.

3.3 Specifications and verifications for Problem 1

3.3.1 Problem 1(a): Specification of two memory components

In this section, we present the formal specification of the memory component and the reliable memory component.

Data types We start the specification with a description of the various data types that play a role. We assume a typed signature Σ_1 and a Σ_1-algebra $\mathcal{A}_1$ which consist of the following components:

- a type **Bool** of booleans with constant symbols true and false, and a standard repertoire of function symbols ($\wedge$, $\vee$, $\neg$, $\rightarrow$), all with the standard interpretation over the booleans. Also, we require, for all types **S** in Σ, an equality, inequality, and if-then-else function symbol, with the usual interpretation:

$$
\begin{aligned}
.=. \quad &: \quad \mathbf{S}\times\mathbf{S} \rightarrow \mathbf{Bool} \\
.\neq. \quad &: \quad \mathbf{S}\times\mathbf{S} \rightarrow \mathbf{Bool} \\
\mathsf{if}\ .\ \mathsf{then}\ .\ \mathsf{else}\ . \quad &: \quad \mathbf{Bool}\times\mathbf{S}\times\mathbf{S} \rightarrow \mathbf{S}
\end{aligned}
$$

  Note the (harmless) overloading of the constants and function symbols of type **Bool** with the propositional connectives used in formulas. We will frequently view boolean valued expressions as formulas, i.e., we use b as an abbreviation of b=true.

- a type **Process** of process identifiers. We frequently use the variable P ranging over **Process** as a subscript.

- a type **MemLocs** of legal memory locations.

- a type **MemVals** of legal memory values, with constant symbol $\mathsf{InitVal}$. None of the memory values is equal to BadArg.

- a type **Locs** of memory locations, such that **MemLocs** $\subseteq$ **Locs**, and a function memloc : **Locs** $\rightarrow$ **Bool**, telling us whether an element of **Locs** is also an element of **MemLocs**.

- a type **Vals** of memory values, such that **MemVals** $\subseteq$ **Vals**, and a function memval : **Vals** $\rightarrow$ **Bool**, telling us whether an element of **Vals** is also an element of **MemVals**.

- a type **Ack** of acknowledgement values, such that **Ack** = **MemVals** $\cup$ {WriteOk}.

- a type **Memory** of functions from **MemLocs** to **MemVals**. We need two functions to actually access the memory: find : **Locs**$\times$**Memory** $\rightarrow$ **MemVals** and change : **Locs**$\times$**Vals**$\times$**Memory** $\rightarrow$ **Memory**. These operations are fully characterised by the axioms:

$$
\begin{aligned}
\mathsf{find}(l, m) \quad &= \quad \mathsf{if}\ \mathsf{memloc}(l)\ \mathsf{then}\ m(l)\ \mathsf{else}\ \mathsf{InitVal} \\
\mathsf{change}(l, v, m) \quad &= \quad \mathsf{if}\ \mathsf{memloc}(l) \wedge \mathsf{memval}(v)\ \mathsf{then}\ m'\ \mathsf{else}\ m \\
&\qquad \text{where } m'(l)=v \wedge \forall l' : (l' \neq l \rightarrow m'(l')=m(l'))
\end{aligned}
$$

  (l, l' are variables of type **Locs**, v is a variable of type **Vals**, and m, m' are variables of type **Memory**)

- a type **Mpc** of program counter values of the memory component, with constant symbols WC, R and W. The intended meaning of these constants will be explained further on in this section.

The memory component

We present the fair I/O automaton *Memory*, which models a memory component. The state variable pc_P of *Memory* gives the current value of the program counter of the memory component for calling process P. Note that there are as many program counters as calling processes. Each of them may have one of the following values:

- WC: Waiting for a $READ_P$ or $WRITE_P$ call,
- R: Reading from memory,
- W: Writing to memory.

Initially, the program counter value is WC for every process P.

Every possible action of *Memory* is indexed with the process that issued the call leading to this action. Since the state variables are also indexed in this manner (except for *memory*!), we can determine in any situation what is going on for each process P.

$READ_P$ and $WRITE_P$ model an incoming read or write call from a process P. They do not change the state when *Memory* is still handling a previous call from the same process. In this case, we call the input action *discarded*. If *Memory* is ready for handling an incoming call, its state is updated according to the parameter(s) of the call.

GET_P actions model an atomic read operation, PUT_P actions model an atomic write operation. Reading is allowed only once between call and return, writing is allowed for an arbitrary number of times.

A $MEM_FAILURE_P$ action can occur in any 'busy' state.

BAD_ARG_P is the only action enabled if the parameters of the call from process P were not legal. $RETURN_P$ delivers the requested memory value or a general WriteOk acknowledgement, after $performed_P$ has been set to true by a GET_P or PUT_P action. The fact that PUT_P actions are in another weak fairness set than $RETURN_P$ and $MEM_FAILURE_P$, ensures that writing will stop at some point.

The code for *Memory* is listed in Figure 3.1.

Liveness We show that fair I/O automaton *Memory* is a live I/O automaton in the sense of [SGSL98]. To do this, we have to check that *Memory* satisfies two conditions. After this, Theorem 1 from [RV96] applies immediately.

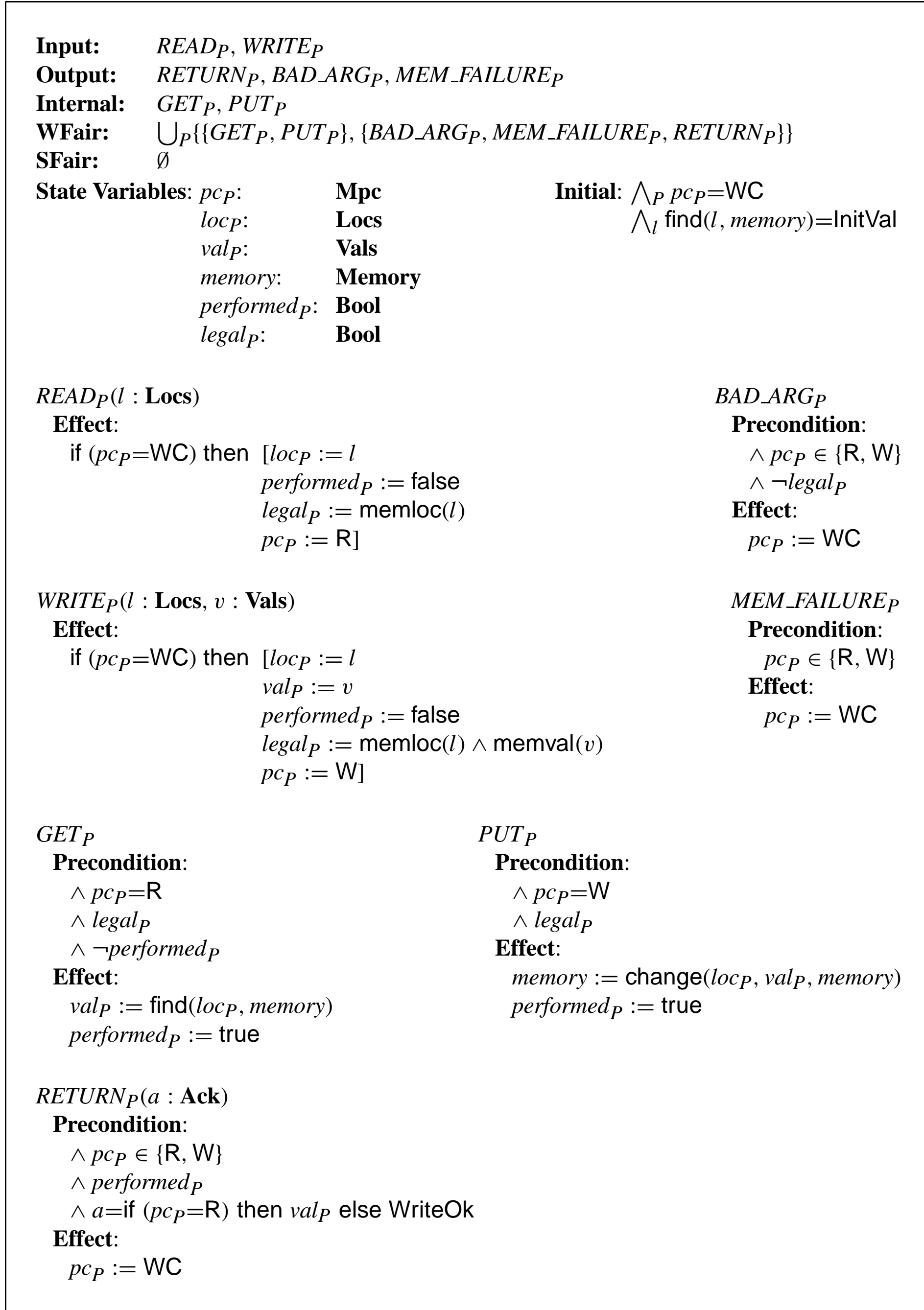
The next lemma checks a restriction of one of the two conditions.

Lemma 3.1 Each reachable state in *Memory* enables at most finitely many locally controlled actions.

Proof For each process P, locally controlled actions can only be enabled if $pc_P \neq$WC. Suppose there is an execution with n actions leading to state s. Then there are at most n processes P such that $s \models pc_P \neq$WC, hence s enables at most $5n$ locally controlled actions. ⊠

Proposition 3.2 *live*(*Memory*) is a live I/O automaton.

Proof We can apply Theorem 1 in [RV96] if we can show that (1) each reachable state of *Memory* enables at most countably many weak and strong fairness sets, and (2) each set in *sfair*(*Memory*) is input resistant.



Input: $READ_P$, $WRITE_P$
Output: $RETURN_P$, BAD_ARG_P, $MEM_FAILURE_P$
Internal: GET_P, PUT_P
WFair: $\bigcup_P \{\{GET_P, PUT_P\}, \{BAD_ARG_P, MEM_FAILURE_P, RETURN_P\}\}$
SFair: $\emptyset$

State Variables:
pc_P: **Mpc**
loc_P: **Locs**
val_P: **Vals**
$memory$: **Memory**
$performed_P$: **Bool**
$legal_P$: **Bool**

Initial: $\bigwedge_P pc_P = \mathsf{WC}$
$\bigwedge_l \mathsf{find}(l, memory) = \mathsf{InitVal}$

$READ_P(l : \mathbf{Locs})$
Effect:
if $(pc_P = \mathsf{WC})$ then $[loc_P := l$
$performed_P := \mathsf{false}$
$legal_P := \mathsf{memloc}(l)$
$pc_P := \mathsf{R}]$

BAD_ARG_P
Precondition:
$\wedge\ pc_P \in \{\mathsf{R}, \mathsf{W}\}$
$\wedge\ \neg legal_P$
Effect:
$pc_P := \mathsf{WC}$

$WRITE_P(l : \mathbf{Locs}, v : \mathbf{Vals})$
Effect:
if $(pc_P = \mathsf{WC})$ then $[loc_P := l$
$val_P := v$
$performed_P := \mathsf{false}$
$legal_P := \mathsf{memloc}(l) \wedge \mathsf{memval}(v)$
$pc_P := \mathsf{W}]$

$MEM_FAILURE_P$
Precondition:
$pc_P \in \{\mathsf{R}, \mathsf{W}\}$
Effect:
$pc_P := \mathsf{WC}$

GET_P
Precondition:
$\wedge\ pc_P = \mathsf{R}$
$\wedge\ legal_P$
$\wedge\ \neg performed_P$
Effect:
$val_P := \mathsf{find}(loc_P, memory)$
$performed_P := \mathsf{true}$

PUT_P
Precondition:
$\wedge\ pc_P = \mathsf{W}$
$\wedge\ legal_P$
Effect:
$memory := \mathsf{change}(loc_P, val_P, memory)$
$performed_P := \mathsf{true}$

$RETURN_P(a : \mathbf{Ack})$
Precondition:
$\wedge\ pc_P \in \{\mathsf{R}, \mathsf{W}\}$
$\wedge\ performed_P$
$\wedge\ a = \mathsf{if}\ (pc_P = \mathsf{R})\ \mathsf{then}\ val_P\ \mathsf{else}\ \mathsf{WriteOk}$
Effect:
$pc_P := \mathsf{WC}$

Figure 3.1: Fair I/O automaton *Memory*

Condition (1) is satisfied by Lemma 3.1, since each locally controlled action is in exactly one weak fairness set. Condition (2) is trivially satisfied, since there are no strong fairness sets. ⊠

The reliable memory component
We present the fair I/O automaton *RelMemory*, which models a reliable memory component. This component behaves exactly like the memory component, except that it can never issue a *MEM_FAILURE*.

Since the code for *RelMemory* can be obtained from the code for *Memory* by omitting the *MEM_FAILURE* action, *wfair*(*RelMemory*) becomes
$\bigcup_P \{\{GET_P, PUT_P\}, \{BAD_ARG_P, RETURN_P\}\}$

Liveness Knowing that *Memory* is a live I/O automaton, it is easy to prove that *RelMemory* is also a live I/O automaton.

Proposition 3.3 *live*(*RelMemory*) is a live I/O automaton.

Proof The proof is almost identical to the proof of Proposition 3.2, since the only difference between *Memory* and *RelMemory* is the absence of $MEM_FAILURE_P$ actions. ⊠

3.3.2 Problem 1(b): *RelMemory* implements *Memory*

We show that *fairtraces*(*RelMemory*) $\subseteq$ *fairtraces*(*Memory*), using the properties safety and deadlock freeness.

Safety Since *RelMemory* and *Memory* are so very much alike, a weak refinement appears the most natural construction for proving safety.

Theorem 3.4 The function REF, which is the identity function on state variables with the same name, is a weak refinement from *RelMemory* to *Memory*, with respect to the reachable states in both *RelMemory* and *Memory*.

Proof The requirements in [LV95] are trivially fulfilled, since REF is the identity function, and the actions in *RelMemory* form a subset of those in *Memory*. ⊠

Corollary 3.5 *RelMemory* is safe with respect to *Memory*.

Proof Directly from Theorem 3.4 in this chapter and Theorem 6.2 in [LV95]. ⊠

Deadlock freeness

Theorem 3.6 For each reachable and quiescent state s of *RelMemory*, REF(s) is a quiescent state of *Memory*.

Proof Suppose s is a quiescent state of *RelMemory*. Observing the preconditions of *RelMemory*, we see that $s \models \bigwedge_P RelMemory.pc_P = \mathsf{WC}$.
Clearly, REF(s) $\models \bigwedge_P Memory.pc_P = \mathsf{WC}$, hence REF($s$) is quiescent. ⊠

Corollary 3.7 *RelMemory* is deadlock free with respect to *Memory*.

Proof By Theorems 3.4 and 3.6 we can, for each quiescent execution of *RelMemory*, construct a corresponding quiescent execution of *Memory* with the same trace. ⊠

Implementation

Theorem 3.8 *RelMemory* implements *Memory*.

Proof We prove $fairtraces(RelMemory) \subseteq fairtraces(Memory)$.
Assume that $\beta \in fairtraces(RelMemory)$. Let α be a fair execution of *RelMemory* with trace β.

If α is finite then α is quiescent and it follows by Corollary 3.7 that *Memory* has a quiescent execution with trace β. Since each quiescent execution is also fair, this implies $\beta \in fairtraces(Memory)$. So we may assume without loss of generality that α is infinite.

Using the fact that REF is a weak refinement (Theorem 3.4) we can easily construct an execution α' of *Memory* with trace β. It remains to prove that α' is fair.

The only case in which α is fair but α' is not, is obtained as follows. In a infinite suffix β' of α', for some P, $MEM_FAILURE_P$ is enabled continuously, but no action from $\{RETURN_P, BAD_ARG_P, MEM_FAILURE_P\}$ is performed. In this case, α must contain an infinite suffix β in which no action from $\{RETURN_P, BAD_ARG_P\}$ is performed. Since α is weakly fair, β is also weakly fair. Since in β', $MEM_FAILURE_P$ is enabled continuously, by definition of REF, in β, the set $\{GET_P, PUT_P, RETURN_P, BAD_ARG_P\}$ is enabled continuously. Since both $RETURN_P$ and BAD_ARG_P, once enabled, can only be disabled by being performed and since no action from $\{RETURN_P, BAD_ARG_P\}$ occurs in β and β is fair, the set $\{RETURN_P, BAD_ARG_P\}$ is not enabled in any state in β. So the set $\{GET_P, PUT_P\}$ is enabled continuously in β. Since any occurrence of an action from $\{GET_P, PUT_P\}$ enables $RETURN_P$, no action from $\{GET_P, PUT_P\}$ occurs in β. Since β is fair and $\{GET_P, PUT_P\}$ is enabled continuously but no action from $\{GET_P, PUT_P\}$ is performed in β, we have a contradiction.

The interpretation of all the other actions are equal in both automata, even with respect to the weak fairness sets, so the weak fairness requirements for α' are satisfied by the weak fairness requirements for α.

Since *Memory* has no strong fairness sets, the above shows that α' is fair. ⊠

3.3.3 Problem 1(c): Nothing but $MEM_FAILURE_P$ actions?

We can construct a very trivial automaton that implements *Memory*, and does nothing but raise $MEM_FAILURE_P$ actions. It can have the same state variables as *Memory*, but only actions $READ_P$, $WRITE_P$ and $MEM_FAILURE_P$. A weak refinement like REF will provide us safety and deadlock freeness results. Such a refinement is even enough to show that this automaton implements *Memory*, since each fair execution in this automaton can be imitated by a fair execution in *Memory*, using the refinement.

Is it reasonable that such an implementation is possible? Since the specification of the memory component is presented as a black box that does not remember success nor failure, it is reasonable to think of it as a dice, harbouring the same chances at success with every throw. So while one can expect such a memory component to yield the right return at some time in an infinite sequence of trials, the possibility of infinitely many failures exists and must therefore be included in the specification we have presented here.

3.4 Specifications and verifications for Problem 2

3.4.1 Problem 2: Specification of the RPC component

Data types We assume a typed signature Σ_2 and a Σ_2-algebra $\mathcal{A}_2$ which consist of the following components:

- the type **Bool** as defined in Section 3.3.1
- a type **Nat** of natural numbers
- a type **Procs** of procedure names
- a type **Names**, such that **Procs** $\subseteq$ **Names**, and a function $\mathsf{legal_proc}$: **Names** $\rightarrow$ **Bool**, telling us whether a given name is a legal procedure name (that is, an element of **Procs**), and a function $\mathsf{arg_num}$: **Names** $\rightarrow$ **Nat**, giving the expected number of arguments for each name.
- a type **Args** of function arguments, and a function num : **Args** $\rightarrow$ **Nat**, giving the number of actual arguments.
- a function $\mathsf{legal_call}$: **Names**$\times$**Args** $\rightarrow$ **Bool**, such that $\mathsf{legal_call}(p, a) = \mathsf{legal_proc}(p) \wedge (\mathsf{arg_num}(p) = \mathsf{num}(a))$ for each p in **Names** and a in **Args**.
- a type **ReturnVal** of possible return values. All exceptions raised by remote procedure calls are expected to be included in this type.
- a type **Rpc** of program counter values of the RPC component, with constant symbols WC, IC, WR and IR.

Specification We present the fair I/O automaton *RPC*, which models an RPC component. RPC stands for Remote Procedure Call. The program counters in *RPC*may have one of the following values:

- WC: Wait for remote calls from the sender
- IC: Issue a call to the receiver or an exceptional return to the sender
- WR: Wait for a return from the receiver
- IR: Issue a return (possibly exceptional) to the sender

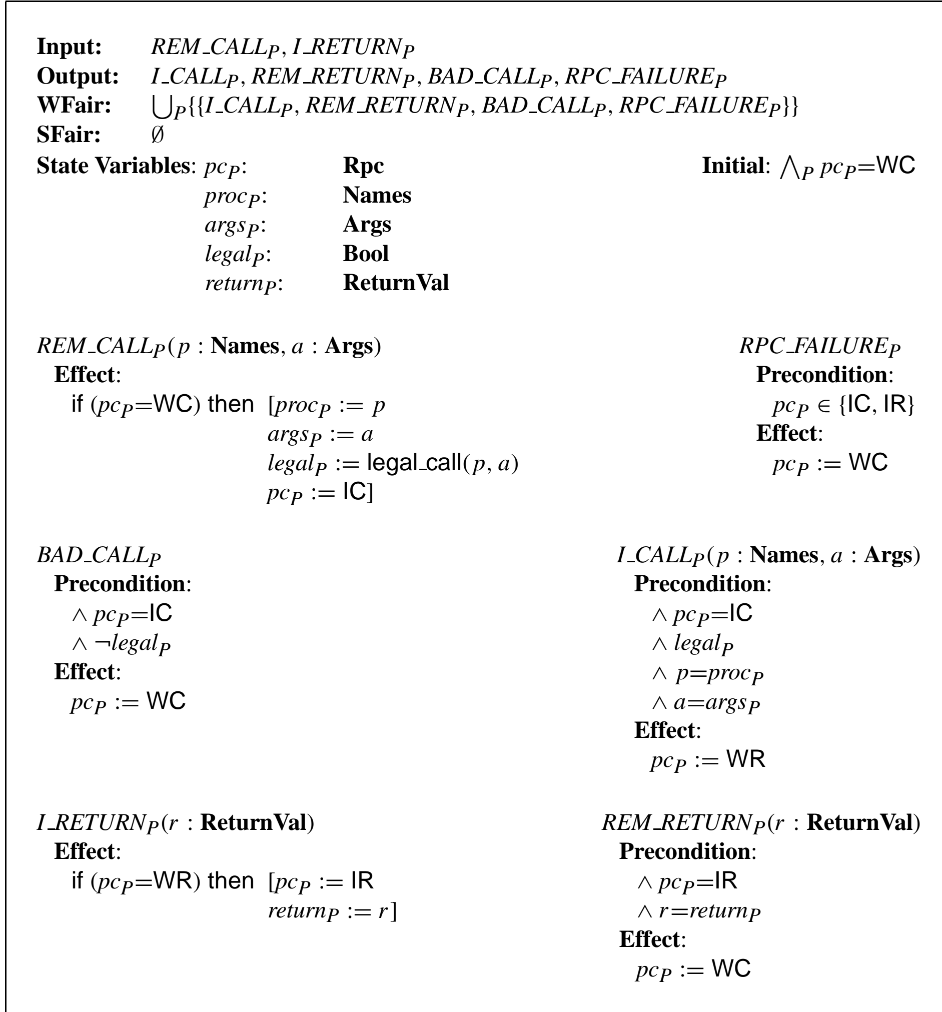
Initially, the program counter value is WC for every process P.

The code for *RPC* is listed in Figure 3.2.

Liveness *RPC* is a live I/O automaton.

Lemma 3.9 Each reachable state in *RPC* enables at most finitely many locally controlled actions.

Proposition 3.10 *live*(*RPC*) is a live I/O automaton.



Input: REM_CALL_P, I_RETURN_P
Output: I_CALL_P, REM_RETURN_P, BAD_CALL_P, $RPC_FAILURE_P$
WFair: $\bigcup_P \{\{I_CALL_P, REM_RETURN_P, BAD_CALL_P, RPC_FAILURE_P\}\}$
SFair: $\emptyset$
State Variables: pc_P: **Rpc**; $proc_P$: **Names**; $args_P$: **Args**; $legal_P$: **Bool**; $return_P$: **ReturnVal**
Initial: $\bigwedge_P pc_P = \mathsf{WC}$

$REM_CALL_P(p : \mathbf{Names}, a : \mathbf{Args})$
Effect:
if $(pc_P = \mathsf{WC})$ then $[proc_P := p$
$args_P := a$
$legal_P := \mathsf{legal_call}(p, a)$
$pc_P := \mathsf{IC}]$

$RPC_FAILURE_P$
Precondition:
$pc_P \in \{\mathsf{IC}, \mathsf{IR}\}$
Effect:
$pc_P := \mathsf{WC}$

BAD_CALL_P
Precondition:
$\wedge\ pc_P = \mathsf{IC}$
$\wedge\ \neg legal_P$
Effect:
$pc_P := \mathsf{WC}$

$I_CALL_P(p : \mathbf{Names}, a : \mathbf{Args})$
Precondition:
$\wedge\ pc_P = \mathsf{IC}$
$\wedge\ legal_P$
$\wedge\ p = proc_P$
$\wedge\ a = args_P$
Effect:
$pc_P := \mathsf{WR}$

$I_RETURN_P(r : \mathbf{ReturnVal})$
Effect:
if $(pc_P = \mathsf{WR})$ then $[pc_P := \mathsf{IR}$
$return_P := r]$

$REM_RETURN_P(r : \mathbf{ReturnVal})$
Precondition:
$\wedge\ pc_P = \mathsf{IR}$
$\wedge\ r = return_P$
Effect:
$pc_P := \mathsf{WC}$

Figure 3.2: Fair I/O automaton *RPC*

3.5 Specifications and verifications for Problem 3

3.5.1 Problem 3: Specification of the composition

Data types We reuse Σ_1 (section 3.3.1) and Σ_2 (section 3.4.1) to obtain a typed signature Σ_3 and a Σ_3-algebra, such that:

- Read and Write are different constants of type **Procs** (and therefore also of type **Names**)
- arg_num(Read) $= 1$, and arg_num(Write) $= 2$
- the domain of **ReturnVal** is equal to the domain of **Ack**, plus an extra constant BadArg
- for each l, l' of type **Locs** and v, v' of type **Vals**, (l) and (l, v) are elements of type **Args**, $(l) = (l') \rightarrow l = l'$, $(l, v) = (l', v') \rightarrow l = l' \wedge v = v'$, $\mathsf{num}((l)) = 1$ and $\mathsf{num}((l, v)) = 2$.

A front end for the RPC component We need another component to make the RPC component retry a call to the reliable memory component. This component is a clerk, which can translate incoming calls to the format accepted by *RPC*, and reissue such a call if *RPC* should fail. Therefore we present the fair I/O automaton *ClerkR*, which models a front end to the RPC component *RPC*. The program counters of *ClerkR* are of type **Rpc**, and therefore have the same possibilities as the program counters of *RPC*. Initially, the program counter value is WC for every process *P*.
The code for *ClerkR* is listed in Figure 3.3.

Liveness Fair I/O automaton *ClerkR* is a live I/O automaton.

Lemma 3.11 Each reachable state in *ClerkR* enables at most finitely many locally controlled actions.

Proposition 3.12 *live*(*ClerkR*) is a live I/O automaton.

Proof As before, we apply Theorem 1 in [RV96] after showing that (1) each reachable state of *RPC* enables at most countably many weak and strong fairness sets, and (2) each set in *sfair*(*ClerkR*) is input resistant.

Condition (1) is satisfied by Lemma 3.11, since each locally controlled action is in exactly one weak fairness set.

Condition (2) relies upon the input resistance of action *MEM_FAILURE*. Suppose that $MEM_FAILURE_P$ is enabled in the reachable state s. Clearly, $s \models ClerkR.pc_P = \mathsf{IC}$. If an input action a for P occurs in s, by definition of *ClerkR* the transition $s \xrightarrow{a} s$ is taken, and $MEM_FAILURE_P$ is still enabled. If an input action a for another P' occurs in s, the transition taken does not affect $ClerkR.pc_P$. Hence $MEM_FAILURE_P$ is input resistant and the second condition is satisfied. ⊠

Renaming component *RelMemory* The front end *ClerkR* is not enough to establish the intended implementation. We also need to rename *RelMemory* to avoid name clashing, and

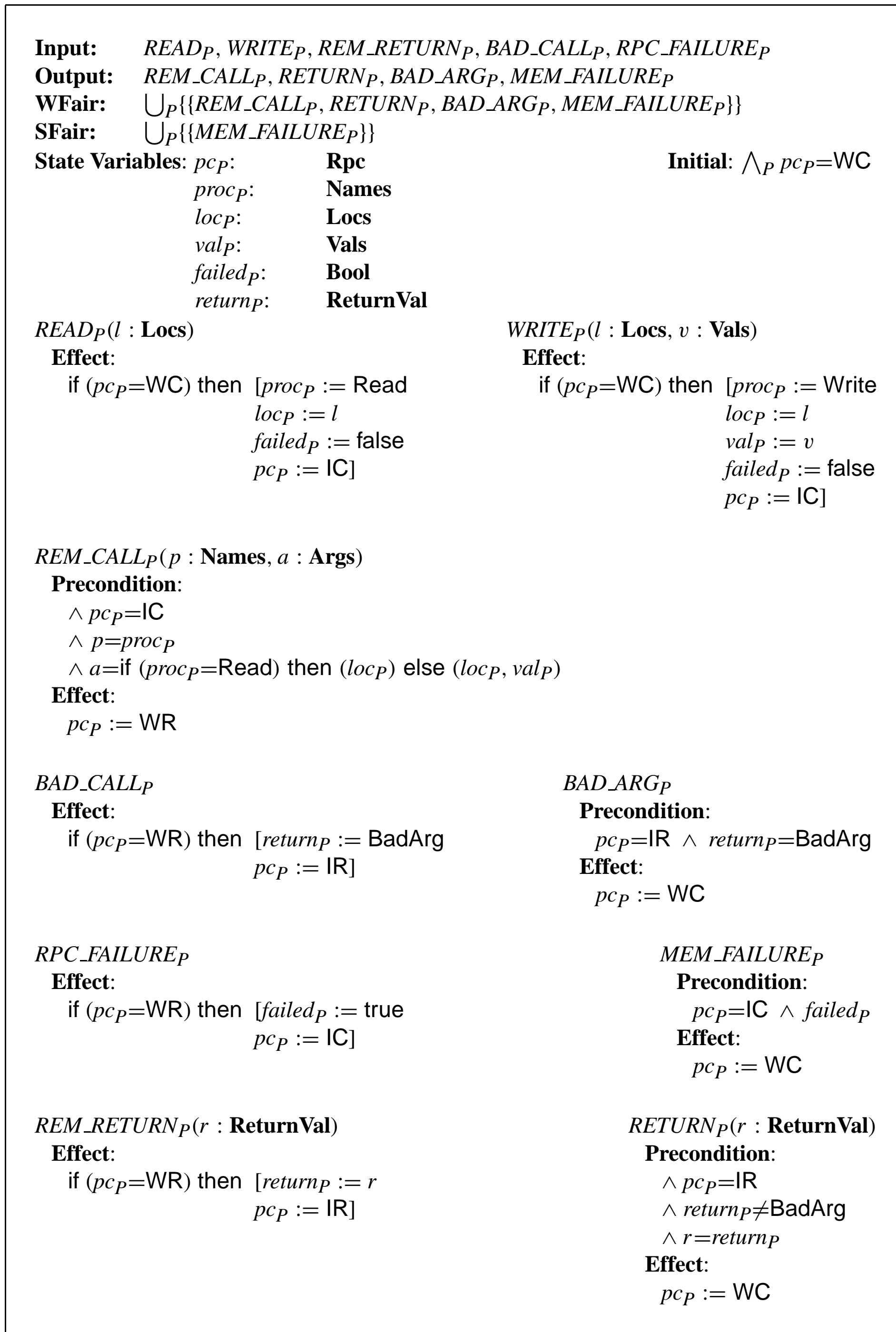


Figure 3.3: Fair I/O automaton *ClerkR*

to get the proper synchronisation. So we define a new fair I/O automaton $RMemory' \triangleq$ rename($RelMemory$), where for every P:

$$
\begin{array}{lll}
\mathsf{rename}(READ_P(l)) & = I_CALL_P(\mathsf{Read}, (l)) & \\
\mathsf{rename}(WRITE_P(l, v)) & = I_CALL_P(\mathsf{Write}, (l, v)) & \\
\mathsf{rename}(RETURN_P(a)) & = I_RETURN_P(a) & \\
\mathsf{rename}(BAD_ARG_P) & = I_RETURN_P(\mathsf{BadArg}) & \\
\mathsf{rename}(x) & = x & \text{otherwise}
\end{array}
$$

(l is a variable of type **Locs**, v is a variable of type **Vals**, a is a variable of type **Ack**, and x is a action variable)

The code for $RMemory'$ is listed in Figure 3.4.

Liveness It is easily shown that $RMemory'$ is a live I/O automaton.

Proposition 3.13 *live*($RMemory'$) is a live I/O automaton.

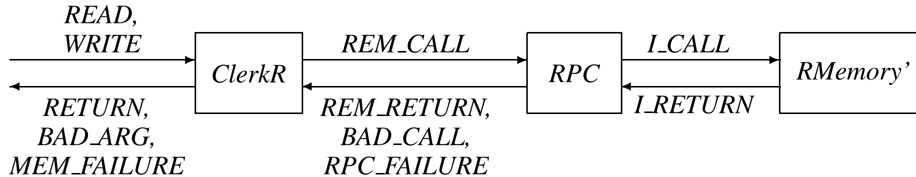
Proof Trivially, $live(RMemory') = rename(live(RelMemory))$. Combining this with Theorem 3.3 in this chapter and Proposition 3.23 in [SGSL98], we obtain that $live(RMemory')$ is a live I/O automaton. ⊠

The implementation *MemoryImp* is defined as the parallel composition of I/O automata *ClerkR*, *RPC* and $RMemory'$, with all communication between those components hidden:

$$MemoryImp \triangleq \mathsf{HIDE}\ I\ \mathsf{IN}\ (ClerkR \| RPC \| RMemory')$$

where $I \triangleq \bigcup_P \{REM_CALL_P(p, a), REM_RETURN_P(r), BAD_CALL_P,$
$RPC_FAILURE_P, I_CALL_P(p, a), I_RETURN_P(r)$
$\mid p$ in **Names**, a in **Args**, r in **ReturnVal**$\}$.

The behaviour of *RPCImp* is illustrated in the following figure.



Liveness

Proposition 3.14 *live*(*MemoryImp*) is a live I/O automaton.

Proof (Sketch) We use Propositions 3.10, 3.12 and 3.13 in this chapter, Propositions 3.22 and 3.28 in [SGSL98], and Theorem 2 in [RV96]. ⊠

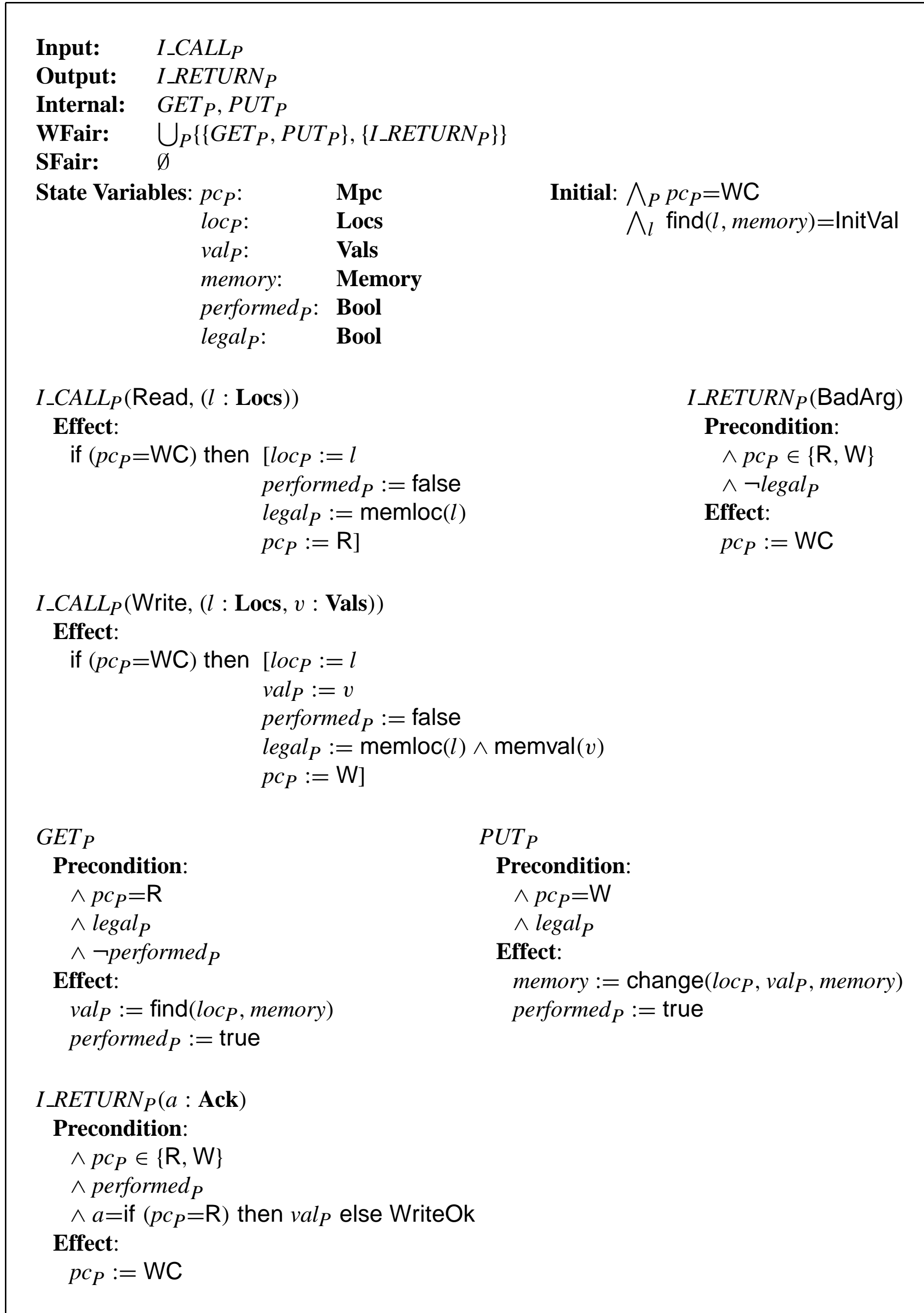


Figure 3.4: Fair I/O automaton *RMemory′*

3.5.2 Set-up for the verification

A direct proof of trace inclusion between *MemoryImp* and *Memory* is not very straightforward. This stems from the fact that *Memory* can only read its memory once for every read call. However, by the fail/retry mechanism of *MemoryImp*, it is able to read multiple times for one read call.

An intermediate automaton To show trace inclusion, we seem to need a forward backward simulation. However, since this is rather complicated, and Theorem 4.1 in [LV95] states that we can just as well look for an intermediate automaton, we will keep things clear by constructing an intermediate automaton, which we allow to read its memory multiple times for one read call. This intermediate automaton will be called *Memory**, the $*$ indicating the possibility of multiple reads instead of one. The code for *Memory** is obtained from *Memory* as follows. The precondition for GET_P is weakened, and a new state variable $hist_P$ is added, in which the value of val_P is stored each time a return is issued. Figure 3.5 lists the code for fair I/O automaton *Memory**. Boxes highlight the places where the code for *Memory** differs from *Memory*.

A forward simulation establishes trace inclusion between *MemoryImp* and *Memory**; a backward simulation does the same for *Memory** and *Memory*. The use of the new state variable *Memory**.$hist_P$ substantially simplifies the backward simulation and also makes it image-finite.

Liveness Fair I/O automaton *Memory** is a live I/O automaton.

Lemma 3.15 Each reachable state in *Memory** enables at most finitely many locally controlled actions.

Proposition 3.16 *live*(*Memory**) is a live I/O automaton.

3.5.3 Problem 3: *MemoryImp* implements *Memory*

In this section, we will first show that *Memory** implements *Memory*, then we will show that *MemoryImp* implements *Memory**. Both results are reached via safety and deadlock freeness. Transitivity of the implementation relation yields the desired result in Section 3.5.3.

*Memory** **implements** *Memory* We need an invariant to show that between the previous output action and the next internal action, the history variable $hist_P$ in *Memory** is up to date with respect to val_P for each P.

Lemma 3.17 The following property Inv1 is an invariant of *Memory**.

$$\bigwedge_P (pc_P \in \{\mathsf{WC}, \mathsf{R}\} \wedge \neg performed_P) \rightarrow val_P{=}hist_P$$

The next invariant expresses that *Memory** will not read or write if it has received illegal arguments.

Lemma 3.18 The following property Inv2 is an invariant of *Memory**.

$$\bigwedge_P pc_P{\neq}\mathsf{WC} \rightarrow (\neg legal_P \rightarrow \neg performed_P)$$

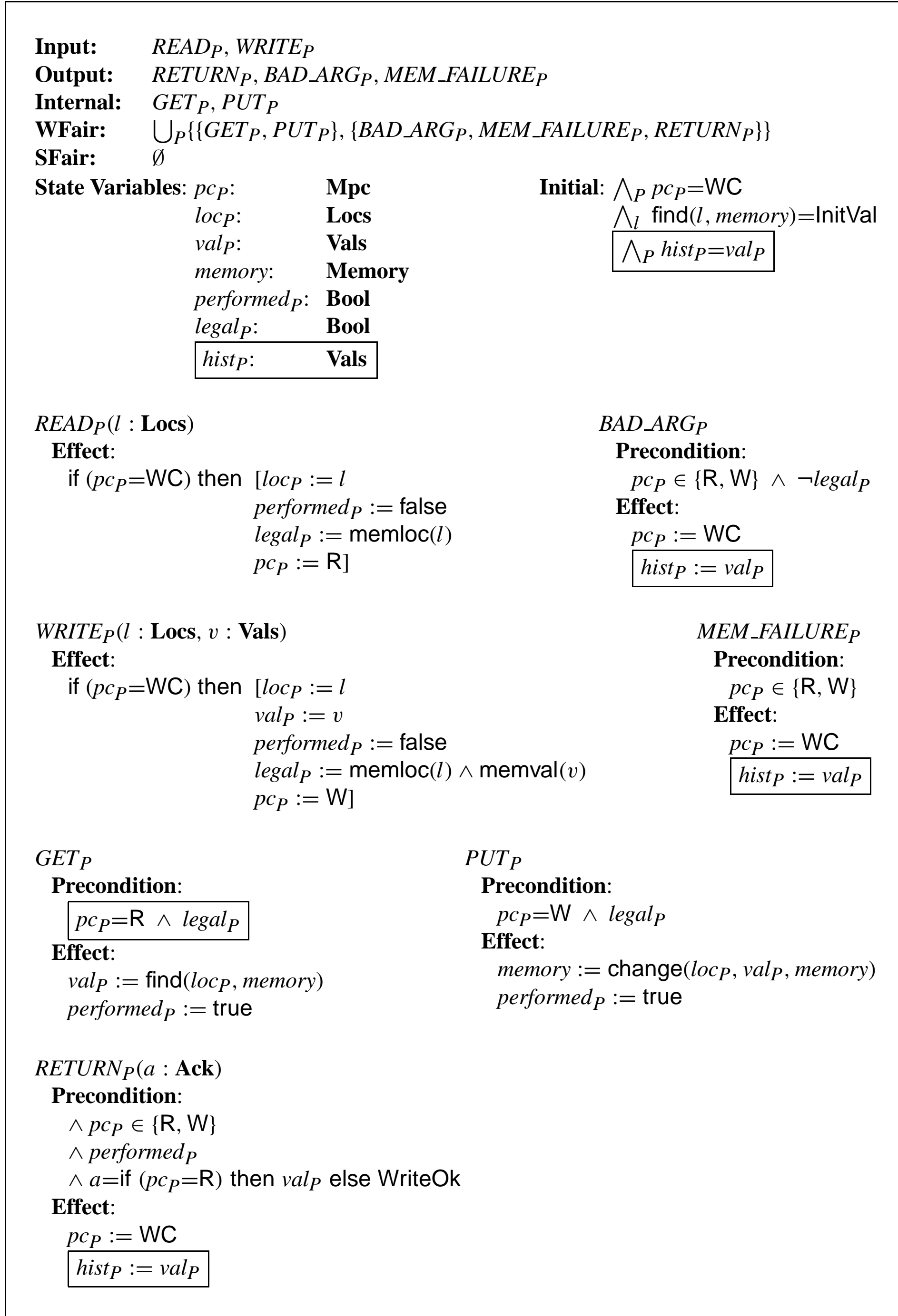


Figure 3.5: Fair I/O automaton *Memory**

A weak backward simulation enables us to construct the behaviour of *Memory*, given the behaviour of *Memory**. We can start in the last state of such a sequence, and work our way back to the beginning. The relation that induces this simulation needs to be image-finite.

Lemma 3.19 The relation BACK defined by the following formula is an image-finite relation over *rstates*(*Memory**) and *states*(*Memory*).

$$\begin{array}{lcl}
\bigwedge_P \mathit{Memory.pc}_P & = & \mathit{Memory}^*.\mathit{pc}_P \\
\bigwedge_P \mathit{Memory.loc}_P & = & \mathit{Memory}^*.\mathit{loc}_P \\
\bigwedge_P \mathit{Memory.val}_P & = & \mathsf{if}\quad \mathit{Memory.pc}_P{=}\mathsf{R} \wedge \neg \mathit{Memory.performed}_P \\
 & & \mathsf{then} \mathit{Memory}^*.\mathit{hist}_P \\
 & & \mathsf{else} \mathit{Memory}^*.\mathit{val}_P \\
\bigwedge_P \mathit{Memory.legal}_P & = & \mathit{Memory}^*.\mathit{legal}_P \\
\wedge \quad \mathit{Memory.memory} & = & \mathit{Memory}^*.\mathit{memory} \\
\multicolumn{3}{l}{\bigwedge_P \neg \mathit{Memory}^*.\mathit{performed}_P \rightarrow \neg \mathit{Memory.performed}_P} \\
\multicolumn{3}{l}{\bigwedge_P \mathit{Memory}^*.\mathit{pc}_P{\neq}\mathsf{R} \rightarrow (\mathit{Memory}^*.\mathit{performed}_P \rightarrow \mathit{Memory.performed}_P)}
\end{array}$$

Theorem 3.20 Relation BACK is a weak backward simulation from *Memory** to *Memory*, with respect to the reachable states in *Memory**.

Proof (Sketch) We satisfy the three requirements in [LV95], which is a bit complicated and takes a lot of paper. The most difficult part is caused by the *GET* action, since *Memory* does not always perform this action along with *Memory**. Here, the history variable of *Memory** proves its value. ⊠

Corollary 3.21 *Memory** is safe with respect to *Memory*.

Proof Combining Lemma 3.19 and Theorem 3.20 in this chapter with Theorem 6.2 in [LV95], we obtain the desired result. ⊠

Theorem 3.22 For each reachable, quiescent state s of *Memory**, each state $r \in \mathrm{BACK}(s)$ is a quiescent state of *Memory*.

Proof Considering the preconditions of *Memory**, in each quiescent state s, $\mathit{Memory}^*.\mathit{pc}_P$ must be equal to WC for every P. For each $r \in \mathrm{BACK}(s) : r \models \bigwedge_P \mathit{Memory.pc}_P{=}\mathsf{WC}$, hence r is quiescent. ⊠

Corollary 3.23 *Memory** is deadlock free with respect to *Memory*.

Proof By Theorems 3.20 and 3.22 we can construct, for each quiescent execution of *Memory**, a corresponding quiescent execution of *Memory* with the same trace. ⊠

Theorem 3.24 *Memory** implements *Memory*.

Proof (Sketch) Assume that $\beta \in \mathit{fairtraces}(\mathit{Memory}^*)$. Let α be a fair execution of *Memory** with the same trace β. If α is finite then α is quiescent and it follows by Corollary 3.23 that *Memory* has a quiescent execution with trace β. Since each quiescent execution is also fair, this implies $\beta \in \mathit{fairtraces}(\mathit{Memory})$. So we may assume without loss of generality that α is infinite.

Using the fact that BACK is a weak image-finite backward simulation (see Lemma 3.19, Theorem 3.20), we can easily construct an execution α' of *Memory* with trace β. It remains to

prove that α' is fair.

We need to show that α' must be infinite. Again, the GET_P action causes trouble, since *Memory* does not always perform it when *Memory** does. However, fairness helps us establish the fact that *Memory** cannot perform infinitely many GET_P actions for P, without performing other actions for P in between. Since *Memory* imitates each of these other actions, the infinity of α' is inevitable.

Using the above, the fairness of α' is satisfied quite trivially because of three facts. Firstly, $wfair(Memory) = wfair(Memory^*)$ and $sfair(Memory) = sfair(Memory^*) = \emptyset$. Secondly, if a weak fairness set is not enabled in *Memory**, it is certainly not enabled in *Memory*. Thirdly, infinitely many occurrences of action a in α cause infinitely many occurrences of a in α'. ⊠

MemoryImp **implements** *Memory**

Invariants The following list of invariants is rather dull. They are necessary for ensuring that the arguments of an incoming call are transmitted properly among the components of *MemoryImp*, and no component will act before it receives permission to do so.

Component *RPC* will remain quiescent until a request is issued by component *ClerkR*:

Lemma 3.25 The following property Inv3 is an invariant of *MemoryImp*.

$$\bigwedge_P ClerkR.pc_P \neq \mathsf{WR} \rightarrow RPC.pc_P = \mathsf{WC}$$

Component *RMemory'* will remain quiescent until a request is issued by component *RPC*:

Lemma 3.26 The following property Inv4 is an invariant of *MemoryImp*.

$$\bigwedge_P RPC.pc_P \neq \mathsf{WR} \rightarrow RMemory'.pc_P = \mathsf{WC}$$

Component *ClerkR* only handles read or write calls:

Lemma 3.27 The following property Inv5 is an invariant of *MemoryImp*.

$$\bigwedge_P ClerkR.pc_P \neq \mathsf{WC} \rightarrow \begin{array}{l} \vee \wedge ClerkR.proc_P = \mathsf{Read} \\ \quad \wedge \exists l : ClerkR.loc_P = l \\ \vee \wedge ClerkR.proc_P = \mathsf{Write} \\ \quad \wedge \exists l : ClerkR.loc_P = l \\ \quad \wedge \exists v : ClerkR.val_P = v \end{array}$$

Component *RPC* receives the same calls and arguments from *ClerkR*, as *ClerkR* received from the environment:

Lemma 3.28 The following property Inv6 is an invariant of *MemoryImp*.

$$\bigwedge_P RPC.pc_P \neq \mathsf{WC} \rightarrow \begin{array}{l} \wedge RPC.proc_P = ClerkR.proc_P \\ \wedge RPC.args_P = \begin{array}{ll} \mathsf{if} & ClerkR.proc_P = \mathsf{Read} \\ \mathsf{then} & (ClerkR.loc_P) \\ \mathsf{else} & (ClerkR.loc_P, ClerkR.val_P) \end{array} \end{array}$$

Component *RPC* only receives read or write calls:

Corollary 3.29 The following property Inv7 is an invariant of *MemoryImp*.

$$\bigwedge_P RPC.pc_P \neq \mathsf{WC} \rightarrow \begin{array}{l} \vee\ RPC.proc_P = \mathsf{Read} \wedge \exists l : RPC.args_P = (l) \\ \vee\ RPC.proc_P = \mathsf{Write} \wedge \exists l, v : RPC.args_P = (l, v) \end{array}$$

Proof Directly from invariants Inv3, Inv5 and Inv6. ⊠

Since Read and Write are proper procedure names, and *RPC* receives no other procedure calls, the action BAD_CALL_P is never enabled:

Corollary 3.30 The following property Inv8 is an invariant of *MemoryImp*.

$$\bigwedge_P \neg enabled(BAD_CALL_P)$$

If *RMemory′* is busy, it is by request of *RPC*, and the arguments have been transmitted properly:

Lemma 3.31 The following property Inv9 is an invariant of *MemoryImp*.

$$\begin{array}{l} \bigwedge_P RMemory'.pc_P = \mathsf{R} \rightarrow \begin{array}{l} \wedge\ RPC.pc_P = \mathsf{WR} \\ \wedge\ RPC.proc_P = \mathsf{Read} \\ \wedge\ RPC.args_P = (l) \rightarrow RMemory'.loc_P = l \end{array} \\ \bigwedge_P RMemory'.pc_P = \mathsf{W} \rightarrow \begin{array}{l} \wedge\ RPC.pc_P = \mathsf{WR} \\ \wedge\ RPC.proc_P = \mathsf{Write} \\ \wedge\ RPC.args_P = (l, v) \rightarrow \begin{array}{l} \wedge\ RMemory'.loc_P = l \\ \wedge\ RMemory'.val_P = v \end{array} \end{array} \end{array}$$

RPC can only issue a return to *ClerkR*, following a (possibly exceptional) return by *RMemory′*, and the return value is transmitted properly:

Lemma 3.32 The following property Inv10 is an invariant of *MemoryImp*.

$$\bigwedge_P RPC.pc_P = \mathsf{IR} \rightarrow \begin{array}{l} \vee \begin{array}{l} \wedge\ RMemory'.performed_P \\ \wedge\ RPC.return_P = \begin{array}{ll} \text{if} & RPC.proc_P = \mathsf{Read} \\ \text{then} & RMemory'.val_P \\ \text{else} & \mathsf{WriteOk} \end{array} \end{array} \\ \vee \begin{array}{l} \wedge\ \neg RMemory'.legal_P \\ \wedge\ RPC.return_P = \mathsf{BadArg} \end{array} \end{array}$$

Inv11 states the same result as Inv10, for component *ClerkR*:

Lemma 3.33 The following property Inv11 is an invariant of *MemoryImp*.

$$\bigwedge_P ClerkR.pc_P = \mathsf{IR} \rightarrow \begin{array}{l} \vee \begin{array}{l} \wedge\ RMemory'.performed_P \\ \wedge\ ClerkR.return_P = \begin{array}{ll} \text{if} & ClerkR.proc_P = \mathsf{Read} \\ \text{then} & RMemory'.val_P \\ \text{else} & \mathsf{WriteOk} \end{array} \end{array} \\ \vee \begin{array}{l} \wedge\ \neg RMemory'.legal_P \\ \wedge\ ClerkR.return_P = \mathsf{BadArg} \end{array} \end{array}$$

$RMemory'.legal_P$ behaves just like we expect it to, as long as *RMemory′* is busy:

Lemma 3.34 The following property Inv12 is an invariant of *MemoryImp*.

$$\begin{array}{l}\bigwedge_P RMemory'.pc_P{=}\mathsf{R} \ \rightarrow RMemory'.legal_P{=}\ \mathsf{memloc}(RMemory'.loc_P)\\ \bigwedge_P RMemory'.pc_P{=}\mathsf{W} \rightarrow RMemory'.legal_P{=} \wedge \mathsf{memloc}(RMemory'.loc_P)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathsf{memval}(RMemory'.val_P)\end{array}$$

$RMemory'.legal_P$ is not changed after $RMemory'$ returns to RPC:

Lemma 3.35 The following property Inv13 is an invariant of *MemoryImp*.

$$\begin{array}{l}\bigwedge_P RPC.pc_P{\in}\{\mathsf{WR},\mathsf{IR}\} \vee ClerkR.pc_P{=}\mathsf{IR}\\ \quad\rightarrow \vee \wedge ClerkR.proc_P{=}\mathsf{Write}\\ \qquad\quad \wedge RMemory'.legal_P{=} \wedge \mathsf{memloc}(ClerkR.loc_P)\\ \qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathsf{memval}(ClerkR.val_P)\\ \qquad \vee \wedge ClerkR.proc_P{=}\mathsf{Read}\\ \qquad\quad \wedge RMemory'.legal_P{=}\mathsf{memloc}(ClerkR.loc_P)\end{array}$$

$Memory^*.legal_P$ behaves just like we expect it to, as long as $Memory^*$ is busy:

Lemma 3.36 The following property Inv14 is an invariant of $Memory^*$.

$$\begin{array}{l}\bigwedge_P pc_P{=}\mathsf{R} \ \rightarrow legal_P{=}\mathsf{memloc}(loc_P)\\ \bigwedge_P pc_P{=}\mathsf{W} \rightarrow legal_P{=}\mathsf{memloc}(loc_P) \wedge \mathsf{memval}(val_P)\end{array}$$

Safety We use a weak forward simulation, instead of a weak refinement. In fact, a weak refinement does not exist from *MemoryImp* to $Memory^*$. Suppose *ClerkR* receives a read call for P for the first time, and *MemoryImp* transits to state s. $Memory^*$ imitates this step, and ends up in an image state of s with $Memory^*.performed_P$ equal to false. Suppose *ClerkR* forwards the call to *RPC*, which forwards it to $RMemory'$. Suppose $RMemory'$ performs some reading activity. We can only ensure that $Memory^*$ returns the same value as $RMemory'$ if they read and write simultaneously. So in the image state of s, $Memory^*.performed_P$ must be false. Suppose after this reading activity, *RPC* returns a fail to *ClerkR*. This may lead to the same state s again. However, $Memory^*$ has imitated the read actions performed by $RMemory'$, and $Memory^*.performed_P$ must therefore be true. So a refinement should map s onto a state in which $Memory^*.performed_P$ is both true and false, which is a contradiction.

Theorem 3.37 The relation SIM defined by the following formula is a weak forward simulation from *MemoryImp* to $Memory^*$, with respect to the reachable states in both *MemoryImp* and $Memory^*$.

$$\begin{array}{l}\bigwedge_P Memory^*.pc_P \qquad = \mathsf{if}\quad ClerkR.pc_P{=}\mathsf{WC}\\ \qquad\qquad\qquad\qquad\quad\ \ \mathsf{then}\ \mathsf{WC}\\ \qquad\qquad\qquad\qquad\quad\ \ \mathsf{else}\ \mathsf{if}\ ClerkR.proc_P{=}\mathsf{Read}\ \mathsf{then}\ \mathsf{R}\ \mathsf{else}\ \mathsf{W}\\ \bigwedge_P Memory^*.loc_P \qquad = ClerkR.loc_P\\ \wedge \quad Memory^*.memory = RMemory'.memory\\ \bigwedge_P ClerkR.proc_P{=}\mathsf{Write} \rightarrow Memory^*.val_P{=}ClerkR.val_P\\ \bigwedge_P RMemory'.performed_P \wedge \vee RPC.pc_P{\in}\{\mathsf{WR},\mathsf{IR}\}\\ \qquad\qquad\qquad\qquad\qquad\qquad\ \vee ClerkR.pc_P{=}\mathsf{IR}\\ \quad\rightarrow \wedge Memory^*.performed_P\\ \qquad \wedge Memory^*.val_P{=}RMemory'.val_P\end{array}$$

Proof (Sketch) We use the following property.
For each two reachable states s in *MemoryImp*, r in *Memory**:

$$
\begin{aligned}
r, s \models & \bigwedge_P \mathit{Memory}^*.\mathit{pc}_P{=}\mathsf{R} \rightarrow \mathit{Memory}^*.\mathit{legal}_P{=}\mathsf{memloc}(\mathit{ClerkR}.\mathit{loc}_P) \\
& \bigwedge_P \mathit{Memory}^*.\mathit{pc}_P{=}\mathsf{W} \rightarrow \mathit{Memory}^*.\mathit{legal}_P{=} \wedge \mathsf{memloc}(\mathit{ClerkR}.\mathit{loc}_P) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \wedge \mathsf{memval}(\mathit{ClerkR}.\mathit{val}_P)
\end{aligned}
$$

This follows directly from Inv5, Inv14 and the definition of SIM. Using this property, and the invariants Inv3, Inv5, Inv6, Inv8, Inv9, Inv11 and Inv13, the proof is obtained by fulfilling the two requirements in [LV95] in a straightforward manner. ⊠

Corollary 3.38 *MemoryImp* is safe with respect to *Memory**.

Proof Directly from theorem 3.37 and Theorem 6.2 in [LV95]. ⊠

Deadlock freeness In order to establish that *MemoryImp* is deadlock free with respect to *Memory**, we need an additional invariant. It expresses that as long as *ClerkR* is waiting for a return, *RPC* is busy. Likewise, if *RPC* is waiting for a return, *RMemory*′ is busy.

Lemma 3.39 The following property Inv15 is an invariant of *MemoryImp*.

$$
\begin{aligned}
& \bigwedge_P \mathit{ClerkR}.\mathit{pc}_P{=}\mathsf{WR} \rightarrow \mathit{RPC}.\mathit{pc}_P{\neq}\mathsf{WC} \\
& \bigwedge_P \mathit{RPC}.\mathit{pc}_P{=}\mathsf{WR} \quad\; \rightarrow \mathit{RMemory}'.\mathit{pc}_P{\in}\{\mathsf{R}, \mathsf{W}\}
\end{aligned}
$$

Theorem 3.40 For each reachable and quiescent state s of *MemoryImp*, each reachable state $r \in \mathit{Memory}^*$ such that $r, s \models$ SIM is a quiescent state of *Memory**.

Proof (Sketch) From the action types of *MemoryImp* and Inv15, we see that *MemoryImp* is quiescent in state s iff $s \models \mathit{ClerkR}.\mathit{pc}_P{=}\mathsf{WC}$. Since $r, s \models$ SIM, obviously $r \models \mathit{Memory}^*.\mathit{pc}_P$ $=\mathsf{WC}$, hence r is quiescent. ⊠

Corollary 3.41 *MemoryImp* is deadlock free with respect to *Memory**.

Proof By Theorems 3.37 and 3.40 we can construct for each quiescent execution of *MemoryImp*, a corresponding quiescent execution of *Memory** with the same trace. ⊠

Theorem 3.42 *MemoryImp* implements *Memory**.

Proof (Sketch) We prove $\mathit{fairtraces}(\mathit{MemoryImp}) \subseteq \mathit{fairtraces}(\mathit{Memory}^*)$.
Assume that $\beta \in \mathit{fairtraces}(\mathit{MemoryImp})$. Let α be a fair execution of *MemoryImp* with trace β. If α is finite then α is quiescent and it follows by Corollary 3.41 that *Memory** has a quiescent execution with trace β. Since each quiescent execution is also fair, this implies $\beta \in \mathit{fairtraces}(\mathit{Memory}^*)$. So we may assume without loss of generality that α is infinite.

Using the fact that SIM is a weak forward simulation (Theorem 3.37) we can easily construct an execution α' of *Memory** with the same trace β. It remains to prove that α' is fair.

First we show that α' is infinite. Then we observe that each non-discarded call to *MemoryImp* will lead to a return within a finite number of steps. Using these two facts, we can easily show for each class in $\mathit{wfair}(\mathit{Memory}^*)$, that α' satisfies the requirements for weak fairness. Since $\mathit{sfair}(\mathit{Memory}^*)$ is empty, this is enough to show that α' is fair. ⊠

The main result

Theorem 3.43 *MemoryImp* implements *Memory*.

Proof Theorems 3.24, 3.42 yield *fairtraces*(*MemoryImp*) $\subseteq$ *fairtraces*(*Memory*). ⊠

3.6 Specifications for Problem 4

3.6.1 Problem 4: Specification of a lossy RPC

The lossy RPC is a timed component whose behaviour is similar to the behaviour of the RPC component from section 3.4.1.

The difference between timed and untimed I/O automata is that time-passage is made explicit by the action *TIME*, and that the fairness constraints are translated into timing restrictions. However, we will see in Section 3.7 that fairness restrictions are still needed for a timed I/O automaton. To avoid comparing different types of timed I/O automata, all specifications are in the format of the *fair timed I/O automaton*. Brief introductions to ordinary and fair timed I/O automata are given in Appendices A.3 and A.4.

Data types We reuse the ingredients of Σ_2 and $\mathcal{A}_2$, given in section 3.4.1, and add the data type **Time** to obtain a typed signature Σ_4 and a Σ_4-algebra $\mathcal{A}_4$. **Time** is the set R^+ of nonnegative real numbers, with the usual interpretation and functions for addition (+) and multiplication (.).

We now present the fair timed I/O automaton *Lossy*, which models a lossy RPC component. It has a new state variable $clock_P$ for each calling process, to keep track of the time elapsed since the last call was received from the sender, or issued to the receiver. Its behaviour is almost equal to that of the fair I/O automaton *RPC* component, except that the output action *RPC_FAILURE* is replaced by an internal action *LOSE*. After each $LOSE_P$ action, *Lossy* is ready for a new call from the sender for process P. Also a time-passing action *TIME* is added. We let time pass without bounds, except in states where a certain output action should be issued within δ seconds. Here we forbid time passing if it violates this bound. Clocks are only updated in states where their value is actually used. Since all fairness restrictions have been replaced by timing constraints, both *wfair*(*Lossy*) and *sfair*(*Lossy*) are empty.

The code for *Lossy* is given in Figure 3.6. Boxes highlight the places where the code differs from the code for *RPC*.

3.7 Specifications and verifications for Problem 5

To model an implementation as specified, we need more than the specification of *Lossy*. There has to be some sort of clerk component, that signals the need for a failure output action and issues this failure.

Suppose *Lossy* performs a $LOSE_P$ action. Now, $Lossy.pc_P = \mathsf{WC}$, and new calls from the sender for process P can be handled. However, the clerk will wait until the $2\delta + \epsilon$ bound is reached to issue the necessary $RPC_FAILURE_P$. Before this $RPC_FAILURE_P$, all calls for process P should be discarded. So the clerk must also monitor the calls from the sender to *Lossy*.

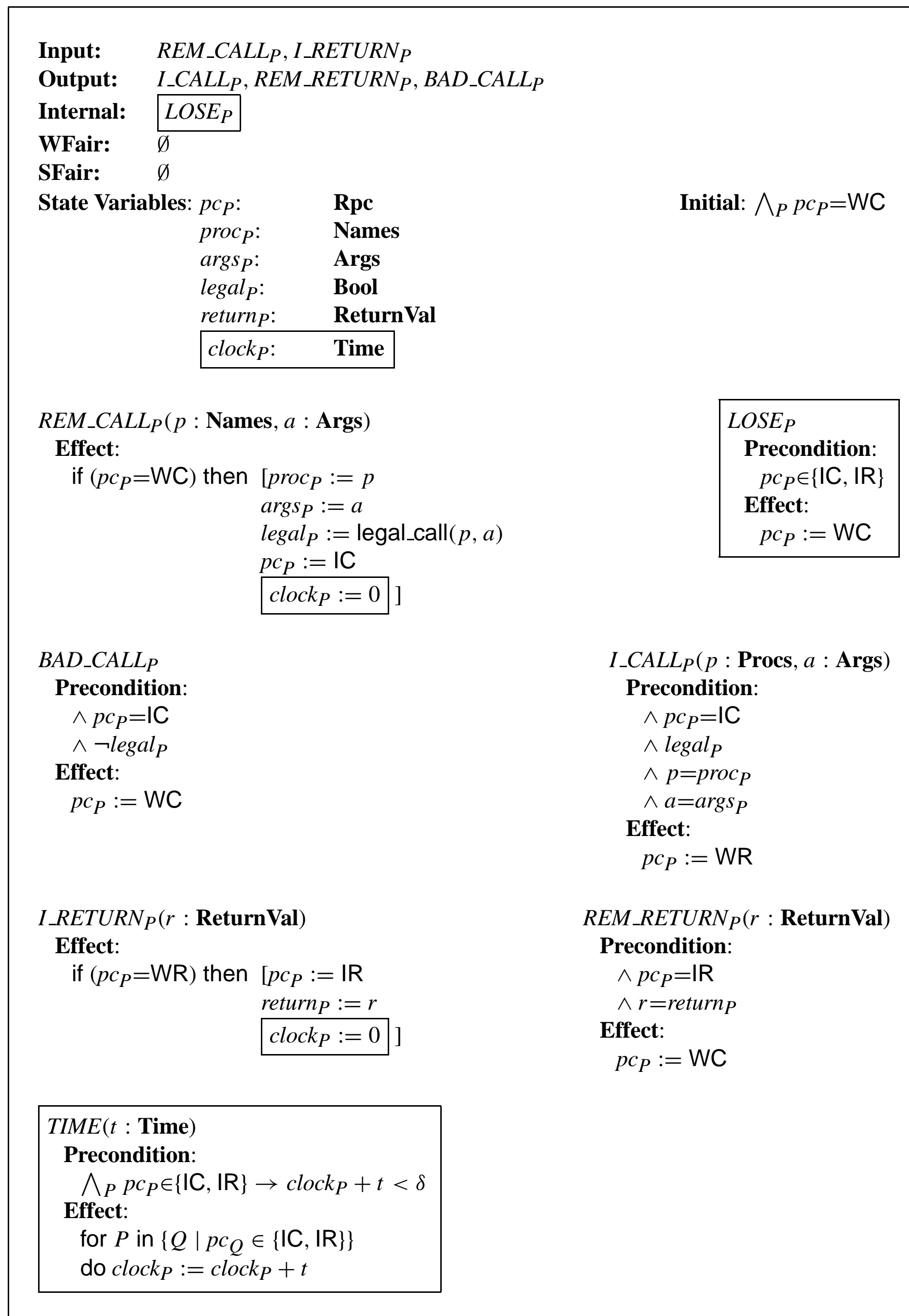


Figure 3.6: I/O automaton *Lossy*

3.7.1 Problem 5(a): The RPC implementation *RPCImp*

Data types We reuse the ingredients of Σ_4 and $\mathcal{A}_4$, given in Section 3.6.1, and add the data type **Cpc** to obtain a typed signature Σ_5 and Σ_5-algebra $\mathcal{A}_5$. **Cpc** contains the constants WC, IC and WR. Note that the domain of **Cpc** is included in the domain of **Rpc**.

Specification We will now present the fair timed I/O automaton *ClerkL*, which models a clerk for the lossy RPC component *Lossy*.

ClerkL catches each *REM_CALL* from the sender. If it is ready for an incoming call, then this call is forwarded with *PASS* to *Lossy*. At the moment of forwarding, a clock is started to check the response time of *Lossy*. If *ClerkL* is busy, incoming calls are discarded. *ClerkL* signals all output actions from *Lossy*, to ensure that, whenever the $2\delta + \epsilon$ bound is reached, the corresponding $RPC_FAILURE_P$ is really needed. To signal the $2\delta + \epsilon$ bound, *ClerkL* has a clock for each process P. Time action may pass without bounds, except when the clerk is waiting for a return from *Lossy*, or when an incoming call should be forwarded. In the last case, the bound ζ is used, which has no other purpose than to ensure that the call to *Lossy* is forwarded within some time limit. The delay of forwarding the call is not added to the response time of *Lossy*.

The code for *ClerkL* is listed in Figure 3.7.

The composition Since *REM_CALL* is an input action for both *Lossy* and *ClerkL*, but should only be received by the latter, we need to rename *Lossy*:

$$Lossy' \triangleq \mathsf{rename}(Lossy)$$

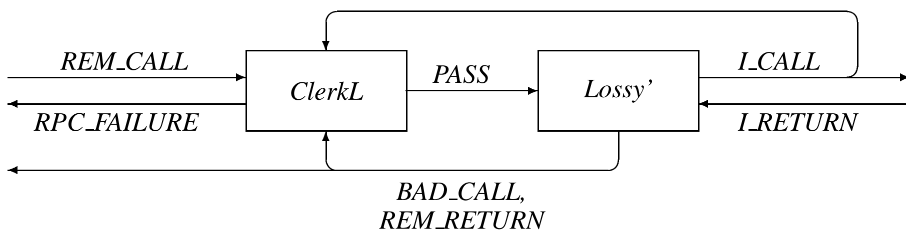
$$\text{where} \quad \begin{array}{lcll} \mathsf{rename}(REM_CALL_P(p,a)) & = & PASS_P(p,a) & \\ \mathsf{rename}(x) & = & x & \text{otherwise} \end{array}$$

Note that the output actions *BAD_CALL*, *I_CALL* and *REM_RETURN* should be received by both *ClerkL* and the environment. The only output action which must be hidden is *PASS*.

The implementation *RPCImp* is defined as follows:

$$RPCImp \quad \triangleq \quad \mathsf{HIDE}\ (\bigcup_P \{PASS_P\})\ \mathsf{IN}\ (ClerkL \| Lossy')$$

The behaviour of *RPCImp* is illustrated in the following figure.



Input: REM_CALL_P, BAD_CALL_P, I_CALL_P, REM_RETURN_P
Output: $PASS_P$, $RPC_FAILURE_P$
WFair: ∅
SFair: ∅
State Variables: pc_P: **Cpc**; $proc_P$: **Names**; $args_P$: **Args**; $call_P$: **Bool**; $clock_P$: **Time**
Initial: $\bigwedge_P pc_P{=}\mathsf{WC}$

$REM_CALL_P(p:$ **Names**, $a:$ **Args**)
Effect:
if ($pc_P{=}\mathsf{WC}$) then [$pc_P := \mathsf{IC}$
$proc_P := p$
$args_P := a$
$clock_P := 0$]

$PASS_P(p:$ **Names**, $a:$ **Args**)
Precondition:
$\wedge\ pc_P{=}\mathsf{IC}$
$\wedge\ p{=}proc_P$
$\wedge\ a{=}args_P$
Effect:
$pc_P := \mathsf{WR}$
$call_P :=$ false
$clock_P := 0$

BAD_CALL_P
Effect:
$pc_P := \mathsf{WC}$

$I_CALL_P(p:$ **Procs**, $a:$ **Args**)
Effect:
$call_P :=$ true

$REM_RETURN_P(r:$ **ReturnVal**)
Effect:
if ($pc_P{=}\mathsf{WR} \wedge (clock_P < 2\delta{+}\epsilon)$)
then [$pc_P := \mathsf{WC}$]

$RPC_FAILURE_P$
Precondition:
$\wedge\ pc_P{=}\mathsf{WR}$
$\wedge\ clock_P{=}2\delta{+}\epsilon$
Effect:
$pc_P := \mathsf{WC}$

$TIME(t:$ **Time**)
Precondition:
$\bigwedge_P pc_P{=}\mathsf{IC} \rightarrow clock_P{+}t < \zeta$
$\bigwedge_P pc_P{=}\mathsf{WR} \rightarrow clock_P{+}t \leq 2\delta{+}\epsilon$
Effect:
for P in $\{Q \mid pc_Q{\in}\{\mathsf{IC}, \mathsf{WR}\}\}$
do $clock_P := clock_P{+}t$

Figure 3.7: Timed I/O automaton *ClerkL*

Note that in the behaviour of *RPCImp*, the following scenario is included. *ClerkL* issues a $RPC_FAILURE_P$ and *Lossy'* issues a REM_RETURN_P for a call from process P, that is, one call leads to two returns. This situation arises whenever the receiver takes too long before returning a procedure call to *Lossy'*. However, since the specification is only required to implement specification *RPC* under the assumption that each *I_CALL* from *Lossy'* is followed by a *I_RETURN* within ϵ seconds, this situation is excluded from the desired behaviour.

3.7.2 Problem 5(b): *RPCImp* implements *RPC*

At this point we have an implementation *RPCImp* with real-time aspects, and an untimed specification *RPC*. To be able to compare these, we can add time to *RPC* and prove admissible trace inclusion. Since we already specified all components as fair timed I/O automata, we are able to keep the weak and strong fairness sets in *RPC*.

The timed I/O automaton *TimeRPC* combines the code for *RPC* with the action $TIME(t : \mathbf{Time})$. The precondition of *TIME* is true, the effect is empty (no state variables change).

If we could prove that each fair admissible trace of *RPCImp* is in the intersection of the admissible traces and fair timed traces of *TimeRPC*, we would be done. However, we still need to formalise the restriction on the environment, namely that the receiver will return each forwarded procedure-call within ϵ seconds.

Since there is no straightforward way to express this type of restrictions in I/O automata theory, we choose to specify a very general receiver by means of the fair timed I/O automaton *Rec*. *Rec* returns some answer for each call from *Lossy'* within ϵ seconds.
Now, *RPCImp* implements *TimeRPC* if the behaviour of *RPCImp* is included in the fair behaviour of *TimeRPC*, provided that both are communicating with the receiver *Rec*.

The code for *Rec* is listed in Figure 3.8.

A new implementation and specification For the new implementation and specification, we take two copies of *Rec*, and call them *RecLossy* and *RecRPC*.
The composition for the implementation is

$$Imp \triangleq \mathsf{HIDE}\ I\ \mathsf{IN}\ (RPCImp \| RecLossy)$$

where $I \triangleq \bigcup_P \{\mathsf{IC}(p,a), \mathsf{IR}(r) \mid p \text{ in } \mathbf{Names}, a \text{ in } \mathbf{Args}, r \text{ in } \mathbf{ReturnVal}\}$.

The composition for the specification is

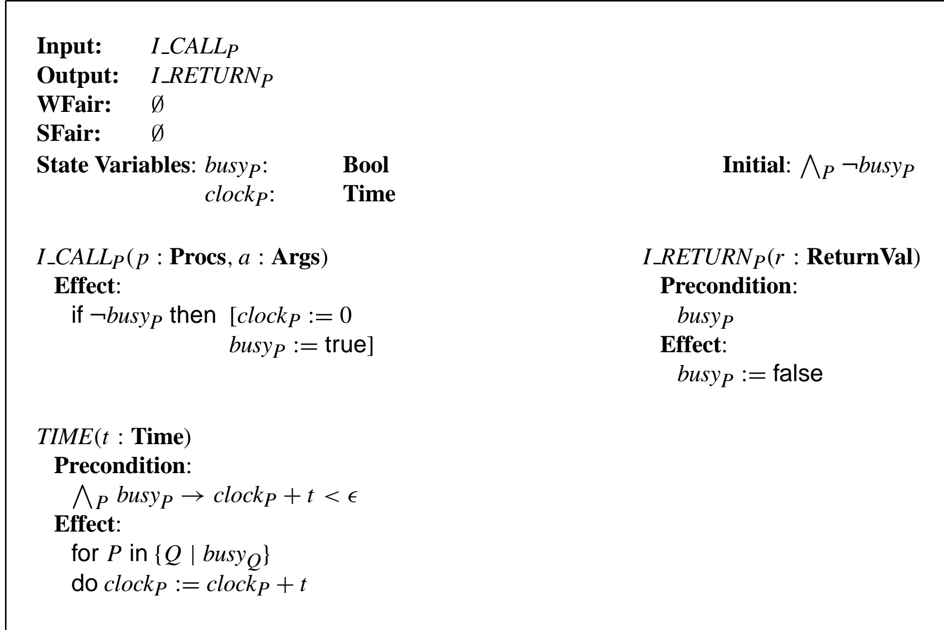
$$Spec \triangleq \mathsf{HIDE}\ I\ \mathsf{IN}\ (TimeRPC \| RecRPC)$$

where $I \triangleq \bigcup_P \{\mathsf{IC}(p,a), \mathsf{IR}(r) \mid p \text{ in } \mathbf{Names}, a \text{ in } \mathbf{Args}, r \text{ in } \mathbf{ReturnVal}\}$.

Note that each discrete action in *Spec* is persistent, and that each admissible execution of *Imp* is fair. Using these two facts, the implementation relation is proved by the inclusion

$$t\text{-}traces(Imp) \subseteq (t\text{-}traces(Spec) \cap fair\text{-}t\text{-}traces(Spec))$$

First we prove $t\text{-}traces(Imp) \subseteq t\text{-}traces(Spec)$, by means of a weak refinement, and then $t\text{-}traces(Imp) \subseteq fair\text{-}t\text{-}traces(Spec)$.



Input: I_CALL_P
Output: I_RETURN_P
WFair: $\emptyset$
SFair: $\emptyset$
State Variables: $busy_P$: **Bool**; $clock_P$: **Time**
Initial: $\bigwedge_P \neg busy_P$

$I_CALL_P(p : \mathbf{Procs}, a : \mathbf{Args})$
Effect:
if $\neg busy_P$ then $[clock_P := 0$
$busy_P :=$ true]

$I_RETURN_P(r : \mathbf{ReturnVal})$
Precondition:
$busy_P$
Effect:
$busy_P :=$ false

$TIME(t : \mathbf{Time})$
Precondition:
$\bigwedge_P busy_P \rightarrow clock_P + t < \epsilon$
Effect:
for P in $\{Q \mid busy_Q\}$
do $clock_P := clock_P + t$

Figure 3.8: Timed I/O automaton *Rec*

In the remainder, we will mostly reason about 'sampling' executions instead of timed executions. Since Lemmas 2.11 - 2.13 in [LV96] state that both induce the same set of timed traces, and we only consider inclusion on sets of traces, this does not make a difference.

Admissible trace inclusion Some invariants are needed to enable the use of a weak timed refinement. The first one states that in the particular states, no clocks violate their bounds.

Lemma 3.44 The following property InvT1 is an invariant of *Imp*:

$$\begin{array}{ll}\bigwedge_P ClerkL.pc_P{=}\mathsf{IC} & \rightarrow (ClerkL.clock_P{<}\zeta) \\ \bigwedge_P ClerkL.pc_P{=}\mathsf{WR} & \rightarrow (ClerkL.clock_P{\leq}2\delta{+}\epsilon) \\ \bigwedge_P Lossy'.pc_P{\in}\{\mathsf{IC},\mathsf{IR}\} & \rightarrow (Lossy'.clock_P{<}\delta) \\ \bigwedge_P Lossy'.pc_P{=}\mathsf{WR} & \rightarrow (RecLossy.clock_P{<}\epsilon)\end{array}$$

The next invariant states that all components synchronise in some way, which is reflected in their program counters and clocks. The formula looks rather complicated, but readability cannot be improved by splitting it into smaller pieces. This is due to the precondition of action *RPC_FAILURE*.

Lemma 3.45 The following property InvT2 is an invariant of *Imp*:

$$\begin{array}{l}\bigwedge_P Lossy'.pc_P{\neq}\mathsf{WC} \rightarrow ClerkL.pc_P{=}\mathsf{WR} \\ \bigwedge_P Lossy'.pc_P{=}\mathsf{WR} \leftrightarrow RecLossy.busy_P \\ \bigwedge_P Lossy'.pc_P{=}\mathsf{IC} \quad \rightarrow (ClerkL.clock_P{=}Lossy'.clock_P) \\ \bigwedge_P Lossy'.pc_P{=}\mathsf{WR} \rightarrow (ClerkL.clock_P{<}RecLossy.clock_P{+}\delta)\end{array}$$

$$\bigwedge_P \mathit{Lossy}'.pc_P{=}\mathsf{IR} \;\rightarrow (\mathit{ClerkL}.clock_P{<}\mathit{Lossy}'.clock_P{+}\delta{+}\epsilon)$$

Whenever *Lossy′* or *ClerkL* is ready to issue a return to the sender, the other of the two is not doing something unexpected.

Corollary 3.46 The following property InvT3 is an invariant of *Imp*:

$$\begin{array}{ll} \bigwedge_P enabled(BAD_CALL_P) & \rightarrow \mathit{ClerkL}.pc_P{=}\mathsf{WR} \\ \bigwedge_P enabled(RPC_FAILURE_P) & \rightarrow \mathit{Lossy}'.pc_P{=}\mathsf{WC} \\ \bigwedge_P enabled(REM_RETURN_P) & \rightarrow \wedge\ \mathit{ClerkL}.pc_P{=}\mathsf{WR} \\ & \quad\;\; \wedge\ (C.clock_P{<}2\delta{+}\epsilon) \end{array}$$

ClerkL records every call from *Lossy′* to *RecLossy* correctly.

Lemma 3.47 The following property InvT4 is an invariant of *Imp*:

$$\begin{array}{ll} \bigwedge_P \mathit{Lossy}'.pc_P{=}\mathsf{IC} & \rightarrow \neg \mathit{ClerkL}.call_P \\ \bigwedge_P \mathit{Lossy}'.pc_P{\in}\{\mathsf{IR}, \mathsf{WR}\} & \rightarrow \mathit{ClerkL}.call_P \end{array}$$

Lossy′ does not unexpectedly change its state variables.

Lemma 3.48 The following property InvT5 is an invariant of *Lossy'*:

$$\bigwedge_P pc_P{\neq}\mathsf{WC} \rightarrow legal_P{=}\mathsf{legal_call}(proc_P, args_P)$$

Lossy′ and *ClerkL* agree on the arguments of the last call.

Lemma 3.49 The following property InvT6 is an invariant of *Imp*:

$$\begin{array}{ll} \bigwedge_P \mathit{Lossy}'.pc_P{\neq}\mathsf{WC} \rightarrow & \wedge\ \mathit{Lossy}'.proc_P{=}\mathit{ClerkL}.proc_P \\ & \wedge\ \mathit{Lossy}'.args_P{=}\mathit{ClerkL}.args_P \end{array}$$

Corollary 3.50 The following property InvT7 is an invariant of *Imp*:

$$\begin{array}{l} \bigwedge_P \mathit{Lossy}'.pc_P{\neq}\mathsf{WC} \\ \quad \rightarrow\ \mathit{Lossy}'.legal_P{=}\mathsf{legal_call}(\mathit{ClerkL}.proc_P, \mathit{ClerkL}.args_P) \end{array}$$

Weak refinement The weak timed refinement does not look very straightforward. This is due to the possibility of *Lossy′* to *LOSE* every now and then. If this happens, the program counter value in *TimeRPC* suddenly relies on the information in *ClerkL*.

Theorem 3.51 The function TREF, which is defined by the identity function on variables with the same name from *RecLossy* to *RecRPC* and by the following formula, is a weak timed refinement from *Imp* to *Spec*, with respect to the reachable states in *Imp* and *Spec*.

$$\begin{array}{lll} \bigwedge_P \mathit{TimeRPC}.pc_P & = & \mathsf{if}\ \ \mathit{Lossy}'.pc_P{\neq}\mathsf{WC} \\ & & \mathsf{then}\ \mathit{Lossy}'.pc_P \\ & & \mathsf{else\ if}\ \ \mathit{ClerkL}.pc_P{\in}\{\mathsf{WC}, \mathsf{IC}\} \\ & & \quad\quad \mathsf{then}\ \mathit{ClerkL}.pc_P \\ & & \quad\quad \mathsf{else\ if}\ \ \mathit{ClerkL}.call_P \\ & & \quad\quad\quad\quad \mathsf{then}\ \mathsf{IR} \\ & & \quad\quad\quad\quad \mathsf{else}\ \mathsf{IC} \end{array}$$

$$\begin{array}{ll}\bigwedge_P \mathit{TimeRPC.proc}_P & = \mathit{ClerkL.proc}_P \\ \bigwedge_P \mathit{TimeRPC.args}_P & = \mathit{ClerkL.args}_P \\ \bigwedge_P \mathit{TimeRPC.legal}_P & = \mathsf{legal_call}(\mathit{ClerkL.proc}_P, \mathit{ClerkL.args}_P) \\ \bigwedge_P \mathit{TimeRPC.return}_P & = \mathit{Lossy}'.\mathit{return}_P\end{array}$$

Proof (Sketch) Using the invariants, this is not too hard. We simply check the requirements in [LV96]. ⊠

Corollary 3.52 *t-traces*(*Imp*) $\subseteq$ *t-traces*(*Spec*)

Proof Directly from Theorem 3.51 in this chapter and Theorem 8.2 in [LV96]. ⊠

Fairness is preserved We prove that each *REM_CALL* to *Imp* leads to a return (*BAD_CALL*, *REM_RETURN* or *RPC_FAILURE*) within bounded time.

Lemma 3.53 Let $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ be an admissible execution of *Imp*.
Then $a_i = \mathit{REM_CALL}_P$ and $s_{i-1} \models pc_P{=}\mathsf{WC}$ implies that there is some j such that $j > i$, $a_j \in \{\mathit{BAD_CALL}_P, \mathit{REM_RETURN}_P, \mathit{RPC_FAILURE}_P\}$, and the sum of time elapsing between s_{i-1} and s_j is bounded.

Proof First we observe that all discrete actions in *Imp* are persistent. This is easily checked by examining the effect of *TIME*, the preconditions of the discrete actions and invariant InvT1.

Suppose $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ is an admissible execution of *Lossy'*,
$a_i = \mathit{REM_CALL}_P$ and $s_{i-1} \models pc_P{=}\mathsf{WC}$.

Clearly, $s_i \models pc_P{=}\mathsf{IC} \wedge clock_P{=}0$. So either s_i enables *TIME*, $\mathit{BAD_CALL}_P$ and LOSE_P or s_i enables *TIME*, $\mathit{I_CALL}_P$ and LOSE_P. By persistency, InvT1, InvT2 and the action types we know that idling after state s_i can only disable *TIME*, but not enable other discrete actions. Since α is admissible, time must pass without bound. So within bounded time, one of the discrete actions mentioned must be performed:

$$\begin{array}{ll}\exists k : & \wedge\ k > i \\ & \wedge\ a_k \in \{\mathit{BAD_CALL}_P, \mathit{I_CALL}_P, \mathit{LOSE}_P\} \\ & \wedge\ \text{the sum of time elapsing between } s_{i-1} \text{ and } s_k \text{ is bounded}\end{array}$$

Take such a k.

1. Suppose $a_k = \mathit{BAD_CALL}_P$. The lemma is fulfilled.

2. Suppose $a_k = \mathit{I_CALL}_P$.
   Then $s_k \models \mathit{RecLossy.busy}_P \wedge \mathit{RecLossy.clock}_P{=}0$, so s_k enables $\mathit{I_RETURN}_P$ and *TIME*. As before, idling after state s_k can only disable *TIME*, but not enable other discrete actions. So within bounded time, $\mathit{I_RETURN}_P$ must be performed:

   $$\begin{array}{ll}\exists l : & \wedge\ l > k \\ & \wedge\ a_l = \mathit{I_RETURN}_P \\ & \wedge\ \text{the sum of time elapsing between } s_k \text{ and } s_l \text{ is bounded}\end{array}$$

   Take such an l.
   We know that $s_l \models \mathit{Lossy}'.pc_P{=}\mathsf{IR} \wedge \mathit{Lossy}'.\mathit{clock}_P{=}0$ and s_l enables $\mathit{REM_RETURN}_P$,

$LOSE_P$ and $TIME$. Again we see that within bounded time, REM_RETURN_P or $LOSE_P$ must be performed:

$$\exists m : \wedge\, m > l$$
$$\wedge\, a_m \in \{REM_RETURN_P, LOSE_P\}$$
$$\wedge \text{ the sum of time elapsing between } s_l \text{ and } s_m \text{ is bounded}$$

Take such an m.

(a) Suppose $a_m = REM_RETURN_P$. The lemma is fulfilled.

(b) Suppose $a_m = LOSE_P$. We know that $m > l > k > i$, so Case 3 applies.

3. Suppose $a_k = LOSE_P$.
Then $s_k \models ClerkL.pc_P{=}\mathsf{WR} \wedge Lossy'.pc_P{=}\mathsf{WC}$. Now s_k enables only $TIME$, but idling is only allowed up to (and not beyond!) the state that enables $RPC_FAILURE_P$. So within bounded time, $RPC_FAILURE_P$ must be performed:

$$\exists l : \wedge\, l > k$$
$$\wedge\, a_l = RPC_FAILURE_P$$
$$\wedge \text{ the sum of time elapsing between } s_k \text{ and } s_l \text{ is bounded}$$

The lemma is fulfilled.

⊠

Theorem 3.54 *t-traces*(*Imp*) $\subseteq$ *fair-t-traces*(*Spec*)

Proof Suppose β is a timed trace of *Imp*, and $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ is an admissible execution of *Imp* such that $t\text{-}trace(\alpha) = \beta$. Using the fact that TREF is a weak timed refinement (Theorem 3.51), we can easily construct an admissible execution α' of *Spec* such that $t\text{-}trace(\alpha') = \beta$. It remains to prove that α' is fair.

Initially, $Lossy'.pc_P{=}\mathsf{WC}$ for each P. Whenever $Lossy'.pc_P{=}\mathsf{WC}$, the first action that changes $Lossy'.pc_P$ must be I_CALL_P. By Lemma 3.53, we know that each occurrence of I_CALL_P is followed within bounded time by a state in which $Lossy'.pc_P = \mathsf{WC}$. Combining this with InvT2 and the fact that α is admissible, we see that for each P, α must contain infinitely many occurrences of states such that both $Lossy'.pc_P{=}\mathsf{WC}$ and $RecLossy.busy_P{=}\mathsf{false}$.

Using the above and the fact that α' is admissible, we see that for each P, α' must contain infinitely many occurrences of states such that $TimeRPC.pc_P{=}\mathsf{WC}$ and $RecRPC.busy_P{=}\mathsf{false}$. Since in such a state no locally controlled actions are enabled for P, α' must be weakly fair. Combining this with the fact that there are no strong fairness sets in *Spec*, we obtain that α' is fair. ⊠

Chapter 4

A two-level approach to automated conformance testing of VHDL designs

Summary

For manufacturers of consumer electronics, conformance testing of embedded software is a vital issue. To improve performance, parts of this software are implemented in hardware, often designed in the Hardware Description Language VHDL. Conformance testing is a time consuming and error-prone process. Thus automating (parts of) this process is essential.
There are many tools for test generation and for VHDL simulation. However, most test generation tools operate on a high level of abstraction and applying the generated tests to a VHDL design is a complicated task. For each specific case one can build a layer of dedicated circuitry and/or software that performs this task. It appears that the ad-hoc nature of this layer forms a bottleneck of the testing process. We propose a *generic* solution for bridging this gap: a generic layer of software dedicated to interface with VHDL implementations. It consists of a number of Von Neumann-like components that can be instantiated for each specific VHDL design.
This chapter reports on the construction of and some initial experiences with a concrete tool environment based on these principles.

4.1 Introduction

As is well-known, the software embedded in consumer electronics is becoming increasingly voluminous and complex. Accordingly, testing the software takes up an increasing part of the product development process – and hence of the costs of products. Therefore, Philips considers automating (parts of) the test process a vital issue.

More and more, manufacturers of consumer electronics do not completely develop the software themselves but import parts from other manufacturers. To guarantee well-functioning and interoperability of these parts, it is essential that they are tested for functional conformance w.r.t. internationally agreed standards. Therefore, testing efforts in this area concentrate on functional conformance testing (see [Hol91, ISO91, Kni93] for testing terminology and methodology).

To optimise performance (in terms of speed or bandwidth), the lower layers of protocol stacks are often implemented directly in hardware. Testing these layers would imply hardware testing. However, Philips is interested in detecting design errors *before* implementation in silicon, which would mean testing hardware *designs* rather than their implementations.

Nowadays, hardware is designed using internationally standardised Hardware Description Languages. Testing a design then is testing a program in the description language at hand. Among the Hardware Description Languages, VHDL [IEE93] is prominent.

There are many tools for test generation on the one hand and VHDL simulation, analysis and synthesis on the other hand. Moreover a lot of effort is put into extending and refining these tools. Ideally, therefore, the testing process could be automated by generating tests with a test generation tool, and then executing these tests using a simulation tool. However, most test generation tools expect behaviour to be modelled in clean-cut events with a high level of abstraction. Applying such tests to a VHDL design whose interface behaviour consists of complex patterns of signals on ports is by no means a trivial task. Now, it is always possible to solve this problem by adding a layer of dedicated circuitry and/or software to bridge the gap between low-level events and high-level events, but it appears that the ad-hoc nature of this dedicated circuitry and software forms one of the bottlenecks of the testing process.

We propose a *generic* solution for bridging the gap between generating tests on the abstract level and executing tests on the simulation level. This makes it possible for each of the two different tasks (test generation and test execution) to be performed at the appropriate level within one test trajectory, with a higher degree of automation. The idea is to build a generic layer of software (written in VHDL), dedicated to interface with VHDL implementations. We call this layer the *test bench*. It consists of a number of components that fulfill various tasks: to offer inputs to interfaces of the implementation, to observe outputs at these interfaces and to supervise the test process. The components are Von Neumann-like in the sense that for each *specific* VHDL design they are loaded with sets of instructions. These sets are compiled from user-supplied mappings between high level and low level events and abstract test cases derived from the specification. In order to be maximally generic, the test bench should accept tests described in a standardised test language. In this way, any tool that complies with this test description language can be used for test generation.

Of course, this test bench will not solve all the problems involved in interpreting abstract tests. But by performing many of the routine (and repetitive) tasks, it enables the tester to concentrate on the specific properties of the interface behaviour of the protocol under test.

This chapter reports on the construction of and some initial experiences with a concrete tool environment based on these principles. This prototype tool environment is called *Phact* and has been developed at Philips Research Laboratories Eindhoven, in cooperation with CWI Amsterdam and the universities of Eindhoven and Nijmegen. It consists of a test generation part and a test execution part. The intermediate language between the two parts is the standardised test description language TTCN (*Tree and Tabular Combined Notation* [ISO91, Part 3]). In the test execution part we find the test bench written in VHDL, with a front-end that accepts TTCN test suites.

In our tool environment, test generation is done by the *KPN Conformance Kit* [BKKW90, KWKK91]. This tool takes as input a specification in the form of an Extended Finite State Machine (EFSM) and generates a TTCN test suite for the specification. The *Leapfrog* tool from [Cad] is used for VHDL simulation.

This chapter is organised as follows. In Section 4.2, we globally describe the tool environment and the testing process it supports. Section 4.3 highlights each important step in the test process. In Section 4.4, we describe our experiences with the use of the environment and discuss its limits. In Section 4.5, we compare our approach with other approaches for analysis of VHDL designs. Finally, Section 4.6 gives a short account of what happened after this research was published.

4.2 Global description of test environment and test process

In this section, we give an overview of the tool environment and the testing process it supports. The next section treats some interesting aspects in more detail. We begin with a short digression on *functional conformance testing*.

Conformance testing aims to check that an implementation conforms to a specification. *Functional* conformance testing only considers the external (input/output) behaviour of the implementation. Often the implementation is given as a *black box* with which one can only interact by offering inputs and observing outputs.

In the theory of functional conformance testing many notions of conformance have been proposed. The differences between these notions arise from (at least) two issues. The first issue is the language in which the specification is described (and the (black box) implementation is assumed to be described). Specifications can be described, e.g., by means of automata, labelled transition systems, or by temporal logic formulas. Secondly, the differences arise from the precise relation between implementation and specification that is required. Typically the different conformance notions differ in the extent to which the external behaviour of the implementation should match the specification.

Thus conformance testing always assumes a specific notion of conformance. However, for most conformance relations, exhaustive testing is infeasible in realistically sized cases: some kind of selection on the total test space is inevitable. So it is generally not possible to fully establish that an implementation conforms to the specification; the selected tests rather aim to show that the implementation approximately conforms to the specification. Conformance then simply means: the resulting test method has detected no errors. An appropriate mixture of theoretical considerations and practical experience should then justify this approach. This holds in particular for the test process supported by our tool environment.

Following ISO methodology [ISO91, Kni93], the conformance test process can be divided in the sequence of steps given in Figure 4.1.

Our prototype tool environment automates the test generation and test execution phases and to a lesser extent the test realisation phase. It expects two inputs: the VHDL code for the Implementation Under Test (henceforth called IUT) and the (abstract, formal) functional specification, in the form of a deterministic Extended Finite State Machine (EFSM). From the EFSM specification abstract test cases are derived. These test cases are translated to the VHDL level and executed on the IUT. The history of the test execution is written to a log file and the analysis phase just consists of inspecting this file and the verdicts it contains.

Note that the EFSM is required to be deterministic. We believe that the restriction to deterministic machines is not a real restriction since we are mostly interested in testing a single deterministic VHDL implementation.

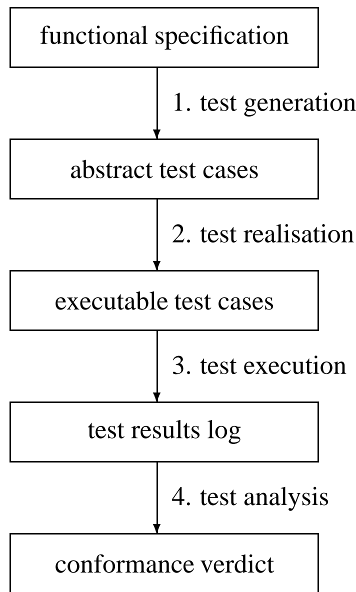


Figure 4.1: Global conformance testing process

The tool environment consists of two parts, taking care of test generation and test execution, respectively. Each one contains an already existing tool. Test generation is done by the *Conformance Kit*, developed by Dutch PTT Research [BKKW90, KWKK91]. When given an EFSM as input, this tool returns a test suite for this EFSM in TTCN notation. The user can to a certain extent determine the parts of the EFSM that are tested and the particular test generation method used. We elaborate on this in Section 4.3.1.

The test cases in the test suite are applied to the IUT by a *test bench*, which is, like the IUT, written in VHDL. The *Leapfrog* tool from [Cad] simulates the application of the test suite to the IUT using the test bench. Thus testing an IUT here means: simulating it together with the test bench. The test bench, which is described in more detail in Section 4.3.3 and in [Sie96], consists of several components connected by a *bus*: the *stimulators*, the *observers*, and the *supervisor* take care of feeding input to the IUT, observing output from the IUT, and coordinating the test cases and handing out the verdict, respectively. The test bench has been designed generically and only needs to be instantiated for each particular IUT.

Compilers connect the test generation part, the output of which is in TTCN notation, to the test execution part, the input of which must be readable for VHDL programs. There are three compilers, one for each type of component of the test bench. The compiler for the supervisor translates the TTCN test suite to an executable format. The compilers for the stimulators and observers map abstract events from the EFSM to patterns of bit vectors at the VHDL level. They require user-supplied translations (comparable to PIXITs in ISO terminology). Section 4.3.2 discusses this in more detail.

Given an IUT written in VHDL and a specification or standard to test against, the global test set-up from Figure 4.1 leads in our setting to the following sequence of steps, also depicted

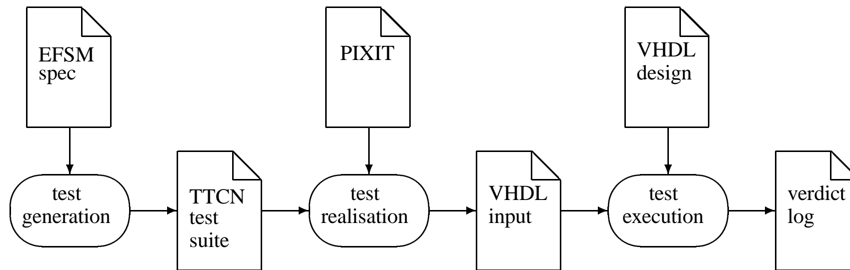


Figure 4.2: Overview of the test trajectory using *Phact*

in Figure 4.2:

0. (Manual) Write an abstract specification EFSM of the IUT.
1. (Automatic) Use the Conformance Kit to derive a test suite for this EFSM, specifying which parts of the EFSM must be tested and what test generation method must be used.
2. (a) (Automatic) Compile the test suite to the executable format for the supervisor.
   (b) (Manual) Define translations between abstract events and patterns of bit vectors (in Figure 4.2 called PIXITs).
   (c) (Automatic) Compile the translations to input files for the stimulator and observer, respectively.
   (d) (Manual) Instantiate the test bench as appropriate for the IUT. That is: enter the number of stimulator/observer pairs, the precise name and location of the compiled translation files, etc.
3. (Automatic) Run the Leapfrog tool on the instantiated test bench together with the IUT.
4. (Manual) Inspect the resulting conformance log file.

We end this section by remarking that the Leapfrog tool also allows the use of the Hardware Description Language *Verilog* [IEE95]. In particular, the Leapfrog can simulate combinations of VHDL and Verilog programs, which makes it possible to plug a Verilog program as IUT into the VHDL test bench.

4.3 Stepwise through the testing process

The following sections explain the consecutive steps in the testing process more thoroughly.

4.3.1 Generating tests with the Conformance Kit

The Conformance Kit consists of a collection of tools for test generation.

The Extended Finite State Machine model supported by the Kit is a slight extension of the traditional Mealy-style FSM model. Transitions are labelled with input/output pairs, where

input and output are treated as simultaneous events (inputs without outputs are allowed). In addition to states and transitions, an EFSM may contain a finite set of variables that range over the booleans or over finite, convex subsets of the integers. Transitions may modify the values of the variables and may be guarded by simple formulas over the variables. There is also the option to mark transitions. For instance, it often happens that certain transitions are added to the EFSM only to make it complete. These transitions are artificial and should not be tested. This is achieved by marking them with a certain marker and excluding all transitions marked thus from the test generation. Finally, it is possible to specify Points of Control and Observation (PCOs) where inputs and outputs occur. They correspond to interfaces of the IUT.

To allow for test generation, the EFSM should be deterministic. Given a deterministic EFSM, one of the tools in the tool set builds a deterministic, trace-equivalent, and minimal FSM (i.e., the FSM exhibits the same external behaviour as the EFSM and contains no pair of distinct but trace-equivalent states). Test generation tools proper take this FSM as input and return a TTCN test suite.

We highlight two of the test generation methods (for more information on test generation methods in general we refer to [FBK+91, Hol91]).

- The *Transition Tour* method. This method yields a finite test sequence (i.e., a sequence of input/output pairs) that performs every transition of the FSM at least once. Thus it checks whether there are no input/output errors.

- The *Partition Tour* method. In addition to the previous method this method also checks for each transition whether the target state is correct. It is similar to the UIO-method [ADLU91, SD88] which in its turn is a variant of the classical W-method [Cho78, Vas73]. Unlike the Transition Tour method, this method yields a number of finite test sequences, one for each transition of the FSM. Each one is a concatenation of the following kinds of sequences:

  – A *synchronising sequence*, that transfers the FSM to its (unique) start state. Theoretically, such a sequence need not always exist. In practice however, most machines have a reset option and hence a synchronising sequence.

  – A *transferring sequence*, that transfers the FSM from the start state to the initial state of the transition to be tested.

  – The input/output pair of the transition.

  – A *Unique Input/Output sequence* (UIO) which verifies that the target state is correct (that is, all other states will show different output behaviour when given the input sequence corresponding to the UIO). If this sequence does not exist it is omitted.

  Although theoretically the fault-coverage of this method is not total, not even when one correctly estimates the number of states of the implementation [CVI89], the counter-examples are academic and we expect that the fault coverage in practice is quite satisfactory.

4.3.2 From abstract tests to executable tests

In the EFSM specification the input and output events of the IUT are described at a very abstract level. For instance, a complicated pattern of input vectors, taking several clock cycles,

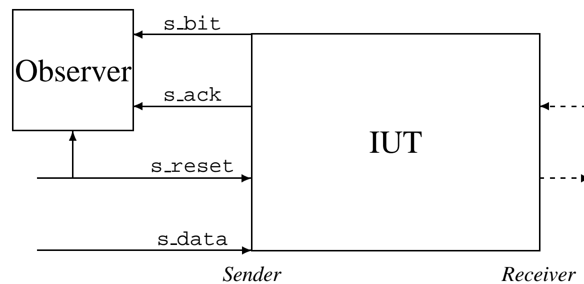


Figure 4.3: An example IUT

may have been abbreviated to a single event `Input_Datum_1`. The abstraction is needed to get a manageable set of meaningful tests. But when one wants to use the TTCN test suite derived from the EFSM to execute tests on the IUT, one has to go back from the abstract level of the EFSM to the concrete level of the VHDL implementation. This translation must be such that the VHDL test bench knows for each abstract event exactly what input should be fed to the IUT or what output from the IUT should be observed. For stimulators, the abstract input events have to be translated to patterns of input bit vectors. For the observers we have to write parser-code to recognise a pattern of output bit vectors as constituting a single abstract output event.

These user-supplied translations may be quite involved and hence sensitive to subtle errors. We expect that in the approach outlined here, this is the part that consumes most of the user's effort.

The translation is constructed in four steps:

1. All abstract events used in the EFSM are grouped per PCO in input and output event groups.

2. All ports of the IUT are grouped into the input or output port group of one interface. Each interface should be associated with exactly one PCO.

3. Each event of an input (output) event group at one PCO is translated to sequences of values of the ports in the input (output) port group at the associated IUT interface. This is done for each interface.

4. All event translations are fed to the compilers that generate code which is understood by the test bench during simulation.

We will give a very simple example of a user-supplied translation that is input for the observer compiler.

The IUT for which the example file is intended is a protocol that transfers data from a Sender to a Receiver and, when successful, sends an acknowledgement back to the Sender. For synchronisation purposes, the acknowledgement is an alternating bit. The IUT has two interfaces (PCOs): *Sender* and *Receiver*. We consider the observer at the *Sender* interface, which should observe acknowledgement events. This situation is depicted in Figure 4.3.

The *Sender* interface has two output ports (which are connected to the input ports of the observer): `s_bit`, through which the alternating bit is delivered, and `s_ack`, through which arrival and presence of an acknowledgement is indicated. Furthermore, the interface has two input ports: `s_data`, a 4 bit wide port through which the *Sender* communicates data to the IUT, and `s_reset`, which has the value `1` whenever the *Sender* resets the IUT.

An acknowledgement event consists of an announcement that an acknowledgement is coming, followed by the acknowledgement itself. The announcement is indicated by the signal at `s_ack` having the value `1`; the value at the `s_bit` port is not yet relevant. Subsequently, the acknowledgement is delivered: port `s_ack` still carries `1`, and port `s_bit` has the value `0` or `1` for the alternating bit.

Now we have all information needed to construct the translation that is input for the observer compiler. The translation code is given in Figure 4.4. Note that the lines preceded with `//` are comments.

First, the translation contains two so-called *qualifiers*, conditions that determine when the parsing of the output of the IUT at this interface should be started or aborted. Parsing should start when an acknowledgement is coming, so the start qualifier uses the value of the `s_ack` port. Parsing should be aborted whenever the IUT is reset, so the abort qualifier uses the value of the `s_reset` port.

Next, the event translation proper is given. Bit masks are defined to recognise individual output bit vectors. In this case the vectors represent two one-bit ports with `s_bit` at the first position and `s_ack` at the second. So mask `ack_coming` has `1` for `s_ack`, and `x` for `s_bit`, indicating that both `11` and `01` match here. Mask `ack_0` only matches when `s_bit` is `0` and `s_ack` is `1`. Output events are defined as regular expressions over the (names for the) bit masks. Here, the arrival of an acknowledgement is recognised by consecutive matching of the two relevant bit masks. This two-phase definition of events reflects the way the observer parses the output from the IUT during execution.

4.3.3 Executing tests at the VHDL level

In order to test the VHDL implementation with the generated tests, we need to execute the VHDL implementation. Executing VHDL code means hardware simulation, for which we use the Cadence Leapfrog tool.

When simulating a VHDL program, which models a reactive system, the program should be surrounded by an environment that behaves – from the program's point of view – exactly like the environment in which the program eventually must operate. This environment should also be able to observe whether the program is operating correctly, and to hand out verdicts reflecting these observations. Finally, since the execution is done by VHDL simulation, the environment itself should be programmed in VHDL too.

Creating the proper environment in VHDL is hard work. However, many tasks remain the same when testing different IUTs. We have therefore created a *generic* VHDL environment, which can easily be instantiated to suit any IUT. The environment we created to perform these tasks is referred to as the *test bench*.

The test bench consists of three kinds of components: a supervisor, some stimulators and some observers. The components communicate with each other by means of a bus. Figure 4.5 shows the structure of the test bench.

Each component type is dedicated to perform its particular task for any IUT. To achieve

```
// Observer bit patterns for the PCO at the Sender side

// Observed ports, with number of bits:
// s_bit(1) s_ack(1)

PCO Sender

QUALIFIERS

// Start parsing output when this qualifier is true
  [(:s_ack = '1')]

// Abort parsing when this qualifier is true
  [(:s_reset = '1')]

MASKS

  ack_coming = 'x1'
  ack_0      = '01'
  ack_1      = '11'

EVENTS

  ACK_OUT_0 = ack_coming ack_0;
  ACK_OUT_1 = ack_coming ack_1;
```

Figure 4.4: Example user-supplied translation for observer

this, each component type has its own instruction set. When plugging an IUT into the test bench, each component is loaded with a sequence of instructions which are specific to the IUT in question. Thus the components can be viewed as small Von Neumann machines.

In the following paragraphs we explain the task of each component type in detail. Thereafter, we describe how the generic test bench is instantiated for testing a certain IUT.

The *supervisor* component has control over the whole test bench. It takes the generated TTCN test suite as input, works its way through each test case and outputs a log file with the verdict and some simulation history. While traversing a test case, it steers the stimulator and observer components and uses a number of timers. Each test case is executed in the following way.

When the current TTCN test case states that input should be provided to the IUT, the supervisor notifies the stimulator at the designated interface. After the stimulator indicates that it has completed this task, the supervisor goes on with the remainder of the test case.

When the TTCN test case states that output should be generated by the IUT, the supervisor checks with the observer at the designated interface to see if this output has been observed. If the output has been observed, the supervisor goes on with the remainder of the test case. If nothing was observed, the supervisor will wait for the observer's notification of new output from the IUT. If output other than the desired output is observed, the TTCN code indicates what action should be taken. The TTCN generated by the Conformance Kit typically hands

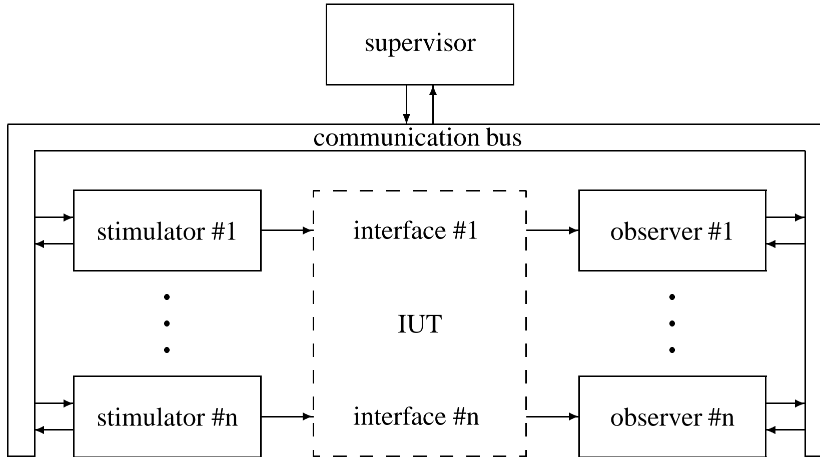


Figure 4.5: Structure of the VHDL test bench

out the verdict *fail* in such a situation.

When the TTCN test case states that a verdict should be handed out, the supervisor logs this verdict to the output file, and quits the current test case.

The other TTCN commands handled by the supervisor are timer commands. TTCN offers the possibility to use timers for testing timing aspects of the behaviour of a system. These timers may be started, stopped and checked for a time-out. At the start of the TTCN test suite, all timers with their respective duration are declared. The supervisor handles these timer instructions in the obvious way. It can instantiate any number of timers with different durations and use them in the prescribed way.

The TTCN produced by the Conformance Kit, however, employs the timer construction in only two ways. It uses one timer for the maximum time a test case should take. This ensures that the test bench will not get stuck in the simulation. A second timer is used to test transitions from the EFSM that have an input event but no output. Since no output event is specified, the IUT should not generate one. This is tested by letting a timer run for some time, during which the IUT should not generate output. Any output observed before the timer expires is considered erroneous and leads to the verdict *fail*. The precise value to which the no-output timer should be set is gleaned from the specification.

The *stimulator* component provides input to the IUT. It waits until the supervisor commands it to start providing a certain abstract event, then drives the input ports of the IUT with the appropriate signals. It has access to the user-defined translation of abstract input events to VHDL input signals.

The *observer* component observes output from the IUT and notifies the supervisor of the abstract events it has observed. Like the stimulator component, it has access to the user-defined translation of VHDL output signals to abstract output events.

Observing the ports of a VHDL component and recognising certain predescribed events is no trivial task. The observer must parse the output of the IUT such that the patterns provided by the user are recognised. Parsing is done with the help of a parser automaton, constructed

with the UNIX tool Lex (and the user-defined translation). The observer uses this automaton to decide which event matches the current output. When the IUT outputs a sequence of values that does not fit into any of the patterns, the supervisor is notified of an error using a special error event.

The supervisor and stimulators communicate directly in a synchronous way – the supervisor always waits for the stimulators to end their activity before resuming its own task – while the supervisor and observers communicate in an asynchronous way via FIFO queues.

In order to plug an arbitrary VHDL implementation into the test bench as the current IUT, some *instantiating* has to take place. The test bench must have as many instantiations of the observer and the stimulator component as the IUT has interfaces. These instantiations must each be connected to the proper interface of the IUT. The IUT may need some external clock inputs, these have to be provided with the correct speed. The supervisor must have the desired number of timers at its disposal, as specified in the TTCN test suite. Each observer (stimulator) must be given access to the compiled version of the user-defined translation. Likewise, the supervisor must be given access to the compiled version of the TTCN test suite.

When these instantiating actions have been performed, the test bench is ready for simulation.

4.4 Experiences

We experimented with our tool environment by running it on a small protocol example. The protocol was derived from the Alternating Bit Protocol [BSW69], with some modifications to test crucial features of the test bench. The features tested mostly concerned the synchronising mechanisms in the test bench.

During the test runs, the VHDL implementation we constructed for the example protocol proved not to conform to its abstract specification. Among other things, the toggling of the alternating bit was not implemented correctly. Already in this small protocol, multiple errors were detected that were subtle enough to escape a manual inspection of the VHDL code.

After conformance was shown for the corrected implementation, we modified the abstract specification EFSM to have discrepancies the other way around. All of these were detected.

Besides this small protocol, we considered a fair-sized, more complex and industrially relevant design. For this we selected a part of the 1394 Serial Bus Protocol, which has been standardised in [IEE96]. The 1394 protocol implements a high speed, low cost bus that can handle communication between video and audio equipment, computers, etc. It supports multimedia applications, allows for "plug-and-play", and provides data transfer rates ranging from 100 Mbit/s to 400 Mbit/s.

The experiments were not carried to completion, because of several problems encountered along the way. We started off with a natural and abstract specification EFSM suggested by the standard document. However, when constructing the translation from abstract events to low-level events, we found that the interface behaviour of the implementation had a very high degree of interleaving of input and output events at different interfaces. In fact, the low-level representation of one abstract event often turned out to be a complete protocol in itself, involving low-level synchronisation schemas and corresponding handshake mechanisms. To enable the test bench to deal with this behaviour, these protocols should be encoded into the stimulator and observer components. Given the simple, generic set-up of the stimulator and observer

components, this appeared to be virtually impossible. This problem was worsened by the fact that the documentation of the protocol and the PIXIT information both lacked the degree of precision required to construct the translation. It remains to be investigated whether the problems encountered with the complicated interface behaviour are specific to the 1394 protocol or occur more frequently and require a refinement or extension of the test bench.

The remainder of this section is devoted to the limits of the test generation method supported by the Conformance Kit.

The EFSM specification format imposes certain restrictions. It has difficulties in modelling, e.g., output events without an input, events occurring simultaneously at multiple interfaces, data parameters of events, and timers. Solutions here require more research in the theory of testing.

Regarding the Conformance Kit itself, it would be convenient if the test generation process could be steered more directly by the user. For instance, one may want to transfer the implementation to a certain interesting state, and perform certain experiments in that state, whereas the Kit moves in a completely autonomous way through the state space.

4.5 Related work

Our tool environment has a modular structure and integrates two well-known techniques: one for automatic generation of TTCN test suites based on finite state machines and the other for the simulation of VHDL hardware designs.

Prior to this research, a number of papers that employ similar techniques for analysing VHDL designs have appeared. From these, only [GFL$^+$96] seems to follow a similar approach to conformance testing. When keeping the phased trajectory from Figure 4.1 in mind, the focus in [GFL$^+$96] is on the test generation phase, the other phases are not described in detail. The method used for test generation is quite different from the classical graph-algorithmic approach such as applied by the Conformance Kit. Model checking techniques are used to derive the tests automatically from an FSM model of either the implementation or the specification. To test a certain transition, a model checking tool is fed with the FSM and a query asserting the non-existence of this transition. The tool derives a counterexample containing the path to the transition. This path is then used as a test sequence. More general temporal formulas can be used to direct the counterexample to check certain situations. Selection of interesting transitions is based on a ranking of state variables, as opposed to the transition marking supported by the Kit (see Section 4.3.1). Although coverage is obtained w.r.t. the ‘interesting’ state variables, there is no measure for coverage w.r.t. exhaustive testing. It seems that theoretic support for dealing with the state explosion problem is as much an issue for this approach, as it is for ours.

In [HYHD95] a tool is described for exhaustive state exploration and simulation of VHDL designs. The VHDL design is transformed into an FSM for which a transition tour is generated (see Section 4.3.1). This tour induces a finite set of finite sequences of bit vectors which together exercise every transition of the VHDL design. As this tool only concerns simulation, there is no notion of conformance w.r.t. a specification, or a mechanism for automatic error detection.

In [WH96] a tool environment is described for the automatic execution of test scripts on VHDL components. There is no support for the automation of test script generation itself.

Finally, there exist many tools for the *verification* of VHDL designs (e.g., [BBDEL96, BJ96a, BBD+96]). Each of them maps VHDL code to some semantical domain, on which the verification algorithms operate. It may be worthwhile to see whether our approach can benefit from techniques used in these tools.

4.6 Later developments

Following up on this research, other projects have worked with the test environment. In 1997, the test environment was used to test an MPEG2 decoder chip in the DIVA project [FMMW98]. In 1998, the test environment was used to test a 64 inch projection TV produced by Philips Consumer Electronics [Hol98a, TLH+99]. Further developments and experiments are still taking place at Philips.

After the research presented here was published, some very similar research was presented in [KVZ98, KVZ99]. In these papers, the test generation tool TGV [FJJV96] is used to generate high-level tests which are translated and executed at the VHDL/Verilog level. The structure of the test environment is based on three component types which seem to have the same tasks as the supervisor, stimulator and observer in our test bench. A difference with our method is that test generation is done on-the-fly, such that the selection of the next test step depends on the results of executing the last test step. The test generation method is based on Lotos [ISO89].

Chapter 5

Exploiting symmetry in protocol testing

Summary

Test generation and execution are often hampered by the large state spaces of the systems involved. In automata (or transition system) based test algorithms, taking advantage of symmetry in the behaviour of specification and implementation may substantially reduce the amount of tests. We present a framework for describing and exploiting symmetries in black box test derivation methods based on finite state machines (FSMs). An algorithm is presented that, for a given symmetry relation on the traces of an FSM, computes a subautomaton that characterises the FSM up to symmetry. This machinery is applied to the classical W-method [Vas73, Cho78] for test derivation. Finally, we focus on symmetries defined in terms of repeating patterns.

5.1 Introduction

It has long been recognised that for the proper functioning of components in open and distributed systems, these components have to be thoroughly tested for interoperability and conformance to internationally agreed standards. For thorough and efficient testing, a high degree of automation of the test process is crucial. Unfortunately, methods for automated test generation and execution are still seriously hampered by the often very large state spaces of the implementations under test. One of the ways to deal with this problem is to exploit structural properties of the implementation under test that can be safely assumed to hold. In this chapter we focus on taking advantage of *symmetry* that is present in the structure of systems. The symmetry, as it is defined here, may be found in any type of parameterised system: such parameters may for example range over IDs of components, ports, or the contents of messages.

We will work in the setting of test theory based on finite state machines (FSMs). Thus, we assume that the specification of an implementation under test is given as an FSM and the implementation itself is given as a black box. From the explicitly given specification automaton a collection of tests is derived that can be applied to the black box. Exploiting symmetry will allow us to restrict the test process to subautomata of specification and implementation that characterise these systems up to symmetry and will often be much smaller. The symmetry is

defined in terms of an equivalence relation over the trace set of the specification. This definition is inspired by symmetry-based reductions in the field of verification [AHI98, CFJ93, EJP97, ES93, ES97, God96, GS97, ID96]. Some requirements are imposed to ensure that such a symmetry indeed allows to find the desired subautomata. We instantiate this general framework by focusing on symmetries defined in terms of repeating *patterns*. Some experiments with pattern-based symmetries, supported by a prototype tool implemented using the OPEN/CÆSAR tool set [Gar98], have shown that substantial savings may be obtained in the number of tests.

Since we assume that the black box system has some symmetrical structure (cf. the *uniformity hypothesis* in [CG97, Gau95]), it is perhaps more appropriate to speak of *gray* box testing. For the specification FSM it will generally be possible to *verify* that a particular relation is a symmetry on the system, but for the black box implementation one has to *assume* that this is the case. The reliability of this assumption is the tester's responsibility. In this respect, one may think of exploiting symmetry as a structured way of test case selection [BTV91, FBK+91] for systems too large to be tested exhaustively, where at least some subautomata are tested thoroughly.

This research is not the first to deal with symmetry in protocol testing. In [MAD96], similar techniques have been developed for a test generation methodology based on labeled transition systems, success trees and canonical testers [Bri88, Tre89]. Like in our case, symmetry is an equivalence relation between traces, and representatives of the equivalence classes are used for test generation. Since our approach and the approach in [MAD96] start from different testing methodologies, it is not easy to compare them. In [MAD96], the symmetry relation is defined through bijective renamings of action labels; our pattern-based definition generalises this approach. On the other hand, since in our case a symmetry relation has to result in subautomata of specification and implementation that characterise these systems up to the symmetry, we have to impose certain requirements, which are absent in [MAD96].

In [KK97], symmetrical structures in the product automaton of interoperating systems are studied. It is assumed that the systems have already been tested in isolation and attention is focused on pruning the product automaton by exploiting symmetry arising from the presence of identical peers. In the present approach, we abstract away from the internal composition of the system and focus on defining a *general* framework for describing and using symmetries on FSMs.

This chapter is organised as follows. Section 5.2 contains some basic definitions concerning FSMs and their behaviour. In Section 5.3, we introduce and define a general notion of trace based symmetry. We show how, given such a symmetry on the behaviour of a system, a subautomaton of the system can be computed, a so-called *kernel*, that characterises the behaviour of the system up to symmetry. In Section 5.5 we apply the machinery to the classical W-method [Cho78, Vas73] for test derivation. In Section 5.6 we will instantiate the general framework by focusing on symmetries defined in terms of repeating patterns. Section 5.7 contains an extensive example, inspired by [TPHT96]. Finally, we discuss future work in Section 5.8.

5.2 Finite state machines

In this section, we will briefly present some terminology concerning finite state machines and their behaviour, that we will need in the rest of this chapter.
We let **Nat** denote the set of natural numbers. (Finite) Sequences are denoted by Greek letters.

Concatenation of sequences is denoted by juxtaposition; ϵ denotes the empty sequence and the sequence containing a single element a is simply denoted a. If σ is non-empty then *first*(σ) returns the first element of σ and *last*(σ) returns the last element of σ.

If V and W are sets of sequences and σ is a sequence, then $\sigma\, W = \{\sigma\,\tau \mid \tau \in W\}$ and $V\, W = \bigcup_{\sigma \in V} \sigma\, W$. For X a set of symbols, we define $X^0 = \{\epsilon\}$ and, for $i > 0$, $X^i = X^{i-1} \cup X\, X^{i-1}$. As usual, $X^* = \bigcup_{i \in \mathbf{Nat}} X^i$.

Definition 5.1 A *finite state machine (FSM)* is a structure $\mathcal{A} = (S, \Sigma, E, s^0)$ where

- S is a finite set of *states*
- Σ a finite set of *actions*
- $E \subseteq S \times \Sigma \times S$ is a finite set of *edges*
- $s^0 \in S$ is the *initial state*

We require that $\mathcal{A}$ is *deterministic*, i.e., for every pair of edges (s, a, s'), (s, a, s'') in E, $s' = s''$. We write $S_{\mathcal{A}}$, $\Sigma_{\mathcal{A}}$, etc., for the components of an FSM $\mathcal{A}$, but often omit subscripts when they are clear from the context. We let s, s' range over states, $a, a', b, c, \ldots$ over actions, and e, e' over edges. If $e = (s, a, s')$ then $e = a$. We write $s \xrightarrow{a} s'$ if $(s, a, s') \in E$ and with $s \xrightarrow{a}$ we denote that $s \xrightarrow{a} s'$ for some state s'. A *subautomaton* of an FSM $\mathcal{A}$ is an FSM $\mathcal{B}$ such that $s^0_{\mathcal{B}} = s^0_{\mathcal{A}}$, $S_{\mathcal{B}} \subseteq S_{\mathcal{A}}$, $\Sigma_{\mathcal{B}} \subseteq \Sigma_{\mathcal{A}}$, and $E_{\mathcal{B}} \subseteq E_{\mathcal{A}}$.

An *execution fragment* of an FSM $\mathcal{A}$ is a (possibly empty) alternating sequence $\gamma = s_0\, a_1\, s_1 \cdots a_n\, s_n$ of states and actions of $\mathcal{A}$, beginning and ending with a state, such that for all i, $0 \leq i < n$, we have $s_i \xrightarrow{a_{i+1}} s_{i+1}$. If $s_0 = s_n$ then γ is a *loop*, if $n \neq 0$ then γ is a *non-empty* loop. An *execution* of $\mathcal{A}$ is an execution fragment that begins with the initial state of $\mathcal{A}$. The *trace* for execution fragment $\gamma = s_0\, a_1\, s_1 \cdots a_n\, s_n$ of $\mathcal{A}$ is defined as *trace*$(\gamma) = a_1\, a_2 \cdots a_n$. If σ is a sequence of actions, then we write $s \xrightarrow{\sigma} s'$ if $\mathcal{A}$ has an execution fragment γ with *first*$(\gamma) = s$, *last*$(\gamma) = s'$, and *trace*$(\gamma) = \sigma$. If γ is a loop, then σ is a *loop-inducing trace*. We write $s \xrightarrow{\sigma}$ if there exists an s' such that $s \xrightarrow{\sigma} s'$, and write *traces*$(s)$ for the set $\{\sigma \in (\Sigma_{\mathcal{A}})^* \mid s \xrightarrow{\sigma}\}$. We write *traces*$(\mathcal{A})$ for *traces*$(s^0_{\mathcal{A}})$.

5.3 Symmetry

In this section we introduce the notion of symmetry.

We want to be able to restrict the test process to subautomata of specification and implementation that characterise these systems up to symmetry. In papers on exploiting symmetry in model checking [AHI98, CFJ93, EJP97, ES93, ES97, God96, GS97, ID96], such subautomata are constructed for explicitly given FSMs by identifying and collapsing symmetrical *states*. We are concerned with black box testing, and, by definition, it is impossible to refer directly to the states of a black box. In traditional FSM based test theory, FSMs are assumed to be deterministic and hence a state of a black box is identified as the unique state of the black box that is reached after a certain trace of the system. Thus it seems natural to define symmetry as a relation over *traces*.

For our basic notion of symmetry on an FSM $\mathcal{A}$, we use an equivalence relation on $(\Sigma_{\mathcal{A}})^*$, such that $\mathcal{A}$ is *closed* under the relation, i.e., if a sequence of actions is related to a trace of $\mathcal{A}$ then the sequence is a trace of $\mathcal{A}$ too.

The idea is to construct from the specification automaton an automaton such that its trace set is included in the trace set of the specification and contains a representative trace for each equivalence class of the equivalence relation on the traces of the specification. In order to be able to do this, we define a symmetry to be the pair consisting of the equivalence relation and a representative-choosing function. We impose some requirements on the symmetry. For the specification we demand (1) that each equivalence class of the symmetry is represented by a unique trace, (2) that prefixes of a trace are represented by prefixes of the representing trace, and (3) that representative traces respect loops. The third requirement means that if a representative trace is a loop-inducing trace, then removing the loop-inducing part also yields a representative trace. This requirement introduces some state-based information in the definition of symmetry.

These requirements enable us to construct a subautomaton of the specification, a so-called *kernel*, such that every trace of the specification is represented by a trace from the kernel. Of course, it will often be the case that the symmetry itself is preserved under prefixes and respects loops, so the requirements will come almost for free.

For the black box implementation, we will, w.r.t. symmetry, only demand that it is closed under symmetry. So if tests have established that the implementation displays certain behaviour, then by assumption it will also display the symmetrical behaviour. In Section 5.5, where the theory is applied to Mealy machines, we will in addition need a way to identify a subautomaton of the implementation that is being covered by the tests derived from the kernel of the specification.

Definition 5.2 A *symmetry* S on an FSM $\mathcal{A}$ is pair $\langle \simeq, ()^r \rangle$ where $\simeq$ is a binary equivalence relation on $(\Sigma_{\mathcal{A}})^*$, and $()^r : (\Sigma_{\mathcal{A}})^* \to (\Sigma_{\mathcal{A}})^*$ is a *representative function* for $\simeq$ such that:

1. $\mathcal{A}$ is closed under $\simeq$: If $\sigma \in \mathit{traces}(\mathcal{A})$ and $\sigma \simeq \tau$, then $\tau \in \mathit{traces}(\mathcal{A})$.
2. Only traces of the same length are related: If $\sigma \simeq \tau$, then $|\sigma| = |\tau|$.
3. $()^r$ satisfies:
   (a) $\sigma^r \simeq \sigma$
   (b) $\tau \simeq \sigma \Rightarrow \tau^r = \sigma^r$
   (c) $()^r$ is prefix closed on $\mathcal{A}$: If $\sigma\, a \in \mathit{traces}(\mathcal{A})$ and $(\sigma\, a)^r = \tau\, b$, then $\sigma^r = \tau$
   (d) $()^r$ is loop respecting on representative traces: If $(\sigma_1\, \sigma_2\, \sigma_3)^r = \sigma_1\, \sigma_2\, \sigma_3 \in \mathit{traces}(\mathcal{A})$ and σ_2 is a loop-inducing trace, then $(\sigma_1\, \sigma_3)^r = \sigma_1\, \sigma_3$.

As mentioned above, we will demand that there exists a symmetry on the specification, while the implementation under test is required only to be closed under the symmetry.

Proposition 5.3 $(\sigma^r)^r = \sigma^r$

Definition 5.4 Let $S = \langle \simeq, ()^r \rangle$ be a symmetry on FSM $\mathcal{A}$. A *kernel* of $\mathcal{A}$ w.r.t. S is a minimal subautomaton $\mathcal{K}$ of $\mathcal{A}$, such that for every $\sigma \in \mathit{traces}(\mathcal{A})$, $\sigma^r \in \mathit{traces}(\mathcal{K})$.

Note that although the kernel is unique since it is minimal, the unicity is relative to the representative function.

5.4 Construction of a kernel

In this section, we fix an FSM $\mathcal{A}$ and a symmetry $S = \langle \simeq, ()^r \rangle$ on $\mathcal{A}$. Figure 5.1 presents an algorithm that constructs a kernel of $\mathcal{A}$ w.r.t. S. It basically explores the state space of $\mathcal{A}$, while keeping in mind the trace that leads to the currently visited state. As soon as such a trace contains a loop, the algorithm will not explore it any further.

In Figure 5.1, $enabled(s, \mathcal{A})$ denotes the set of actions a such that $E_{\mathcal{A}}$ contains an edge (s, a, s'), and for such an a, $eff(s, a, \mathcal{A})$ denotes s'. Furthermore, $repr(\sigma, E)$ denotes the set F of actions such that $a \in F$ iff there exists an action $b \in E$ such that $\sigma^r\, a = (\sigma\, b)^r$, that is, all candidate actions that extend σ^r to a greater representative trace. We will only call this function for σ such that $\sigma^r = \sigma$ (see Lemma 5.7). By definition of $()^r$, for some action c, $(\sigma\, b)^r = \sigma^r\, c = \sigma\, c$. So, since $\mathcal{A}$ is deterministic and closed under $\simeq$, $F \subseteq E$ and if E is non-empty, F is non-empty. This justifies the function $choose(F)$ which nondeterministically chooses an element from F. In the algorithm, the global variable $\mathcal{K}$ is the growing state space which is returned at the end of the algorithm, and updated during the execution of the procedure Build_It. The local variable F for Build_It is not significant for the execution of the algorithm but is useful for proving correctness.

The remainder of this section is devoted to the correctness of algorithm Kernel. In order to prove that the algorithm works properly, we first prove that it terminates, that it creates a subautomaton of $\mathcal{A}$ and that Build_It uses its parameters properly.

Lemma 5.5 The execution of the algorithm Kernel($\mathcal{A}$, S) terminates.

Proof The number of states in $\mathcal{A}$ is finite, and for each nested call of Build_It(s', σ', $Seen'$) within Build_It(s, σ, $Seen$), $Seen' = Seen \cup \{s'\}$ with $s' \notin Seen$. So there can be only finitely many levels of such nested calls. Furthermore, the number of enabled transitions in s is finite, so the while loop that empties E (E decreases strictly monotonically during this loop until it's empty) can make finitely many nested calls to Build_It. ⊠

Lemma 5.6 During execution of Build_It($s^0_{\mathcal{A}}$,ϵ,$\emptyset$), automaton $\mathcal{K}$ is a subautomaton of $\mathcal{A}$, and $\mathcal{K}$ grows monotonically.

Proof Obvious from the algorithm. ⊠

The following lemma concerns the value of the variable $\mathcal{K}$ at the moment that the call to Build_It is made.

Lemma 5.7 When Kernel($\mathcal{A}$,S) during its execution calls Build_It(s,σ,$Seen$), then at that moment $s^0_{\mathcal{K}} \xrightarrow{\sigma}_{\mathcal{K}} s$ and $\sigma^r = \sigma$.

Proof By induction on the length n of σ.

- $n = 0$. Then $\sigma = \epsilon$. From observing the algorithm Kernel and procedure Build_It, it is clear that the only call of Build_It(s,ϵ,$Seen$) is with $s = s^0_{\mathcal{A}}$ and $Seen = \emptyset$. In the initialisation of $\mathcal{K}$, $s^0_{\mathcal{K}}$ has been defined equal to $s^0_{\mathcal{A}}$. As $s^0_{\mathcal{K}} \xrightarrow{\epsilon}_{\mathcal{K}} s$ and $(\epsilon)^r = \epsilon$, the result follows.

- $n = m + 1$.

```
 1 function Kernel(𝒜, S): FSM;
 2 var 𝒦: FSM;
 3
 4    procedure Build_It(s, σ, Seen);
 5    var a, b, s′, E, F;
 6    begin
 7       if s ∉ Seen then
 8          E := enabled(s, 𝒜);
 9          F := ∅;
10          while E ≠ ∅ do
11             a := choose(repr(σ, E));
12             s′ := eff(s, a, 𝒜);
13             S_𝒦 := S_𝒦 ∪ {s′};
14             Σ_𝒦 := Σ_𝒦 ∪ {a};
15             E_𝒦 := E_𝒦 ∪ {(s, a, s′)};
16             Build_It(s′, σ a, Seen ∪ {s});
17             F := F ∪ {a};
18             for each b ∈ E . σ a ≃ σ b do
19                E := E \ {b};
20             od;
21          od;
22       fi;
23    end;
24
25 begin
26    s⁰_𝒦 := s⁰_𝒜;
27    S_𝒦 := {s⁰_𝒜};
28    Σ_𝒦 := ∅;
29    E_𝒦 := ∅;
30    Build_It(s⁰_𝒜, ϵ, ∅);
31    return 𝒦;
32 end.
```

Figure 5.1: The algorithm Kernel

Suppose $\sigma = \sigma' a$ is a trace of length $m + 1$ and Kernel($\mathcal{A}$,S) calls Build_It(s,σ, *Seen*). Since $\sigma \neq \epsilon$, the call Build_It(s,σ,*Seen*) must occur within the execution of a call Build_It(s',σ',*Seen*$'$). By the induction hypothesis, we know that $s^0_{\mathcal{K}} \xrightarrow{\sigma'}_{\mathcal{K}} s'$. When Build_It($s'$,$\sigma'$,*Seen*$'$) calls Build_It($s$,$\sigma' a$,*Seen*) then (s', a, s) has just been added to $E_{\mathcal{K}}$, with a from *enabled*$(s, \mathcal{A})$ and $s = \mathit{eff}(s', a, \mathcal{A})$. So $s' \xrightarrow{a}_{\mathcal{K}} s$ when the call Build_It(s,σ,*Seen*) is made, and it follows that $\sigma \in \mathit{traces}(\mathcal{K})$.

As to $\sigma^r = \sigma$. When Build_It(s',σ',*Seen*$'$) calls Build_It(s,$\sigma' a$,*Seen*) then by definition of $\mathit{choose}(\mathit{repr}(\sigma', E))$, $(\sigma')^r a = (\sigma' a)^r$. Since, by induction hypothesis, $(\sigma')^r = \sigma'$, $(\sigma' a)^r = \sigma' a$, which completes the proof.

⊠

Lemma 5.8 If Kernel($\mathcal{A}$,S)=$\mathcal{K}$ and during its execution has called Build_It(s,σ,*Seen*), then $s^0_{\mathcal{K}} \xrightarrow{\sigma}_{\mathcal{K}} s$ and $\sigma^r = \sigma$.

Proof Follows immediately from Lemmas 5.6 and 5.7. ⊠

Lemma 5.9 If Kernel($\mathcal{A}$,S)=$\mathcal{K}$ and $s \xrightarrow{a}_{\mathcal{K}} t$ then there is a σ such that $s^0_{\mathcal{K}} \xrightarrow{\sigma}_{\mathcal{K}} s$ and $(\sigma a)^r = \sigma a$.

Proof If $s \xrightarrow{a}_{\mathcal{K}} t$, then we know by Lemma 5.6 and by the fact that execution starts with $\mathcal{K}$ empty, that the transition (s, a, t) has been added to $\mathcal{K}$ by execution of line 15 of algorithm Kernel. This happens during the execution of the call Build_It(s,σ,*Seen*) for some σ and some *Seen*, so by Lemma 5.8 we may conclude that upon completion, $s^0_{\mathcal{K}} \xrightarrow{\sigma}_{\mathcal{K}} s$ and $\sigma^r = \sigma$. By the definition of a during execution of the call Build_It(s,σ,*Seen*) at line 11, we see that $(\sigma a)^r = \sigma a$. ⊠

Lemma 5.10 When Kernel($\mathcal{A}$,S) calls Build_It(s,σ,*Seen*), then during the execution of Build_It the following holds:

1. at termination of the while loop, the following property holds:

$$a \in F \Rightarrow \text{Kernel}(\mathcal{A}, S) \text{ has called Build_It}(\mathit{eff}(s, a, \mathcal{K}), \sigma\, a, \mathit{Seen} \cup \{s\})$$

2. at termination of the while loop, the following property holds:

$$s \xrightarrow{b}_{\mathcal{A}} \Rightarrow \exists a \in F.\, \sigma\, a = (\sigma\, b)^r$$

Proof

1. When the while loop is started, F is empty. The only statement that adds a to F is at line 17, which is executed after lines 12 through 16 have been executed, hence the edge (s, a, s') has been added to $E_{\mathcal{K}}$ and Build_It(s', σa, *Seen* $\cup \{s\}$) has been called.

2. At the start of the while loop, $E = \mathit{enabled}(s, \mathcal{A},)$, so $b \in E$ at the start of the while loop. At termination of the while loop, E is empty. Actions are never added to E, only removed from E at line 19. So during the execution of the while loop, certainly $E \subseteq \mathit{enabled}(s, \mathcal{A},)$. We observe that during the execution of the while loop for each a, if $a \in$

$repr(\sigma, E)$, then $a \in repr(\sigma, enabled(s, \mathcal{A},))$. At the moment that b is removed from E (at line 19), the condition that $\sigma\, a \simeq \sigma\, b$ holds. Since a was defined at line 11 and has not changed since, and by our observation, it holds that $a \in repr(\sigma, enabled(s, \mathcal{A},))$. So there is a $c \in enabled(s, \mathcal{A},)$ such that $\sigma\, a = (\sigma\, c)^r$. Since $\sigma\, a \simeq \sigma\, b$, and by definition of representative, $\sigma\, b \simeq \sigma\, c$. By uniqueness of representative, $\sigma\, a = (\sigma\, b)^r$, and the result follows.

⊠

Lemma 5.11 If Kernel($\mathcal{A}$,S) during execution calls Build_It(s,σ,$Seen$) with $\sigma = a_1 a_2 \ldots a_n$, $s_0 \xrightarrow{a_1}_{\mathcal{A}} s_1 \xrightarrow{a_2}_{\mathcal{A}} s_2 \ldots \xrightarrow{a_n}_{\mathcal{A}} s_n$, and $s_0 = s^0_{\mathcal{A}}$, then

1. Kernel($\mathcal{A}$,S) calls Build_It(s_0,ϵ,$\emptyset$)
2. for $0 \le i < n$, Build_It(s_i,σ_i,$Seen_i$) calls Build_It(s_{i+1},σ_{i+1},$Seen_{i+1}$) with $\sigma_i = a_1 a_2 \ldots a_i$ and for $0 \le i \le n$, $Seen_i = \bigcup_{j \in \{0,1,\ldots,i-1\}} \{s_j\}$
3. $s = s_n$ and $Seen = Seen_n$

Proof By induction on the length n of σ.

- $n = 0$. Then $\sigma = \epsilon$, and the result follows immediately.
- $n = m + 1$.

  Suppose $\sigma = a_1 a_2 \ldots a_{m+1}$ and Kernel($\mathcal{A}$,S) calls Build_It(s,σ,$Seen$) with $s_0 \xrightarrow{a_1}_{\mathcal{A}} s_1 \xrightarrow{a_2}_{\mathcal{A}} s_2 \ldots \xrightarrow{a_{m+1}}_{\mathcal{A}} s_{m+1}$ and $s_0 = s^0_{\mathcal{A}}$. Let $\sigma' = a_1 a_2 \ldots a_m$. Since $\sigma \ne \epsilon$, the call Build_It(s,σ,$Seen$) must occur within the execution of a call Build_It(s',σ',$Seen'$) and $Seen = Seen' \cup \{s'\}$. By the induction hypothesis, we know that $Seen' = Seen_m$, that $s' = s_m$, that Kernel($\mathcal{A}$,S) calls Build_It(s_0,ϵ,$\emptyset$), that for $0 \le i < m$, Build_It(s_i,σ_i,$Seen_i$) calls Build_It(s_{i+1},σ_{i+1},$Seen_{i+1}$) with $\sigma_i = a_1 a_2 \ldots a_i$ and that for $0 \le i \le m$, $Seen_i = \bigcup_{j \in \{0,1,\ldots,i-1\}} \{s_j\}$.

  So Build_It(s_m,σ_m,$Seen_m$) calls Build_It(s_{m+1},σ_{m+1},$Seen$), and we need to prove that $s = s_{m+1}$ and $Seen_{m+1} = Seen = \bigcup_{j \in \{0,1,\ldots,m\}} \{s_j\}$. Looking at the statements in Build_It(s_m,σ_m,$Seen_m$) that call Build_It(s_{m+1},σ_{m+1},$Seen$), we see that $s = s_{m+1}$ and $Seen = Seen_m \cup \{s_m\}$. So $Seen = (\bigcup_{j \in \{0,1,\ldots,m-1\}} \{s_j\}) \cup \{s_m\} = \bigcup_{j \in \{0,1,\ldots,m\}} \{s_j\}$ and the result follows.

⊠

Lemma 5.12 If Kernel($\mathcal{A}$,S) during execution calls Build_It(s,σ,$Seen$), then

$$s \in Seen \Leftrightarrow \exists \sigma_1, \sigma_2.\, \sigma = \sigma_1\, \sigma_2 \wedge \sigma_2 \ne \epsilon \wedge s^0_{\mathcal{A}} \xrightarrow{\sigma_1}_{\mathcal{A}} s \xrightarrow{\sigma_2}_{\mathcal{A}} s$$

Proof From Lemmas 5.6 and 5.8 it follows that $\sigma \in traces(\mathcal{A})$. The lemma then follows from Lemma 5.11. ⊠

The next theorem completes the proof of the fact that the algorithm Kernel($\mathcal{A}$,S) returns a kernel for $\mathcal{A}$ w.r.t. S.

Theorem 5.13 Let $\mathcal{K}$ = Kernel($\mathcal{A}$,S).

1. If $\mathcal{K}'$ is a subautomaton of $\mathcal{A}$ such that $\forall\sigma \in \mathit{traces}(\mathcal{A}), \sigma^r \in \mathit{traces}(\mathcal{K}')$, then then $\mathcal{K}$ is a subautomaton of $\mathcal{K}'$.

2. If $\sigma \in \mathit{traces}(\mathcal{A})$, then $\sigma^r \in \mathit{traces}(\mathcal{K})$.

Proof First we prove Item 1. From the algorithm Kernel it is obvious that for each state s in $\mathcal{K}$, either s is the initial state, or there is a transition leading to s. Since $\mathcal{K}$ and $\mathcal{K}'$ are subautomata of $\mathcal{A}$, their initial states are equal. Since $\mathcal{K}$ and $\mathcal{K}'$ are deterministic and representative traces are unique, and since by Lemma 5.9 each transition in $\mathcal{K}$ is part of a representative trace, we see that each transition in $\mathcal{K}$ must be present in $\mathcal{K}'$. Combining all these observations, we see that each state in $\mathcal{K}$ is present in $\mathcal{K}'$. We conclude that $\mathcal{K}$ is a subautomaton of $\mathcal{K}'$.

As to Item 2. Let $\tau = \sigma^r$. Since $\mathcal{A}$ is closed under S, $\tau \in \mathit{traces}(\mathcal{A})$; say that $s^0_{\mathcal{A}} \xrightarrow{\tau}_{\mathcal{A}} t$. We prove a stronger property $\mathit{Inv}(\tau)$ by induction on the length n of σ (= the length of τ).

$\mathit{Inv}(\tau) = \wedge\ \tau \in \mathit{traces}(\mathcal{K})$
$\quad\wedge\ \exists\, \mathit{Seen}.$
$\quad\quad\vee$ Kernel($\mathcal{A}$, S) during execution calls Build_It(t, τ, Seen)
$\quad\quad\vee\ \wedge\ \tau = \tau_1\, \tau_2\, a\, \tau_3$
$\quad\quad\quad\wedge\ s^0_{\mathcal{A}} \xrightarrow{\tau_1}_{\mathcal{A}} t' \xrightarrow{\tau_2 a}_{\mathcal{A}} t' \xrightarrow{\tau_3}_{\mathcal{A}} t$
$\quad\quad\quad\wedge\ \tau_1\, \tau_2$ contains no non-empty loop-inducing trace in $\mathcal{A}$
$\quad\quad\quad\wedge$ Kernel($\mathcal{A}$, S) during execution calls Build_It(t', $\tau_1\, \tau_2\, a$, Seen)

- $n = 0$.

  Then $\sigma = \epsilon$, and also $\tau = \epsilon$. So $t = s^0_{\mathcal{A}}$. Since $s^0_{\mathcal{A}} \in S_{\mathcal{K}}$, $\epsilon \in \mathit{traces}(\mathcal{K})$. It suffices to observe that Kernel($\mathcal{A}$,S) calls Build_It($s^0_{\mathcal{A}}$,ϵ,$\emptyset$).

- $n = m + 1$.

  Induction Hypothesis (IH): $0 \leq |\rho| \leq m \Rightarrow \mathit{Inv}(\rho^r)$

  Suppose $\sigma = \sigma' b$, $\tau = \tau' c$, and $|\sigma| = |\tau| = m + 1$. Since $()^r$ is prefixed closed, $(\sigma')^r = \tau'$. Since $\tau \in \mathit{traces}(\mathcal{A})$, $\tau' \in \mathit{traces}(\mathcal{A})$. We distinguish two cases.

  - τ' does not contain a non-empty loop-inducing trace.

    We show that, for some set Seen, Kernel($\mathcal{A}$, S) calls Build_It(t, $\tau' c$, Seen). By Lemma 5.8 we then know that $\tau' c \in \mathit{traces}(\mathcal{K})$, which proves $\mathit{Inv}(\tau)$.

    Assume $s^0_{\mathcal{A}} \xrightarrow{\tau'}_{\mathcal{A}} t'$. Since $(\sigma')^r = \tau'$, $\mathit{Inv}(\tau')$ holds by IH, so $\tau' \in \mathit{traces}(\mathcal{K})$. There is no loop-inducing trace in τ', and by $\mathit{Inv}(\tau')$, for some Seen', Kernel($\mathcal{A}$, S) calls Build_It(t', τ', Seen'). We now inspect the execution of procedure Build_It for this call. By Lemma 5.12, we know that $t' \notin \mathit{Seen}'$. By Lemma 5.10 we know that upon completion of the while loop Build_It(t'',$\tau' c'$,$\mathit{Seen}' \cup \{t'\}$) has been called, for some state t'' and action c' such that $t' \xrightarrow{c'}_{\mathcal{K}} t''$ and $(\tau' c)^r = \tau' c'$. By Lemma 5.3, we know that $(\tau' c)^r = \tau' c$, so $c' = c$ and hence $t'' = t$. Thus, Build_It(t,$\tau' c$,$\mathit{Seen}' \cup \{t'\}$) has been called.

  - τ' contains a non-empty loop-inducing trace.

    Then there exist $\tau_1, \tau_2, \tau_3, \sigma_1, \sigma_2, \sigma_3, a$, and t' such that

$\wedge \quad \tau = \tau_1\, \tau_2\, a\, \tau_3\, c \wedge \sigma = \sigma_1\, \sigma_2\, \sigma_3$
$\wedge \quad |\tau_1| = |\sigma_1| \wedge |\tau_2\, a| = |\sigma_2| \wedge |\tau_3\, c| = |\sigma_3|$
$\wedge \quad s_0 \xrightarrow{\tau_1}_{\mathcal{A}} t' \xrightarrow{\tau_2 a}_{\mathcal{A}} t' \xrightarrow{\tau_3 c}_{\mathcal{A}} t$
$\wedge \quad \tau_1 \tau_2$ contains no non-empty loop-inducing trace in $\mathcal{A}$

We show that, for some set *Seen*, Kernel($\mathcal{A}$, S) calls Build_It(t', $\tau_1\, \tau_2\, a$, *Seen*), and that $\tau \in traces(\mathcal{K})$. Trivially, $|\tau_1\, \tau_2\, a| < |\tau_1\, \tau_2\, a\, \tau_3\, c|$, and $|\tau_1\, \tau_3\, c| < |\tau_1\, \tau_2\, a\, \tau_3\, c|$. Since $()^r$ is prefix closed and $\tau^r = \tau$, $\tau_1\, \tau_2\, a = (\tau_1\, \tau_2\, a)^r$. Since $()^r$ is loop respecting, $\tau_1\, \tau_3\, c = (\tau_1\, \tau_3\, c)^r$. So we may apply IH and obtain that $Inv(\tau_1\, \tau_2\, a)$ and $Inv(\tau_1\, \tau_3\, c)$ hold. This means that $\tau_1\, \tau_2\, a \in traces(\mathcal{K})$, $\tau_1\, \tau_3\, c \in traces(\mathcal{K})$, and since there is no loop-inducing trace in $\tau_1\, \tau_2$, that, for some set *Seen*, Kernel($\mathcal{A}$, S) calls Build_It(t', $\tau_1\, \tau_2\, a$, *Seen*). Since $\mathcal{K}$ is a subautomaton of $\mathcal{A}$ (Lemma 5.6), we know that $s^0_{\mathcal{K}} \xrightarrow{\tau_1}_{\mathcal{K}} t' \xrightarrow{\tau_2 a}_{\mathcal{K}} t' \xrightarrow{\tau_3 c}_{\mathcal{K}} t$, and hence $\tau_1\, \tau_2\, a \tau_3\, c \in traces(\mathcal{K})$.

⊠

5.5 Test derivation from symmetric Mealy machines

In this section we will apply the machinery developed in the previous sections to Mealy machines. There exists a wealth of test generation algorithms based on the Mealy machine model [ADLU91, Cho78, CVI89, Vas73]. We will show how the classical W-method [Cho78, Vas73] can be adapted to a setting with symmetry. The main idea is that test derivation is not based on the entire specification automaton, but only on a kernel of it. A technical detail here is that we do not require Mealy machines to be minimal (as already observed by [PHK95] for the setting without symmetry). We will use the notation from Chow's paper.

Definition 5.14 A *Mealy machine* is a (deterministic) FSM $\mathcal{A}$ such that

$$\Sigma_{\mathcal{A}} = \{(i/o) \mid i \in I_{\mathcal{A}} \wedge o \in O_{\mathcal{A}}\}$$

where $I_{\mathcal{A}}$ and $O_{\mathcal{A}}$ are two finite and disjoint sets of *inputs* and *outputs*, respectively. We require that $\mathcal{A}$ is *input enabled* and *input deterministic*, i.e., for every state $s \in S_{\mathcal{A}}$ and input $i \in I_{\mathcal{A}}$, there exists precisely one output $o \in O_{\mathcal{A}}$ such that $s \xrightarrow{(i/o)}$.

Input sequences of $\mathcal{A}$ are elements of $(I_{\mathcal{A}})^*$. For ξ an input sequence of $\mathcal{A}$ and $s, s' \in S_{\mathcal{A}}$, we write $s \stackrel{\xi}{\Longrightarrow}_{\mathcal{A}} s'$ if there exists a trace σ such that $s \xrightarrow{\sigma}_{\mathcal{A}} s'$ and ξ is the result of projecting σ onto $I_{\mathcal{A}}$. In this case we write $outcome_{\mathcal{A}}(\xi, s) = \sigma$, and the execution fragment γ with $first(\gamma) = s$ and $trace(\gamma) = \sigma$ is denoted by $exec_{\mathcal{A}}(s, \xi)$. A *distinguishing sequence* for two states s, s' of $\mathcal{A}$ is an input sequence ξ such that $outcome_{\mathcal{A}}(\xi, s) \neq outcome_{\mathcal{A}}(\xi, s')$. We say that ξ distinguishes s from s'.

In Chow's paper, conformance is defined as the existence of an isomorphism between specification and implementation. Since we do not assume automata to be minimal, we will show the existence of a *bisimulation* between specification and implementation. Bisimilarity is a well-known process equivalence from concurrency theory [Mil89]. For minimal automata, bisimilarity is equivalent to isomorphism, while for deterministic automata, bisimilarity is equivalent to equality of trace sets.

Definition 5.15 Let $\mathcal{A}$ and $\mathcal{B}$ be FSMs. A relation $R \subseteq S_{\mathcal{A}} \times S_{\mathcal{B}}$ is a *bisimulation on* $\mathcal{A}$ *and* $\mathcal{B}$ iff

- $R(s_1, s_2)$ and $s_1 \xrightarrow{a}_{\mathcal{A}} s_1'$ implies that there is a $s_2' \in S_{\mathcal{A}}$ such that $s_2 \xrightarrow{a}_{\mathcal{B}} s_2'$ and $R(s_1', s_2')$,
- $R(s_1, s_2)$ and $s_2 \xrightarrow{a}_{\mathcal{B}} s_2'$ implies that there is a $s_1' \in S_{\mathcal{A}}$ such that $s_1 \xrightarrow{a}_{\mathcal{A}} s_1'$ and $R(s_1', s_2')$.

$\mathcal{A}$ and $\mathcal{B}$ are *bisimilar*, notation $\mathcal{A} \leftrightarroweq \mathcal{B}$, if there exists a bisimulation R on $\mathcal{A}$ and $\mathcal{B}$ such that $R(s^0_{\mathcal{A}}, s^0_{\mathcal{B}})$. We call two states $s_1, s_2 \in S_{\mathcal{A}}$ bisimilar, notation $s_1 \leftrightarroweq_{\mathcal{A}} s_2$, if there exists a bisimulation R on $\mathcal{A}$ (and $\mathcal{A}$) such that $R(s_1, s_2)$. The relation $\leftrightarroweq_{\mathcal{A}}$ is an equivalence relation on $S_{\mathcal{A}}$; a *bisimulation class* of $\mathcal{A}$ is an equivalence class of $S_{\mathcal{A}}$ under $\leftrightarroweq_{\mathcal{A}}$.

The main ingredient of Chow's test suite is a *characterising set* for the specification, i.e., a set of input sequences that distinguish inequivalent states by inducing different output behaviour from them. In our case, two states are inequivalent if they are non-bisimilar, i.e. have different trace sets. In the presence of symmetry we will need a characterising set not for the entire specification automaton but only for a kernel of it. However, a kernel need not be input enabled, so two inequivalent states need not have a common input sequence that distinguishes between them. Instead we will use a characterising set that contains for every two states of the kernel that are inequivalent in the original specification automaton, an input sequence that these states have in common in the specification and distinguishes between them.

Constructing distinguishing sequences in the specification automaton rather than in the smaller kernel is of course potentially as expensive as in the setting without symmetry, and may lead to large sequences. However, if the number of states of the kernel is small we will not need many of them, so test *execution* itself may still benefit considerably from the restriction to the kernel. Moreover, we expect that in most cases distinguishing sequences can be found in a subautomaton of the specification that is easily identified and that envelopes the kernel.

Definition 5.16 A *test pair* for a Mealy machine $\mathcal{A}$ is a pair $\langle \mathcal{K}, W \rangle$ where $\mathcal{K}$ is a kernel of $\mathcal{A}$ and W is a set of input sequences of $\mathcal{A}$ such that the following holds. For every pair of states $s, s' \in S_{\mathcal{K}}$ such that $s \not\leftrightarroweq_{\mathcal{A}} s'$, W contains an input sequence ξ such that $outcome_{\mathcal{A}}(\xi, s) \neq outcome_{\mathcal{A}}(\xi, s')$.

The proof that Chow's test suite has complete fault coverage crucially relies on the assumption that (an upper bound to) the number of states of the black box implementation is correctly estimated. Since specification and implementation are also assumed to have the same input sets and to be input enabled, this is equivalent to a correct estimate of the number of states of the implementation that can be reached from the start state by an input sequence from the specification. Similarly, we will assume that we can give an upper bound to the number of states of the black box that are reachable from the start state by an input sequence from the kernel of the specification. We call the subautomaton of the implementation generated by these states the *image* of the kernel.

Technically, the assumption on the state space of the black box is used in [Cho78] to bound the maximum length of distinguishing sequences needed for a characterising set for the implementation. Since, like the kernel, the image of the kernel need not be input enabled, it may be that distinguishing sequences for states of the image cannot be constructed in the image itself. Thus, it is not sufficient to estimate the number of states of the image, but we must in addition

estimate how long the suffix of a distinguishing sequence can be which starts with the first step outside the image of the kernel.

Definition 5.17 Let $\mathcal{A}$ and $\mathcal{B}$ be Mealy machines with the same input set and let $\mathcal{K}$ be a kernel of $\mathcal{A}$. A *$\mathcal{K}$-sequence* is an input sequence ξ such that $s^0_{\mathcal{K}} \stackrel{\xi}{\Longrightarrow}_{\mathcal{K}}$. A state s of $\mathcal{B}$ is called *$\mathcal{K}$-related* if there exists a $\mathcal{K}$-sequence ξ such that $s^0_{\mathcal{B}} \stackrel{\xi}{\Longrightarrow}_{\mathcal{B}} s$.
We define $im_{\mathcal{K}}(\mathcal{B})$ as the subautomaton (S, Σ, E, s^0) of $\mathcal{B}$ defined by:

- $S = \{s \in S_{\mathcal{B}} \mid s \text{ is } \mathcal{K}\text{-related}\}$
- $E = \{(s, a, s') \in E_{\mathcal{B}} \mid s, s' \in S\}$
- $\Sigma = \{a \in \Sigma_{\mathcal{B}} \mid \exists\, s, s'.\ (s, a, s') \in E\}$
- $s^0 = s^0_{\mathcal{B}}$

In the following definition, the parameter n is the upper bound to the length of that part of the distinguishing sequence which steps outside the image of the kernel.

Definition 5.18 A subautomaton $\mathcal{B}$ of a Mealy machine $\mathcal{A}$ is *n-self-contained in $\mathcal{A}$* when the number of bisimulation classes Q of $\mathcal{A}$ such that $Q \cap S_{\mathcal{B}} \neq \emptyset$ is m, and for every pair of states s, s' of $\mathcal{B}$ such that $s \not\simeq_{\mathcal{A}} s'$, there exist input sequences ξ_1, ξ_2 of $\mathcal{A}$ of length at most m, n, respectively, such that $s \stackrel{\xi_1}{\Longrightarrow}_{\mathcal{B}}$, $s' \stackrel{\xi_1}{\Longrightarrow}_{\mathcal{B}}$, and $outcome_{\mathcal{A}}(\xi_1\xi_2, s) \neq outcome_{\mathcal{A}}(\xi_1\xi_2, s')$.

The next lemma is a generalisation of [Cho78]'s Lemma 0.

Lemma 5.19 Let $\mathcal{A}$ and $\mathcal{B}$ be Mealy machines with the same input set I and let $\langle \mathcal{K}, W\rangle$ be a test pair for $\mathcal{A}$. Let $\mathcal{C} = im_{\mathcal{K}}(\mathcal{B})$. Suppose that:

1. The number of bisimulation classes Q of $\mathcal{B}$ such that $Q \cap S_{\mathcal{C}} \neq \emptyset$ is bounded by m_1.
2. $\mathcal{C}$ is m_2-self-contained in $\mathcal{B}$.
3. W distinguishes between n bisimulation classes Q of $\mathcal{B}$ such that $Q \cap S_{\mathcal{C}} \neq \emptyset$.

Then for every two states s and s' of $\mathcal{C}$ such that $s \not\simeq_{\mathcal{B}} s'$, $I^{m_1-n}\, I^{m_2}\, W$ distinguishes s from s'.

Proof By induction on $j \in \{0, \dots, m_1-n\}$ we prove that there exist $j+n$ bisimulation classes Q of $\mathcal{B}$ with $Q \cap S_{\mathcal{C}} \neq \emptyset$ such that $I^j\, I^{m_2}\, W$ distinguishes between them. This proves the result, since, by assumption 2, the number of bisimulation classes Q of $\mathcal{B}$ such that $Q \cap S_{\mathcal{C}} \neq \emptyset$ is bounded by m_1.

- $j = 0$. By assumption 3, W already distinguishes between n bisimulation classes of $\mathcal{B}$ with $Q \cap S_{\mathcal{C}} \neq \emptyset$, so surely $I^{m_2}\, W$ distinguishes at least these n classes.
- $j = k+1$. If $I^k\, I^{m_2}\, W$ already distinguishes between $k+n+1$ bisimulation classes Q of $\mathcal{B}$ such that $Q \cap S_{\mathcal{C}} \neq \emptyset$, we are done. So suppose not. Then there exist two distinct bisimulation classes Q_1 and Q_2 of $\mathcal{B}$ whose intersection with $S_{\mathcal{C}}$ is non-empty, such that $I^k\, I^{m_2}\, W$ does not distinguish Q_1 from Q_2. So there exist states $s_1 \in Q_1 \cap S_{\mathcal{C}}$ and $s_2 \in Q_2 \cap S_{\mathcal{C}}$ of $\mathcal{C}$ such that $s_1 \not\simeq_{\mathcal{B}} s_2$ but $I^k\, I^{m_2}\, W$ does not distinguish s_1 from s_2.

Since $\mathcal{C}$ is m_2-self-contained in $\mathcal{B}$, we can define the smallest number $l \leq m_1$ such that $I^l\, I^{m_2}\, W$ contains an input sequence ξ such that $outcome_{\mathcal{B}}(\xi, s_1) \neq outcome_{\mathcal{B}}(\xi, s_2)$. So there exist states t_1 and t_2 of $\mathcal{C}$ (among the $(l-(k+1))^{th}$ successors of s_1 and s_2, respectively) such that $I^k\, I^{m_2}\, W$ does not distinguish t_1 from t_2 whereas $I^{k+1}\, I^{m_2}\, W$ does distinguish t_1 from t_2. Hence $I^{k+1}\, I^{m_2}\, W$ distinguishes the bisimulation classes of $\mathcal{B}$ to which t_1 and t_2 belong.

⊠

This result allows us to construct a characterising set $Z = I^{m_1-n}\, I^{m_2}\, W$ for the image of the kernel in the implementation. The test suite resulting from the W-method consists of all concatenations of sequences from a *transition cover* P for the specification with sequences from Z.

Definition 5.20 A *transition cover* for the kernel of a Mealy machine $\mathcal{A}$ is a finite collection P of input sequences of $\mathcal{A}$, such that $\epsilon \in P$ and, for all transitions $s \xrightarrow{(i/o)} s'$ of $\mathcal{K}$, P contains input sequences ξ and $\xi\, i$ such that $s^0_{\mathcal{K}} \overset{\xi}{\Longrightarrow}_{\mathcal{K}} s$.

Now follows the main theorem.

Theorem 5.21 Let *Spec* and *Impl* be Mealy machines with the same input set I, and assume $\langle \simeq, ()^r \rangle$ is a symmetry on *Spec* such that *Impl* is closed under $\simeq$. Let $\langle \mathcal{K}, W \rangle$ be a test pair for *Spec*. Write $\mathcal{C} = im_{\mathcal{K}}(Impl)$. Suppose

1. The number of bisimulation classes Q of *Spec* such that $Q \cap S_{\mathcal{K}} \neq \emptyset$ is n.
2. The number of bisimulation classes Q of *Impl* such that $Q \cap S_{\mathcal{C}} \neq \emptyset$ is bounded by m_1.
3. $\mathcal{C}$ is m_2-self-contained in *Impl*.
4. For all $\sigma \in P$ and $\tau \in I^{m_1-n}\, I^{m_2}\, W$

$$outcome_{Spec}(\sigma\, \tau, s^0_{Spec}) = outcome_{Impl}(\sigma\, \tau, s^0_{Impl}) \qquad (\star)$$

Then $Spec \leftrightarrow Impl$.

Proof *Spec* and *Impl* are deterministic, so it suffices to prove $traces(Spec) = traces(Impl)$. Since *Spec* is input enabled and *Impl* is input deterministic, it then suffices to prove that $traces(Spec) \subseteq traces(Impl)$. Using that *Impl* is closed under S, this follows immediately from the first item of the following claim.

Claim For every $\sigma \in traces(Spec)$, with $\sigma^r = \tau$ and $s^0_{\mathcal{K}} \xrightarrow{\tau}_{\mathcal{K}} r$ we have:

1. $\tau \in traces(Impl)$
2. For every $\xi \in P$ such that $s^0_{\mathcal{K}} \xrightarrow{\xi}_{\mathcal{K}} r$: if $s^0_{Impl} \xrightarrow{\tau}_{Impl} u$ and $s^0_{Impl} \overset{\xi}{\Longrightarrow}_{Impl} u'$ then $u \leftrightarrow_{\mathcal{I}} u'$.

where $\mathcal{I}$ abbreviates *Impl*.

Proof of claim Write $Z = I^{m_1-n}\, I^{m_2}\, W$. Note that, by construction of W, W distinguishes between n bisimulation classes of *Spec* whose intersection with $S_{\mathcal{K}}$ is non-empty. So, since

($\star$) holds, W distinguishes between at least n bisimulation classes of *Impl* whose intersection with $S_{\mathcal{C}}$ is non-empty. Thus we can use Lemma 5.19.

The proof of the claim proceeds by induction on the length n of σ.

- $n = 0$. So $\sigma = \epsilon = \tau$. Then certainly $\tau \in traces(Impl)$. As to item 2). Consider an input sequence $\xi \in P$ such that $s^0_{\mathcal{K}} \stackrel{\xi}{\Longrightarrow}_{\mathcal{K}} s^0_{\mathcal{K}}$ and assume $s^0_{Impl} \stackrel{\xi}{\Longrightarrow}_{Impl} u'$. We have to show that $s^0_{Impl} \leftrightarroweq_{\mathcal{I}} u'$.

  Since ξ and ϵ are elements of P and lead in *Spec* to the same state, it follows from ($\star$) that for all $\rho \in Z$, $outcome_{Impl}(\rho, s^0_{Impl}) = outcome_{Impl}(\rho, u')$. Hence, by Lemma 5.19, $s^0_{Impl} \leftrightarroweq_{\mathcal{I}} u'$.

- $n > 0$. Write $\sigma = \sigma'\,(i/o)$. By induction hypothesis $(\sigma')^r = \tau' \in traces(\mathcal{K}) \cap traces(Impl)$. Say that $s^0_{\mathcal{K}} \stackrel{\tau'}{\longrightarrow}_{\mathcal{K}} r'$. Since $\mathcal{K}$ is a kernel of *Spec*, there exists an action (i'/o') such that $(\sigma'\,(i/o))^r = \tau'\,(i'/o')$ and, for some state r, $r' \stackrel{i'/o'}{\longrightarrow}_{\mathcal{K}} r$. Since $r' \in S_{\mathcal{K}}$, there exist input sequences $\xi', \xi'\,i' \in P$ such that $s^0_{\mathcal{K}} \stackrel{\xi'}{\Longrightarrow}_{\mathcal{K}} r'$.

  Let $s^0_{Impl} \stackrel{\tau'}{\longrightarrow}_{Impl} u$ and $s^0_{Impl} \stackrel{\xi'}{\Longrightarrow}_{Impl} u'$. By induction hypothesis, item 3), $u \leftrightarroweq_{\mathcal{I}} u'$. Since $outcome_{Spec}(\xi'\ i', s^0_{Spec}) = outcome_{Impl}(\xi'\ i', s^0_{Impl})$, there exists a (unique) state v' such that $u' \stackrel{i'/o'}{\longrightarrow}_{\mathcal{I}} v'$. Since $u \leftrightarroweq_{\mathcal{I}} u'$, there exists a (unique) state v such that $u \stackrel{i'/o'}{\longrightarrow}_{\mathcal{I}} v$. So $\tau'\,(i'/o') \in traces(Impl)$. Because *Impl* is input deterministic, $v \leftrightarroweq_{\mathcal{I}} v'$.

  Finally, we have to prove, for all $\xi \in P$ such that $s^0_{\mathcal{K}} \stackrel{\xi}{\Longrightarrow}_{\mathcal{K}} r$: for the unique state w such that $s^0_{Impl} \stackrel{\xi}{\Longrightarrow}_{Impl} w$, we have $w \leftrightarroweq_{\mathcal{I}} v$. Consider such a ξ. Since $v' \leftrightarroweq_{\mathcal{I}} v$ it suffices to prove that $w \leftrightarroweq_{\mathcal{I}} v'$. Since $\xi'\,i'$ and ξ are elements of P and lead to the same state in *Spec*, it follows from ($\star$) that, for all $\rho \in Z$, $outcome_{Impl}(\rho, v) = outcome_{Impl}(\rho, w)$. Hence, by Lemma 5.19, $v' \leftrightarroweq_{\mathcal{I}} w$.

⊠

⊠

5.6 Patterns

In this section we describe symmetries based on *patterns*. A pattern is an FSM, together with a set of permutations of its set of actions, so-called *transformations*. The FSM is a *template* for the behaviour of a system, while the transformations indicate how this template may be filled out to obtain symmetric variants that cover the full behaviour of the system.

In [KK97] an interesting example automaton is given for a symmetric protocol, representing the behaviour of two peer hosts that may engage in the ATM call setup procedure. This behaviour is completely symmetric in the identity of the peers. An FSM representation is given in Figure 5.2. Here, !<action>(i) means output of the ATM service to caller i, and ?<action>(i) means input from caller i to the ATM service. So, action ?set_up(1) denotes the

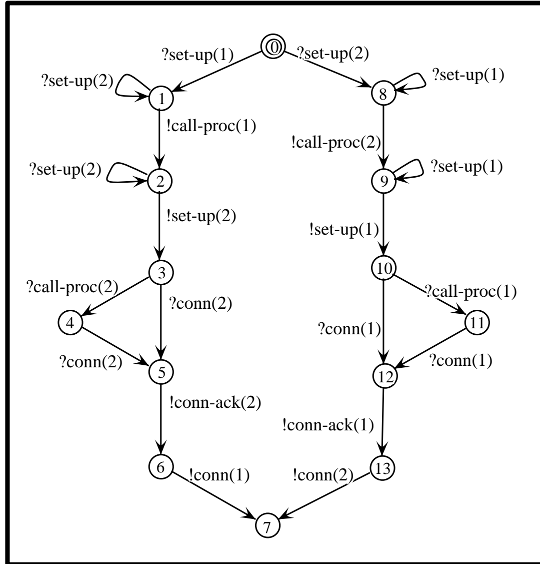


Figure 5.2: The ATM call setup protocol

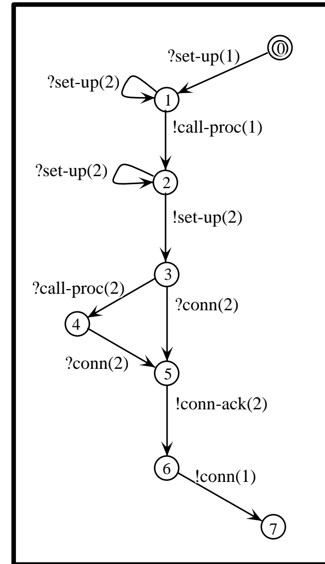


Figure 5.3: A template

request from caller 1 to the ATM service, to set up a call to caller 2. A set_up request is followed by an acknowledgement in the form of call_proc if the service can be performed. Then, action conn indicates that the called side is ready for the connection, which is acknowledged by conn_ack. A caller may skip sending call_proc, if it can already send conn instead (transition from state 3 to 5 and from 10 to 12 in Figure 5.2).

Here, a typical template is the subautomaton representing the call set up as initiated by a single initiator (e.g. caller 1), and the transformation will be the permutation of actions generated by swapping the roles of initiator and responder. Such a template is displayed in Figure 5.3.

In the example of Section 5.7, featuring a *chatbox* that supports multiple conversations between callers, the template will be the chatting between two callers, while the transformations will shuffle the identity of the callers.

The template FSM may be arbitrarily complex; intuitively, increasing complexity indicates a stronger symmetry assumption on the black box implementation.

Definition 5.22 A *pattern* $\mathcal{P}$ is a pair $\langle \mathcal{T}, \Pi \rangle$ where $\mathcal{T}$ is an FSM, called the *template* of $\mathcal{P}$, and Π is a finite set of permutations of $\Sigma_{\mathcal{T}}$, which we call *transformations*.

Given a sequence $\langle f_1, \ldots, f_n \rangle$ of (partial) functions $f_1, \ldots, f_n : \Pi \rightarrow E_{\mathcal{T}}$, we denote with $exec(\langle f_1, \ldots, f_n \rangle, \pi)$ the sequence of edges obtained by taking for each function f_i, $0 \leq i \leq n$, the edge e (if any) such that $f_i(\pi) = e$.

In the example in Figures 5.2 and 5.3, the set of permutations is $\{\{1 \mapsto 1, 2 \mapsto 2\}, \{1 \mapsto 2, 2 \mapsto 1\}\}$.

In the remainder of this section, we fix an FSM $\mathcal{A}$ and a pattern $\mathcal{P} = \langle \mathcal{T}, \Pi \rangle$. Below we will explain how $\mathcal{P}$ defines a symmetry of the behaviour of $\mathcal{A}$. Each transformation $\pi \in \Pi$

gives rise to a copy $\pi(\mathcal{T})$ of $\mathcal{T}$ obtained by renaming the actions according to π. Each such copy is a particular instantiation of the template. Intuitively, the trace set of $\mathcal{A}$ is included in the trace set of the parallel composition of the copies $\pi(\mathcal{T})$, indexed by elements of Π, with enforced synchronisation over all actions of $\mathcal{A}$. Using that traces of $\mathcal{A}$ are traces of the parallel composition, we will define the symmetry relation on traces in terms of the behaviour of the copies and permutations of the index set Π.

The following definition rephrases the intuitive requirement above in such a way that the relation $\simeq$ and a representative function for it can be formulated succinctly. In particular, if $\mathcal{A}$ *is* the parallel composition of the copies of $\mathcal{T}$, both the intuitive requirement and the formal rephrasing apply. In this definition (and the remainder of this section), the following terminology for partial functions and multisets is used. If $f : A \to B$ is a partial function and $a \in A$, then $f(a) \downarrow$ means that $f(a)$ is defined, while $f(a) \uparrow$ means that $f(a)$ is not defined. A *multiset* over A is a set of the form $\{(a_1, n_1), \ldots, (a_k, n_k)\}$ where, for $1 \leq i \leq k$, a_i is an element of A and $n_i \in \mathbf{Nat}$ denotes its *multiplicity*. We use $[f(x)|\, \mathrm{cond}(x)]$ as a shorthand for the multiset over A that is created by adding, for every single $x \in A$, a copy of $f(x)$ if the condition $\mathrm{cond}(x)$ holds.

Definition 5.23 Let $\sigma = a_1 \cdots a_n$ be an element of $(\Sigma_{\mathcal{A}})^*$. A *covering* of σ by $\mathcal{P}$ is a sequence $\langle f_1, \ldots, f_n \rangle$ of partial functions $f_i : \Pi \to E_{\mathcal{T}}$ with *non-empty* domain such that for every $\pi \in \Pi$ and $1 \leq i \leq n$:

1. If $f_i(\pi) = e$ then $a_i = \pi(e)$.
2. The sequence $exec(\langle f_1, \ldots, f_i \rangle, \pi)$ induces an execution γ_i of $\mathcal{T}$.
3. If the sequence $trace(\gamma_{i-1})\, a_i$ is a trace of $\pi(\mathcal{T})$ then $f_i(\pi) \downarrow$.

We say that $\mathcal{P}$ covers σ if there exists a covering of σ by $\mathcal{P}$.

We call $\mathcal{P}$ *loop preserving* when the following holds. Suppose $\sigma_1\, \sigma_2 \in traces(\mathcal{A})$ is covered by $\langle f_1, \ldots, f_n, g_1, \ldots, g_m \rangle$ and σ_2 is a loop-inducing trace. Then for all $\pi \in \Pi$,

$$last(exec(\langle f_1, \ldots, f_n \rangle, \pi)) = last(exec(\langle f_1, \ldots, f_n, g_1, \ldots, g_m \rangle, \pi))$$

Intuitively, these requirements mean the following. The 'non-empty domain' requirement for the partial functions f_i ensures the inclusion of the trace set of $\mathcal{A}$ in the trace set of the parallel composition of copies of $\mathcal{T}$. Requirements 1 and 2 express that a covering should not contain 'junk'. Requirement 3 corresponds to the enforced synchronisation of actions of the parallel composition.

Lemma 5.24 For every trace σ, there exists at most one covering of σ by $\mathcal{P}$.

Proof Since $\mathcal{T}$ is deterministic, coverings of σ are uniquely determined by $\mathcal{T}$. ⊠

Two traces σ and τ of the same length n that are covered by $\mathcal{P}$, are *variants* of each other if at each position i, $1 \leq i \leq n$, of σ and τ the following holds. The listings for σ and τ, respectively, of the copies $\pi(\mathcal{T})$ that participate in the action at position i, the states these copies are in before participating, and the edge they follow by participating, are equal up to a permutation of Π. Then, two traces of the same length are *symmetric* iff they are either both not covered by $\mathcal{P}$ or are covered by coverings that are variants of each other.

Definition 5.25 Let σ and τ be elements of $(\Sigma_{\mathcal{A}})^n$, which $\mathcal{P}$ covers by $cov_1 = \langle f_1, \ldots, f_n \rangle$ and $cov_2 = \langle g_1, \ldots, g_n \rangle$, respectively. Then cov_1 and cov_2 are said to be *variants* of each other if for every $1 \leq i \leq n$, $[f_i(\pi) \mid \pi \in \Pi] = [g_i(\pi) \mid \pi \in \Pi]$.
We define the binary relation $\simeq_{\mathcal{P}}$ on $(\Sigma_{\mathcal{A}})^*$ by:

$$\begin{array}{lll} \sigma \simeq_{\mathcal{P}} \tau & \Leftrightarrow & \wedge \quad |\sigma| = |\tau| \\ & & \wedge \quad \vee \text{ both } \sigma \text{ and } \tau \text{ are not covered by } \mathcal{P} \\ & & \qquad \vee\ \mathcal{P} \text{ covers } \sigma \text{ and } \tau \text{ by variant coverings} \end{array}$$

It is easy to check that $\simeq_{\mathcal{P}}$ is an equivalence relation. As in Section 5.3, we will write $\simeq$ instead of $\simeq_{\mathcal{P}}$.

An important special case is the following. Suppose $\mathcal{A}$ consists of the parallel composition of components C_i, indexed by elements of a set I, that are identical up to their indices (which occur as parameters in the actions). Let σ and τ be traces of $\mathcal{A}$. If there exists a permutation ρ of the index set I such that for all indices $i \in I$, σ induces (up to renaming of indices in actions) the same execution of C_i as τ induces in $C_{\rho(i)}$, then σ and τ are symmetric.

Lemma 5.26 If $\mathcal{P}$ covers σa by $\langle f_1, \ldots, f_n \rangle$, then $\mathcal{P}$ covers σ by $\langle f_1, \ldots, f_{n-1} \rangle$.

Lemma 5.27 If $\mathcal{P}$ covers $\sigma\, a$ and $\tau\, b$ and $\sigma\, a \simeq \tau\, b$, then $\sigma \simeq \tau$.

Proof Let $\sigma\, a$ and $\tau\, b$ be covered by $\langle f_1, \ldots, f_n \rangle$ and $\langle g_1, \ldots, g_n \rangle$, respectively. By Lemma 5.26, these coverings induce the coverings $\langle f_1, \ldots, f_{n-1} \rangle$ and $\langle g_1, \ldots, g_{n-1} \rangle$ of σ and τ, respectively, which are clearly variants of each other. ⊠

The previous two lemmas together imply the following result.

Corollary 5.28 The relation $\simeq$ is *prefix closed* on $\mathcal{A}$, i.e., for every two traces $\sigma\, a,\ \ \tau\, b \in$ *traces*$(\mathcal{A})$, if $\sigma\, a \simeq \tau\, b$ then $\sigma \simeq \tau$.

Given the definition of $\simeq$, it is reasonable to demand that every trace of $\mathcal{A}$ is covered by $\mathcal{P}$. We will also need the following closure property. We call a binary relation R on $(\Sigma_{\mathcal{A}})^*$ *persistent on* $\mathcal{A}$ when $R(\sigma, \tau)$ and $\sigma\, a \in$ *traces*$(\mathcal{A})$ implies that there exists an action b such that $R(\sigma\, a, \tau\, b)$.
Now we define a representative function for $\simeq$. We assume given a total, irreflexive ordering $<$ on $\Sigma_{\mathcal{A}}$. Such an ordering of course always exists, but the choice for $<$ may greatly influence the size of the kernel constructed for a symmetry based on $\mathcal{P}$.

Definition 5.29 Let $<$ be a total, irreflexive ordering on $\Sigma_{\mathcal{A}}$. This ordering induces a reflexive, transitive ordering $\leq$ on traces of the same length in the following way:

$$a\,\sigma \leq b\,\tau \Leftrightarrow a < b \vee (a = b \wedge \sigma \leq \tau)$$

We define σ^r as the least element of $\{\tau | \sigma \simeq \tau\}$ under $\leq$.

We will show that $()^r$ is a representative function for $\simeq$. First we prove that $()^r$ is prefix closed.

Lemma 5.30 Suppose $\simeq$ is persistent on $\mathcal{A}$ and $\mathcal{A}$ is closed under $\simeq$. If $(\tau\, b)^r = \sigma\, a \in$ *traces*$(\mathcal{A})$, then $(\tau)^r = \sigma$.

Proof Suppose that there exists a trace ρ such that $\rho = (\tau)^r$. Note that, since $\mathcal{A}$ is closed under $\simeq$, $\tau\, b \in$ *traces*$(\mathcal{A})$. By persistence of $\simeq$, $\rho \simeq \tau$ implies that there exists an action c

such that $\rho\, c \simeq \tau\, b$. Since $\simeq$ is prefix closed on $\mathcal{A}$ (Corollary 5.28) and $\sigma\, a \simeq \tau\, b$, it follows that $\sigma \simeq \tau$. By definition of $()^r$, $\rho \leq \sigma$. On the other hand, $\sigma a \leq \rho c$, and, by definition of $\leq$, $\sigma \leq \rho$. So $\rho = \sigma$. ⊠

To show that $()^r$ is loop respecting, we first prove two auxiliary results.

Lemma 5.31 If $\mathcal{P}$ covers σ and τ by $\langle f_1, \ldots, f_n \rangle$ and $\langle g_1, \ldots, g_n \rangle$, respectively, and $\sigma \simeq \tau$, then for every $1 \leq i \leq n$:

$$\begin{aligned} &[\mathit{last}(\mathit{exec}(\langle f_1, \ldots, f_i \rangle, \pi)) \mid \pi \in \Pi \wedge f_i(\pi) \downarrow] \\ &\quad = \quad [\mathit{last}(\mathit{exec}(\langle g_1, \ldots, g_i \rangle, \pi)) \mid \pi \in \Pi \wedge g_i(\pi) \downarrow] \end{aligned}$$

Proof Since $\sigma \simeq \tau$ we know that for every $1 \leq i \leq n$, $[f_i(\pi) \mid \pi \in \Pi] = [g_i(\pi) \mid \pi \in \Pi]$. Now the result follows immediately. ⊠

Lemma 5.32 Suppose $\mathcal{P}$ is a loop preserving pattern on $\mathcal{A}$ and let $<$ be a total, irreflexive ordering on $\Sigma_{\mathcal{A}}$. Let $()^r$ be as in Definition 5.29. Suppose every trace of $\mathcal{A}$ is covered by $\mathcal{P}$, $\mathcal{A}$ is closed under $\simeq$, and $\simeq$ is persistent on $\mathcal{A}$. If $\sigma_1\, \sigma_2\, \sigma_3 \in \mathit{traces}(\mathcal{A})$ and σ_2 is a loop-inducing trace, then

$$\sigma_1\, \sigma_3 \simeq \sigma_1\, \tau \text{ iff } \sigma_1\, \sigma_2\, \sigma_3 \simeq \sigma_1\, \sigma_2\, \tau.$$

Proof Write $|\sigma_1| = n$, $|\sigma_2| = m$, and $|\sigma_3| = |\tau| = k$.
Let $\langle f_1, \ldots, f_n, g_1, \ldots, g_m, h_1, \ldots, h_k \rangle$ cover $\sigma_1\, \sigma_2\, \sigma_3$.
By Lemma 5.26, $\langle f_1, \ldots, f_n, g_1, \ldots, g_m \rangle$ covers $\sigma_1\, \sigma_2$ and $\langle f_1, \ldots, f_n \rangle$ covers σ_1. Since $\simeq$ is loop preserving on $\mathcal{A}$, we know that for every $\pi \in \Pi$

$$\mathit{last}(\mathit{exec}(\langle f_1, \ldots, f_n \rangle, \pi)) = \mathit{last}(\mathit{exec}(\langle f_1, \ldots, f_n, g_1, \ldots, g_m \rangle, \pi)) \tag{5.1}$$

So $\langle f_1, \ldots, f_n, h_1, \ldots, h_k \rangle$ covers $\sigma_1\, \sigma_3$.

"⇒" Since $\sigma_1\, \sigma_3 \simeq \sigma_1\, \tau$ and $\sigma_1\, \sigma_3 \in \mathit{traces}(\mathcal{A})$, $\sigma_1\, \tau \in \mathit{traces}(\mathcal{A})$.
Let $\langle f_1, \ldots, f_n, h'_1, \ldots, h'_k \rangle$ cover $\sigma_1\, \tau$.
From Equation 5.1 and the fact that $\langle f_1, \ldots, f_n, g_1, \ldots, g_m \rangle$ covers $\sigma_1\, \sigma_2$, it follows that $\langle f_1, \ldots, f_n, g_1, \ldots, g_m, h'_1, \ldots, h'_k \rangle$ covers $\sigma_1\, \sigma_2\, \tau$. Since $\sigma_1\, \sigma_3 \simeq \sigma_1\, \tau$, we obtain, for every $0 \leq i \leq k$:

$$[h_i(\pi) \mid \pi \in \Pi] = [h'_i(\pi) \mid \pi \in \Pi] \tag{5.2}$$

Now it follows that $\sigma_1\, \sigma_2\, \sigma_3 \simeq \sigma_1\, \sigma_2\, \tau$.

"⇐" Since $\sigma_1\, \sigma_2\, \sigma_3 \simeq \sigma_1\, \sigma_2\, \tau$ and $\sigma_1\, \sigma_2\, \sigma_3 \in \mathit{traces}(\mathcal{A})$, $\sigma_1\, \sigma_2\, \tau \in \mathit{traces}(\mathcal{A})$.
Let $\langle f_1, \ldots, f_n, g_1, \ldots, g_m, h'_1, \ldots, h'_k \rangle$ cover $\sigma_1\, \sigma_2\, \tau$. From Equation 5.1, it follows that $\langle f_1, \ldots, f_n, h'_1, \ldots, h'_k \rangle$ covers $\sigma_1\, \tau$. Since $\sigma_1\, \sigma_2\, \sigma_3 \simeq \sigma_1\, \sigma_2\, \tau$, we obtain, for every $0 \leq i \leq k$:

$$[h_i(\pi) \mid \pi \in \Pi] = [h'_i(\pi) \mid \pi \in \Pi] \tag{5.3}$$

Now it follows that $\sigma_1\, \sigma_3 \simeq \sigma_1\, \tau$.

⊠

Finally, we prove that $()^r$ is loop respecting.

Lemma 5.33 Suppose $\mathcal{P}$ is a loop preserving pattern on $\mathcal{A}$ and let $<$ be a total, irreflexive ordering on $\Sigma_{\mathcal{A}}$. Let $()^r$ be as in Definition 5.29. Suppose every trace of $\mathcal{A}$ is covered by $\mathcal{P}$, $\mathcal{A}$ is closed under $\simeq$, and $\simeq$ is persistent on $\mathcal{A}$. If $(\sigma_1\,\sigma_2\,\sigma_3)^r = \sigma_1\,\sigma_2\,\sigma_3 \in \mathit{traces}(\mathcal{A})$ and σ_2 is a loop-inducing trace, then $(\sigma_1\,\sigma_3)^r = \sigma_1\,\sigma_3$.

Proof By contradiction. Suppose that $(\sigma_1\,\sigma_3)^r = \tau_1\,\tau_3$ and $\tau_1\,\tau_3 \neq \sigma_1\,\sigma_3$. By Lemma 5.30, $(\sigma_1)^r = \sigma_1$, and $\tau_1 = (\sigma_1)^r$, so $\tau_1 = \sigma_1$. By definition of $()^r$, $\sigma_1\,\tau_3 \leq \sigma_1\,\sigma_3$ and $\sigma_1\,\tau_3 \simeq \sigma_1\,\sigma_3$. By Lemma 5.32, $\sigma_1\,\sigma_2\,\sigma_3 \simeq \sigma_1\,\sigma_2\,\tau_3$. Since $\sigma_1\,\sigma_2\,\sigma_3 = (\sigma_1\,\sigma_2\,\sigma_3)^r$, $\sigma_1\,\sigma_2\,\sigma_3 \leq \sigma_1\,\sigma_2\,\tau_3$, and by definition of $\leq$, $\sigma_1\,\sigma_3 \leq \sigma_1\,\tau_3$. Since also $\sigma_1\,\tau_3 \leq \sigma_1\,\sigma_3$, $\sigma_1\tau_3 = \sigma_1\sigma_3$, and we have a contradiction. So $\sigma_1\,\sigma_3 = (\sigma_1\,\sigma_3)^r$. ⊠

The next result allows us to use the pattern-approach for computing a kernel. In our example of the ATM switch, we have computed the kernel from the FSM in Figure 5.2, using the symmetry induced by the template in Figure 5.3 and an ordering $<$ that obeys the relation ?set_up(1) $<$?set_up(2). Not surprisingly, the resulting kernel is identical to the template.

Theorem 5.34 Suppose $\mathcal{P}$ is a loop preserving pattern on $\mathcal{A}$ and let $<$ be a total, irreflexive ordering on $\Sigma_{\mathcal{A}}$. Let $()^r$ be as in Definition 5.29. Suppose every trace of $\mathcal{A}$ is covered by $\mathcal{P}$, $\mathcal{A}$ is closed under $\simeq$, and $\simeq$ is persistent on $\mathcal{A}$. Then $\langle \simeq, ()^r \rangle$ is a symmetry on $\mathcal{A}$.

Proof We have to show that $()^r$ is a representative function for $\simeq$. It is immediate that $\sigma^r \simeq \sigma$ and for all τ such that $\sigma \simeq \tau$, $\tau^r = \sigma^r$. The requirement that $()^r$ is prefix closed follows from Lemma 5.30. That $()^r$ is loop respecting follows from Lemma 5.33. ⊠

The following two lemmas give the justification for making the implementation of the algorithm Kernel from Section 5.4 more efficient. The implementation itself is described in Section 5.7. Lemma 5.36 enables us to stop exploring as soon as state s is visited for trace σ, under the condition that s has been visited already by the algorithm for another trace τ, and σ and τ steer each copy of the template to the same state.

Lemma 5.35 Suppose $\mathcal{P} = \langle \mathcal{T}, \Pi \rangle$ is a pattern on $\mathcal{A}$, that covers σ and τ by $\langle f_1, \ldots, f_n \rangle$ and $\langle g_1, \ldots, g_m \rangle$, respectively.
If $s^0_{\mathcal{A}} \xrightarrow{\sigma}_{\mathcal{A}} s$, $s^0_{\mathcal{A}} \xrightarrow{\tau}_{\mathcal{A}} s$ and for each π in Π: $\mathit{last}(\mathit{exec}(\langle f_1, \ldots, f_n \rangle, \pi)) = \mathit{last}(\mathit{exec}(\langle g_1, \ldots, g_m \rangle, \pi))$, then for each ρ such that $s \xrightarrow{\rho}_{\mathcal{A}}$:

$$\langle f_1, \ldots, f_n, h_1, \ldots, h_k \rangle \text{ covers } \sigma\rho \;\Leftrightarrow\; \langle g_1, \ldots, g_m, h_1, \ldots, h_k \rangle \text{ covers } \tau\rho$$

Lemma 5.36 Suppose $\langle \mathcal{P}, ()^r \rangle$ is a symmetry on $\mathcal{A}$, $()^r$ is as in Definition 5.29, and $\mathcal{P} = \langle \mathcal{T}, \Pi \rangle$ covers σ and τ by $\langle f_1, \ldots, f_n \rangle$ and $\langle g_1, \ldots, g_m \rangle$, respectively.
If $s^0_{\mathcal{A}} \xrightarrow{\sigma}_{\mathcal{A}} s$, $s^0_{\mathcal{A}} \xrightarrow{\tau}_{\mathcal{A}} s$, for each π in Π: $\mathit{last}(\mathit{exec}(\langle f_1, \ldots, f_n \rangle, \pi)) = \mathit{last}(\mathit{exec}(\langle g_1, \ldots, g_m \rangle, \pi))$, and $\sigma = \sigma^r$ and $\tau = \tau^r$, then for each ρ such that $s \xrightarrow{\rho}_{\mathcal{A}}$:

$$\sigma\rho = (\sigma\rho)^r \;\Leftrightarrow\; \tau\rho = (\tau\rho)^r$$

Proof We only prove "⇒", the other direction then follows immediately.

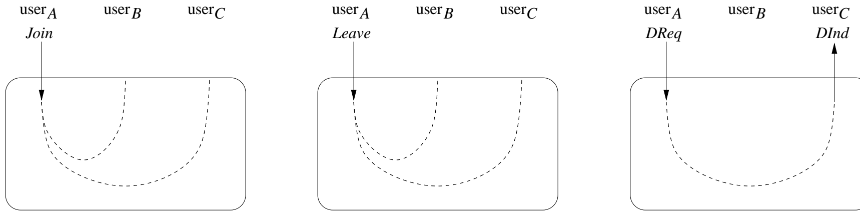


Figure 5.4: The chatbox protocol service

By contradiction. Suppose $s \xrightarrow{\rho}_{\mathcal{A}}$, $\sigma\rho = (\sigma\rho)^r$ and $(\tau\rho)^r = \tau\rho'$ with $\rho \neq \rho'$. By definition of $()^r$, we know that $\tau\rho \simeq \tau\rho'$. By Lemma 5.35, we know that the covering of the ρ-part in $\tau\rho$ must be equal to the covering of the ρ-part in $\sigma\rho$, and likewise for the ρ'-part in $\tau\rho'$ and $\sigma\rho'$. Then certainly $\sigma\rho \simeq \sigma\rho'$ must hold. By unicity of representatives, $\sigma\rho = (\sigma\rho')^r$. From Definition 5.29 we then obtain that $\sigma\rho \leq \sigma\rho'$ and $\tau\rho' \leq \tau\rho$, so $\rho \leq \rho'$ and $\rho' \leq \rho$. This yields a contradiction with the assumption that $\rho \neq \rho'$. ⊠

5.7 Examples

In this section we report on some initial experiments in the application of symmetry to the testing of two examples. Section 5.7.1 presents the example of a chatbox, and Section 5.7.2 presents the example of a cyclic train.

Part of the test generation trajectory was implemented: we used the tool environment OPEN/CÆSAR[Gar98] for prototyping the algorithm Kernel from Section 5.3. Section 5.7.3 relates some prototyping experiences.

We work with a pattern based symmetry (Section 5.6) and apply the test derivation method from Section 5.5.

5.7.1 A chatbox service

In this section we report on some experiments in the application of symmetry to the testing of a *chatbox*.

A chatbox offers the possibility to talk with users connected to the chatbox. After one joins (connects to) the chatbox, one can talk with all other connected users, until one leaves (disconnects). One can only join if not already present, and one can leave at any time. For simplicity, we assume that every user can at each instance talk with at most one user. Moreover, we demand that a user waits for a reply before talking again (unless one of the partners leaves). Finally, we abstract from the contents of the messages, and consider only one message. The service primitives provided by the chatbox are thus the following; Join, Leave, DReq, and DInd, with the obvious meaning (see Figure 5.4). For lack of space, we do not give the full formal specification of the chatbox or its template.

What we test for is the service of the chatbox as a whole, such as it may be offered by a vendor, rather than components of its implementation, which we (the “customers”) are not allowed to, or have no desire to, inspect.

```
des (0, 20, 3)
(0, "1 !DREQ !0 !1 !MES !DIND", 1)
(0, "3 !DREQ !1 !0 !MES !DIND", 1)
(0, "9 !DREQ !0 !1 !MES !NO_OUTPUT", 0)
(0, "10 !DREQ !0 !1 !ACK !NO_OUTPUT", 0)
(0, "11 !DREQ !0 !0 !MES !NO_OUTPUT", 0)
(0, "12 !DREQ !0 !0 !ACK !NO_OUTPUT", 0)
(1, "2 !DREQ !1 !0 !ACK !DIND", 0)
(1, "4 !DREQ !0 !1 !ACK !DIND", 0)
(1, "5 !DREQ !1 !0 !MES !DIND", 2)
(1, "7 !DREQ !0 !1 !MES !DIND", 2)
(1, "13 !DREQ !0 !1 !MES !NO_OUTPUT", 1)
(1, "14 !DREQ !0 !1 !ACK !NO_OUTPUT", 1)
(1, "15 !DREQ !0 !0 !MES !NO_OUTPUT", 1)
(1, "16 !DREQ !0 !0 !ACK !NO_OUTPUT", 1)
(2, "6 !DREQ !0 !1 !ACK !DIND", 1)
(2, "8 !DREQ !1 !0 !ACK !DIND", 1)
(2, "17 !DREQ !0 !1 !MES !NO_OUTPUT", 2)
(2, "18 !DREQ !0 !1 !ACK !NO_OUTPUT", 2)
(2, "19 !DREQ !0 !0 !MES !NO_OUTPUT", 2)
(2, "20 !DREQ !0 !0 !ACK !NO_OUTPUT", 2)
```

Figure 5.5: The template for a chatbox with three users and no joining/leaving

This example was inspired by the conference protocol presented in [TPHT96]. Some changes were made, all stemming from the need to keep the protocol manageable for experiments without losing the symmetry pursued. We mention the absence of queues and multicasts and the restriction to the number of outstanding messages. Also, we ignore the issues of test contexts, test architectures, and points of control and observation. A Lotos [ISO89] model and a μCRL [GP95] model were constructed for 3 and 4 users.

The symmetry inherent in the protocol is immediate: pairs of talking users can be replaced by other pairs of talking users, as long as this is done systematically according to Definitions 5.23 and 5.25. As an example, the trace in which user 1 joins, leaves and joins again, is symmetric to the trace in which user 1 joins and leaves, after which user 2 joins. The essence is that after user 1 has joined and left, this user is at the same point as all the other users that are not present, so all new join actions are symmetric. Note that this symmetry is more general than a symmetry induced solely by a permutation of actions or IDs of users. Thus the template $\mathcal{T}$ used for the symmetry basically consists of the conversation between two users, including joining and leaving, while the transformations π in the set Π shuffle the identity of users. We feel that it is a reasonable assumption that the black-box implementation offering the service indeed is symmetric in this sense.

We have applied the machinery to chatboxes with up to 4 users. We also considered a (much simpler) version of the protocol without joining and leaving.

In Figure 5.5, we display the template for the chatbox with three users and no joining/leaving. The template is in the Aldébaran state space format, in which `(i,<label>,j)` indicates the transition from state `i` to state `j` with action label `<label>`. The set of permutations is $\{\{1 \mapsto 1, 2 \mapsto 2\}, \{1 \mapsto 2, 2 \mapsto 1\}, \{1 \mapsto 1, 2 \mapsto 3\}, \{1 \mapsto 3, 2 \mapsto 1\}, \{1 \mapsto 2, 2 \mapsto 3\}, \{1 \mapsto 3, 2 \mapsto 2\}\}$. So, the template gives the chatting between user 0 and 1, from the point of view of user 0. There are three states, namely where no message has been sent (or all

	model			kernel	
	states	*trans*	*minimal?*	*states*	*trans*
3 users	512	12288	yes	213	3722
4 users	65536	2621440	yes	16385	263000
no joining/leaving					
3 users	64	1152	yes	10	84
4 users	4096	131072	yes	112	1296

Table 5.1: Kernel statistics for the chatbox

messages have been acknowledged), a state in which a message has been sent from one to the other, which has not yet been acknowledged, and a state in which two messages have been sent (in the two different directions) which have not been acknowledged. The self-loops are there to cover unsuccessful chatting attempts. The permutations shuffle the identities such that all possibilities are covered.

We start the test generation by computing a kernel for these specifications. In Table 5.1, the results of applying our prototype implementation of the algorithm Kernel can be found. Our prototype is able to find a significantly smaller Mealy machine as a kernel for each of the models, provided that it is given a suitable ordering $<$ (see Definition 5.29) on the actions symbols for its representative function. The kernels constructed consist of interleavings of transformations of the pattern, constrained by the symmetry and the ordering $<$.

For instance, in a chatbox with 3 users and no joining and leaving, we take the ordering $<$ defined as follows. “Sending a message from i_1 to j_1” $<$ “sending a message from i_2 to j_2” if $(i_1 < i_2)$ or if $(i_1 = i_2$ and $j_1 < j_2)$, and “sending a reply from i_1 to j_1” $<$ “sending a reply from i_2 to j_2” if $(i_1 > i_2)$ or if $(i_1 = i_2$ and $j_1 > j_2)$.

Using this ordering, the kernel only contains those traces in which first messages from user 1 are sent, then messages from user 2 and finally messages from user 3, while the sending of replies is handled in the reverse order. Each trace with different order of sending messages can then be computed from a trace of this kernel, which is exactly what Theorem 5.13 states. This technique of dealing with traces is reminiscent of partial ordering techniques [God96].

From Table 5.1 we see that the kernel size is relatively smaller when considering chatboxes without joining and leaving. This difference is due to the fact that, since one cannot send a message to a user that has left, joining and leaving obstructs the symmetry in messages being sent.

Given the computed kernels, we can construct test pairs by determining for each kernel a set of input sequences W that constitutes a *characterising set* for the kernel (as defined in Definition 5.16). Although this part has not yet been automated, it is easily seen by a generic argument that for every pair of inequivalent (non-bisimilar) states very short distinguishing sequences exist. It is easy to devise a transition cover for a kernel, the size of which is proportional to the size of the kernel.

As shown in Theorem 5.21, the size of the test suite to be generated will depend on the magnitude of two numbers m_1 and m_2, indicating the search space for distinguishing sequences for the image of the kernel in the implementation. This boils down to the following questions: (1) What is the size of the image part of the implementation for this kernel? (2) What is the size of a minimal distinguishing experience for each two inequivalent (non-bisimilar) states in

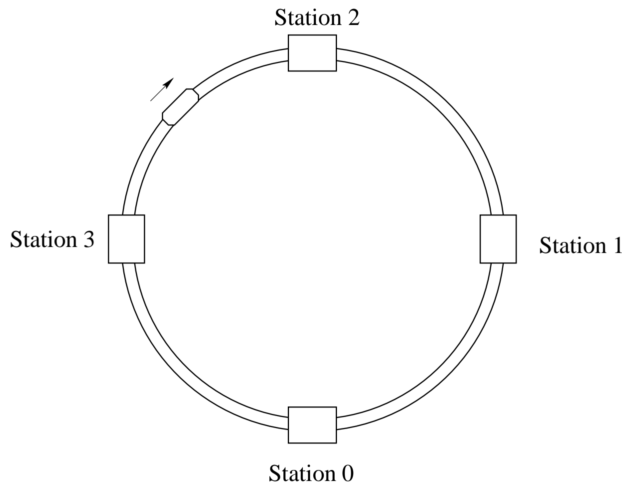


Figure 5.6: A cyclic train with 4 stations

the image part of the implementation? (3) How many steps does a distinguishing sequence perform outside the image of the kernel? These questions are variations of the classical state space questions for black box testing. For practical reasons, these numbers are usually taken to be not much larger than the corresponding numbers for the specification.

5.7.2 A cyclic train

In this section we report on some initial experiments in the application of symmetry to the testing of a *cyclic train*. This example was inspired by the elevator specification used by Frits Vaandrager in the course ‘Declarative Specifications and Systems’ at the University of Nijmegen in spring 1998. Since the symmetry in an elevator obviously must be sought in the floor numbers, and at the lowest (highest) floor it is not possible to go any lower (higher), we modified the example a little to make the elevator cyclic: from the lowest floor, the elevator can reach the highest floor by moving one floor down, and vice versa. To make the example a bit more intuitive, we rename the cyclic elevator to a cyclic train, and floors to stations.

See Figure 5.6. The train runs on a cyclic track, from station to station. It can change direction if needed, and can be sent to a destination if a button inside the train is pressed, and called to a station if a button in the station is pressed. We consider as running example a cyclic train running between four stations. In Figure 5.6, the train is moving from station 3 to station 2.

The symmetry inherent in the protocol is immediate: the behaviour of the train requested to go to a station or moving from one station to another is symmetric to the same behaviour when other stations are involved. As an example, the trace in which the train starts at station 1, is called to station 2 and then sent to station 0 is symmetric to the trace in which the train starts at station 3, is called to station 0 and then sent to station 2. Thus the template $\mathcal{T}$ used for the symmetry basically consists of the train arriving at the current station (from left or right), opening its doors, closing its doors and moving away again, while the transformations π in the

```
des (0, 20, 5)
(0, "1!REQUEST(CALL,1)", 1)
(0, "2!REQUEST(SEND,1)", 1)
(0, "3!MOVELEFT(1)", 2)
(0, "4!MOVERIGHT(1)", 2)
(1, "5!REQUEST(CALL,1)", 1)
(1, "6!REQUEST(SEND,1)", 1)
(1, "7!MOVELEFT(1)", 3)
(1, "8!MOVERIGHT(1)", 3)
(2, "9!REQUEST(CALL,1)", 3)
(2, "10!REQUEST(SEND,1)", 3)
(2, "11!MOVELEFT(0)", 0)
(2, "12!MOVERIGHT(2)", 0)
(3, "13!REQUEST(CALL,1)", 3)
(3, "14!REQUEST(SEND,1)", 3)
(3, "15!MOVELEFT(0)", 1)
(3, "16!MOVERIGHT(2)", 1)
(3, "17!OPENDOOR(1)", 4)
(4, "18!REQUEST(CALL,1)", 4)
(4, "19!REQUEST(SEND,1)", 4)
(4, "20!CLOSEDOOR(1)", 2)
```

Figure 5.7: The template for a train with three stations

set Π shuffle the identity of the station.

In Figure 5.7, we display the template for a train with three (or more) stations. Again, the template is in the Aldébaran state space format, in which `(i,<label>,j)` indicates the transition from state `i` to state `j` with action label `<label>`. The set of permutations is $\{\{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 2\}, \{0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 0\}, \{0 \mapsto 2, 1 \mapsto 0, 2 \mapsto 1\}\}$. So, the template gives the possibilities for the train arriving at station 1, and passing by or opening and closing its doors. The neighbour stations are 0 and 2. The permutations shuffle these station identities in a roundabout way.

We have applied the machinery to trains with up to 8 stations. We also considered a version of the train in the Mealy style, where each transition consists of an input and an output action. Here we have assumed that in each state, one can give an input by pressing a button, and that the output for such an input depends on the state of the train. If no input is given, the train may still want to move from station to station. This is modeled with the input action `WAIT`.

In Table 5.2, the results of applying our prototype implementation of the algorithm Kernel can be found. We work with state spaces generated from μCRL code which have not been minimised. The kernel is significantly smaller for each of the models, provided that it is given a suitable ordering $<$ (see Definition 5.29) on the actions symbols for its representative function. The orderings in the table refer to the ordering of symmetric request actions for the train to go to a certain station. The numbers indicate the identity of the station to which the train should go. For the Mealy style models, it turns out that the orderings listed in the table work better than others. For the models with 4 and 5 stations these are in fact the best orderings. For the other models, only some orderings were tested. Naturally, if the state space from which the kernel is constructed and the kernel itself get larger, the process takes longer.

	model			kernel		
	states	*trans*	*minimal?*	*states*	*trans*	*representative ordering*
7 stations	286725	4300874	no	12685	79594	6<5<4<3<2<1<0
8 stations	1310725	22282324	no	30945	195404	7<6<5<4<3<2<1<0
Mealy style						
4 stations	2808	24312	no	1548		0<2<1<3
5 stations	13950	148650	no	5193		0<2<4<3<1
6 stations	72036	912276	no	16959		0<2<5<3<4<1
7 stations	336042	4927734	no	49941	79594	0<3<6<4<5<2<1
8 stations	1563696	26060592	no	146394	887208	0<2<7<3<6<4<5<1

Table 5.2: Kernel statistics for the cyclic train

5.7.3 Implementing the algorithm Kernel

The algorithm Kernel (see Figure 5.1) was implemented using the OPEN/CÆSAR [Gar98] tool set. An exploration algorithm like this is implemented by writing the essence of the algorithm in C, using library functions and data types from the OPEN/CÆSAR interface in the prescribed manner. The routines and datatypes from the OPEN/CÆSAR library take care of the data structures for exploring the state space. The core of this is a table of states with two pointers, one pointing at the state that is being explored, and one pointing at the end of the table, where new states may be inserted. As soon as the first pointer passes the second one, the exploration is finished.

Since we based our implementation on the pattern approach, the input to the algorithm consists of two finite state machines: one for the specification that is reduced to a kernel, and one for the template of the symmetry, which is used to determine (as an oracle) whether two traces are symmetric. To enable this, the OPEN/CÆSAR interface had to be generalised somewhat so that it is now able to explore several labeled transition systems at the same time.

Our implementation differs a little from the presentation in Figure 5.1, in that it does not only keep track of the trace that it is exploring, but also of the current state for each copy of the template. This enables us to use Lemma 5.36, and search in the part of the table that was already explored for the current state, together with the current set of states for the copies of the template. Also, the set of possible representative actions leading from the current state is not determined using the trace leading to the current state, but using the state of each copy of the template.

The implementation was tried on the examples described in Sections 5.7.1 and 5.7.2, giving the results mentioned in Tables 5.1 and 5.2.

We have the experience that OPEN/CÆSAR is suitable for prototyping exploration algorithms such as Kernel.

5.8 Future work

We have introduced a general, FSM based, framework for exploiting symmetry in specifications and implementations in order to reduce the amount of tests needed to establish correctness. The feasibility of this approach has been shown in a few experiments.

However, a number of open issues remain. We see the following steps as possible, necessary and feasible. On the theoretical side we would like to (1) construct algorithms for computing and checking symmetries, and (2) determine conditions that are on the one hand sufficient to guarantee symmetry, and on the other hand enable significant optimisations of the algorithms. On the practical side we would like to (1) generate and execute tests for real-life implementations, and (2) continue prototyping for the whole test generation trajectory.

Chapter 6

Model checking the HAVi leader election protocol

Summary

The HAVi specification [GHM+98] proposes an architecture for audio/video interoperability in home networks. Part of the HAVi specification is a distributed leader election protocol. We have modelled this leader election protocol in Promela and Lotos and have checked several properties with the tool Spin and the tool Xtl (from the Cæsar/Aldébaran package).
It turns out that the protocol does not meet some safety properties and that there are situations in which the protocol may never converge to designate a leader. Our conclusion is that realistic timing requirements on sending and processing of messages should be added to the HAVi specification. Then a (timed) formal verification could give a definite answer with respect to correctness of the leader election protocol.

6.1 Introduction

The Home Audio/Video Interoperability (HAVi) project [GHM+98] is a joint effort by eight consumer electronics companies to solve interoperability problems for audio/video networks in the home environment.

The HAVi specification specifies a set of Application Programming Interfaces (APIs) and protocols that allow consumer electronics manufacturers and third parties to develop applications for the home network. Thus the home network is viewed as a distributed computing platform, and the primary goal of the HAVi architecture is to assure that products from different vendors can cooperate to perform application tasks. The HAVi architecture is supposed to work on top of an IEEE 1394 serial bus [IEE96, IEE99].

There are two types of HAVi devices: controllers and controlled devices. The controller acts as a host for controlled devices via a Device Control Module (DCM). Installation and allocation of such DCMs is done by a HAVi software element which is called the Device Control Module Manager (DCM Manager). Each controller is supposed to have a DCM Manager. All DCM Managers have to cooperate with each other to ensure that the installation and allocation of DCMs works properly. A complicating factor here is the dynamic plug-and-play character

of the 1394 network. Each time when a change in the 1394 network occurs, the DCM Managers restart their activities by first choosing a leader among them, and then under the control of the designated leader, complete their DCM controlling tasks.

The purpose of the leader election is that the DCM Manager with the best capabilities will play a central role in the DCM controlling tasks. Since not all of these capabilities are persistent and globally available, the DCM Managers need to communicate to find out which one is the best candidate for leadership.

In this chapter, we study the leader election protocol between the DCM Managers. Our goal is to analyse this protocol with several model checking tools, to determine whether the protocol is correct, and to compare the model checking tools. Our approach is to construct a model of the behaviour of the protocol in a suitable formal language, and to establish certain properties through model checking. Model checking is a verification approach where one checks whether a property holds by exploring the reachable state space of the model. The manual construction of such proofs is a tedious and error-prone process. Nowadays, there are several tools that fully automate the model checking process.

We present several models of the protocol leader election protocol in the formal languages Promela [Hol91] and Lotos [ISO89]. Several properties have been checked with the model checking tools Spin [Hol91, Hol97] and Xtl [Mat98, MG98] (part of the Cæsar/Aldébaran distribution [FGK+96]).

We have found errors in the formal models with both Spin and Xtl. It turns out that some safety properties are not met by the protocol and that there are situations in which the protocol may never converge to designate a leader. The cause of these errors is that the HAVi specification is not detailed enough to ensure that HAVi compliant implementations are faultless. The errors occur when communication between different devices is faster than communication between components in one device. Besides our conclusions on the correctness of the HAVi protocol, we compare the two model checking tools.

As far as we know, the only other paper in which the HAVi leader election protocol between DCM Managers is studied is [Use99]. Here, a comparison is made between the performance of state space exploration of Spin and the μCRL tool set [GP95]. The model of the protocol differs from ours and no model checking has been performed.

It should be noted that although [GHM+98] is not the most recent version of the HAVi specification, it is the only version publicly available. Newer versions are subject to constant change and confidential. Therefore this research is based on [GHM+98].

This chapter is organised as follows. Section 6.2 gives an informal description of the HAVi leader election protocol. Section 6.3 introduces the tools and languages used. Section 6.4 describes our model of the protocol. Section 6.5 gives the details of all the model checking experiments. Finally, Section 6.6 gives several conclusions that we drew from this experiment.

The full version of the research presented in this chapter is available as CWI report SEN-R9915. It contains relevant excerpts from the HAVi and 1394 specifications and several code listings.

6.2 The DCM Manager leader election protocol

The DCM Manager leader election protocol is described in the HAVi specification [GHM+98] at page 160. The protocol tries to find a suitable leader that can manage the actual task of the

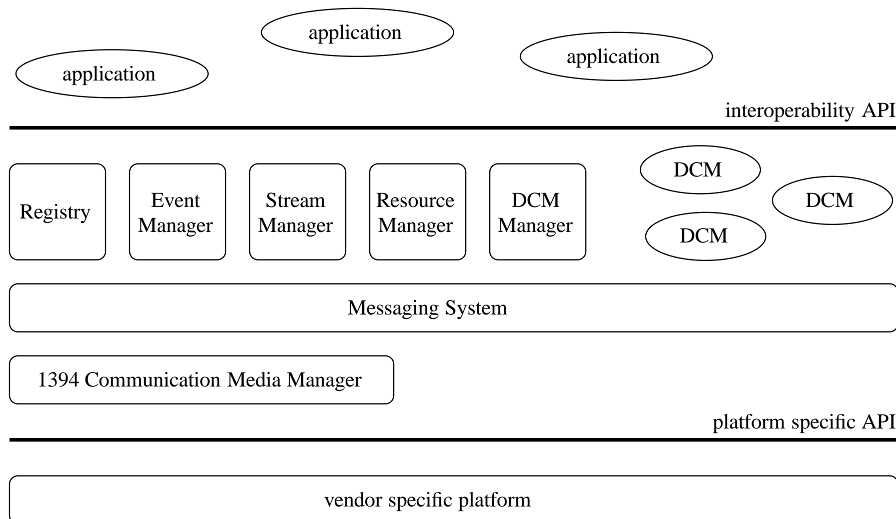


Figure 6.1: The HAVi architecture

DCM Managers, which is performed in the autonomous operation phase. We only study the leader election phase.

The parts of the HAVi specification and the IEEE 1394 standard that are relevant for this protocol can be found in the report version. Here, we give an informal explanation of the protocol, and the services that it requires from several HAVi components. We start with the latter.

6.2.1 HAVi components

In Figure 6.1, the HAVi architecture is depicted. The different services in the middle layer of the architecture are described in the HAVi specification; they are referred to as HAVi elements. Local elements reside in the same device. The DCM Managers use the services of the local elements Messaging System, Communication Media Manager, and Event Manager. These elements will be available at each HAVi device that contains a DCM Manager.

The **Messaging System** provides two services and two modes of sending messages to software elements, whether local or not. The service choices are to block while waiting for a response by the receiver or not to wait for a response. The modes are reliable or simple. The reliable mode implicates that the sender is informed by the Messaging Systems involved whether the message reached the receiver. The sender is blocked until such an acknowledgement arrives or a timeout occurs. The simple mode implicates no acknowledgement information from the Messaging Systems is given to the sender. The Messaging System on the device of the receiver delivers the message to the receiver via a call back function, which the receiver has dispensed to the Messaging System at start-up time. The Messaging System uses the 1394 network for the actual message passing. From the 1394 specification we learn that at the 1394 level, no messages can be sent between different devices while a bus reset is taking place.

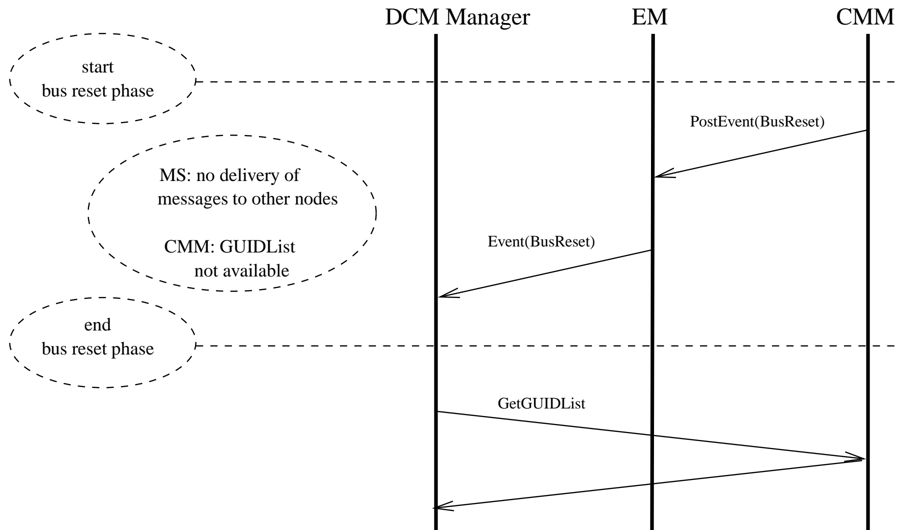


Figure 6.2: A bus reset scenario

The DCM Managers communicate with each other using the reliable method and the response service. The HAVi specification does not limit the nature of the call back function that the DCM Managers use. The DCM Managers use a timeout of 3 seconds on all messages.

The **Event Manager** accepts requests to post events and sends a message with the event through the Messaging System to every local software element that has subscribed to the event. A posting request must be sent through the Messaging System. The DCM Managers all subscribe to the BusReset event during initialisation.

The **Communication Media Manager** provides information on the network configuration which it gets from the 1394 layer. Upon the start of a bus reset phase, it posts the event BusReset. Since each FAV or IAV device has its own Communication Media Manager to signal the bus reset start, the BusReset event only needs to be sent to software elements on the same device. This means that the Messaging System can at all times deliver the messages containing this event to the interested parties, as long as the device is powered up.

The Communication Media Manager also allows software elements to request network information in the form of a GUIDList. This service is only available outside bus reset phases, after the Communication Media Manager has received the information from 1394. This information is to be asked with a message through the Messaging System.

An example scenario In Figure 6.2 we show an example scenario in which the following happens. A bus reset period starts. The Communication Media Manager posts the BusReset to the Event Manager. The Event Manager delivers the BusReset to the DCM Manager. The DCM Manager reacts by requesting the GUIDList from the Communication Media Manager. This list is available only when the bus reset period has ended.

6.2.2 Protocol

Each DCM Manager enters the leader election phase upon initialisation and each time a bus reset event is received. First it obtains information on the current network topology, by sending a request to another HAVi element, the Communications Media Manager, which returns a list with all the devices that are active in the (1394) network. The list contains the Global Unique ID (GUID) of all devices in the network. The DCM Manager then questions the 1394 level of each active device to find out some more information. The information needed for this protocol is the HAVi type of the device (FAV, IAV, BAV or LAV), and whether there is a DCM Manager present at the device (at FAV compulsory, at IAV optional). Based on this information, the DCM Manager selects an initial leader from the GUIDs of devices on which a DCM Manager is present. Since each DCM Manager uses the same procedure for the selection, all of them choose the same initial leader without communicating with each other. Each DCM Manager which is not the initial leader is called initial follower.

The initial leader waits for initialisation requests from all initial followers, in which they state their capability. Using this new information and the HAVi type of the devices, the initial leader decides which DCM Manager is the best candidate for the final leadership. One of the criterions is the HAVi controller type, which is found in the (static) information of the HAVi device and which can be accessed from outside the device. The other criterion is Internet access which is found in the request messages from the followers. Each initial follower is informed of the decision with an initialisation reply, and the DCM Manager that has been elected as the final leader is informed last. After this, the leader election phase ends and the autonomous operation phase is entered. Here, each DCM Manager which is not the final leader is called final follower.

During or after the leader election phase, the network topology may change, which causes a bus reset phase to start. Whenever this happens, the DCM Managers should start anew with the leader election because the previously elected leader may have disappeared from the network or a more suitable candidate may have appeared. The DCM Managers are informed of a bus reset phase by the Communications Media Manager with an event. The HAVi specification does not lay down any implementation rules for the delivery of this event, such as timing requirements. So it is possible that the bus reset event is delivered after the bus reset phase has already ended. If multiple bus reset phases occur (almost) adjacently, the DCM Managers may get out of phase in their leader election. Then one DCM Manager might be sending its initialisation request to an initial leader which is not aware of any bus reset phase having taken place, or vice versa. To keep things in order, the DCM Manager which is to be the initial leader, must remember this role and answer initialisation requests with an initialisation reply, even after leader election has ended. During and after the protocol, all unexpected messages are ignored.

6.3 Languages and tools

This section gives a short introduction to the languages and tools used for formalisation and verification of the leader election protocol. For details we refer to the documentation cited below.

6.3.1 Spin and Promela

Spin [Hol91, Hol97] is a tool that supports simulation and verification of Promela [Hol91] models of distributed systems. Models in Promela (a Process Meta Language) consist of definitions of process behaviour, with variable assignments, sequential and alternative composition, repetition and dynamic process creation. Communication between processes happens on synchronous or asynchronous channels. Synchronous communication always involves two processes. The support of data types is very limited: basic types are booleans and naturals, from which arrays and record structures can be built.

Verification is supported through detection of deadlocks, invalid end-states or non-progress loops, through violation of assertions and through LTL [Pnu77, MP92] properties. The verification is done on the fly: the global state space is not constructed, but explored directly from an interpreted version of the Promela code.

6.3.2 Lotos, Cæsar/Aldébaran and Xtl

Lotos [ISO89] is a standardised language for abstract modelling of distributed systems. Lotos models consist of a data part and a behaviour part: the data part is expressed in ACT-ONE, an algebraic formalism for abstract data types, and the behaviour part is expressed in process algebra with sequential, alternative and parallel composition, and recursion. Communication happens on synchronous gates and can involve more than two processes.

The Cæsar/Aldébaran tool set [FGK$^+$96] facilitates simulation and verification of Lotos models. Simulation and detection of deadlocks, livelocks et cetera can be done on the fly.

The Xtl tool [Mat98, MG98] (which is part of the Cæsar/Aldébaran tool set) facilitates the verification of temporal properties over Lotos models. First the global state space must be generated (with Cæsar), then Xtl can verify a property given in one of the following logics: HML [HM85], CTL [CES86], LTAC [QS83], ACTL [DNFGR93, DNV90] and the modal μ-calculus [Koz83]. It is even possible to define one's own modal logic in terms of the libraries provided by Xtl (including greatest and least fixpoint operators).

6.4 Modelling decisions

In this section, our model of the protocol is explained. What is presented here is the result of a process of experimenting with different models, imposing and lifting restrictions until a satisfactory model with a manageable size was obtained.

In the remainder of this section we abbreviate DCM Manager (DM), Communication Media Manager (CMM), and Messaging System (MS).

Restrictions on the network Each of the following choices is a restriction on what is allowed by the HAVi model. These restrictions are imposed in order to obtain a model of manageable size.

We study only situations with one network in which maximally three devices are active, and demand that in the start state no device is powered on.

The HAVi device types are FAV, IAV, BAV and LAV. We assume that there only are FAV devices in the network, and that on each of these devices, a DM is present.

A bus reset in the 1394 network may be caused by a change in the network topology (a device being added to or removed from the network), by a device in the network being powered up or down, by race conditions in the 1394 protocol or by other error situations. We model the cause of a bus reset as the power change of zero or more devices in the network. Here, zero power changes represent some other cause of bus reset, and the power change of a device also represents the connecting or disconnecting of that device (when a device is disconnected but still powered up, it operates in a new network consisting of just itself; we only study one network). The network behaviour is modelled with the process Bus_Reset.

From IEEE 1394 we learn that the worst-case time delay between the start of the bus reset phase and the moment that the last device in the network notices the bus reset is less than 167 microseconds. The duration of the bus reset phase until normal operation resumes is at least 414 and maximally 581 microseconds. We restrict the bus reset phase delay to zero, which means that the bus reset phase starts at the same time at all devices in the network. For our verification purposes we only want to consider properties that concern situations in which a bus reset is not taking place. Therefore it is convenient to have the start of the bus reset phase actually precede the change of network which causes the bus reset phase.

In the HAVi design, each DM uses a capability and a preference in the leader election protocol. We restrict ourselves to the capability UrlCapable, which indicates whether a device has Internet access (true) or not (false). We assume that the value of UrlCapable does not change.

In a 1394 network a device may be unplugged (powered off), and then plugged back in (powered on). This may cause the device to get a different 1394 physical ID and HAVi SEID (Software Element ID) once it is back in the network, than the 1394 and HAVi IDs it had before. Since each device has a globally unique ID (GUID) which does not change, and other devices can find out about this through the GUIDList which is managed by their CMM, we only identify devices with their GUID and do not model the physical ID.

Which HAVi components? We model the DM, the MS and the CMM with separate processes, which are described below. We do not include a process for the Event Manager. The only event posted to this component will be the BusReset, and all different scenarios of delivery of this event can be modelled by one synchronous communication between the CMM and the DM. If the delivery is unsuccessful, the communication does not occur. An extra process Bus_Reset is needed to model the behaviour of the 1394 network.

Process Bus_Reset This process determines whether a new bus reset period will start, and which devices (hence which DMs and CMMs) will be powered up or down. Both of these choices are non-deterministic, hence in a verification all possibilities will be considered. Whenever a device is powered up or down, the DM, CMM and MS on that device are informed by Bus_Reset in a synchronous manner. The power changes are determined in increasing order of device ID.

Process CMM This process controls the GUIDList, in which all devices present in the network are listed. It also signals any start or end of a bus reset period on the 1394 network, and passes this information on to the DM and the MS on the same device.

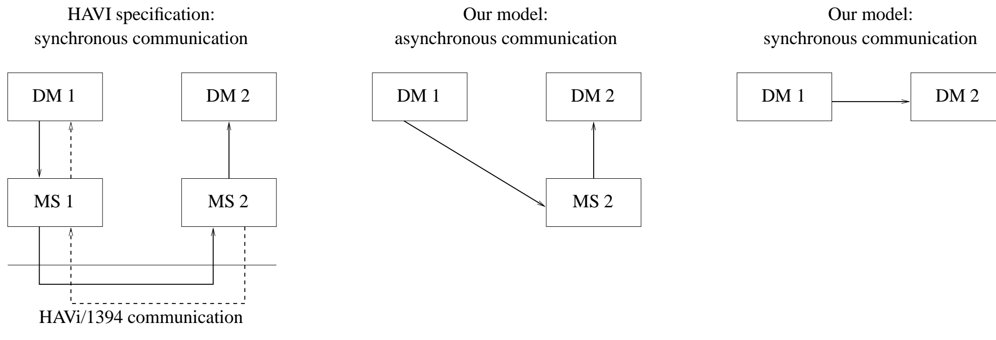


Figure 6.3: DCM Manager to DCM Manager communication

When several bus reset periods follow each other with little time in between, it is possible that a CMM has not posted the occurrence of a previous bus reset, when the next is already taking place. The HAVi specification does not define whether both bus reset events should be posted or just one. We choose to have the new bus reset overrule the previous one, and have only the last bus reset notification being posted and delivered.

Process MS This process takes care of the communication between the DMs and acts as a buffer. All message transfers that use the MS, are performed in reliable mode, therefore we model such a message transfer as one communication involving just the sending and the receiving component. The message transfer is shown in Figure 6.3.

The HAVi design is that DM 1 sends a message, intended for DM 2, to the MS 1 (which is on the same device as DM 1). MS 1 sends the message on the network to MS 2, which delivers it to DM 2. After sending the message, DM 1 will wait for an error message, a timeout or an acknowledgement of successful delivery to DM 2. DM 1 only continues its operation after such a notification/timeout. In Figure 6.3, continuous arrows show how a message is transported through the HAVi architecture from DM 1 to DM 2, and the dashed arrows show how the notifications are generated and returned. In case of erroneous transfer, the message may not reach MS 2 or DM 2, but DM 1 is aware that something is wrong because the proper acknowledgement was not received. Successful delivery to DM 2 means that either DM 2 is interrupted to receive the message (synchronous communication) or the message is put into a buffer designated by DM 2 (asynchronous communication).

We have modelled the synchronous version of this communication with direct synchronous communication between DM 1 and DM 2 (and then there is no need for any MS process), and the asynchronous version by synchronous communication between DM 1 and MS 2. In the latter case, DM 2 can get the message from MS 2 by synchronous communication. Note that MS 1 is not used in this communication scheme. This modelling choice is made to limit the possibilities for the communication, which is reasonable since we are only interested in the communication succeeding (modelled by the message put into the buffer) or failing (modelled by the communication not occurring at all). Of course, the size of the buffer maintained by the MS limits the number of messages that can be sent to a DM before it actually receives them.

So, in short, in the case of synchronous communication between DMs, there will be no MS

process in our model. In the case of asynchronous communication between DMs, there will be an MS process which acts as a buffer for incoming messages directed to the DM at the same device. The buffer size is a parameter for the model; in all our models the buffer size is 1. In case of asynchronous communication, the DM will empty the buffer in the event of a bus reset period or whenever the power is switched off.

Process DCM_Manager The general task of the DM is explained in Section 6.2. Our model follows this procedure as closely as possible, except for a few modelling choices.

1. In our model we skip the subscription that the DM uses to inform the Event Manager that it wants to receive all bus reset events. We also skip the registration of the call back function that the DM must dispense to the MS.

2. From the two parameters that the DM uses in the protocol, we only consider UrlCapable (Internet access).

3. The HAVi method of electing the initial leader, is to choose the DM on the device with the highest *bit order reversed* ID. Since our assignment of IDs to DMs is arbitrary, we just choose the DM with the lowest ID for initial leader.

4. The selection of the final leader in the HAVi design should be an arbitrary choice of the devices with the best capabilities. We study networks with only FAV devices on which a DM is present, hence we let the device with the lowest ID and UrlCapable set to true be the final leader (which is not arbitrary, but does limit the size of the state space). If no device has special capabilities, the HAVi design allows the initial leader to elect an arbitrary device for final leader. In this case, we have the initial leader elect itself for final leader (which also limits the state space size).

5. In the HAVi protocol, each initial follower will send its initialisation request to the initial leader, and will resend the request if a reply was not received before a timeout occurs (which is after 3 seconds). All our models are without timing information. Hence we let the initial follower choose arbitrarily between resending the request and receiving the reply. In this manner we cover all possibilities. Note that this choice does not introduce new behaviour, that is, behaviour that is not permitted by the HAVi specification.

6.5 Model checking experiments

In order to check that the protocol works as intended, we have checked four properties on several models of the protocol. Each of the following sections is dedicated to one property. The properties are listed in this section in an informal manner and in a notation slightly different from the actual input for the tools. For the exact definitions of the properties, we refer to the report version.

The properties presented here were devised after the models of the protocol had been constructed. This has both advantages and disadvantages. A disadvantage is that it turns out to be rather difficult to express properties for our specific models. In fact we have had to change them slightly to make some information visible. An advantage is that the models have not been tailored towards the properties that should be checked except the changes mentioned. A

potential danger is that the model does not resemble the protocol close enough anymore, and the properties to be checked trivially hold.

Since the behaviour of the protocol is unpredictable during bus resets or the period that the CMMs need to deliver the bus reset event, we only demand that the properties be true for stable situations, that is, in states where it is not the case that a bus reset is taking place or a bus reset event should still be delivered. Since a new bus reset period may start at any moment after the previous bus reset has ended and since we have included this possibility in our models with non-deterministic choice, we get the behaviour depicted in Figure 6.4 from our models. Suppose that $s_1, s_2, s_3, \ldots, s_n$ in Figure 6.4 are stable states, which means that no bus reset is taking place, and all events concerning the last bus reset have been delivered. We see that from s_1 it is possible that a new bus reset period starts, but it is also possible that some other behaviour takes place on the transition to s_2. If we establish a property in terms of behaviour, we can only capture the desired behaviour from s_1 by using an *exists* quantifier: from s_1 there exists a behaviour which satisfies a certain requirement. Moreover, in our models the amount of activity that concerns the protocol is bounded. After a certain point, the protocol is stuck or completed, and the only possible behaviour is that a new bus reset period starts. So it is not possible to express a property as follows: "for all behaviours: if no bus reset starts in this behaviour then fulfill a requirement".

Expressing properties for Promela models Safety properties can be checked in Spin through the use of `assertion` statements. We use a process with only such an assertion statement in the verification for checking whether there is a state in which the assertion is false. If this happens, Spin reports this as an error and stops the verification. An error trace is saved which can be used for diagnostic purposes.

Liveness properties can be checked in Spin through the use of LTL [Pnu77, MP92] formulas, which are translated into `never` claims. A never claim is a process which will only terminate if the corresponding LTL formula was violated. Actually, never claims represent ω-regular properties. Spin checks whether never claims hold in the initial state. This means that if a never claim is already satisfied by the initial state, no further exploration of the state space is needed.

Both assertions and LTL formulas are expressed in terms of predicates, which range over values of variables. It is also possible to check a pattern of communications, but not in combination with checks of state variable values. Since in our case, it is by far the easiest to find error situations by referencing the state variable values, we stick to the assertions and never claims.

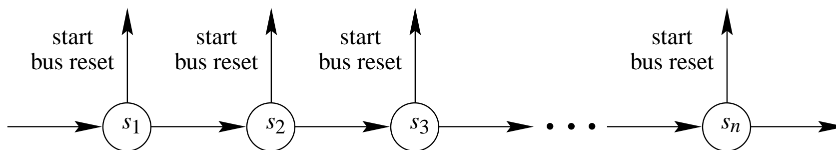


Figure 6.4: Protocol behaviour

Expressing properties for Lotos models We express safety and liveness properties in ACTL [DNFGR93, DNV90]. A property is checked by Xtl on the reachable state space, by checking for each reachable state whether the property holds in that state.

Since the model checker Xtl is only used on state spaces which have been generated from the Lotos model, the information of state variables is lost. Actually, the states are identified by natural numbers in the state graph accepted by Xtl. This means that we cannot express properties in terms of values of state variables, but can only observe the occurrences of actions. A consequence of this approach is that some safety properties are expressed with patterns of action occurrences, which are normally only used for liveness properties. With the ACTL logic we are able to observe such patterns. In order to still reference state variable values, one could build self-loops into the Lotos model, which give the values of the state variables in the corresponding state. However, this was not a feasible approach in our case (See the discussion in Section 6.6).

An action can be observed by comparing an action label from a transition to a label set in the property that is being checked. Comparing an action label to the label set **T** (**F**) always succeeds (fails). Label sets can be constructed from syntactic expressions that capture one or more action labels, and boolean operators. For instance, it is straightforward to construct a label set that succeeds when compared to the label `BUS_RESET_START` or the label `POWER_CHANGE` and fails otherwise.

In order to enable the checking of not just communications between the DCM Managers, but also other important actions, the model contains a few extra observable events. These are modelled by the occurrences of communication on the special gate `GEvent`. In this way we observe a DCM Manager electing itself for initial or final leader.

We now give an overview of the ACTL operators used, and their informal meaning[1].

$\mathbf{T}, \neg, \wedge, \vee, \rightarrow$ Boolean true, negation, and, or, implication

$[a]\ \phi$ For every transition $s \xrightarrow{a} t$ from the current state: formula ϕ must hold in the target state t

$\forall \mathbf{G}_a \phi$ For each (possibly finite) path from the current state where all actions are either a or τ, formula ϕ must hold in every state

$\exists(\phi_a \mathbf{U}_b \psi)$ There exists a path from the current state along which for a finite fragment formula ϕ holds in each state and all actions are either a or τ, and this fragment is immediately followed by a transition $s \xrightarrow{b} t$, and in state t formula ψ holds.

For a complete list of ACTL operators and a formal definition, we refer to [DNFGR93, DNV90, HM85].

The standard library in the Cæsar/Aldébaran distribution for using these operators is the `actl.xtl` library (implemented by Mateescu [Mat98, MG98]) which establishes the validity of a formula by checking whether the formula holds in all reachable states of the Lotos model. This library is not implemented in such a way that it gives diagnostics in case a property is not true. Diagnostics can be obtained by using the `walk_actl.xtl` library (implemented

[1] Note that here, [] is not a pure ACTL operator, but an operator from the Hennessy-Milner modal logic [HM85]. Since the Xtl library for ACTL is defined using the Xtl libraries for the Hennessy-Milner modal logic and the modal μ-calculus [Koz83], we can use operators from these logics in any ACTL expression.

by Pecheur [Pec98]), which also implements the ACTL operators mentioned, and which tries to find an error trace. This implementation establishes the validity of a formula by checking whether the formula holds in the initial state of the Lotos model. Of course, in general the use of this library is more costly since there is more administration involved in finding the trace, and a lot of backtracking occurs.

6.5.1 Safety: At most one leader

It is never the case that more than one DCM Manager is a (initial or final) leader.

Spin We use an `assertion` statement, and check the following formula:

$$\forall d, d'.(\neg bus_reset \wedge leader(d) \wedge leader(d') \ \rightarrow \ (d = d'))$$

This property does not hold for any of the models. In Figure 6.5 an error trace constructed by Spin for the model with two DCM Managers and synchronous communication is listed[2]. This trace describes the following behaviour. In the first bus reset period both DCM Managers are powered up. They start the leader election protocol, in which DCM Manager A is the initial leader and DCM Manager B is the initial follower. B is UrlCapable and A is not. B sends A an InitRequest, A computes the final leader which is B, and sends the InitReply to B. A new bus reset period starts and ends without change in the network topology. The CMM on the device of B delivers the bus reset event to B, and B starts the leader election protocol anew. B is again initial follower and sends A an InitRequest. A does not know about the second bus reset period so it is in its final follower phase where it answers any InitRequest with the same InitReply as before. A sends B the InitReply and B concludes it is the final leader. Now the CMM on the device of A delivers the bus reset event to A, and A starts the leader election protocol anew. A is again initial leader and does not know the identity of the final leader to be elected, while B still thinks it is final leader. In this state the property checked is violated.

The question is now whether this scenario is also possible within the HAVi specification. The problems apparently arise when the delivery of a bus reset event message is delayed beyond the duration of both a message and a response between different devices, and when the GUIDList is available before the corresponding bus reset event has been delivered. It may be argued that such delays are not 'realistic' and 'will not occur in practice'. However, since the HAVi specification does not put constraints on the duration of communication between devices, or on the delay between posting and delivery of the bus reset event, it is possible that the bus reset event is delivered first to other HAVi elements which have subscribed to it, before the DCM Manager receives it. Thus, it seems that in HAVi compliant implementations this erroneous scenario may occur, and we conclude that this error indeed traces back to a design flaw. We refer to Section 6.5.5 for a more elaborate discussion whether the HAVi protocol is wrong.

Xtl What we want to establish, is that there are not multiple InitialLeader or FinalLeader events in between of bus reset periods. Since we can check for patterns of actions, we formulate the property as follows: if a bad pattern of Initial or FinalLeader events occurs, then we are

[2] In fact, this trace was generated when model checking the property from Section 6.5.3. It turns out that it is also an error trace for the property discussed here.

not in a stable situation (where no bus reset is taking place and the last bus reset events have all been delivered). This boils down to expressing that when a bad pattern does occur outside bus reset periods, apparently a bus reset event must still be delivered.

We check the following formula:

$$([b_1]\ \forall\mathbf{G}_{i_1}([i_3]\ \forall\mathbf{G}_{i_1}([i_3]\ \exists(\mathbf{T}_{i_2}\mathbf{U}_{b_2}\mathbf{T})))\ \wedge\ ([b_1]\ \forall\mathbf{G}_{i_1}([f]\ \forall\mathbf{G}_{i_1}([i_4]\ \exists(\mathbf{T}_{i_2}\mathbf{U}_{b_2}\mathbf{T})))$$

$$\begin{aligned}
\text{where}\quad & b_1 = \mathit{BusResetEnd}\\
& b_2 = \mathit{BusResetEvent}\\
& i_1 = \mathit{Ignore}_1 = \neg(\mathit{BusResetEvent} \vee \mathit{BusResetStart} \vee \mathit{Initleader} \vee \mathit{FinalLeader})\\
& i_2 = \mathit{Ignore}_2 = \neg(\mathit{BusResetEvent} \vee \mathit{BusResetStart})\\
& i_3 = \mathit{InitLeader}\\
& i_4 = \mathit{InitLeader} \vee \mathit{FinalLeader}\\
& f = \mathit{FinalLeader}
\end{aligned}$$

This formula expresses two patterns that should be followed by a bus reset event being delivered. Both patterns start with the end of a bus reset period, and do not allow the start of a new bus reset period by the use of the action label sets Ignore_1 and Ignore_2. The first pattern checks the double occurrence of the InitialLeader event. The second pattern checks the occurrence of a FinalLeader event, followed by either an InitialLeader or FinalLeader event. The action label sets in the subscript of the **G** and **T** symbols enable the actions in the subscripts to occur in any sequence in between.

This property holds for all models. Since we found errors in the Promela models for this property using Spin (See earlier in this section) two questions remain, namely whether the error behaviour found with Spin also occurs here and if so, why it is not found with the ACTL formula used. Simulating the behaviour from the Spin error trace is possible for the Lotos model with two DCM Managers and synchronous behaviour. As to the second question. The answer is that the label set Ignore_1 is too restrictive. The idea of checking a pattern when a bus reset event has completed turns out counterproductive. We might have checked *all* occurrences of the FinalLeader event followed by bad patterns, and qualified the occurrence of a BusResetStart, BusResetEnd or BusResetEvent as a good pattern. In any case, it appears that the formulation of the property in this setting is very complicated. We refer to Section 6.5.5 for the discussion whether the HAVi protocol is wrong.

6.5.2 Safety: Best candidate becomes final leader

It is never the case that a final leader is selected which is not UrlCapable, while there is a DCM Manager active in the network which is UrlCapable.

Spin We use an `assertion` statement, and check the following formula:

$$\neg \mathit{bus_reset} \wedge \forall d.((\mathit{f_leader}(d) \wedge \neg \mathit{url_capable}(d))\ \rightarrow\ \forall d'.(\mathit{up}(d') \rightarrow \neg \mathit{url_capable}(d')))$$

This property holds for all models except for the setting with three DCM Managers and asynchronous communication. However, the error found here reveals problems with the interpretation and execution of the Promela code rather than an error in the protocol. In fact, we can

reason why in our model the property should be true for any number of DCM Managers with either synchronous or asynchronous communication. The idea is that upon receipt of a bus reset event, each DCM Manager will clear the information of being final leader and ask for the new network topology (the GUIDList). Since the start of a bus reset period causes the delivery of a bus reset event at some time, in a stable situation all bus reset events have been delivered, and each DCM Manager must have the correct network topology information. So after the last bus reset event delivery to a DCM Manager, it cannot choose a non UrlCapable final Leader if there is a UrlCapable DCM Manager present. So the only way in which a non UrlCapable DCM Manager can still be the final leader in a stable situation, while a UrlCapable DCM Manager is present, is to receive an InitReply with its identity from the initial leader, when the initial leader has not received the latest bus reset event. But we have modelled the final leader election by having the initial leader choose itself, if no UrlCapable Manager is present. So it cannot ever send an InitReply with the identity of another, non UrlCapable DCM Manager. It is clear that although the property must hold in our models, it does not hold when we lift the restriction that the initial leader chooses itself for final leader when no UrlCapable DCM Manager is present. Since there is no such restriction in the HAVi specification, we expect that this property does not hold for HAVi compliant implementations in general. As in Section 6.5.1 the error scenarios require that the delivery of a bus reset event message is delayed beyond the duration of the sending and delivery of both a message and a response between different devices, and possibly also that the GUIDList is available before the delivery of the corresponding bus reset event. We refer to Section 6.5.5 for a more elaborate discussion whether the HAVi protocol is wrong.

Xtl The situation that a DCM Manager is up and UrlCapable is signalled by the request from such a DCM Manager to the initial leader, in which the UrlCapable parameter is true. Whenever such a request is followed by the election of a final leader which is not UrlCapable, there must be a bus reset event pending that needs to be delivered.

We check the following formula:

$$[u]\,\forall \mathbf{G}_{i_1}\,([f]\,\exists(\mathbf{T}_{i_2}\mathbf{U}_b\mathbf{T}))$$

$$\begin{aligned} \text{where}\quad & b = \mathit{BusResetEvent} \\ & i_1 = \mathit{Ignore}_1 = \neg(\mathit{BusResetEvent} \vee \mathit{BusResetStart} \vee \mathit{BusResetEnd} \vee \mathit{FinalLeader}) \\ & i_2 = \mathit{Ignore}_2 = \neg(\mathit{BusResetEvent} \vee \mathit{BusResetStart}) \\ & f = \mathit{FinalLeaderNotUrlCapable} \\ & u = \mathit{RequestUrlCapable} \end{aligned}$$

This property holds for all models. We refer to the paragraph above on Spin experiments for this property, for a discussion whether this property holds in general or not, and to Section 6.5.5 for the discussion whether the HAVi protocol is wrong.

6.5.3 Safety: All agree on the final leader

Whenever a final leader is selected, all DCM Managers agree on the identity of this leader. Of course this can only be checked as soon as all DCM Managers have been informed of the decision of the initial leader. Since the final leader is informed last of the decision (and

whenever this happens to be also the initial leader, it will 'inform itself last'), this can be checked as soon as one of the DCM Managers has been elected for final leader.

Spin We use an `assertion` statement, and check the following formula:

$$\forall d.(\neg \mathit{bus_reset} \wedge \mathit{f_leader}(d) \;\rightarrow\; \forall d'.(\mathit{up}(d') \rightarrow \mathit{leader_id}(d') = d))$$

This property does not hold for any of the models. The error trace constructed by Spin for the model with two DCM Managers and synchronous communication, which is depicted in Figure 6.5, is discussed in Section 6.5.1. We refer to Section 6.5.5 for the discussion whether the HAVi protocol is wrong.

Xtl We can only check that everyone has the same leader identity by checking the parameters of messages/events concerning the final leader. We require the leader identity parameter to be equal for all such actions in stable situations. So the property must express that whenever two actions carry a different leader identity outside a bus reset period, apparently a bus reset event must still be delivered.

We check the following formula:

$$\forall d.\ [l_d]\ \forall \mathbf{G}_{i_1}([l_{\neg d}]\ \exists(\mathbf{T}_{i_2}\mathbf{U}_b\mathbf{T}))$$

where $b = \mathit{BusResetEvent}$

$i_1 = \mathit{Ignore}_1$

$= \neg(\mathit{BusResetEvent} \vee \mathit{BusResetStart} \vee \mathit{BusResetEnd} \vee \mathit{InitReply} \vee \mathit{FinalLeader})$

$i_2 = \mathit{Ignore}_2 = \neg(\mathit{BusResetEvent} \vee \mathit{BusResetStart} \vee \mathit{BusResetEnd})$

$l_d = (\mathit{InitReply} \vee \mathit{FinalLeader})$ with leader identity d

$l_{\neg d} = (\mathit{InitReply} \vee \mathit{FinalLeader})$ with leader identity not equal to d

This property holds only when communication between DCM Managers is synchronous. In the asynchronous case an erroneous initialisation reply may be lingering in someones input queue, after the corresponding bus reset event has been handled by the sender of the erroneous message. In Figure 6.6 an error trace constructed with the `walk_actl` library is listed. The behaviour described by this trace is as follows. In the first bus reset period DCM Manager A is powered up. A is not UrlCapable. A starts the leader election protocol and elects itself for initial leader. In the second bus reset period DCM Manager B is powered up. B is UrlCapable. After the second bus reset, A has not received the bus reset event yet. A elects itself for final leader which completes the leader election. B elects A for initial leader and sends an InitRequest. A receives the InitRequest from the MS and sends an InitReply with its own identity for final leader. Now A receives the bus reset event and starts the leader election protocol anew. B has not received the InitReply from the MS yet and sends a second InitRequest to A. Now B receives the InitReply from the MS and concludes that A is the final leader. A elects itself for initial leader, and receives the second InitRequest that B sent from the MS. A elects B for final leader and sends an InitReply with the identity of B for final leader. The property is violated.

Since we found errors for the Promela models with synchronous communication using Spin, two questions remain, namely whether the error behaviour found with Spin also occurs here and if so, why it is not found with the ACTL formula used. In Section 6.5.1 we mention that it is possible to simulate the Lotos model with two DCM Managers and synchronous

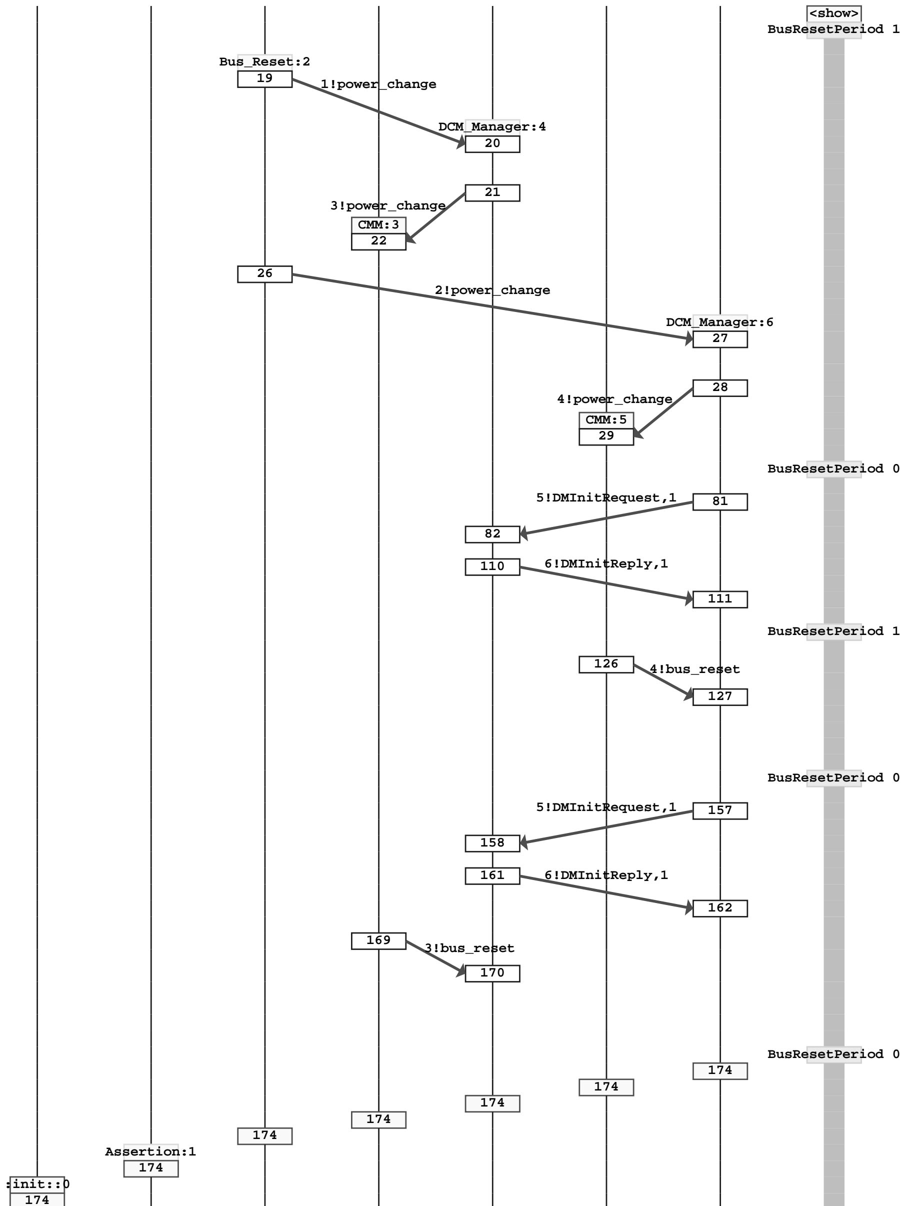


Figure 6.5: The Spin error trace for ‘one leader’ and ‘same final leader’

```
    AG_A(A, F) is FALSE
 0:(0, "GBUSRESET !BUS_RESET_START", 5036)
 1:(5036, "GUPDOWN !1 !POWER_CHANGE", 3437)
 2:(3437, i, 4798)
 3:(4798, "GBUSRESET !BUS_RESET_END !CONSNET(CONSN(TRUE),CONSN(FALSE))", 4797)
 4:(4797, "GINFO !1 !GUID_LIST !CONSNET(CONSN(TRUE),CONSN(FALSE))", 4790)
 5:(4790, "GBUSRESET !BUS_RESET_START", 4789)
 6:(4789, "GEVENT !INIT_LEADER !1", 4769)
 7:(4769, i, 133)
 8:(133, "GUPDOWN !2 !POWER_CHANGE", 142)
 9:(142, "GBUSRESET !BUS_RESET_END !CONSNET(CONSN(TRUE),CONSN(TRUE))", 658)
10:(658, "GEVENT !FINAL_LEADER !1 !FALSE", 150)
11:(150, "GINFO !2 !GUID_LIST !CONSNET(CONSN(TRUE),CONSN(TRUE))", 2542)
12:(2542, "GDMOUT !1 !CONSM(DMINITREQUEST,2,TRUE)", 552)
13:(552, "GDMIN !1 !CONSM(DMINITREQUEST,2,TRUE)", 2524)
14:(2524, "GDMOUT !2 !CONSM(DMINITREPLY,1,FALSE)", 316)
15:(316, "GINFO !1 !BUS_RESET_EVENT", 303)
16:(303, "GDMOUT !1 !EMPTY", 1924)
17:(1924, "GDMOUT !1 !CONSM(DMINITREQUEST,2,TRUE)", 1921)
    Box(A, F) is FALSE
18:(1921, "GDMIN !2 !CONSM(DMINITREPLY,1,FALSE)", 1909)
    AG_A(A, F) is FALSE
19:(1909, "GINFO !1 !GUID_LIST !CONSNET(CONSN(TRUE),CONSN(TRUE))", 1486)
20:(1486, "GEVENT !INIT_LEADER !1", 1662)
21:(1662, "GDMIN !1 !CONSM(DMINITREQUEST,2,TRUE)", 1507)
    Box(A, F) is FALSE
22:(1507, "GDMOUT !2 !CONSM(DMINITREPLY,2,FALSE)", 1548)
    EU_A_B(F, A, B, G) is FALSE
    *Failure.*
```

Figure 6.6: The Xtl error trace for ‘same final leader’

communication and reproduce the error behaviour found by Spin and depicted in Figure 6.5. As to the second question. The ACTL formula used only checks communication involving leader identities. Here we are really hampered by the fact that for the current Lotos models it is not possible to include state information in the formula. It turns out that in the synchronous Lotos models a bus reset event will appear in between of the two events carrying a different leader identity. Since such a pattern is in general not erroneous, it is not possible with this approach to find the erroneous behaviours constructed with Spin. We refer to Section 6.5.1 for a discussion of the behaviour that violates this property, and to Section 6.5.5 for the discussion whether the HAVi protocol is wrong.

6.5.4 Liveness: Eventually there will always be a final leader

Whenever there is at least one DCM Manager active in the network, there should eventually be a final leader. The property we check is whether from each stable state in which at least one DCM Manager is up there exists a path on which no bus reset period starts and a final leader is chosen. It may be argued that this property is too strong since it assumes that there exists a path on which bus reset periods can be delayed until after the election of the final leader. If the environment would violate this assumption, the property would be false even when the protocol was correct. There are two reasons for our approach. First, we know that in our models the

choice between a bus reset period starting and any other activity is non-deterministic. So bus reset periods can be delayed as long as other activity is possible. Second, the alternative property to be checked would be: 'After the handing out of the GUIDList, each path leads to a new bus reset period or a final leader being elected'. This formula requires that during and after the leader election activity, the DCM Managers can perform idle/internal actions indefinitely, in order to distinguish between situations where leader election is interrupted by a bus reset period and situations where leader election does not terminate for some other reason, i.e. livelock rather than deadlock, since in case of a deadlock a bus reset period is forced to start. Moreover, the models already contain a livelock when there are more two initial followers of which one keeps sending InitRequests and the other never gets a turn. The problem with livelocks is that the property should then be checked under certain fairness aspects. This makes the situation increasingly complex, and we have chosen to stick with the first formulation.

Spin The only way to model a liveness property like this and have Spin check its validity, is with an LTL formula. We have been able to express this without too much trouble in ACTL, as can be seen below. However, the expressivity of LTL and branching time logics like ACTL is not comparable [Sti92]. When we try to express the property to be checked in LTL and formulate it as follows, we get an expression which is not in LTL syntax:

$$\Box((\neg bus_reset \wedge (\exists d.\, up(d))) \;\rightarrow\; \exists(\neg bus_reset\, \mathbf{U} \neg bus_reset \wedge \exists d.\, f_leader(d)))$$

Because of the $\exists$ operator, this is not an LTL formula. However, we do need an $\exists$ operator to express the behaviour that the Promela models should have (See also Figure 6.4). The reason is that an LTL formula is interpreted to be true if and only if it holds for each behaviour of the model. So if it is only possible to express desired or undesired properties for one behaviour. But the property that we desire to have is that there always exists a good path. The property that we desire not to have is that there is no state from which there are only bad paths. This cannot be expressed in LTL. This problem has been discussed via e-mail [Dam98, Hol98b], but no solution was found, other than to change the model such that there is a fixed number of bus reset periods, after which the network remains stable. Then Spin's capability to find invalid end states can be used to check that the protocol ends up with a leader, or identify a finite path as undesirable with LTL. A drawback of this approach is that it is not a priori clear how many bus reset periods should be allowed to obtain correctness for the more general model. However, we already found errors in the Spin models for other properties, and in the Lotos models for this property. In the Spin models, errors occur already after two bus reset periods. We have changed all models such that at most two bus reset periods can take place, and added labels to indicate what states in the model are valid end states. Then it turns out that all new models have an invalid end state, which indicates that the protocol ends without electing a final leader even though at least one DCM Manager is up.

In Figure 6.7 the error trace constructed by Spin for the model with two DCM Managers and synchronous communication is listed. This trace describes the following behaviour. In the first bus reset period DCM Manager A is powered up. The first bus reset period is immediately followed by a second, in which DCM Manager B is powered up. A and B are both not UrlCapable. After the end of the second bus reset period, A does not receive the bus reset event yet. Now both A and B start the leader election protocol, in which DCM Manager A is the initial leader and DCM Manager B is the initial follower. B sends A an InitRequest, A computes the final leader which is A, and sends the InitReply to B. B concludes that A is the final leader

Figure 6.7: The Spin error trace for 'always final leader'

which completes the leader election. Now the CMM on the device of A delivers the bus reset event to A, and A starts the leader election protocol anew. A is again initial leader and waits for the InitRequest from B, while B has already completed leader election. Since there is no action possible we are in an end state, and since for A the leader election has not been completed, it is an invalid end state.

As in Section 6.5.1 the error scenario requires that the delivery of a bus reset event message is delayed beyond the duration of the sending and delivery of both a message and a response between different devices, and that the GUIDList is available before the delivery of the corresponding bus reset event. In Section 6.5.1, we argue that this behaviour is allowed by the HAVi specification and hence may occur in HAVi compliant implementations. We refer to Section 6.5.5 for a more elaborate discussion whether the HAVi protocol is wrong.

Xtl We check whether a DCM Manager is up in a stable state by observing the transaction in which the CMM hands out the GUIDList. We check whether a final leader is elected by observing the FinalLeader event. We demand that there exists a path from each GUIDList transaction on which no bus reset period starts and on which a FinalLeader event occurs.

We check the following formula:

$$[g]\ \exists(\mathbf{T}_i \mathbf{U}_f \mathbf{T})$$

where $i = \mathit{Ignore} = \neg(\mathit{BusResetStart} \vee \mathit{FinalLeader})$

$g = \mathit{GetGUIDList}$

$f = \mathit{FinalLeader}$

```
    AG_A(A, F) is FALSE
 0:(0, "GBUSRESET !BUS_RESET_START", 962)
 1:(962, "GUPDOWN !1 !POWER_CHANGE", 72)
 2:(72, i, 1024)
 3:(1024, "GBUSRESET !BUS_RESET_END !CONSNET(CONSN(TRUE),CONSN(FALSE))", 1023)
 4:(1023, "GBUSRESET !BUS_RESET_START", 820)
 5:(820, i, 612)
 6:(612, "GUPDOWN !2 !POWER_CHANGE", 542)
 7:(542, "GBUSRESET !BUS_RESET_END !CONSNET(CONSN(TRUE),CONSN(TRUE))", 288)
 8:(288, "GINFO !2 !GUID_LIST !CONSNET(CONSN(TRUE),CONSN(TRUE))", 97)
 9:(97, "GINFO !1 !GUID_LIST !CONSNET(CONSN(TRUE),CONSN(TRUE))", 335)
10:(335, "GEVENT !INIT_LEADER !1", 231)
11:(231, "GDM !1 !2 !DMINITREQUEST !FALSE", 199)
12:(199, "GDM !2 !1 !DMINITREPLY !1", 995)
13:(995, "GINFO !1 !BUS_RESET_EVENT", 95)
    Box(A, F) is FALSE
14:(95, "GINFO !1 !GUID_LIST !CONSNET(CONSN(TRUE),CONSN(TRUE))", 1003)
    EU_A_B(F, A, B, G) is FALSE
    *Failure.*
```

Figure 6.8: The Xtl error trace for 'always final leader'

This formula does not hold for any of the models.

In Figure 6.8 an error trace constructed with the `walk_actl` library is listed. By coincidence, the behaviour described by this trace is the same as the behaviour described by the error trace found by Spin for this property. See earlier in this section for an explanation of the behaviour. We refer to Section 6.5.5 for the discussion whether the HAVi protocol is wrong.

6.5.5 Is the HAVi protocol wrong?

The error traces given in Figures 6.5, 6.6, 6.7 and 6.8 show that either our model of the protocol or the HAVi specification itself must be wrong.

The error traces indicate that problems occur when the delivery of a bus reset event message is delayed beyond the duration of the sending and delivery of both a message and a response between different devices. In the case of synchronous communication, another cause of problems is the availability of the GUIDList before the delivery of the corresponding bus reset event.

If all assumptions and restrictions that we made in our model are correct, then these scenarios may occur in an implementation that is totally compliant with this version of the HAVi specification, because of two reasons. First, the HAVi specification does not lay down how long messages may be on their way in the system. Second, the delivery of any event has to go through the Event Manager. The Event Manager may cause a delay of the event for several reasons. It is not known how many events the Event Manager may get due to a bus reset period, which need to be delivered, and in what manner these events are processed. Furthermore, there may be many components that listen to the bus reset event and in a sequential approach to delivery of the events, the DCM Manager may very well be the last of them to receive this message.

If our assumptions are not correct, then obviously it is hard to say whether the protocol

would be correct or not. However, all of the assumptions we made are restrictions on configurations or scenarios permitted by the HAVi document which means that we only exclude some HAVi behaviour. So the error behaviour we found would almost certainly be present in a model with fewer restrictions. In fact, the chances are high that with fewer restrictions more erroneous behaviour could be found in the protocol. We already argued in Section 6.5.2 that lifting the restriction that the initial leader chooses itself for final leader when no UrlCapable DCM Managers are present, will lead to violations of the property 'the best candidate becomes final leader'. Other generalisations we could make are: several types of devices in the network, physical IDs that change, bus reset periods that start and end at different moments in different devices, no difference between processing of events and messages, et cetera. Also, it may still be the case that one or more of the software elements used for this protocol have a potential deadlock in their behaviour, and thus prevent the DCM Managers from completing their leader election.

Our conclusion is that for the HAVi leader election protocol to be correct (meaning that any implementation that complies with HAVi works correctly), the HAVi specification should have requirements added on the duration of delivery of events related to the duration of communication between devices. Since the disruption by bus reset periods makes it difficult to establish such requirements, we think the easiest solution is to establish real-time constraints on the duration of sending and processing messages and events, which are realistic for HAVi-compliant implementations. This information should then be checked in a timed formal verification. Since timed model checking is beyond the scope of this experiment, we cannot give an estimate of time bounds that would work, or say whether such time bounds exist.

6.5.6 Statistics

The statistics for model checking the different models with the Spin tool set (version 3.2.4, version 3.3.0 beta-13 May 1999) and the Cæsar/Aldébaran tool set (Cæsar version 5.3, Aldébaran version 6.4, Xtl version 1.1) are given in Tables 6.1, 6.2 and 6.3. All experiments with Spin were done on an SGI IRIX64 6.5 machine with 48 Gbyte of memory. All Cæsar/Aldébaran experiments were done on a SUN Ultra 5_10 SunOS 5.6 machine with 1 Gbyte of memory.

A few remarks are in order.

- All memory entries are in Megabyte. All time entries are in *hours:minutes:seconds* format.
- Spin, Cæsar, Aldébaran and Xtl all generate C code which after compilation performs the state space generation, minimisation and/or exploration.
- The memory numbers mentioned in Table 6.3 indicate the amount of memory used by the verifier generated by Xtl in C code, compiled to executable form. However, C compilation takes at least 6 Mb. For the `walk_actl` library, C compilation takes at least 12 Mb for the models with 2 DCM Managers.
- For the Spin experiments, the memory usage is provided in the output of Spin. Note that this is always a little higher than the memory usage observed with the UNIX command 'top'. For the Cæsar, Aldébaran and Xtl experiments, the memory usage is obtained by observing the outcome of the UNIX command 'top'.

- For all experiments, the timing information is obtained by the UNIX command ‘time’.

- Normally, Lotos state space generation is done with Cæsar in the `.bcg` format, which is very compact. However, Cæsar sometimes creates a state space of much greater size than the corresponding minimal state space under strong bisimulation, and for the models in our case this means that state space generation gets stuck at an unknown portion of the desired total, and fails due to lack of memory. So we turned to an alternative route, and generated the state spaces separately for each instance of each process in the main parallel composition expression. This again is done with Cæsar. The state spaces generated are first minimised with respect to strong bisimulation equivalence (with Aldébaran and the bmin criterion), which is also done in the `.bcg` format. Then these minimised state spaces must be combined into one state space. This is done with Aldébaran and works only if the separate state spaces are in the `.aut` format. The target state space is then also in the `.aut` format. The `.bcg` version is computed and then minimised.

  When generating the state space for one of the communicating processes, often the receipt of a message is not constrained other than by all possible instantiations of the parameters of the communication. This means for instance that when a parameter is of type Natural, that this parameter is instantiated with all constructor values provided by the library for type Natural, when in fact there are only a few values possible in the context of two or three communicating DCM Managers. These parameter values had to be constrained in the separate process definitions to make state space generation manageable. This was done by making a new library for the data types used, and by modifying some conditions on communications, for instance by ruling out the receipt of messages from oneself. Without such constraints, it was not possible to generate a state space for the DCM Manager process with the lowest identity, in the case of asynchronous communication and three DCM Managers.

- All state space generation sizes in Table 6.2 are for a state space in the `.bcg` format, except the *comb network* entries which represent a state space in the `.aut` format. Minimised state spaces are always in the `.bcg` format. In some cases, the `.bcg` version has fewer states for the same state space than the original `.aut` version.

- In Table 6.3, the full state space size is listed for each model. When using the `actl` library, the full state space is explored, even when errors are found. When using the `walk_actl` library, the verification stops after the construction of the first diagnostic trace. We do not know how many states and transitions were explored by `walk_actl` to construct the diagnostic traces.

- The Promela models for 2 DCM Managers are more efficient than the ones with 3 DCM Managers in the sense that they use the data type `bit` instead of `byte` for the Id parameter in the general process `DCM_Manager`.

- With Spin we first tried to explore the whole state space. Whenever an error was found, we reran the verification with a smaller search depth (option ‘-m’ at run time) to see if a smaller error trail could be found. In this way we found the trails reported in Table 6.1, which are the shortest trails we could find. Sometimes the search for a shorter trail

involves the exploration of more states and transitions, due to the order in which the depth-first search is performed.

Only after completing the verification experiments, we learned that option '-DREACH' (to be used at compile time) guarantees a complete search of the truncated state space. This explains why we found a shorter error trail with Spin version 3.2.4 in one case than with Spin version 3.3.0 beta. The '-DREACH' option may increase memory usage and duration of verification experiments. It is very well possible that with this option we would have been able to find the error in the model with three DCM Managers and asynchronous communication for the property 'best final leader' with a much smaller search depth. Without the '-DREACH' option we did not find an error with search depth '-m1000' but ran out of memory.

- Checking the property 'best final leader' for the Promela model with 3 DCM Managers and asynchronous communication was done with the new Spin 3.3.0 beta option '-DSC' to keep the major part of the depth first search stack on disk, and not in memory. Otherwise this experiment would have taken much more memory. The stack file size was 281 Mbyte.

- All experiments with Spin were first done on Promela models in which the global variable `m` was 'hidden', which means that it is not part of the state vector. In this situation Spin did not explore the entire state space. Major parts of the code were unreachable because of using the hidden variable inside two branches of an 'if' statement inside an atomic statement. The predicate 'hidden' should not be used this way but this was not listed in the manuals (it is in the Spin on-line manual now). The difference in semantics between the simulator and the verifier made the situation increasingly unclear, since the parts of the state space that were unreachable to the verifier, were reachable in simulation. Some improvements have been made in Spin 3.3.0 beta to the semantics of the simulator.

- All experiments in Spin were done without partial order reduction by using the compile time option '-DNOREDUCE'. The reason for this is that the use of synchronous communication in the escape guard of an unless command is not compatible with the partial order reduction, hence when using partial order reduction it is possible that error behaviour is missed.

- The error traces produced by Spin can be simulated interactively. The figures in this section are the message sequence charts that were created during such simulation. The figures have been adjusted a little to improve the presentation in black and white. Each thin vertical line in the figure refers to a process in the Promela model, arrows between process lines refers to communication. The thick vertical line refers to the global variable BusResetPeriod in the Promela model. The numbers in the figures refer to steps in the error trail.

- The error traces produced by Xtl were found with the use of the `walk_actl` library. Traces are only produced in case of a universal property that does not hold, or an existential property that does hold. Since we used universal properties, we got traces only in case of an error. The error traces were constructed from end to beginning, and have been reversed in the figures to improve the presentation. The layout of the steps is:

```
<step nr>:(<source state>, <transition label>, <target state>)
```

The transition labels consist of the gate and the offers exchanged at the gate (each offer is preceded by `!`). In between of the steps, messages occur that indicate that a temporal operator from the formula checked does not hold at that point.

one leader						
model	*states*	*trans*	*holds?*	*memory*	*time*	*Spin*
2 sy	18K	93K	F	136	0:00:04	3.2.4
2 as	23K	108K	F	135	0:00:06	3.3.0 beta
3 sy	781K	4.7M	F	161	0:03:59	3.3.0 beta
3 as	2.8M	18M	F	230	0:15:30	3.3.0 beta
best final leader						
2 sy	167K	806K	T	140	0:00:43	3.3.0 beta
2 as	418K	2.1M	T	149	0:01:57	3.3.0 beta
3 sy	44M	279M	T	1767	7:32:22	3.3.0 beta
3 as	194M	3.8G	F	7778	49:43:03	3.3.0 beta
same final leader						
2 sy	16K	77K	F	135	0:00:04	3.3.0 beta
2 as	19K	91K	F	135	0:00:05	3.3.0 beta
3 sy	407K	2.5M	F	148	0:02:05	3.3.0 beta
3 as	1.7M	10M	F	190	0:08:54	3.3.0 beta
always final leader						
2 sy	17K	58K	F	135	0:00:04	3.3.0 beta
2 as	27K	98K	F	135	0:00:06	3.3.0 beta
3 sy	674K	3.2M	F	134	0:03:32	3.3.0 beta
3 as	1.5M	7.5M	F	182	0:07:52	3.3.0 beta

Table 6.1: Spin statistics: state space generation + model checking

6.6 Conclusions

We have modelled the leader election protocol among DCM Manager components in the HAVi architecture, and found that this protocol does not meet some safety requirements and that it does not always converge to a situation with a leader actually elected. The errors are due to the absence of requirements on how long it takes for messages and events to reach their destination. It is expected that if these requirements are added, a formal verification will be able to show whether the restricted protocol works correctly.

6.6.1 Concerning Spin

Using Promela and Spin Promela is an easy language at first, and more difficult at second sight. The basic language constructs have an intuitive meaning, but combining many aspects such as rendez-vous communication, the atomic and the unless construct makes behaviour more fuzzy. The treatment of data is manageable as long as the data is not too involved. In

2 DCM Managers, synchronous								
	generating				minimising			
per process	*states*	*trans*	*memory*	*time*	*states*	*trans*	*memory*	*time*
DM 1	16K	79K	3	0:00:11	32	144	9	0:00:04
DM 2	1.0K	5.4K	3	0:00:03	21	96	4	0:00:02
Bus_Reset	46	59	3	0:00:02	16	24	4	0:00:01
CMM 1,2	12K	55K	3	0:00:09	12	49	5	0:00:02
Other 1,2	2	8	3	0:00:02	1	4	4	0:00:01
comb network	1.5K	5.0K	5	0:00:01	1.2K	4.0K	4	0:00:01
2 DCM Managers, asynchronous								
DM 1	404K	3.0M	28	0:05:12	37	233	183	0:01:18
DM 2	2.1K	18K	3	0:00:05	27	170	4	0:00:01
Bus_Reset	46	59	3	0:00:02	16	24	4	0:00:01
CMM 1,2	12K	55K	3	0:00:09	12	49	5	0:00:02
MS 1,2	2.3K	19K	3	0:00:04	27	159	4	0:00:02
comb network	6.3K	23K	7	0:00:07	5.1K	19K	5	0:00:02
3 DCM Managers, synchronous								
DM 1	2.1M	16M	140	0:43:46	63	474	897	0:08:52
DM 2	177K	1.4M	12	0:02:31	37	255	92	0:00:39
DM 3	4.6K	38K	3	0:00:07	25	174	4	0:00:02
Bus_Reset	186	243	2	0:00:02	40	64	3	0:00:02
CMM 1,2,3	297K	2.1M	79	0:14:00	20	93	139	0:00:49
Other 1,2,3	2	40	2	0:00:02	1	20	3	0:00:01
comb network	58K	247K	47	0:00:52	44K	193K	24	0:00:21
3 DCM Managers, asynchronous								
DM 1	2.0M	11M	109	0:24:29	55	360	620	0:05:33
DM 2	509K	3.8M	31	0:06:52	35	199	233	0:01:36
DM 3	9.8K	105K	3	0:00:13	31	254	10	0:00:03
Bus_Reset	186	243	3	0:00:02	40	64	4	0:00:02
CMM 1,2,3	297K	2.1M	111	0:13:57	20	93	139	0:00:47
MS 1,2,3	3.9K	47K	3	0:00:08	35	279	6	0:00:02
comb network	1.0M	5.2M	358	0:31:12	748K	3.9M	423	0:10:21

Table 6.2: Cæsar/Aldébaran statistics: state space generation

our case, we are clearly overstepping the bounds of the type of model for which Promela was designed.

The graphical interface of Spin is attractive, and it is easy to use. The semantics of the simulator and the verifier have been made more alike recently, which is very important since simulation is often used as a justification for having modelled things right. We are in favour of the semantics being exactly the same for simulator and verifier. After a while, we turned to the command-line use of the tools rather than the graphical interface. This was partly due to the experimental use of Spin on a 64 bit machine.

Expressing safety properties in assertions is very straightforward. Expressing liveness properties in LTL is rather cumbersome and proved impossible in our case, mostly because of the nature of LTL and the nature of the protocol. However, the possibility to track invalid

model	*states*	*trans*	*holds?*	*memory*	*time*
one leader					
2 sy	1.2K	4.0K	T	3	0:00:08
2 as	5.1K	19K	T	3	0:00:13
3 sy	44K	193K	T	4	0:05:11
3 as	748K	3.9M	T	68	25:12:18
best final leader					
2 sy	1.2K	4.0K	T	3	0:00:05
2 as	5.1K	19K	T	3	0:00:08
3 sy	44K	193K	T	4	0:02:17
3 as	748K	3.9M	T	68	10:31:08
same final leader					
2 sy	1.2K	4.0K	T	3	0:00:05
2 as	5.1K	19K	F	3	0:00:11
3 sy	44K	193K	T	4	0:05:51
3 as	748K	3.9M	F	69	29:05:41
error trace same f_leader					
2 as	5.1K	19K	F	5	0:00:26
3 as	748K	3.9M	F	199	15:22:48
always final leader					
2 sy	1.2K	4.0K	F	3	0:00:05
2 as	5.1K	19K	F	3	0:00:08
3 sy	44K	193K	F	4	0:03:28
3 as	748K	3.9M	F	69	18:57:39
error trace always final leader					
2 sy	1.2K	4.0K	F	3	0:00:05
2 as	5.1K	19K	F	3	0:00:08
3 sy	44K	193K	F	6	0:05:32
3 as	748K	3.9M	F	199	14:16:50

Table 6.3: Xtl statistics: model checking

end states was a simple way around this, although it implied changing the models.

Performance of Spin As can be seen in Table 6.1, the performance of Spin is quite good, as long as the number of DCM Managers remains small, and there are no asynchronous channels. We achieved the best performance by using all the advice given in Spin's Help Section on reducing the state space size. Of course, when the communication channels in a Promela model are asynchronous rather than synchronous, the state space grows tremendously because of all possibilities of interleaving the sending and receiving of messages with other activities. Spin uses a partial order reduction technique [HP94] to reduce the model checking effort. This technique identifies transitions as independent and takes only one of the many orders in which these transitions might be explored. The independence criterion holds for transitions that (1) access only local variables, (2) access only communication channels to which the executing process has exclusive read or write access. In our case, we could not use the partial order reduction because we had synchronous communication in the escape guard of an unless

command. If we had been able to use partial order reduction, then we would not have had a great benefit for the following reasons. In our models, most variables have to be used in the verification and are global, and all communication channels for which exclusive read or write access can be guaranteed, are declared as arrays of channels which prohibits the use of the exclusive access declaration construct. The latter is a syntactical restriction for which some escape routes are available, such as the creation of a process where a channel from an array is bound to an ordinary channel, on which the exclusive read or write access can be declared. The other restriction is at the core of the reduction method, and cannot be lifted.

Another important memory usage-increasing factor for our models is probably that, whereas using `atomic` sequences does reduce the number of states, still the number of steps performed in one such `atomic` sequence is reflected in the 'search depth' of the tool. This search depth is limited by the user, and determines the portion of the state space to be explored, the size of the heap that is to be allocated for the search, and hence the amount of memory used for the verification.

What one would like to have (and what might help to improve the performance of Spin tremendously) is to be able to define functions that perform computations without adding to the state space size, and `atomic` sequences to be truly atomic. One would then lose the possibility of exactly tracing down a statement where error situations occur or simulating per statement, but we feel that when using `atomic` sequences, it is fair to not have those possibilities anymore. Since the focus of Spin is on synchronisation and not on computation, there is no plan to improve Spin in this respect [Hol99].

Multi-way synchronisation It is difficult to model multi-way synchronisation in Promela and keep the state space small. Channels are by definition one-to-one, and several processes glancing a global variable or a channel cannot be forced to do this in one atomic action. There is no plan to improve Spin in this respect [Hol99].

Data structures Spin forbids the initialisation of processes with a parameter which is a non-basic data structure, such as an array or record. This hampers the construction of generic models. Recently, the sending of messages with an array as parameter became possible.

Never claims and traces A mixture of 'never claim' and 'trace' processes will probably affect the performance of Spin very badly. Nevertheless, the possibility to use assertions (that reference global state variables) in the 'trace' process seems like a desirable and useful feature for Spin. This is also a planned improvement for Spin [Hol99].

6.6.2 Concerning Cæsar/Aldébaran and Lotos

Using Lotos, Cæsar, Aldébaran and Xtl Lotos is a hard language at first, and a precise language at second sight. It can be hard to grasp the meaning of the language constructs at first, but they have a clear semantics and do not become more complicated when combined. Modelling data is not very hard as long as the data is of a constructive and simple nature. Constructions like sets are not easy to model, but lists are.

The graphical interface of the tool set is easy to use. The simulator has the same semantics as the verifiers, which makes simulation a good means for validation of models. After a while, we turned to the command-line use of the tools rather than the graphical interface.

Expressing properties in an action based logic like ACTL turned out to be quite hard. This is partly due to the nature of the protocol, with bus reset periods disrupting normal behaviour, and partly due to the fact that we cannot use state information in the formulas. Using ACTL, we were not able to find some violations of safety properties which we found with assertions in the Promela models.

Performance of Cæsar For this protocol, the performance of the Cæsar generator is poor. It does not produce the minimal graph under strong bisimulation equivalence, but generates far more states. Judging from the Lotos code and Table 6.2, we think this may be caused by the use of the abstract data types. Perhaps terms which are equal on the basis of the data models are not recognised as such during state space generation.

If it were not for the Aldébaran possibility to compose several communicating components, we would not have been able to construct a complete state space even for 2 DCM Managers. Actually, for the Lotos model with synchronous communication between DCM Managers, and 2 DCM Managers in the network, Cæsar generated about 1.3M states and 2.5M transitions in one hour, and then got stuck due to lack of memory. It is hard to say whether the error traces present in this model, would have been found with a far more restricted model of the protocol.

In order to use the Aldébaran facility of combining state spaces, we had to enumerate some data types, which affected the genericity of the Lotos model. We also had to restrict the possibilities for communication, which proved essential when generating the state space for the asynchronous case with 3 DCM Managers.

6.6.3 Comparison of the tools

Models, state spaces The models in Promela or Lotos are hard to compare. Some tasks can be performed in one `atomic` sequence in Promela (but do increase the size of the verification itself), which takes several atomic actions in Lotos. In Lotos, the data types and process parameters allow for computations being made without state space enlargement. In Promela, most computations must be translated into (parts of) `atomic` sequences. In Promela one would like a little more support for data types and functions. The Lotos models with asynchronous communication and 3 DCM Managers are about as general as they can be. With the current tool support, state space generation becomes impossible with any generalisation of the behaviour.

LTL versus CTL We have found an error in the protocol with an ACTL property which we cannot express in LTL, and which we could only find with Spin by changing the models. The LTL versus CTL issue is the inspiration of many papers and discussions of which we only cite [EL87, KV98, Pnu85, Sti92]. Some attempts have been made at unification of the two approaches (See for instance [KV98]). However, the property that we expressed in ACTL turns out to be a classical example of the difference in expressivity between the two paradigms.

State space sizes The state spaces are smaller for the Lotos models than for the Promela models, when the models are fully explored. On the one hand, this definitely is a flattered view, since generating the state space for a complete Lotos model as such gives tremendously high numbers. On the other hand, the Spin sizes hide the actual number of statements that must be executed to reach a certain state. Because of the `atomic` predicate, the number of

statements may be much higher. This does not affect the state space size, but it does affect the amount of memory used for the verification.

When errors are found, Spin stops immediately, hence explores only part of the state space. In Xtl, the library `actl` always explores the full model. The library `walk_actl` stops immediately when a diagnostic trace is constructed.

Memory usage when model checking It turns out that we needed much less memory for the Xtl verifications than with the Spin tool, which is probably due to the state space sizes being larger for Promela, and `atomic` sequences consisting of more steps causing more memory to be used than one statement. When verifying a property with the `walk_actl` library, much more memory is used than with the `actl` library, which we think is due to backtracking and overhead for the diagnostic trace.

Size of generated code The size of C code generated by Spin is manageable considering the state space size. For state space generation from Lotos models, the C files become larger. Finally, large state spaces cause Xtl to generate very large C files in which very many variables are allocated (a stack size greater than 2 Gbyte).

Expressing the properties to be checked The properties verified with Spin and with Xtl are not comparable. In Spin we used assertions (and tried in vain to use LTL) in terms of state variable values. In Xtl we used ACTL properties in terms of observable actions.

We would like to use state information from the Lotos process parameters in the properties to be verified with Xtl.

In Spin one would like to reference the values of state variables in a `trace` process, where the occurrences of communications can be checked. The combination of these features, which is as yet forbidden, would be very useful. This is a planned improvement.

Comparing model checking times When errors can be found, Spin is overall faster than Xtl except when the models become very large. For full state space exploration, Xtl is faster, probably due to the state space sizes. It should be noted that Spin builds the state space anew during exploration whereas Xtl checks properties on state spaces that have already been built, so in this case one should add the state space generation times to the model checking time. Both approaches have advantages and drawbacks in terms of efficiency.

Tailoring models for model checking We had slightly different Promela models depending on whether we were checking properties concerning final leadership, properties concerning any type of leadership, or the property which we could not express in LTL. In the former two cases, differences were only in the variables used for observation. In the latter, a fundamental change was made to the environment behaviour by having maximally two bus reset periods instead of arbitrarily many. If we had used one general model for all properties, we would have had a much larger state space. This was not necessary with the Lotos model because the experiments there were based on observing actions rather than state variable values and it was possible to express all properties in ACTL. The addition of the events that signal the election of a leader in the Lotos models do not seem to enlarge the state space size as much as the state variables in the Promela models do.

Efficiency of model checking When verifying an ACTL property with the `actl.xtl` library, Xtl visits all reachable states, thus verification does not stop as soon as the property is found to be false, and it cannot become true anymore, or vice versa. When using the `walk_actl.xtl` library, Xtl will stop as soon as a diagnostic trace has been constructed. This will be a trace showing truth in the case of an 'exists' property, and it will be a trace showing falsity in the case of a 'for all' property.

Spin uses partial order reduction [HP94] to improve efficiency. We already mentioned that a small change in the Promela syntax accepted by Spin can increase the benefit of this reduction technique. In our case the partial order reduction cannot be exploited because of the combination of rendez-vous communication and unless constructs. This may be a consideration when constructing models.

Spin stops the verification as soon as an error is found. A diagnostic trace leading to the error situation is presented to the user. The trace may reveal the falsity of the property to be checked, but also a dynamic error because an array index is out of range, et cetera.

6.6.4 Concerning this experiment

It appears that the combined approach of having different models of the same protocol and different verification techniques, gives better results, for several reasons:

1. The restrictions of the different modelling languages force one to think carefully about how to model all the aspects of the protocol.

2. The different verification techniques enable establishing different kinds of properties for the protocol.

3. One approach acts as a debugger for the other, in the sense that

   - Mistakes at the syntactic or semantic level are generally not made in the exact same manner during the different modelling efforts.
   - Results can be checked in two different situations.
   - Negative results obtained on one side and not on the other can still be 'checked' by simulating with the counterexample, and validating whether the error behaviour is also present in the model for which this could not be verified.

Thus, the results are more convincing than when only one modelling/verification approach is applied.

Chapter 7

A timed verification of the IEEE 1394 leader election protocol

Summary

The IEEE 1394 architecture standard defines a high performance serial multimedia bus that allows several components in a network to communicate with each other at high speed. In the physical layer of the architecture, a leader election protocol is used to find a spanning tree with a unique root in the network topology. If there is a cycle in the network, the protocol treats this as an error situation. This chapter presents a formal model of the leader election protocol in the language IOA as well as a correctness proof. The verification shows that under certain timing restrictions the protocol behaves correctly. The timing constants proposed in the IEEE 1394 standard documentation obey the requirements found in this proof.

7.1 Introduction

The IEEE 1394-1995 serial bus standard [IEE96] defines an architecture that allows several components to communicate at very high speed. Originally, the architecture was designed by Apple (FireWire). Currently, more than 70 companies are involved in the standardisation effort. Although the IEEE 1394-1995 standard has been finalised, the architecture is still being refined and adapted. Part of this ongoing work is reflected in the IEEE P1394a standard proposal document [IEE99], which is intended to be a supplement to IEEE 1394-1995. In this chapter, 1394 will refer to IEEE 1394-1995 unless otherwise stated.

The IEEE 1394 standard allows several components to be connected either with cables and IEEE 1394 chips (cable environment), or with an IEEE 1394 backplane in one physical device (backplane environment). We restrict our attention to the cable environment situation, and refer to the whole of components, cables, etc. as *the network*.

Like in the OSI model, the IEEE 1394 architecture has several layers of which the physical layer is the lowest. This layer takes care of the actual communication on the bus, which happens by sending signals on a wire by asserting voltages. The physical layer is responsible for the knowledge that a component has of the network topology and of components present,

and for issues such as timing of asynchronous and synchronous communication and arbitration for use of the bus. These tasks are taken care of in several phases.

The first phase in the physical layer is the *bus reset phase*, which is entered whenever a component is powered up, when the network topology changes or an error is discovered, or on request of higher layers in the architecture. After completion of the bus reset phase, the *tree identify phase* starts. In the three identify phase the network topology is determined by spanning a tree in the network. The root of the tree will act as the bus master. After the tree identify phase, the *self identify phase* follows in which all components inform the rest of the network of their capabilities and get a unique ID. Finally, in the *normal operation phase*, the arbitration for and actual use of the bus by higher layers and applications takes place.

In this chapter we study the tree identify phase in the physical layer. The components employ a leader election protocol to span a tree in the network, with the root acting as the leader. A side effect of the protocol is that it detects whether there is a cycle in the network, and if so, does not terminate with a leader but halts in the initial phase of the protocol and issues error messages. Our intention is to prove that an abstraction of the protocol, which is as close as possible to the description in the IEEE 1394 documents [IEE96, IEE99], works correctly. There already are some correctness proofs for other abstractions of this protocol [DGRV97, GV98, SV99, SZ98]. We reuse part of this work for proving the correctness of our model of the protocol. This is done by establishing an implementation relation between the most detailed model from [GV98] and our more detailed model of the protocol. In this way, our verification adds to a stepwise refinement of IEEE 1394 in which more detail is added to models in each step.

The verification is carried out by establishing timed trace inclusion between timed I/O automata through a timed refinement [LT87, LT89, LV96]. The I/O automata are presented in the IOA language [GLV97]. We reuse an untimed I/O automaton from [GV98] to which we add a harmless time-passage action to turn it into a timed I/O automaton and use timed refinements as presented in [LV96]. As mentioned in [LV96], we could equally well establish an untimed refinement between the timed I/O automata, so timed trace inclusion follows if the time-passage action is visible in both models. Some related work that is interesting in the timed vs. untimed respect is the work presented in [Sch97], which discusses safety and failure refinements between timed and untimed CSP models [DS95]. Some results are presented for failure refinements between communicating processes, which may be useful in the I/O automata setting.

The proofs show that under the assumptions made, the behaviour of the models is correct when we use the timing constants proposed in IEEE 1394-1195 and IEEE P1394a. It still remains to be seen whether further refinement of the models preserves the correctness.

This chapter is organised as follows. Section 7.2 explains the IEEE 1394 tree identify, discusses related verifications and presents our abstraction. Section 7.3 introduces our I/O automata models of the tree identify protocol and shortly discusses the IOA language. Section 7.4 is an intermezzo about network topologies, in which general results are derived, which we need in the verification. Section 7.5 presents the formal verification of the protocol. In Section 7.6 we sum up the conclusions that can be drawn from this exercise.

Note that to improve readability, we often use Lamport's list notation [Lam94b] for conjunction or disjunction in formulas.

7.2 The protocol

In this section, the IEEE 1394 tree identify phase is described, other verifications of this protocol are discussed, and our abstraction is introduced. The IOA models are presented in Section 7.3. The tree identify phase has already been described in several articles. The following text and pictures are borrowed from [DGRV97].

7.2.1 The IEEE 1394 tree identify phase

We refer to the components connected to 1394 bus as *devices*. Each device has a number of *ports*, which are used for bidirectional *connections* to other devices. Each port has at most one connection. The device at the other side of the connection is called the *peer* device. The tree identify phase follows on completion of the bus reset phase, which is started as soon as a total reset of the network is demanded. This can occur on request of applications, or because the network configuration has changed or an error situation has been detected. The bus reset phase clears all topology information except local information on a device, namely which ports have connections. During the tree identify phase a spanning tree is constructed in the network. After the tree identify phase completes, the tree structure will be used in the normal bus operation. An example of a network topology at the start of the tree identify phase is presented in Figure 7.1.

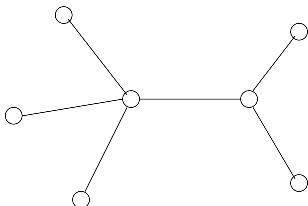


Figure 7.1: Initial network topology

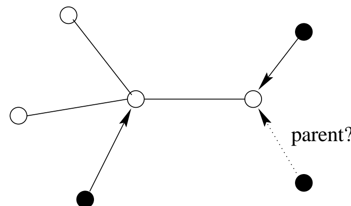


Figure 7.2: Intermediate configuration

Informally, the basic idea of the protocol is as follows: each device starts in the initial phase, in which it may receive a "parent request" on from a peer device on one of its ports. The receiving device then sends an acknowledgement message to the peer device and adds the port to its collection of children. A peer device which is connected to the child port, is then considered to be a child in the tree structure (See Figure 7.2). When a device is in the initial phase and has no more than one port left on which no communication has taken place yet, it can send a parent request on that port and leave the initial phase. It is obvious that leaf devices (i.e. devices with a single connected port) have exactly one such port at the start of the protocol, so they can send their parent request and leave the initial phase immediately. In this manner, a tree is constructed that grows from the leaves inward, until all ports of one device are children, and that device is the *root* of the tree (See Figure 7.4).

It is possible that two devices end up asking each other to be the parent. This situation is called "root contention". The devices both signal the reception of a parent request on a port on which they already sent a parent request, and turn to a symmetry breaking protocol in which random bits are used (See Figure 7.3). This root contention protocol has been formally

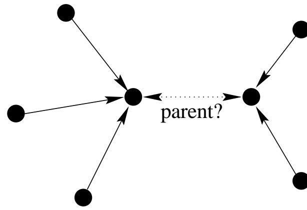


Figure 7.3: Two contending devices

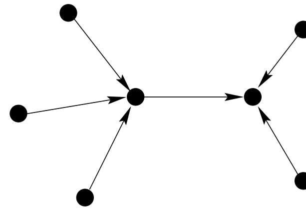


Figure 7.4: Final spanning tree

specified and verified in [SV99].

When a cycle is present in the network, all the devices that are on such a cycle will not get a parent request from their peers on the cycle. So they will have more than one port on which no parent request was received, and can therefore not send a parent request themselves or leave the initial phase. Devices that are not on a cycle, but are wedged between two or more cycles will not get a parent request either on at least two ports, and will not send a parent request themselves or leave the initial phase. Such a situation is solved by a timer, which is started at the start of the tree identify phase, and which is supposed to expire only in the situation of a cycle in the network. When there is a cycle in the network, a root should not be elected, since the operation of the bus in the following phases relies on the topology being a tree structure.

A device may influence its own chances at becoming root by waiting for some time before sending the parent request, even if it is already possible to proceed. A device will only do so if it has the flag force_root set to true.

Devices may enter the tree identify phase at different times. This is due to the difference in the moments at which different devices signal that the bus reset phase (preceding the tree identify phase) should be entered.

7.2.2 Other verifications of the protocol

Parts of the IEEE 1394 architecture have been formally specified and/or verified in several articles [DGRV97, GV98, KHR97, Lut97, SM98, SV99, SZ98]. Of these, [KHR97, Lut97, SM98] focus on the link layer. The articles [DGRV97, GV98, SV99, SZ98] study the tree identify phase of the physical layer, like we do. In Figure 7.5 we give an overview of the results of these articles, and their relation to the research presented here. The results of the different articles are in the dashed boxes. The names of the formal models are listed, arrows between these indicate a (proved) implementation relation. The vertical position of a model name indicates the level of abstraction of that model with respect to the IEEE 1394 documentation. Very abstract models do not consider implementation details such as timing, signals etc. The most detailed models incorporate more detail from the IEEE 1394 documentation. In the picture, we have given some models the same vertical position to indicate that they have a comparable degree of detail. We now explain the results of each article in short.

Devillers, Griffioen, Romijn and Vaandrager [DGRV97] have shown that the election in the tree identify phase works correctly, under the assumption that there are no cycles in the network, that the network topology is fixed throughout the protocol, that a root contention situation is solved in one atomic step, and that no device tries to become root by having the

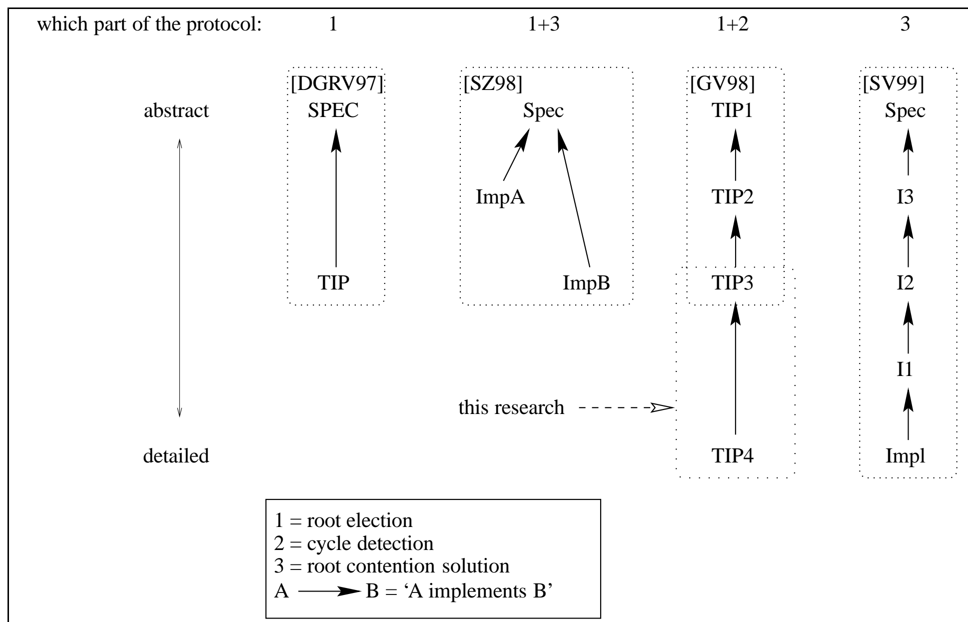


Figure 7.5: An overview of research on the IEEE 1394 tree identify phase

corresponding force_root flag set to true. The models are at a high level of abstraction: there is no timing and communication is modelled with finite queues. The models are I/O automata [LT87, LT89] presented in a precondition/effect style. The proofs use invariants and simulation techniques from [LV95]. The proofs have been checked with the theorem prover PVS [ORSH95].

Shankland and van der Zwaag [SZ98] have also shown that the election in the tree identify phase works correctly, under the assumption that there are no cycles in the network, that the network topology is fixed throughout the protocol, and that no device tries to become root by having the corresponding force_root flag set to true. The models are at a high level of abstraction: there is no timing and communication is modelled with finite queues. The models are presented in μCRL [GP95], a process algebra language with data. The proofs use invariants and the cones and foci method from [GS]. Note that the paper gives no proof that the root contention protocol actually terminates within bounded time, since for the verification it is enough to show that it *can* terminate.

Griffioen and Vaandrager [GV98] have shown that the election in the tree identify phase works correctly, under the assumption that the network topology is fixed throughout the protocol, that a root contention situation is solved in one atomic step, and that no device tries to become root by having the corresponding force_root flag set to true. The models are at a high level of abstraction: there is no timing and communication is modelled with finite queues. The models are I/O automata [LT87, LT89] presented in the IOA language [GLV97]. The paper introduces a new simulation proof technique, called *normed simulations*. The proofs use invariants and the proposed simulation technique. The proofs have been checked with the

theorem prover PVS [ORSH95]. Note that cycle detection is done with a predicate that takes the structure of the whole network into account, and does not use timing information, as in IEEE 1394. The predicate used implies that nodes that are part of a cycle will detect this with an error message. In IEEE 1394 (and in the models presented here), the error situation is also detected by nodes that are not part of a cycle themselves, but wedged in between of two cycles.

Stoelinga and Vaandrager [SV99] have shown that the root contention solving protocol in the tree identify phase works correctly under the assumption that the network topology is fixed throughout the protocol. The models are at an intermediate level of abstraction: on the one hand timers and probabilities are used, but on the other hand communication is modelled with finite queues. The models are probabilistic timed I/O automata [Seg95, SL95] presented in the IOA language [GLV97]. The paper introduces two simulation proof techniques, which are special cases of the simulation techniques in [Seg95, SL95]. The proofs use invariants and the proposed simulation techniques.

The model of the protocol that is presented in [DGRV97] is essentially the same as one of the I/O automata examples in the book *Distributed Algorithms* by Lynch [Lyn96]. A correctness proof of this protocol is not given in [Lyn96]. The models that either include cycle detection or the root contention protocol can be considered refinements of the protocol in [Lyn96].

7.2.3 This verification

As can be seen in Figure 7.5, we aim to give an implementation relation between the most detailed model from [GV98] and a more detailed model. In this way, our verification adds to a layered verification of IEEE 1394 in which models are refined, that is, more and more detail is added in each step. In order to keep our proof obligations manageable, we do not add too much detail, and hence our model has an intermediate degree of detail with respect to IEEE 1394.

The verification is carried out by establishing trace inclusion between timed I/O automata through a refinement [LT87, LT89, LV96]. The I/O automata are presented in the IOA language [GLV97].

The most detailed model of [GV98] is an untimed model. This means that the cycle detection is done with a predicate that takes the structure of the whole network into account. In this verification, we want to establish that cycle detection based on the timing in IEEE 1394 works correctly. In order to do this, we add timers to the model which expire when the leader election takes too much time. We also add timing information to the messages sent, in order to model the delay in communication in IEEE 1394. As argued above, we use a different predicate for cycle detection than the one used in [GV98], in order to conform to the error behaviour of IEEE 1394. As in [GV98] we assume that the network topology is fixed throughout the protocol, that a root contention situation is solved in one atomic step that no device has the force_root flag set to true, and that communication can be modelled with finite queues.

Since our aim is to show that whenever timers in the model expire, there is indeed a cycle in the network, and that the timers will expire in case of a cycle in the network, we are trying to show that the timers do not expire too soon or too late. In our proofs we use invariants that express worst case scenarios in terms of delay. So we are actually performing a worst case analysis on the timing proposed in IEEE 1394. In this way, we establish a relation between the parameters of the protocol in terms of minimal and maximal values.

We expect that in a next refinement step it is possible to include the result from [SV99], to get closer to the IEEE 1394 behaviour without much effort. The next refinement step could then be to add a force_root flag to the model, thus expressing that devices behave a little different to increase their chances at leadership. In order to obtain a correctness statement about IEEE 1394 with all its detail, it still has to be shown that modelling the IEEE 1394 communication of voltages on wires by messages and finite queues is correct. We expect that in this situation, we will not just have a judgement on correctness, but we will also be able to say how the timing constants in IEEE 1394 could/should be adjusted.

Our assumptions As a specification of the desired behaviour, we have taken the most detailed model TIP3 from [GV98]. In [GV98] it is shown that the behaviour of TIP3 meets the requirements for the tree identify phase.

TIP3 is a very abstract model of the tree identify phase, in the sense that it abstracts from a lot of details. We introduce a model TIP4, which is more detailed than TIP3, and prove that it is a refinement of TIP3. In this way, the correctness of the behaviour of TIP4 can be derived from the correctness of the behaviour of TIP3.

Our justification for still leaving out many implementation details that may affect the correctness of the protocol, is that we intend to reuse as much as possible of the proofs already established. This can only be done in a manageable way if we do not add too many details at once. As it is, the proofs for our verification are already quite lengthy and involved. See also Section 7.6 for a discussion of our results.

The abstractions have been chosen as follows.

- In TIP3, it is assumed that the devices signal a cycle by merely checking the network topology. In TIP4, the devices use a timer, which conforms to IEEE 1394.
- In both TIP3 and TIP4, communication between devices is modelled by sending messages on queues. In a IEEE 1394 network, the devices communicate by asserting signals (defined in terms of voltages) on wires for a certain time.
- In both TIP3 and TIP4, it is assumed that no device has the force_root flag set to true.
- In both TIP3 and TIP4, the network is assumed to be connected and to be fixed throughout the protocol. There may be cycles in the topology.
- In both TIP3 and TIP4, the root contention situation is solved in one atomic step, as opposed to the IEEE 1394 protocol which involves picking random bits, and which repeats until the symmetry is broken. Note that the root contention protocol has been formally specified and proved correct in [SV99].
- In both TIP3 and TIP4, all devices enter the tree identify phase at the same time.
- In TIP3, no timing is used whatsoever. In TIP4, timing is used for determining whether the network topology contains a cycle (see above), and for determining the actual delivery time of messages. The IEEE 1394 delay between the moment of sending and reception and processing of a signal is caused by difference in clocks of the devices, the length and propagation delay of the wires, and the difference in the tree identify phase enter moment of the sending and receiving device. In TIP4, the delay of message is

determined at the moment that the message is being received. This delay may vary between the bounds caused by difference in clocks of the devices, and by the length and propagation delay of the wires. Although the second factor is constant, we have modelled the choice of delay to be completely free for each receive operation. Since we are after the bounds on the timing constants in relation to the network topology with respect to detecting cycles, we are establishing the property that the cycle detection timer will not expire too soon or too late. Therefore we are actually performing a worst case analysis. The worst case scenarios for IEEE 1394 and our model are the same, under the assumption that all devices enter the tree identify phase at the same moment.

7.3 IOA models

We present two models in the IOA language [GLV97] of the tree identify protocol, namely TIP3 and TIP4. The IOA model for TIP3 comes (almost) literally from [GV98] and gives an abstract and untimed model of the protocol behaviour. It has been shown in [GV98] that this model has the desired behaviour of electing exactly one device for root if there is no cycle in the network. If there is a cycle in the network, all devices that are part of this cycle will detect this and give an error message.

The IOA language The IOA language facilitates precise and readable descriptions of I/O automata [LT87, LT89]. Since our models are timed, we have added a **time** action, according to the definition in [LV96].

IOA contains the basic types Bool, Nat, Int and Real with their standard operators. In addition type constructors Array, Seq (finite sequences) and Set (finite sets) are part of the language. The notation __[__] is used for array subscripting, an array with a value e in all cells is denoted by const(e). The operation $__\vdash__$ appends an element at the end of a sequence and the operations head and tail have the usual meaning. We assume the type Time which is the (predefined) type Real restricted to nonnegative values.

We assume the extra types Mes to represent the different message contents that may be exchanged between devices, as follows:

Type Mes **enumeration of** parent, ack

In Section 7.4 we give several definitions and operations that concern network topologies. Given a network $N = \langle D, P, \text{dev}, \text{peer}\rangle$, we assume the types Dev=D and Port=P and all operations as defined in Section 7.4.

The TIP models The signature part for both models is shown in Figure 7.6. The connected network $N = \langle D, P, \text{dev}, \text{peer}\rangle$ is a parameter for both models. In addition, the constants MinDelay, MaxDelay, MinLpdtime, and MaxLpdtime are parameters for TIP4. We assume MinDelay $\leq$ MaxDelay and MinLpdtime $\leq$ MaxLpdtime. Any message sent at time t, arrives in the interval $t +$ [MinDelay, MaxDelay]. If a loop is signalled, then this happens in the interval [MinLpdtime, MaxLpdtime].

The IOA description of TIP3 is shown in Figure 7.7. The action definitions are almost equal to those of TIP4, so we refer to the explanation below. The model TIP3 comes (almost) literally from [GV98]. The first change is the addition of the time action, whose precondition

signature
  internal childrenknown(d: Dev),
    addchild(d:Dev, p:Port),
    receivemes(d:Dev, p:Port, m:Mes),
    solverootcontent(d:Dev, p:Port)
  output root(d:Dev),
    loopdetect(d:Dev)
  time δ

Figure 7.6: Signature for TIP3 and TIP4.

is true, and whose effect is empty. The second change is the use of the oncycle predicate, which recognises not just devices that are on an ordinary cycle, but also devices that are on a path between two cycles (see Section 7.4). Our verification shows that these devices also detect a cycle in the protocol and give an error message (see property I_{12} in Definition 7.10, Section 7.5.1).

The IOA description of TIP4 is shown in Figure 7.8. The model TIP4 is a proper timed IOA model: there is a state variable time which is used as a global clock, and per message queue there is a variable delay that is reset for each message sent on the corresponding queue. A message is available at least after MinDelay time units have passed or ultimately after MaxDelay time units have passed. The condition for detecting a cycle in the network also depends on time, and not (as in TIP3) on the predicate oncycle which is based on the structure of the network. It is our goal to show that cycle detection will occur if and only if there really is a cycle present in the network.

We now give a short explanation of each action of TIP4. Whether a device is in the initial phase is reflected in the state variable init. When init is true, only actions addchild and childrenknown can be enabled. With addchild a parent request may be received (if the value of delay indicates that the parent request is available) and the corresponding port is added to the collection of children. The action childrenknown marks the end of the init phase. It can only be performed when there is at most one port left which is not a child port, and when it is performed, an acknowledgement is sent to all peer devices that are connected to a child port and a parent request is sent to the peer device connected to the port that is not a child, if any. If a device is on a cycle, then it does not ever reach the state in which childrenknown is enabled, because two of its ports are connected to peer devices which are also on a cycle. In this situation, the action loopdetect should be performed. In TIP3, the cycle is detected with the oncycle predicate. In TIP4, a timer signals that the device stays in the init phase too long, and therefore must be on a cycle.

As soon as a device has left the init phase, it must wait for a message on the one remaining port that is not a child. If there is no such port, then the device is the root of the tree, and can perform the root action. If there is such a port, then the action receivemes can be performed as soon as the message is available. The expected message is an acknowledgement, after which the contribution of the device to the leader election is over. If an unexpected parent request is received, then the device is in root contention with the peer device that sent the parent request.

```
automaton TIP3
  states
    child: Set[Port] := {}
    mq: Array[Port,Seq[Mes]] := const({})
    init: Array[Dev,Bool] := const(true)
    rc, root, lpd: Array[Dev,Bool] := const(false)
  transitions
    internal childrenknown(d)
      pre  init[d] ∧ size(ports(d)-child) ≤ 1
      eff  init[d] := false;
           for p in ports(d) do if p ∈ child
                                then mq[p] := mq[p] ⊢ ack
                                else mq[p] := mq[p] ⊢ parent fi od
    internal addchild(d,p) where d = dev(p)
      pre  init[d] ∧ head(mq[peer(p)]) = parent
      eff  child := insert(p, child);
           mq[peer(p)] := tail(mq[peer(p)])
    internal receivemes(d,p,m) where d = dev(p)
      pre  ¬ init[d] ∧ ports(d)-child = {p} ∧ head(mq[peer(p)]) = m
      eff  if m = parent then rc[d] := true fi;
           mq[peer(p)] := tail(mq[peer(p)])
    internal solverootcontent(d,p) where d = dev(p)
      pre  rc[d] ∧ rc[dev(peer(p))]
      eff  child := insert(p,child);
           rc[d] := false;
           rc[dev(peer(p))] := false
    output root(d)
      pre  ¬ init[d] ∧ ¬ root[d] ∧ ports(d) ⊆ child
      eff  root[d] := true
    output loopdetect(d)
      pre  oncycle(d) ∧ ¬ lpd[d]
      eff  lpd[d] := true
    time δ where δ > 0
      pre  true
      eff
```

Figure 7.7: Automaton TIP3.

```
automaton TIP4
  states
    child: Set[Port] := {}
    mq: Array[Port,Seq[Mes]] := const({})
    delay: Array[Port,Time] := const(0)
    init: Array[Dev,Bool] := const(true)
    rc, root, lpd: Array[Dev,Bool] := const(false)
    time: Time := 0
  transitions
    internal childrenknown(d)
      pre  init[d] ∧ size(ports(d)-child) ≤ 1
      eff  init[d] := false;
           for p in ports(d) do delay[p] := 0;
                                if p ∈ child
                                then mq[p] := mq[p] ⊢ ack
                                else mq[p] := mq[p] ⊢ parent fi od
    internal addchild(d,p) where d = dev(p)
      pre  init[d] ∧ head(mq[peer(p)])=parent ∧ delay[peer(p)] ≥ Mindelay
      eff  child := insert(p, child); mq[peer(p)] := tail(mq[peer(p)])
    internal receivemes(d,p,m) where d = dev(p)
      pre  ∧ ¬ init[d] ∧ ports(d)-child = {p}
           ∧ head(mq[peer(p)])=m ∧ delay[peer(p)] ≥ Mindelay
      eff  if m = parent then rc[d] := true fi;
           mq[peer(p)] := tail(mq[peer(p)])
    internal solverootcontent(d,p) where d = dev(p)
      pre  rc[d] ∧ rc[dev(peer(p))]
      eff  child := insert(p,child);
           rc[d] := false; rc[dev(peer(p))] := false
    output root(d)
      pre  ¬ init[d] ∧ ¬ root[d] ∧ ports(d) ⊆ child
      eff  root[d] := true
    output loopdetect(d)
      pre  init[d] ∧ ¬ lpd[d] ∧ time ≥ MinLpdtime
      eff  lpd[d] := true
    time δ where δ > 0
      pre  ∀ d,p:
           ∧ ¬ pre(childrenknown(d)) ∧ ¬ pre(root(d))
           ∧ if init[d] ∧ ¬ lpd[d] then time+δ ≤ MaxLpdtime fi
           ∧ if mq[p]≠{} then delay(mq[p])+δ ≤ MaxDelay fi
      eff  time := time+δ
           for p in Port do delay[p] := delay[p]+δ od
```

Figure 7.8: Automaton TIP4.

The peer device has received or will receive the parent request that was sent earlier, and thus has signalled or will signal the root contention. As soon as both devices have signalled root contention, the action solverootcontent can be performed to break the symmetry and add one of the two ports involved to the child collection. The device whose port is added to child can then perform the root action.

The time action signals the passing of time, by increasing the value of time. Time passage may not occur if there are other actions that cannot be delayed any further. Actions children-known and root are urgent, which means that they should happen at the first moment when they are enabled. Actions addchild and receivemes are also urgent, but they are enabled only when a message becomes available. Since the message is available only when the value of delay is in the interval [MinDelay, MaxDelay], we require that the value of delay does not pass beyond the right-hand border of this interval. The action loopdetect depends on the value of time and can happen anywhere in the interval [MinLpdtime, MaxLpdtime], so we require that time does not pass beyond MaxLpdtime. The only action that is not mentioned in the pre-condition of the time action, is solverootcontent. The reason for this is that in the IEEE 1394 documentation, there is a small sub-protocol with timers that is used to break the symmetry, instead of the one action that represents this sub-protocol in TIP4. Since this sub-protocol is not guaranteed to end in finite time (due to randomly drawn bits), we cannot say at what time the action solverootcontent will take place. Hence we have put not requirement on the time action for solverootcontent. The root contention solving protocol is discussed and proved correct in [SV99].

7.4 Network preliminaries

This section gives some definitions and properties of network topologies which are needed in the verification.

7.4.1 Networks

Definition 7.1 A *network* is a quadruple $\langle D, P, \text{dev}, \text{peer}\rangle$, where

- D is a non-empty set of devices.
- P is a set of ports.
- $\text{dev} : P \to D$.
- $\text{peer} : P \to P$ with for all p: $\text{peer}(\text{peer}(p)) = p$ and $p \neq \text{peer}(p)$.

For $d \in D$, we define the abbreviation $\text{ports}(d) = \{p \in P | \text{dev}(p) = d\}$.
Given D' and $d \in D'$, the predicate $\text{leaf}(D', d)$ holds iff $\forall p_1, p_2 \in \text{ports}(d) : \text{dev}(\text{peer}(p_1)) \in D' \wedge \text{dev}(\text{peer}(p_2)) \in D' \to p_1 = p_2$.

The network consists of a collection of devices, each of which has a set of ports. Each port is connected to one other port with a cable, which is captured by the function peer. Each port has a connection and no port is connected to itself. The cable connection itself is referred to as a *cable hop*. Since for each $p \in P$, $\text{dev}(p)$ is defined, it follows that $P = \bigcup_{d \in D} \text{ports}(d)$.

Throughout this paper, we fix a network $N = \langle D, P, \text{dev}, \text{peer}\rangle$ and let variables p, p', p'', $p_0, \ldots$ range over ports in P, and $d, d', d_0, \ldots$ over devices in D.

7.4.2 Paths, cycles

The following definitions and lemmas are necessary to identify paths, cycles, etc. in the network.

Definition 7.2 A *path* π is a non-empty sequence of ports $\pi = p_0 p_1 \ldots p_n$, such that:

- n is odd
- $p_0 \neq p_n$
- for all $i > 0$, if i is odd then $p_{i-1} = \text{peer}(p_i)$ else $\text{dev}(p_{i-1}) = \text{dev}(p_i)$)

We denote the first and last port of π with $\text{first}(\pi) = p_0$ and $\text{last}(\pi) = p_n$. We denote the length of π with $\text{length}(\pi) = (n+1)/2$. We denote the path obtained by reversing π with $\text{reverse}(\pi) = p_n \ldots p_0$.
Path π is a path *from* d_1 *to* d_2 if $\text{dev}(\text{first}(\pi)) = d_1$ and $\text{dev}(\text{last}(\pi)) = d_2$. We say that a device d *is on* π iff there is a port p in π such that $d = \text{dev}(p)$. A *cycle* is a path $\pi = p_0 \ldots p_n$ such that $\text{dev}(p_0) = \text{dev}(p_n)$.
The predicate $\text{oncycle}(p)$ is true iff there is a cycle such that p is on it. The predicate $\text{oncycle}(d)$ is true iff there is a port $p \in \text{ports}(d)$, such that $\text{oncycle}(p)$ holds.

A path reflects a walk through the network by the concatenation of cable hops, in which a $p_1 p_2$ cable hop may not be followed immediately by the reverse hop $p_2 p_1$. The length of a path is the number of cable hops included in that path. A cycle may include a path π which is wedged in between smaller cycles ρ and τ, resulting in the shape of a pair of glasses: $\rho\pi\tau\text{reverse}(\pi)$. Ports that are part of a cycle (of whatever shape) remain inactive during the protocol, as we will show later.

Lemma 7.3 If $\pi = p_0 p_1 p_2 \ldots p_n$ is a cycle, then $p_2 \ldots p_n p_0 p_1$ and $\text{reverse}(\pi)$ are also cycles.

Lemma 7.4 $\text{oncycle}(p) \rightarrow \text{oncycle}(\text{peer}(p))$

Lemma 7.5 $\text{oncycle}(p) \rightarrow \text{size}(\{p' | p' \in \text{ports}(\text{dev}(p)) \wedge \text{oncycle}(p')\}) \geq 2$

Proof Let $\pi = p_0 p_1 \ldots p_n$ be a cycle such that $p = p_i$.
If $i = 0$, then by definition of a cycle, $\text{dev}(p_n) = \text{dev}(p)$, and by definition of a path, $p_n \neq p$.
If i is even and $i > 0$, then by definition of a path, $\text{dev}(p_{i-1}) - \text{dev}(p)$ and $p_{i-1} \neq p$.
If $i = n$, then by definition of a cycle, $\text{dev}(p_0) = \text{dev}(p)$, and by definition of a path, $p_0 \neq p$.
If i is odd and $i < n$, then by definition of a path, $\text{dev}(p_{i+1}) - \text{dev}(p)$ and $p_{i+1} \neq p$. ⊠

Lemma 7.6 Let $N = \langle D, P, \text{dev}, \text{peer} \rangle$ be a connected network, and $d_1, d_2 \in D$.
If $\text{oncycle}(d_1)$, $\text{oncycle}(d_2)$, and π is a path from d_1 to d_2, then for each $p \in \pi : \text{oncycle}(p)$

Proof Let ρ be a cycle such that d_1 is on it. Let τ be a cycle such that d_2 is on it. We will show for each port p in π that $\text{oncycle}(p)$, as follows.

Let $\pi = p_0 \ldots p_n$. If p_i is in ρ or τ, then $\text{oncycle}(p_i)$. We take a fragment $\pi' = p_i \ldots p_j$ from π such that $i > 0$ implies that p_{i-1} in ρ or in τ, and $j < n$ implies that p_{j+1} in ρ or in τ, and for each port p on π', p is not on ρ and not on τ. If we can construct a cycle such that the fragment π' is part of it, we are done.

Note that by definition, $\text{dev}(p_i)$ is on ρ or on τ, and $\text{dev}(p_j)$ is on ρ or on τ.

In the following case distinction we leave out all cases which are symmetric to a case proved earlier.

1. $p_i = p_j$.
   We assume w.l.o.g. that $\text{dev}(p_i)$ is on ρ. Let $\rho = \rho_1\rho_2$ such that $\text{dev}(p_i) = \text{dev}(\text{last}(\rho_1)) = \text{dev}(\text{first}(\rho_2))$ and $\text{length}(\rho_1)$ is even. By assumption, $\text{last}(\rho_1) \neq p_i = p_j \neq \text{first}(\rho_2)$. We construct the path $\rho_2\rho_1\pi'$ and see that it is a cycle.
2. $p_i \neq p_j$.
   We assume w.l.o.g. that $\text{dev}(p_i)$ is on ρ and $\text{dev}(p_j)$ is on τ. Let $\rho = \rho_1\rho_2$ such that $\text{dev}(p_i) = \text{dev}(\text{last}(\rho_1)) = \text{dev}(\text{first}(\rho_2))$ and $\text{length}(\rho_1)$ is even. Let $\tau = \tau_1\tau_2$ such that $\text{dev}(p_j) = \text{dev}(\text{last}(\tau_1)) = \text{dev}(\text{first}(\tau_2))$ and $\text{length}(\tau_1)$ is even. By assumption, $\text{last}(\rho_1) \neq p_i \neq \text{first}(\rho_2)$ and $\text{last}(\tau_1) \neq p_j \neq \text{first}(\tau_2)$. We construct the path $\rho_2\rho_1\pi'\tau_2\tau_1\text{reverse}(\pi')$ and see that it is a cycle.

⊠

7.4.3 Connected networks

The following definitions and lemmas are necessary to identify the distance of devices in the network to the edge of the network, that is, how many times we have to take all the leaf devices away before a device becomes a leaf in the remaining set. The distance measure defined here will be used in the protocol to quantify the worst-case time that it takes for a device to complete its part in the protocol.

Definition 7.7 N is *connected* if for each two devices $d, d' \in D$ there is a path from d to d'. If N is connected, we denote the maximum length of the shortest path in N between any two devices by $\text{MaxHop} = \max(\{n | d_1, d_2 \in D \wedge n = \min(\{\text{length}(\pi) | \pi \text{ is path from } d_1 \text{ to } d_2\})\})$. The function Steps is defined by the following equation:

$$\text{Steps}(D', d) = \begin{cases} 0 & \textit{if } \text{leaf}(D', d) \textit{ or } \text{oncycle}(d) \\ 1 + \text{Steps}(D'', d) & \textit{otherwise} \end{cases}$$
$$\textit{where } D'' = D' - \{d' \in D' | \text{leaf}(D', d')\}$$

We abbreviate $\text{Steps}(d) = \text{Steps}(D, d)$.
The function Shrink is defined by the following equation:

$$\text{Shrink}(D', n') = \begin{cases} D' & \textit{if } n' = 0 \\ \text{Shrink}(D'', n' - 1) & \textit{otherwise} \end{cases}$$
$$\textit{where } D'' = D' - \{d' \in D' | \text{leaf}(D', d')\}$$

We abbreviate $\text{Shrink}(n) = \text{Shrink}(D, n)$.

The value MaxHop, which is an upper bound to the minimum number of cable hops between any two devices, is used in the IEEE documentation as a restriction on the networks on which the protocol is to operate.

The function Steps gives the one but greatest distance between a device and a leaf in the network. This number is determined by the number of steps it takes for such a device to become a leaf, when in each step all leafs are removed. For a device that is part of a cycle, the value of Steps has no meaning and will not be used.

The function Shrink gives the set of devices that remains when in each step the leaf devices are removed and this is repeated for the indicated number of times, starting with the given set. The correspondence between Steps and Shrink is obvious: if $\text{Steps}(d) = n$ then $\text{leaf}(\text{Shrink}(n), d)$ holds and if $\text{Steps}(d) \geq n$ then $d \in \text{Shrink}(n)$.

In the remainder of this paper, we assume that N is connected.

Lemma 7.8 Let $d \in D$ such that $\neg\text{oncycle}(d)$.
If $\text{Steps}(d) = n$ then $\text{size}(\{p' \in \text{ports}(d) | \text{oncycle}(\text{dev}(\text{peer}(p')) \vee \text{Steps}(\text{dev}(\text{peer}(p'))) \geq n\}) \leq 1$.

Proof By contradiction. Assume $\neg\text{oncycle}(d)$ and $\text{Steps}(d) = n$. Let $p, p' \in \text{ports}(d)$ such that $p \neq p'$ and $\text{oncycle}(\text{dev}(\text{peer}(p)) \vee \text{Steps}(\text{dev}(\text{peer}(p))) \geq n$ and $\text{oncycle}(\text{dev}(\text{peer}(p')) \vee \text{Steps}(\text{dev}(\text{peer}(p'))) \geq n$. Since $\text{Steps}(d) = n$, either $\text{oncycle}(d)$ or $\text{leaf}(\text{Shrink}(n), d)$. Since we assumed $\neg\text{oncycle}(d)$, apparently $\text{leaf}(\text{Shrink}(n), d)$. By our assumption $\text{dev}(\text{peer}(p))$ and $\text{dev}(\text{peer}(p'))$ are both in $\text{Shrink}(n)$. But $p \neq p'$, which contradicts $\text{leaf}(\text{Shrink}(n), d)$. We conclude that $\text{size}(\{p' \in \text{ports}(d) | \text{oncycle}(\text{dev}(\text{peer}(p')) \vee \text{Steps}(\text{dev}(\text{peer}(p'))) \geq n\}) \leq 1$. ⊠

Lemma 7.9 For each $d \in D$

$$\text{Steps}(d) \leq \begin{cases} \lfloor \text{MaxHop}/2 \rfloor & \textit{if } \forall d' \in D : \neg\text{oncycle}(d') \\ \max(0, \text{MaxHop} - 1) & \textit{otherwise} \end{cases}$$

Proof By contradiction.

1. Suppose $\forall d \in D : \neg\text{oncycle}(d)$.
   Suppose $\text{Steps}(d) = m > \lfloor n/2 \rfloor$. We show that we can construct a shortest path π with $\text{length}(\pi) > n$, by starting with d and extending the path in each step with one cable hop in two directions. We use induction on $n' \in \{1, \ldots, m\}$ in the following hypothesis: There is a path $p_0 \ldots p_{4n'-1}$ with $\text{Steps}(\text{dev}(p_0)) \geq m - n'$ and $\text{Steps}(\text{dev}(p_{4n'-1})) \geq m - n'$ and there is no other path from $\text{dev}(p_0)$ to $\text{dev}(p_{4n'-1})$.

   - (Base step) $n' = 1$
     Since $m > 0$, certainly $\neg\text{leaf}(\text{Shrink}(m - 1), d)$, and since $\text{leaf}(\text{Shrink}(m), d)$, there must be $p, q \in \text{ports}(d)$ such that $p \neq q$ and $\text{dev}(\text{peer}(p))$ and $\text{dev}(\text{peer}(q))$ in $\text{Shrink}(m - 1)$. Fix p, q.
     Clearly, $\text{Steps}(\text{dev}(\text{peer}(p))) \geq m - 1$ and $\text{Steps}(\text{dev}(\text{peer}(q))) \geq m - 1$. Consider $\text{peer}(p)pq\text{peer}(q)$. This is a path if $\text{peer}(p) \neq \text{peer}(q)$. Since $\neg\text{oncycle}(d')$ for all $d' \in D$, we see that $\text{dev}(\text{peer}(p)) \neq \text{dev}(\text{peer}(q))$, so $\text{peer}(p) \neq \text{peer}(q)$. If was another path from $\text{dev}(\text{peer}(p))$ to $\text{dev}(\text{peer}(q))$ then this would contradict the assumption that $\neg\text{oncycle}(d')$ for all $d' \in D$. We conclude that $\pi = \text{peer}(p)pq\text{peer}(q)$ is a path that meets the requirements.
   - (Induction step) $n' = n'' + 1 \leq m$ and the hypothesis holds for n''
     Let $\pi = p_0 \ldots p_{4n''-1}$ such that $\text{Steps}(\text{dev}(p_0)) \geq m - n''$ and $\text{Steps}(\text{dev}(p_{4n''-1})) \geq m - n''$ and there is no other path from $\text{dev}(p_0)$ to $\text{dev}(p_{4n''-1})$. We abbreviate $d_1 = \text{dev}(p_0)$ and $d_2 = \text{dev}(p_{4n''-1})$ for the first and last device of π. Since $n'' < m$, $\text{Steps}(d_1) > 0$ and $\text{Steps}(d_2) > 0$. So $\neg\text{leaf}(\text{Shrink}(m - n'' - 1), d_1)$ and $\neg\text{leaf}(\text{Shrink}(m - n'' - 1), d_2)$. So there must be $p, p' \in \text{ports}(d_1)$ and $q, q' \in \text{ports}(d_2)$ such that $p \neq p'$, $q \neq q'$, and $\text{dev}(\text{peer}(p))$, $\text{dev}(\text{peer}(p'))$

dev(peer(q)), and dev(peer(q')) in Shrink($m - n'' - 1$). Fix p, p', q and q'. Now for $x \in \{p, p', q, q'\}$: Steps(dev(peer(x))) $\geq m - n'' - 1 = m - (n'' + 1) = m - n'$ We assume without loss of generality that $p \neq p_0$ and $q \neq p_{4n''-1}$. Consider peer(p)$p\pi q$peer(q). This is a path if peer(p) $\neq$ peer(q). Since $\neg$oncycle(d') for any $d' \in D$, we see that dev(peer(p)) $\neq$ dev(peer(q)), so peer(p) $\neq$ peer(q). If there was another path from dev(peer(p)) to dev(peer(q)) then this would contradict the assumption that $\neg$oncycle(d') for all $d' \in D$. So, peer(p)$p\pi q$peer(q) is a path that meets all the requirements for the induction step.

We conclude that there is a shortest path in the network of length $2m$. Since $m > \lfloor n/2 \rfloor$, certainly $2m > (2\lfloor n/2 \rfloor) + 1 \geq n$. So $2m > n$ and we have a contradiction.

2. Suppose $\exists d' \in D$: oncycle(d').
Suppose Steps(d) $= m > \max(0, n - 1)$. Then $m > 0$ and by definition of Steps, certainly $\neg$oncycle(d). We show that we can construct a path π with length(π) $> n$, by starting with d and a neighbour of d on a cycle, and extending the path in each step with one cable hop to a neighbour which is not on a cycle. We use induction on $n' \in \{0, \ldots, m\}$ in the following hypothesis:
There is a path $p_0 p_1 \ldots p_{2n'+1}$ with oncycle(dev(p_0)), Steps(dev($p_{2n'+1}$)) $\geq m - n'$ and for all $1 \leq i \leq 2n' + 1$: $\neg$oncycle(dev(p_i)), and there is no shorter path from dev(p_0) to dev($p_{2n'+1}$).

- (Base step) $n' = 0$
Suppose there is no $p \in$ ports(d) such that oncycle(dev(peer(p))). Since N is connected, there must be π, d' such that π is a path $\pi = p_0 \ldots p_n$ from d' to d with oncycle(d') and for each $i > 0$: $\neg$oncycle(dev(p_i)). Fix d', π. Since oncycle(d') and $\neg$oncycle(dev(p_1)), we can use Lemma 7.8 to conclude that Steps(dev(p_1)) $>$ max($\{$Steps(dev(p')$|p' \in$ ports(dev(p_1)) $\wedge$ $p' \neq p_1\}$). Then it is not hard to show (using induction and Lemma 7.8) that $\forall i \in \{1, 3, \ldots, n - 2\}$: Steps(dev(p_i)) $\geq$ Steps(dev(p_{i+2})). Then we easily have $\forall i \in \{1, 3, \ldots, n\}$: Steps(dev(p_i)) $\geq m$. Since d is chosen arbitrarily with Steps(d) $> n - 1$, any of the devices on π except dev(p_0) would do. So we assume without loss of generality that there is a $p \in$ ports(d) such that oncycle(dev(peer(p))). Fix p.
We now have Steps(d) $\geq m$, $\neg$oncycle(d), and oncycle(dev(peer(p))). We see that peer(p)p is a path that meets the requirements, since there cannot be a shorter path from dev(peer(p)) to d.
- (Induction step) $n' = n'' + 1 \leq m$ and the hypothesis holds for n''
Let $\pi = p_0 \ldots p_{2n''+1}$ such that oncycle(dev(p_0)) , Steps(dev($p_{2n''+1}$)) $\geq m - n''$ and for all $1 \leq i \leq 2n'' + 1$: $\neg$oncycle(dev(p_i), and there is no shorter path from dev(p_0) to dev($p_{2n''+1}$). We abbreviate $d' =$ dev($p_{2n''+1}$) to indicate the last device in π. Since $n'' < m$, Steps(d') > 0. So $\neg$leaf(Shrink($m - n'' - 1$), d. So there must be $p, p' \in$ ports(d') such that $p \neq p'$, and dev(peer(p)) and dev(peer(p')) in Shrink($m - n'' - 1$). Fix p, p'. Now for $x \in \{p, p'\}$: Steps(dev(peer(x))) $\geq m - n'' - 1 = m - (n'' + 1) = m - n'$. We assume without loss of generality that $p \neq p_{2n''+1}$.
Consider πppeer(p). This is a path if peer(p) $\neq p_0$. Suppose that peer(p) $= p_0$. Then $p = p_1$ and dev(p_1) $= d_1$, hence $p_2 \ldots p_n$ is a cycle, which contradicts our

assumption. We conclude that $\text{peer}(p) \neq p_0$ and $\pi p\text{peer}(p)$ is a path.

Suppose that oncycle(dev(peer(p))). Then by Lemma 7.6 we have that for all $1 \leq i \leq 2n'' + 1$, oncycle(dev(p_i)), which contradicts our assumption. So we conclude that ¬oncycle(dev(peer(p))).

Suppose a shorter path than $\pi p\text{peer}(p)$ exists from dev(p_0) to dev(peer(p)). This enables us to conclude that oncycle(dev(peer(p_i))) with $p_i \in \pi$, which contradicts our assumptions. So we conclude that no shorter path than $\pi p\text{peer}(p)$ exists from dev(p_0) to dev(peer(p)).

We see that the path $\pi p\text{peer}(p)$ meets all the requirements for the induction step.

So there is a shortest path in the network of length $m + 1$. Since $m > n - 1$, $m + 1 > n$ and we have a contradiction.

⊠

7.5 Verification

In this section we prove that the IEEE 1394 tree identify protocol is correct relative to our model. In Section 7.5.1 some properties are given which have been proved invariant for the model TIP3in [GV98]. Some additional properties are given, which are be proved invariant for the model TIP4, provided that the invariants for TIP3 are also invariant for TIP4. This provision is solved in the next section, Section 7.5.2, in which it is proved that under certain timing restrictions the behaviour of TIP4 is included in the behaviour of TIP3. The proofs in Section 7.5.2 allow us to conclude that the safety aspects of cycle detection and root election in TIP4 meet the IEEE 1394 requirements. In Section 7.5.3 we prove some liveness properties for TIP4. Finally, in Section 7.5.4 we discuss whether the IEEE 1394 timing constants obey the restrictions that we found in Section 7.5.2.

The proofs in Sections 7.5.1 and 7.5.2 use simulation techniques from [LV96] which are listed in Appendices A.1 and A.3. These appendices also present a new result for using invariants in stepwise refinement, which is useful for this verification because it allows us to reuse invariants properties from [GV98] without extra effort. In Appendix A.3, some new sufficient conditions for feasibility can be found. These lessen the proof burden when proving that there are no time deadlocks in the model.

Throughout this section we fix a connected network $N = \langle D, P, \text{dev}, \text{peer}\rangle$ as the parameter for TIP4. We let s,t range over states of TIP4, δ over Time, and m over Mes.

7.5.1 Invariants for TIP3 and TIP4

We first define the properties, of which some are taken from the PVS code used to check the proofs for [GV98]. All of the following properties are necessary for the proofs in Sections 7.5.2 and 7.5.3.

Definition 7.10 $I_1(d) \triangleq \text{init}[d] \rightarrow \neg\text{rc}[d]$

$I_2(p) \triangleq \text{init}[\text{dev}(p)] \rightarrow \text{mq}[p] = \{\}$

$I_3(p) \triangleq \text{init}[\text{dev}(p)] \rightarrow \text{peer}(p) \notin \text{child}$

$$I_4(d) \triangleq \text{init}[d] \vee \text{size}(\text{ports}(d) - \text{child}) \leq 1$$

$$I_5(p) \triangleq \text{length}(\text{mq}[p]) \leq 1$$

$$I_6(p) \triangleq p \in \text{child} \rightarrow \text{mq}[\text{peer}(p)] = \{\}$$

$$I_7(p) \triangleq \text{rc}[\text{dev}(p)] \rightarrow \text{mq}[\text{peer}(p)] = \{\}$$

$$I_8(p) \triangleq \text{rc}[\text{dev}(p)] \rightarrow \text{peer}(p) \notin \text{child}$$

$$\begin{aligned} I_9(p) \triangleq\ & \vee \text{init}[\text{dev}(p)] \\ & \vee \text{head}(\text{mq}[p]) = \text{parent} \\ & \vee \text{peer}(p) \in \text{child} \\ & \vee \text{rc}[\text{dev}(\text{peer}(p))] \\ & \vee p \in \text{child} \end{aligned}$$

$$I_{10}(p) \triangleq \text{mq}[p] \neq \{\} \rightarrow \text{delay}[p] \leq \text{MaxDelay}$$

$$\begin{aligned} I_{11}(d) \triangleq\ & \wedge \neg\text{oncycle}(d) \wedge \text{init}[d] \rightarrow \text{time} \leq \text{Steps}(d) * \text{MaxDelay} \\ & \wedge \neg\text{oncycle}(d) \wedge \text{time} > \text{Steps}(d) * \text{MaxDelay} \\ & \quad \rightarrow \forall p' \in \text{ports}(d) : \\ & \qquad\quad \text{head}(\text{mq}[p']) = \text{parent} \\ & \qquad\quad \rightarrow \text{time} - \text{delay}[p'] \leq \text{Steps}(d) * \text{MaxDelay} \end{aligned}$$

$$\begin{aligned} I_{12}(d) \triangleq\ & \wedge \text{MinLpdtime} > \max(0, \text{MaxHop} - 1) * \text{MaxDelay} \\ & \wedge \text{init}[d] \\ & \wedge \neg\text{oncycle}(d) \\ & \rightarrow \text{time} < \text{MinLpdtime} \end{aligned}$$

$$I_{13}(d) \triangleq \text{oncycle}(d) \rightarrow \text{init}[d]$$

$$I_{14}(d) \triangleq \text{oncycle}(p) \wedge \neg\text{lpd}[d] \rightarrow \text{time} \leq \text{MaxLpdtime}$$

We let $I_1 \triangleq \bigwedge_d I_1(d)$, $I_2 \triangleq \bigwedge_p I_2(p)$, et cetera.

Some of the properties in Definition 7.10 have been taken from [GV98], from which we also repeat the following result.

Lemma 7.11 Properties I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, and I_9 are invariant for TIP3.

Even though the predicate oncycle has a different meaning in [GV98], we can assume that the proofs [GV98] still hold here, since the oncycle predicate is not used in the proofs.

Now we prove that under the assumption that some of the properties from Definition 7.10 hold in each reachable state for TIP4, it follows that others hold in each reachable state for TIP4 as well. In Section 7.5.2, the assumptions will be fulfilled by the corresponding properties for TIP3.

Lemma 7.12

1. I_{10} is inductive relative to $(I_2 \wedge I_5)$ for TIP4.
2. I_{11} is inductive relative to $(I_1 \wedge I_3 \wedge I_5 \wedge I_9)$ for TIP4.
3. For each $s \in \text{reachable(TIP4)}$, $s \models I_{11}$ implies $s \models I_{12}$.

4. I_{13} is inductive relative to I_3 for TIP4.
5. I_{14} is inductive relative to I_{13} for TIP4.

Proof

1. Trivial.
2. Suppose $\neg\text{oncycle}(d)$.
Initially, $s.\text{time} = 0$ and $\forall p : s.\text{mq}[p] = \{\}$. Since $\text{Steps}(d) \geq 0$, $\text{Steps}(d) * \text{MaxDelay} \geq 0 = s.\text{time}$. Since $\forall p \in \text{ports}(d) : s.\text{mq}[p] = \{\}$, it follows that $s \models I_{11}$.

Suppose from $s \xrightarrow{a} t$ and $s \models I_1 \wedge I_3 \wedge I_5 \wedge I_9 \wedge I_{11}$. We have to show that $t \models I_{11}$. Fix n such that $n * \text{MaxDelay} \leq s.\text{time} < (n+1) * \text{MaxDelay}$.

We make the following case distinction.

(a) $s.\text{time} > \text{Steps}(d) * \text{MaxDelay}$
By $s \models I_{11}$ we see that $\neg s.\text{init}[d]$. By the effect of a, we conclude that $\neg t.\text{init}[d]$. Now we just need to show for each $p' \in \text{ports}(d)$ that if $\text{head}(t.\text{mq}[p']) = \text{parent}$, then $t.\text{time} - t.\text{delay}[p'] \leq \text{Steps}(d) * \text{MaxDelay}$. Assume $p' \in \text{ports}(d)$ and $\text{head}(t.\text{mq}[p']) = \text{parent}$. By $\neg s.\text{init}[d]$, $s \models I_5$ and the precondition and effect of a, $\text{head}(s.\text{mq}[p']) = \text{parent}$. Since $s \models I_{11}$, it follows that $s.\text{time} - s.\text{delay}[p'] \leq \text{Steps}(d) * \text{MaxDelay}$.

i. $\forall \delta > 0 : a \neq \delta$
Then by the effect of a, $t.\text{time} = s.\text{time}$, and by $\neg s.\text{init}[d]$ and the precondition and effect of a, $t.\text{delay}[p'] = s.\text{delay}[p']$. So $t.\text{time} - t.\text{delay}[p'] = s.\text{time} - s.\text{delay}[p'] \leq \text{Steps}(d) * \text{MaxDelay}$ and it follows that $t \models I_{11}(d)$.

ii. $a = \delta$
Then $t.\text{time} = s.\text{time} + \delta$, and $t.\text{delay}[p'] = s.\text{delay}[p'] + \delta$. So $t.\text{time} - t.\text{delay}[p'] = s.\text{time} + \delta - (s.\text{delay}[p'] + \delta) = s.\text{time} - s.\text{delay}[p'] \leq \text{Steps}(d) * \text{MaxDelay}$ and it follows that $t \models I_{11}(d)$.

(b) $s.\text{time} \leq \text{Steps}(d) * \text{MaxDelay}$

i. $\neg s.\text{init}[d]$
By the effect of a, $\neg t.\text{init}[d]$. The remainder is equal to the proof for Step 2a.

ii. $s.\text{init}[d]$

A. $\forall \delta > 0 : a \neq \delta$
Then by the effect of a, $t.\text{time} = s.\text{time}$, so $t.\text{time} \leq \text{Steps}(d) * \text{MaxDelay}$, hence for each $p' \in \text{ports}(d)$, trivially $t.\text{time} - t.\text{delay}[p'] \leq \text{Steps}(d) * \text{MaxDelay}$ and it follows that $t \models I_{11}(d)$.

B. $a = \delta \wedge s.\text{time} + \delta \leq \text{Steps}(d) * \text{MaxDelay}$
Then for each $p' \in \text{ports}(d)$, trivially $t.\text{time} - t.\text{delay}[p'] \leq \text{Steps}(d) * \text{MaxDelay}$ and it follows that $t \models I_{11}(d)$.

C. $a = \delta \wedge s.\text{time} + \delta > \text{Steps}(d) * \text{MaxDelay}$
The effect of a leads to a violation of property I_{11}, so we have to show that our assumption on a leads to a contradiction.
First we prove by contradiction that for each d' with $\neg\text{oncycle}(d')$ and $\text{Steps}(d') < \text{Steps}(d)$, it follows that $\forall p' \in \text{ports}(d') : s.\text{mq}[p'] = \{\} \vee$

head(s.mq[p']) = ack. Suppose $\neg$oncycle(d'), Steps(d') $<$ Steps(d) and head(s.mq[p']) = parent. By $s \models I_{11}$, we see that s.time $-$ s.delay[p'] $\leq$ Steps(d')$*$MaxDelay. As t.time$-t$.delay[p'] = s.time$+\delta-$(s.delay[p']$+\delta$) = s.time$-s$.delay[p'] $\leq$ Steps(d')$*$MaxDelay, and since s.time$+\delta >$ Steps(d)$*$MaxDelay $\geq$ (Steps(d')$+1$)$*$MaxDelay, we get s.delay[p']$+\delta >$ MaxDelay, which in contradiction with our assumption that s enables δ.
Now we prove by contradiction that for each d' with $\neg$oncycle(d') and Steps(d') $\leq$ Steps(d), it follows that $\neg s$.init[d']. Fix a d' such that $\neg$oncycle(d'), s.init[d'] and there is no d'' with Steps(d'') $<$ Steps(d') and s.init[d'']. Let $P' = \{p' \in$ ports(d')$|\neg$oncycle(dev(peer(p'))) $\wedge$ Steps(dev(peer(p'))) $\leq$ Steps(d') $- 1\}$. By Lemma 7.8, size(ports(d') $- P'$) ≤ 1. Fix $p' \in P'$ and d'' = dev(peer(p')). Note that Steps(d'') $<$ Steps(d') $\leq$ Steps(d). By our assumption, $\neg s$.init[d'']. By s.init[d'] and $s \models I_3$, we see peer(p') $\notin s$.child. By s.init[d'] and $s \models I_1$, we see $\neg s$.rc[d']. Combining all of this with our observation s.mq[peer(p')] = $\{\} \vee$ head(s.mq[peer(p')] = ack and $s \models I_9$, we get $p' \in s$.child. So size(ports(d'))$-s$.child = 1. Since s.init[d'], s enables childrenknown(d') which is in contradiction with our assumption that s enables δ. We conclude that $\neg s$.init[d'].
From this observation, it trivially follows that $\neg s$.init[d] which is in contradiction with our assumption. It follows that $a \neq \delta \vee s$.time $+ \delta \leq$ Steps(d) $*$ MaxDelay.

3. Let $s \in$ reachable(TIP4) such that $s \models I_{11}$. Assume MinLpdtime $>$ max(0, MaxHop $-$ 1) $*$ MaxDelay $\wedge$ s.init[d] $\wedge$ $\neg$oncycle(d). By s.init[d] $\wedge$ $\neg$oncycle(d) and $s \models I_{11}$, s.time $\leq$ Steps(d) $*$ MaxDelay. Note that for each $n \geq 0$, $\lfloor n/2 \rfloor \leq \max(0, n-1)$. Combining this with Lemma 7.9, we get Steps(d) $\leq$ max(0, MaxHop $-$ 1). So s.time $\leq$ max(0, MaxHop $-$ 1) $*$ MaxDelay $<$ MinLpdtime and it follows that $s \models I_{12}$.

4. Suppose oncycle(d).

   Initially, s.init[d], hence $s \models I_{13}$.

   Suppose $s \xrightarrow{a} t$ and $s \models I_3 \wedge I_{13}$. By oncycle($d$) and $s \models I_{13}$, we see that s.init[d]. If $a \neq$ childrenknown(d) then t.init[d] = s.init[d], so it suffices to show that $a \neq$ childrenknown(d). By Lemma 7.5 and oncycle(d), there must be $p_1, p_2 \in$ ports(d) such that $p_1 \neq p_2$ and oncycle(dev(peer(p_1))) and oncycle(dev(peer(p_2))). Since $s \models I_{13}$, we see that s.init[dev(peer(p_1))] and s.init[dev(peer(p_2))]. Since $s \models I_3$, we see that $p_1 \notin s$.child and $p_2 \notin s$.child. Since $p_1 \neq p_2$, we see that size(ports(d) $- s$.child ≥ 2, hence s does not enable childrenknown(d).

5. Trivial.

⊠

Note that by Items 2 and 3 it follows that I_{12} is inductive relative to $(I_1 \wedge I_3 \wedge I_5 \wedge I_9 \wedge I_{11})$ for TIP4.

7.5.2 TIP4 **implements** TIP3

We use the properties established in Section 7.5.1 to obtain that TIP4 implements TIP3. As an implementation relation we take inclusion of admissible timed traces. From Section 7.5.1, it follows that the behaviour of TIP4 meets these properties only when the parameters obey the following relation: MinLpdtime $>$ max(0, MaxHop $-$ 1) $*$ MaxDelay. Therefore, we assume throughout this section that this relation holds.

In order to obtain the implementation relation, we construct a function that is to be proved a weak timed refinement from TIP4 to TIP3. Given the complicated relations between the invariants for TIP3 and the properties for TIP4, we have been forced to either prove the properties for TIP4 that depend on invariants for TIP3 anew, or to prove the invariance of the properties for TIP4 and the weak refinement in one proof, or to come up with a more elegant solution. The latter approach has given rise to some new sufficient conditions, which are presented in Appendices A.1 and A.3.

To avoid confusion, all state variables from TIP3 are subscripted with $_3$, and all state variables from TIP4 are subscripted with $_4$. Since the action signatures are equal, we do not use these subscript on the action names.

Definition 7.13 The function ref from states of TIP4 to states of TIP3 is defined to be the identity function on state variables with the same name.

Lemma 7.14 Let $s \in \mathit{states}(\text{TIP3})$. For all $I \in \{I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9\}$, $\text{ref}(s) \models I$ implies $s \models I$.

Proof Trivial. ⊠

Lemma 7.15

1. $s \in \text{Start(TIP4)}$ implies $\text{ref}(s) \in \text{Start(TIP3)}$.
2. $s \xrightarrow{a}_{\text{TIP4}} t$, $s \models I_{10} \wedge I_{11} \wedge I_{12} \wedge I_{13} \wedge I_{14}$ and $\text{ref}(s) \models I_1 \wedge I_2 \wedge I_3 \wedge I_4 \wedge I_5 \wedge I_6 \wedge I_7 \wedge I_8 \wedge I_9$ implies $\text{ref}(s) \xrightarrow{a}_{\text{TIP3}} \text{ref}(t)$.

Proof

1. Suppose $s \in \text{Start(TIP4)}$.

   Since the initial requirements are the same for every state variable in TIP3 as for the state variable with the same name in TIP4, and the state variables with the same name have the same value in s and in $\text{ref}(s)$, $\text{ref}(s) \in \text{Start(TIP3)}$ follows from $s \in \text{Start(TIP4)}$.

2. Suppose $s \xrightarrow{a}_{\text{TIP4}} t$ $s \models I_{10} \wedge I_{11} \wedge I_{12} \wedge I_{13} \wedge I_{14}$ and $\text{ref}(s) \models I_1 \wedge I_2 \wedge I_3 \wedge I_4 \wedge I_5 \wedge I_6 \wedge I_7 \wedge I_8 \wedge I_9$. $s \in \text{reachable(TIP4)}$ and $\text{ref}(s) \in \text{reachable(TIP3)}$.

   Since for each a, the effect in TIP3 is equal to the effect in TIP4 on all state variables from TIP3, it follows that whenever $\text{ref}(s) \xrightarrow{a}_{\text{TIP3}} t'$, then $t' = \text{ref}(t)$.

   If $a \notin \bigcup_d \text{loopdetect}(d)$, then we see that the precondition of a in TIP4 trivially implies the precondition of a in TIP3, hence if $s \xrightarrow{a}_{\text{TIP4}}$, then $\text{ref}(s) \xrightarrow{a}_{\text{TIP3}}$. So we just need to show that if $a = \text{loopdetect}(a)$, then $\text{ref}(s) \xrightarrow{a}_{\text{TIP3}}$.

   Suppose $a = \text{loopdetect}(a)$. By precondition of a in TIP4, $\neg s.\text{lpd}_4[d]$ and $s.\text{time}_4 \geq$ MinLpdtime. From $\neg s.\text{lpd}_4[d]$ we see $\neg\text{ref}(s).\text{lpd}_3[d]$. By $s.\text{time}_4 = \text{lpdtime}_4[d]$ and

$s \models I_{12}$ we see that either $\neg s.\text{init}_4[d]$ or oncycle(d). By precondition of a in TIP4 we see that $s.\text{init}_4[d]$, and we conclude that oncycle(d). Hence ref(s) enables a.

⊠

Corollary 7.16 $I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_{11}, I_{12}, I_{13}$ and I_{14} are invariant for TIP4.

Proof By Lemmas 7.11, 7.12, 7.14 and 7.15 we can use Lemma A.2. ⊠

Corollary 7.17 The function ref is a weak timed refinement from TIP4 to TIP3 with respect to $(I_{10} \wedge I_{11} \wedge I_{12} \wedge I_{13} \wedge I_{14})$ and $(I_1 \wedge I_2 \wedge I_3 \wedge I_4 \wedge I_5 \wedge I_6 \wedge I_7 \wedge I_8 \wedge I_9)$

Proof By Lemmas 7.11, 7.12, 7.14 and 7.15 we can use Lemma A.2. ⊠

The implementation relation now follows easily.

Theorem 7.18 *t-traces*(TIP4) $\subseteq$ *t-traces*(TIP3).

Proof Combine Corollary 7.17 with Theorem 6.14 from [LV96]. ⊠

7.5.3 Liveness results for TIP4

In this section we show some liveness results for model TIP4. As in Section 7.5.2, we assume that the parameters of TIP4 meet the following relation: $\text{MinLpdtime} > \max(0, \text{MaxHop} - 1) * \text{MaxDelay}$.

The liveness results are the following. We first show that TIP4 has no time deadlocks. For this, some new sufficient conditions are used, which are presented in Appendix A.3. Then we prove that when a cycle is present, it will be detected, and that otherwise a root will be elected. Notice that we cannot use results from TIP3, since notions as quiescence and fairness are not present at the timed level.

First we need to define a measure on reachable states, to indicate the number of discrete actions that must be performed before passing of time will be enabled again.

Definition 7.19 The function Measure gives a pair for each state s from TIP4, as follows:

$$\text{Measure}(s) = \langle I, C + R + M + L \rangle$$

where

$$
\begin{aligned}
I &= \text{size}(\{d \mid s.\text{init}[d]\})\\
C &= \text{size}(P - s.\text{child})\\
R &= \text{size}(\{d \mid \neg s.\text{root}[d]\})\\
M &= \text{size}(\{p \mid s.\text{mq}[p] \neq \{\}\})\\
L &= \text{size}(\{d \mid \neg s.\text{lpd}[d]\})
\end{aligned}
$$

The ordering $\prec$ is the lexicographic ordering on pairs of naturals, based on the ordering $<$ on naturals.

Since $<$ is well-founded, $\prec$ is also well-founded.

Now we prove the properties that we need for deadlock freedom, namely that when no discrete action is enabled, then the passage of time is enabled, and that at every moment in time at most a finite amount of discrete activity can occur.

Lemma 7.20 For each $s \in$ reachable(TIP4) the following holds:

1. s enables an action from *acts*(TIP4).

2. If $s \xrightarrow{a} t$ and $\forall \delta > 0 : a \neq \delta$, then Measure$(t) \prec$ Measure(s) otherwise Measure$(t) =$ Measure(s).

Proof

1. It suffices to show that if s does not enable a for all $a \in acts(Tipvier) - \bigcup_{\delta>0}\{\delta\}$, then s enables δ for some $\delta > 0$, which we prove by contradiction.

   Suppose that for all $a \in acts$(TIP4), s does not enable a. Apparently s does not enable any $\delta > 0$, so either s.time = MaxLpdtime and $\exists d : s.\text{init}[d] \wedge \neg s.\text{lpd}[d]$, or $\exists p : s.\text{mq}[p] \neq \{\} \wedge s.\text{delay}([p]) \geq \text{MaxDelay}$.

   Suppose s.time = MaxLpdtime, s.init[d] and $\neg s$.lpd[d]. Then s enables loopdetect(d) and we have a contradiction.

   Suppose $s.\text{mq}[p] \neq \{\}$ and $s.\text{delay}[p] \geq$ MaxDelay. Let head(s.mq[p]) $= m$. Since MaxDelay $\geq$ MinDelay, s.delay[p] $\geq$ MinDelay. Using Invariant I_2, we see that $\neg s$.init[dev(p)], and using Invariant I_6 we get peer(p) $\notin s$.child. Suppose $p \in s$.child. Using Invariant I_3 we get $\neg s$.init[dev(peer(p))]. So, s enables receivemes(dev(peer(p)), peer(p), m) and we have a contradiction. So $p \notin s$.child. Using Invariant I_7, we see that $\neg s$.rc[dev(peer(p))]. Combining all of this with I_9 we get $m =$ parent. Suppose s.init[dev(peer(p))]. Then s enables addchild(dev(peer(p)), peer(p)) and we have a contradiction. So $\neg s$.init[dev(peer(p))]. Then s enables receivemes(dev(peer(p)), peer(p), parent) and we have a contradiction.

2. Let Measure$(s) = \langle I_s, C_s + R_s + M_s + L_s \rangle$ and Measure$(t) = \langle I_t, C_t + R_t + M_t + L_t \rangle$.

   (a) $a =$ childrenknown(d)
   By precondition of a, $\neg s$.init[d] and by effect of a, t.init[d]. So $I_t < I_s$. We conclude that Measure$(t) \prec$ Measure(s).

   (b) $a =$ addchild(d, p)
   By effect of a, t.init $= s$.init, t.root $= s$.root and t.lpd $= s$.lpd, hence $I_t = I_s$, $R_t = R_s$ and $L_t = L_s$. By precondition of a, head(s.mq[peer(p)] = parent. Combining this with Invariant I_5, we get s.mq[peer(p)] $= \{\} \vdash$ parent. By the effect of a, t.mq[peer(p)] = tail(s.mq[peer(p)]) $= \{\}$, so $M_t = M_s - \{\text{peer}(p)\}$, hence $M_t < M_s$. Combining head(s.mq[peer(p)] = parent with Invariant I_6 we get $p \notin s$.child. By effect of a, $t.\text{child} = s.\text{child} \cup \{p\}$. So $C_t < C_s$. By effect of a We conclude that Measure$(t) \prec$ Measure(s).

   (c) $a =$ receivemes(d, p, m)
   By effect of a, t.init $= s$.init, t.child $= s$.child, t.root $= s$.root and t.lpd $= s$.lpd, hence $I_t = I_s$, $C_t = C_s$, $R_t = R_s$ and $L_t = L_s$. By precondition of a, head(s.mq[peer(p)] $= m$. Using Invariant I_5, we get s.mq[peer(p)] $= \{\} \vdash m$. By the effect of a, t.mq[peer(p)] = tail(s.mq[peer(p)]) $= \{\}$, so $M_t = M_s - \{\text{peer}(p)\}$, hence $M_t < M_s$. We conclude that Measure$(t) \prec$ Measure(s).

(d) $a = \text{solverootcontent}(d, p)$
By effect of a, $t.\text{init} = s.\text{init}$, $t.\text{root} = s.\text{root}$, $t.\text{mq} = s.\text{mq}$ and $t.\text{lpd} = s.\text{lpd}$, hence $I_t = I_s$, $R_t = R_s$, $M_t = M_s$ and $L_t = L_s$. By precondition of a, $s.\text{rc}[\text{dev}(\text{peer}(p))]$. Combining this with Invariant I_8 we get $p \notin s.\text{child}$. By effect of a, $t.\text{child} = s.\text{child} \cup \{p\}$. So $C_t < C_s$. We conclude that $\text{Measure}(t) \prec \text{Measure}(s)$.

(e) $a = \text{root}(d)$
By effect of a, $s.\text{init} = t.\text{init}$, $s.\text{child} = t.\text{child}$, $t.\text{mq} = s.\text{mq}$ and $t.\text{lpd} = s.\text{lpd}$, hence $I_t = I_s$, $C_t = C_s$, $M_t = M_s$ and $L_t = L_s$. By precondition of a, $\neg s.\text{root}[d]$, and by effect of a, $t.\text{root}[d]$. So $R_t < R_s$. We conclude that $\text{Measure}(t) \prec \text{Measure}(s)$.

(f) $a = \text{loopdetect}(d)$
By effect of a, $s.\text{init} = t.\text{init}$, $s.\text{child} = t.\text{child}$, $s.\text{root} = t.\text{root}$ and $s.\text{mq} = t.\text{mq}$, hence $I_t = I_s$, $C_t = C_s$, $R_t = R_s$ and $M_t = M_s$. By precondition of a, $\neg s.\text{lpd}[d]$, and by effect of a, $t.\text{lpd}[d]$. So $L_t < L_s$. We conclude that $\text{Measure}(t) \prec \text{Measure}(s)$.

(g) $a = \delta$
By effect of a, $s.\text{init} = t.\text{init}$, $s.\text{child} = t.\text{child}$, $s.\text{root} = t.\text{root}$, $s.\text{mq} = t.\text{mq}$ and $t.\text{lpd} = s.\text{lpd}$. Hence $I_t = I_s$, $C_t = C_s$, $R_t = R_s$, $M_t = M_s$ and $L_t = L_s$, and we conclude that $\text{Measure}(t) = \text{Measure}(s)$.

⊠

Now we show that TIP4 cannot get into a time deadlock by its own discrete activity. A timed I/O automaton that has this property is called *feasible*.

Theorem 7.21 TIP4 is feasible.

Proof It can easily be seen that TIP4 fulfills the requirements for Lemma A.4. This lemma fulfills one of the requirements in Theorem A.5. The other requirement is fulfilled by the Measure function and the result in Proposition 7.20. It follows from Theorem A.5 that TIP4 is feasible. ⊠

We now show that whenever there is a cycle in the network, it is detected by the protocol.

Theorem 7.22 Let α be an admissible execution of TIP4.
If $\text{oncycle}(d)$ then α contains an occurrence of $\text{lpd}(d)$.

Proof Since time proceeds in α without bound, and since initially $s.\text{lpd}[d]$ is false and since $s.\text{lpd}[d]$ can only be made true by an occurrence of $\text{lpd}(d)$, it suffices to show that for each reachable state s in TIP4, if $s.\text{time} > \text{MaxLpdtime}$, then $s.\text{lpd}[d]$. This follows easily from Invariant I_{14}. ⊠

Unfortunately, it is not possible to prove that if there is no cycle in the network, then within finite time a root will be elected. This is due to the unknown duration of the root contention solving sub-protocol. The following theorem shows that if no root contention occurs, then indeed a root is elected in finite time.

Theorem 7.23 Let $\alpha = s_0 a_1 s_1 \ldots$ be an admissible execution of TIP4.
If $\forall d : \neg\text{oncycle}(d)$ and $\forall i, d : \neg s_i.\text{rc}[d]$, then $\exists d$ such that α contains an occurrence of root(d).

Proof Assume $\forall d : \neg\text{oncycle}(d)$. We observe the following:

1. Time proceeds in α without bound.

2. In each reachable state s in TIP4 the following holds. For all d: if s is an initial state then $\neg s.\text{root}[d]$, and if $s.\text{root}[d]$, then each execution leading to s must contain an occurrence of root(d).

3. If there is a state s in α and a d such that $\text{ports}(d) - s.\text{child} = \{\}$, then α contains an occurrence of root(d).

   This is easily seen by a few observations. Fix s and d such that $\text{ports}(d) - s.\text{child} = \{\}$. First, $s.\text{init}[d]$ or $s.\text{root}[d]$ or s enables root(d). If $s.\text{root}[d]$, then by Item 2 we conclude that α contains an occurrence of root(d). If $s.\text{init}[d]$ then s enables childrenknown(d). If s enables childrenknown(d) or root(d), then s does not enable any $\delta > 0$. If s enables childrenknown(d) and $s \overset{a}{\rightarrow} t$ then either $a = \text{childrenknown}(d)$ and t enables root(d) or t enables childrenknown(d). If s enables root(d) and $s \overset{a}{\rightarrow} t$ then $a = \text{root}(d)$ or t enables root(d).

4. In each reachable state s in TIP4 the following holds. If $\forall p \in P$ either $p \in s.\text{child}$ or $\text{peer}(p) \in s.\text{child}$, then there exists a d such that $\text{ports}(d) - s.\text{child} = \{\}$.

   Suppose $\forall p \in P$ either $p \in s.\text{child}$ or $\text{peer}(p) \in s.\text{child}$ and there is no d such that $\text{ports}(d) - s.\text{child} = \{\}$. Construct a longest path $\pi = p_0 \ldots p_n$ such that for each $i \in \{0, 2, \ldots, n-1\} : p_i \notin s.\text{child}$ and for all $i, j \in \{0, 1, 3, 5, \ldots, n\} : i \neq j \rightarrow \text{dev}(p_i) \neq \text{dev}(p_j)$. Since $p_{n-1} \notin s.\text{child}$, and $p_n = \text{peer}(p_{n-1})$ certainly $p_n \in s.\text{child}$. Suppose that $p \in \text{dev}(p_n) : p \notin s.\text{child}$. If $\text{dev}(\text{peer}(p)) = \text{dev}(p_i)$ for some $i \in \{0, \ldots, n\}$, then we can construct a cycle, and we have a contradiction. So $\text{dev}(\text{peer}(p)) \neq \text{dev}(p_i)$ for all $i \in \{0, \ldots, n\}$. But then we can construct a longer path than π to meet the requirements, and we have a contradiction. So we conclude that there is no $p \in \text{dev}(p_n) : p \notin s.\text{child}$, hence $\text{ports}(\text{dev}(p_n)) - s.\text{child} = \{\}$.

We now show that there is a state s in α and a d such that $\text{ports}(d) - s.\text{child} = \{\}$. By definition of α, there exists an i such that $s_i.\text{time} > (\lfloor \text{MaxHop}/2 + 1 \rfloor + 1) * \text{MaxDelay}$ and $\forall j < i : s_j.\text{time} \leq (\lfloor \text{MaxHop}/2 \rfloor + 1) * \text{MaxDelay}$. Fix i.
By Lemma 7.9 and $\forall d : \neg\text{oncycle}(d)$, we have $\forall d : \text{Steps}(d) \leq \lfloor \text{MaxHop}/2 \rfloor$, hence $\forall d : (\text{Steps}(d) + 1) * \text{MaxDelay} < s_i.\text{time}$. Using invariant I_{11} we get $\forall d : \neg s_i.\text{init}[d]$. Using invariant I_4 we get $\forall d : \text{size}(\text{ports}(d) - s_i.\text{child}) \leq 1$.
Suppose $\exists d : \text{size}(\text{ports}(d) - s_i.\text{child}) = 0$. Fix d. It follows that $\text{ports}(d) - s_i.\text{child} = \{\}$. By Item 3 we may conclude that α contains an occurrence of root(d).
Suppose $\forall d : \text{size}(\text{ports}(d) - s_i.\text{child}) = 1$. Suppose $\exists p : p \notin s_i.\text{child} \wedge \text{peer}(p) \notin s_i.\text{child}$. Fix p. By our assumption we have $\neg s.\text{rc}[\text{dev}(\text{peer}(p))]$. Combining this with $\neg s.\text{init}[\text{dev}(p)]$ and By invariant I_9, we get $\text{head}(s_i.\text{mq}[p]) = \text{parent}$. Combining this with $\neg\text{oncycle}(\text{dev}(p))$ and Invariant I_{11}, we get $s_i.\text{time} - s_i.\text{delay}[p]) \leq \text{Steps}(\text{dev}(p)) * \text{MaxDelay}$. Since $s_i.\text{time} > (\text{Steps}(\text{dev}(p)) + 1) * \text{MaxDelay}$, we have $s_i.\text{delay}[p] > \text{MaxDelay}$ which is in contradiction with Invariant I_{10}. We conclude that there is no p such that $p \notin s_i.\text{child} \wedge \text{peer}(p) \notin s_i.\text{child}$.

constant	*value*	*reference*
min cable length	0	no restriction specified
max cable length	4.5 m	Section 1.1, Page 1, 1394-1995
max cable hops	16	Section 1.1, Page 1, 1394-1995
propagation delay	≤5.05 ns/m	Section 4.2.1.4.3, Page 74, 1394-1995
min CONFIG_TIMEOUT	166.6 μs 166.6 μs	Table 7-14, Page 89, 1394-1995 Table 8-14, Page 90, P1394a
max CONFIG_TIMEOUT	166.9 μs 166.9 μs	Table 7-14, Page 89, 1394-1995 Table 8-14, Page 90, P1394a

Table 7.1: IEEE 1394 timing constants

Since $\forall p : p \in s_i$.child $\vee$ peer(p) $\in s_i$.child we can use Item 4 to conclude that there is a d such that ports(d) $- s_i$.child $= \{\}$. Fix d. By Item 3 we may conclude that α contains an occurrence of root(d).

⊠

7.5.4 Are the IEEE 1394 timing constants correct?

Table 7.1 gives the IEEE 1394 timing constants, and a reference to where they are to be found in the documentation. Here, 1394-1995 refers to [IEE96] and P1394a refers to [IEE99]. Note that the constants are the same for 1394-1995 and P1394a. From these numbers, we get the constants used for the formal verification as follows:

$$\begin{aligned}
\text{MinDelay} &= \text{min cable length} * \text{propagation delay} = 0\text{ns} \\
\text{MaxDelay} &= \text{max cable length} * \text{propagation delay} = 22.72\text{ns} \\
\text{MinLpdtime} &= \text{min CONFIG_TIMEOUT} = 166.6\mu\text{s} \\
\text{MaxLpdtime} &= \text{max CONFIG_TIMEOUT} = 166.9\mu\text{s} \\
\text{MaxHop} &\leq \text{max cable hops} = 16
\end{aligned}$$

The question is then, do these constants meet the requirements for a correct implementation? We found in Theorem 7.18 that the model behaves correctly if the relation MinLpdtime $>$ max(0, MaxHop $-$ 1) $*$ MaxDelay holds. Since $(16-1)*22.72$ ns $= 340.80$ ns $< 166.9\ \mu$s, the answer is yes. If the devices in IEEE 1394 enter the tree identify phase at the same time, if there is no device with the force_root flag set to true, and if our model of the IEEE 1394 communication is correct, then we can say the following with certainty: If a loop is in the network, it is detected, and that if there is no loop in the network, no loop will be detected and a root will be chosen.

The difference between the actual MinLpdtime value and the minimal value as required by our relation is rather large. One could wonder whether this implies that the limitations set by IEEE 1394 and P1394a can be tightened. This could be done by decreasing the MinLpdtime value, increasing the number of nodes allowed, increasing the delay between nodes (by allowing greater cable lengths), or a combination of these. However, the times at which the tree identify phase is entered can differ among nodes. The constant responsible for the duration of

the bus reset signal being sent is based on a worst-case scenario for any node to notice that a bus reset period has started. This constant has a value of about 166 μs, and can be used as an indication of the difference in starting times for the tree identify phase. If the times can indeed be that far apart for peer nodes, the loop detection timer should be in the same order of magnitude to not run the risk of detecting a loop when it is not there. Moreover, the use of the force root flag increases the delay in participating in the tree identify phase even further. We conclude that it is not yet clear whether the IEEE 1394 and P1394a bounds are correct and may be tightened.

7.6 Conclusions

The verification shows that under the assumptions made, the IEEE 1394 definition of the tree identify phase meets the requirements. Exactly one root is chosen when there is no cycle present, and a cycle is detected if and only if there is a cycle present in the network. It is obvious from the proofs that the refinement step from an untimed model to a timed model in combination with the desired property of correct cycle detection is a complicated one. More proofs about network topologies are needed to make a quantitative analysis of the worst case scenarios. Also, the invariants that are specific to the model TIP4 are more complicated than the invariants for TIP3. The effort invested in the construction of these proofs adds up to about two months. We hope that in further refinement steps these proofs can be reused with little effort.

As to the remaining IEEE 1394 details that we have not considered, we believe that the addition of the root contention solving protocol with its verification from [SV99] will probably not touch the critical behaviour parts of the root election or cycle detection. However, the correctness of a new model, obtained by adding the delay in entering the tree identify phase or by adding the force_root flag, and the correctness of the assumption that the message queues model the IEEE 1394 signal communication are not that obvious. An extension of this work may show that either IEEE 1394 timing bounds can be tightened or should be loosened.

The advantage of the layered verification in this case is that we do not need to prove anything about the safety properties of root election, since our refinement proof gives us safety immediately. Establishing the refinement was not as easy as expected, because of the complicated reuse of invariants at the abstract level. The extra lemma that was needed shows that the proof obligations can still be divided over small, clear proof steps.

The desired liveness properties, which express that a cycle is detected when there is a cycle in the network topology and a root is elected otherwise, cannot be established with the 'implements' relation alone, since we have only proved inclusion of admissible traces. In an untimed verification, liveness properties are proved by showing a *fair trace* inclusion, that is, each fair trace from the more detailed model is also a fair trace in the more abstract model. In most cases, the liveness property holds trivially for any fair trace of the abstract model, and therefore also for any fair trace of the detailed model. In a timed verification, liveness is often expressed in terms of timing bounds. Then again the (admissible) trace inclusion yields correctness. In our case, neither of these methods works. We are comparing a timed model to an essentially untimed model and hence have no fairness that carries over from the more abstract level to the (timed) detailed level. On the other hand, we have no timing requirement stating when the root should be elected. So, we had to prove the liveness completely on the

level of model TIP4, without reusing proofs from the level of TIP3. For proving feasibility (i.e. no time deadlocks) we added a small result to the I/O automata theory, consisting of sufficient conditions, mostly of a syntactical nature.

We conclude that for proving safety and liveness properties in a situation with only untimed models or with only timed models, a layered verification is a very suitable proof method which allows one to 'divide and conquer' the proof obligations. In a situation where timed and untimed behaviour are compared, we think that other methods should be used in addition, or the degree of refinement should be very low in order for a layered verification to diminish the amount of work to be done in each layer. It would be very useful if the proofs constructed for this verification were checked with a proof checker. Careful manual inspection can never replace the confidence obtained by such automated inspection. Some results have been obtained in checking invariant proofs for I/O automata, both timed and untimed, as can be seen in [AH96], and several papers which are under construction [Arc99]. We expect that such an effort will be considerable, but manageable.

Chapter 8

Conclusions

In this chapter, the results of the project are evaluated with respect to the project objectives and the central hypothesis from Section 1.4. The hypothesis is:

> *Using formal methods to support the industrial software development process can be effective.*

The project objectives are:

1. Development of heuristics about when formal methods should be applied.
2. Improvement of methods and tools so that bigger applications can be dealt with faster.
3. Integration of complementary approaches within formal methods research.
4. Improvement of technology transfer process from formal methods research to practice.

In the following sections, I discuss first per project objective how the cases have or have not contributed to this objective. Then I give my position with respect to the general hypothesis. Finally, I list some directions for the future of this kind of research.

8.1 Project objectives

1. *Development of heuristics about when formal methods should be applied.*

   None of the cases have given quantifiable information from which such heuristics could be deducted. However, from the experience with Case 2, 3, 5 and 6 it can be concluded that formal methods are not (yet) to be applied to complete designs of the size of IEEE 1394 or HAVi. They are suitable to show or refute correctness of small, self-contained parts of such designs. So, if one is interested in applying formal methods to a large design, it is sensible to try to find a small self-contained part of the design with some critical function for the whole design. In Case 2 this was the tree identify phase, since the rest of the protocol depends heavily on the correct completion of this phase. In Case 3 one of the reasons for not being able to test the link layer of 1394 was that this part is not self-contained enough to enable testing in isolation.

Another criterion for applying formal methods is of course the suspicion of errors. When a design looks complicated, correctness is usually not obvious, and errors may be found already in a very abstract formalisation of the design. When constant values are used such as the time bounds in Case 2, a verification may show whether these values are correct or not, and in addition, fault-tolerance bounds may be found which improve the overall effectiveness of the design.

2. *Improvement of methods and tools so that bigger applications can be dealt with faster.*

   Case 3 and 4 are clear attempts at this objective, each from their own specific angle.

   After the completion of Case 3, other projects have been able to test real-life designs in VHDL, in several stages of the development. So it can be concluded that indeed this case has contributed to this objective.

   Case 4 can be seen as the formalisation of a common practice in testing, where one reasons in an ad hoc manner to justify the selection of tests and not execute the whole set. The good news in Case 4 is that a suitable formal basis was found for such reasoning and it does help in eliminating a significant part of the test obligation. A limiting factor is that the symmetry definition and the conditions on the implementation are currently rather complicated, hence engineers cannot be expected to use this method in practice straightaway. We expect that further research will lead to definitions and conditions which are simpler to apply.

   The extension of I/O automata theory that was established during Case 1 is an improvement for problems with possibly uncountable action sets or strong fairness requirements. It remains to be seen how often such problems are encountered in practice, but I expect strong fairness aspects occur more often than uncountable action sets. The proof-checking efforts on my work for Case 1 show that such manual proofs need not be trusted as such, but can be checked in a fairly efficient way to obtain more confidence in the results.

3. *Integration of complementary approaches within formal methods research.*

   In spite of the aim at the start of the project, none of the cases have used complementary approaches to tackle a problem. In Case 6 the following approach was considered: first perform model checking to obtain correctness for finite instances of the HAVi leader election protocol, and then use these results in a formal proof that the protocol is correct for all instances (e.g. with the help of a theorem proving tool). However, since the protocol was not correct and model checking already took up more time than planned, this approach was not feasible.

4. *Improvement of technology transfer process from formal methods research to practice.*

   Both Case 3 and Case 6 aimed at this objective.

   For Case 3 the transfer has clearly succeeded since the tool environment has been used in several projects on industrial designs. The use of the tool environment requires academic skills.

   The results of Case 6 could be used by Philips in two ways: one is to see how protocols can be specified and verified formally, and another is to improve this particular leader

election protocol or tighten the assumptions under which the protocol should behave correct. The latter option implies imposing restrictions on the environment of the protocol in the HAVi environment. It remains to be seen whether this transfer will actually take place.

The ultimate transfer of formal methods research to practice can be achieved when evaluations like in this thesis find their way to the industrial public, and both positive and negative results are used to improve the software development process.

8.2 Does the hypothesis hold?

Now I will discuss the most important conclusion of this thesis, whether using formal methods to support the industrial software development process can be effective. I think the hypothesis is true and will gain strength over the coming years. I have three arguments to support my position:

1. *The potential of formal methods will increase significantly in the coming years.*
2. *The applications of formal methods in this project have given useful results.*
3. *Formal methods can be applied more effectively still than in this project.*

Argument 1 can be justified by several developments. In the academic world, much attention is given to the application of formal methods to industrial cases of increasing size and complexity, and one is working constantly on the improvement of theory and tools. In fact, there is a competition going on in which tool developers will go to great lenghts to be able to handle larger or more complex cases than the rest of the community. The quest for improvement is supported by the ever-growing capacities of computer systems in terms of speed, memory and storage. I think the current trend of growing potential will continue for at least another ten years.

The justification for Argument 2 can be found in Section 1.6. The work on the different case studies has taken a long (effective) time, and neither the results, nor their scope are overwhelming. However, in five out of six case studies, formal methods have given an answer that can or does support the software/hardware development process in Philips. Given the limiting factors of the project, the outcomes are rather good.

Argument 3 is inspired by our experiences with such limiting factors. We list them below since they may help to improve the conditions for applying formal methods and increasing effectivity. In the following list, the symbol • indicates a cause that is related to academic expertise, experience and more whimsical causes like personal taste. The symbol $*$ indicates a cause arising from external factors.

- The learning curve of methods and tools.
  Delay in the project was caused by the time I needed to learn to work with several formal methods and tools. It seems such learning curves can only be shortened by involving people more experienced, which may imply more funding.

- Underestimation of the formalisation effort.
  Understanding the behaviour and properties that are to be formalised and making the

right abstractions is a tough job. Especially the task of finding proper requirements and expressing these in a temporal logic was underestimated. What may help in such situations is the approach proposed in [DAC98], where patterns are given for classes of properties in different temporal logics.

The two next items contributed to the complexity of the formalisation task.

- Ad hoc abstraction methods.
  Once the proper model and specification have been obtained, in many cases the model is too complex for verification by hand or with tools. The abstractions performed to make the model manageable, were in all cases done during the formalisation itself, in an ad hoc manner and justified with intuitive arguments. Only in Case 2 there has been an attempt at undoing some abstractions through the use of refinements. However, when formalising, one should work in the other direction first, by starting to give the most detailed model and then use abstraction methods to obtain more abstract models which are suitable for verification. After establishing verification relations at the abstract level, one could then obtain similar relations at the more detailed level as presented here or as proposed in [KP98].

- Choosing the proper paradigm.
  It is not always clear what formal language paradigm fits a particular case best. Most formal languages are rather disjunct in the sense that none of them have all the properties that a particular case calls for. A good cooperation with the industrial partner is indispensable for this effort.

∗ Lack of precision in standard documents.
  Standard documents often describe protocols in natural language. This gives rise to ambiguities, errors and unclear statements, hence this hampers the formalisation effort. Also, the people who write standards often have implicit ideas about how the standard should be implemented. Even when a standard attempts to be more precise and clear (e.g. the IEEE 1394 standard documents [IEE96, IEE99]), many mistakes and unclarities remain.

∗ The limiting conditions of the project.
  As mentioned in Section 1.4, not much funded manpower was involved in the project, and the cooperating institutes were geographically far apart. This made intense cooperation very hard. For such a modest project, the priority with the Philips Research Laboratories was naturally not very high. So in most cases, the analysis was performed after the development of the corresponding system had already completed. This means that the results of the analysis could only be used for a posteriori evaluation. Also, the cooperation with people from the Philips Research Laboratories who were not involved in the project but acted as external experts, was limited to one person per case. It is desirable to have more people involved with such expertise, because this is beneficial for the availability and objectivity of the information and advice exchanged.

8.3 Future directions

What can be done to improve the effectiveness of using formal methods? The first important point is the improvement of formal methods themselves. The second is to induce industry to use formal methods, for instance by investing in projects like these. The third is the improvement of the application of formal methods in projects like these by meeting some prior conditions.

Much has been said about the road ahead for formal methods. Judging from the experiences gained in this project, I think there are many directions for research which are valuable for the application of formal methods in projects like these.

It seems that in this project, the verification techniques theorem proving and model checking have given different benefits and are both suitable to be used in the future. The advantage of theorem proving is that proofs, once constructed, can be reused and often easily adapted if models are slightly changed. Also it is possible to construct proofs for models of infinite size, which cannot (yet) be done with model checking. The advantage of model checking is that it requires less expertise and is easier to present to industrial partners. The choice between the two approaches may depend on the expected variety in models, the expected size of the models, and the number of people involved that are skilled in constructing proofs. I conclude that extensive research into improvements of both model checking and theorem proving is desired. Issues like composition (of methods, models, proofs), abstraction and refinement, reduction through confluence, and the like are important for both approaches.

As to conformance testing, I think the application to industrial case studies teaches us that complete coverage is too much to ask for. In order to be used in industry, research must be performed in two directions. The first is the quantification of coverage. Testers want to have percentages of errors found or behaviour tested. The assumptions under which such percentages are given, should have a clear, intuitive meaning to people from industry. The other direction is the approach to test by exhaustive exploration, where e.g. artefacts are model checked.

The second item for improving effectiveness, the inducement of industry to invest in formal methods, is perhaps the toughest task. The Philips Research Laboratories have repeatedly shown their interest in formal methods in terms of funding and cooperation, and their willingness to use the outcomes of such cooperation. How can we convince more industrial partners of their need for using formal methods? I think researchers like me are not and should not be qualified for convincing industry that they should use formal methods, and how they should do this. The priorities, budgets, time schedules, working atmosphere, hierarchy, protocols, and such that prevail in an industrial environment are very different from the scientific environment. The expected counter argument is “But you have all these success stories of finding bugs in very important software and hardware designs”. There are indeed a number of success stories, but academic researchers do not have the skill to drive the point home, that is, to teach industry how to use which formal method for which problem, how to keep doing this, and how to do this on a large scale. Therefore I think it would be useful (and in the long term indispensible) to cooperate with experts on business processes in order to find the most effective way to train industry, and to find what still needs to be added to formal methods.

As to the prior conditions to projects like this one, my views are as follows.
The industrial partners must

- Study this project: both results and evaluation.

- Put in more manpower.
- Assign greater priority to possible results.
- Be very clear in what is desired on the whole and per case study.

On the other hand, the researchers from academia must

- Gather and classify results of applications of formal methods and work towards a library in which one can hope to find the appropriate method for specific problems. Currently, this is being attempted on an Internet site which can be found through the URL http://vlib.org/.
- Put in more manpower.
- Put in more effort to cooperate among research institutes and projects.
- Be clear and honest in what is currently possible for different types of case studies.

If and only if the majority of these suggestions for improvement are picked up and implemented, it can be expected that in about ten years, significant parts of the software/hardware development will be analysed formally, and therefore of better quality.

Bibliography

[ADLU91] A.V. Aho, A.T. Dahbura, D. Lee, and M.U. Üyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours. *IEEE Transactions on Communications*, 39(11):1604–1615, 1991. Also appeared in: Proceedings of *Protocol Specification, Testing and Verification* VIII, pp. 75–86, North-Holland, 1988.

[AH96] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proceedings 1996 IEEE Real-Time Technology and Applications Symposium (RTAS'96)*. IEEE Computer Society Press, 1996. A full version is available as Report NRL/MR/5540–98-8180 from URL http://www.itd.nrl.navy.mil/ITD/5540/publications/CHACS/1998/.

[AHI98] K. Ajami, S. Haddad, and J-M. Ilié. Exploiting symmetry in linear time temporal logic model checking: One step beyond. In Steffen [Ste98], pages 52–67.

[AL94] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[ALM96] M. Abadi, L. Lamport, and S. Merz. A TLA solution to the RPC-Memory specification problem. In Broy et al. [BMS96], pages 21–66.

[AR96] E. Astesiano and G. Reggio. A dynamic specification of the RPC-Memory problem. In Broy et al. [BMS96], pages 67–108.

[Arc99] M. Archer. Personal communication, March 1999.

[AS85] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[BBD+96] D. Borrione, H. Bouamama, D. Deharbe, C. Le Faou, and A. Wahba. HDL-based integration of formal methods and CAD tools in the PREVAIL environment. In M. Srivas and A. Camilleri, editors, *FMCAD'96, Palo Alto, CA, USA*, volume 1166 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 1996.

[BBDEL96] I. Beer, Sh. Ben-David, C. Eisner, and A. Landver. RuleBase: An industry-oriented formal verification tool. In *Proceedings of the 33rd ACM Design Automation Conference, Las Vegas, NV, USA*, 1996.

[Bes96] E. Best. A memory module specification using composable high-level Petri Nets. In Broy et al. [BMS96], pages 109–160.

[BH95a] J.P. Bowen and M.G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

[BH95b] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995.

[BJ96a] M. Bickford and D. Jamsek. Formal specification and verification of VHDL. In M. Srivas and A. Camilleri, editors, *FMCAD'96, Palo Alto, CA, USA*, volume 1166 of *Lecture Notes in Computer Science*, pages 310–326. Springer-Verlag, 1996.

[BJ96b] J. Blom and B. Jonsson. Constraint oriented temporal logic specification. In Broy et al. [BMS96], pages 161–182.

[BKKW90] S. van de Burgt, J. Kroon, E. Kwast, and H. Wilts. The RNL Conformance Kit. In *Proceedings 2nd International Workshop on Protocol Test Systems*, pages 279–94. North-Holland, 1990.

[BL96] M. Broy and L. Lamport. The RPC-Memory specification problem – Problem statement. In Broy et al. [BMS96], pages 1–4.

[BMS96] M. Broy, S. Merz, and K. Spies, editors. *Formal Systems Specification – The RPC-Memory Specification Case Study*, volume 1169 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Bri88] E. Brinksma. A theory for the derivation of tests. In S. Aggrawal and K. Sabani, editors, *Protocol Specification Testing and Verification, Volume VIII*, pages 63–74. North-Holland, 1988.

[Bro96] M. Broy. A functional solution to the RPC-Memory specification problem. In Broy et al. [BMS96], pages 183–212.

[BSW69] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full–duplex transmission over half–duplex links. *Communications of the ACM*, 12:260–261, 1969.

[BTV91] E. Brinksma, J. Tretmans, and L. Verhaard. A framework for test selection. In B. Jonsson, J. Parrow, and B. Pehrson, editors, *Protocol Specification Testing and Verification, Volume XI*, pages 233–248. North-Holland, 1991.

[Cad] Cadence. Leapfrog VHDL simulator. Product information at http://www.cadence.com/software/qx-leap.html.

[CBH96] J.R. Cuéllar, D. Barnard, and M. Huber. A solution relying on the model checking of boolean transition systems. In Broy et al. [BMS96], pages 213–252.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CFJ93] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis [Cou93], pages 450–462.

[CG97] O. Charles and R. Groz. Basing test coverage on a formalization of test hypotheses. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems, Volume 10*, pages 109–124. Chapman & Hall, 1997.

[Cho78] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–188, 1978.

[Cou93] C. Courcoubetis, editor. *Proceedings* 5th *International Conference on Computer Aided Verification (CAV '93)*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[CVI89] W.Y.L. Chan, S.T. Vuong, and M.R. Ito. An improved protocol test generation procedure based on UIOs. In *Proceedings of the ACM Symposium on Communication Architectures and Protocols*, pages 283–294, 1989.

[CW96] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[DAC98] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, March 1998. ACM Press.

[Dam98] D. Dams. Personal communication, October 1998.

[DGRV97] M.C.A. Devillers, W.O.D. Griffioen, J.M.T. Romijn, and F.W. Vaandrager. Verification of a leader election protocol — formal methods applied to IEEE 1394. Technical Report CSI-R9728, Computing Science Institute, University of Nijmegen, December 1997. Accepted, subject to revision, for Formal Methods in System Design.

[DNFGR93] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action based framework for verifying logical and behavioural properties of concurrent systems. *Computer Networks and ISDN Systems*, 25:761–778, 1993.

[DNV90] R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Proceedings of Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag, 1990.

[DS95] J.W. Davies and S.A. Schneider. A brief history of timed csp. *Theoretical Computer Science*, 138(2):243–271, 1995.

[EJP97] E.A. Emerson, S. Jha, and D. Peled. Combining partial order and symmetry reductions. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 19–34. Springer-Verlag, 1997.

[EL87] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, June 1987.

[ES93] E.A. Emerson and A.P. Sistla. Symmetry and model checking. In Courcoubetis [Cou93], pages 463–478.

[ES97] E.A. Emerson and A.P. Sistla. Utilizing symmetry when model-checking under fairness assumptions: an automata-theoretic approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, 1997.

[FBK+91] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer-Verlag, August 1996. Information and tool set available from URL http://www.inrialpes.fr/ vasy/pub/cadp.html.

[FJJV96] J. C. Fernandez, C. Jard, T. Jeron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer Verlag, July 1996.

[FMMW98] L.M.G. Feijs, F. Meijs, J.R. Moonen, and J.J. van Wamel. Conformance testing of a multimedia system using PHACT. In A. Petrenko and N. Yevtushenko, editors, *Testing of Communicating Systems*, volume 11, pages 193–210. Kluwer Academic Publishers, September 1998.

[Gar98] H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In Steffen [Ste98], pages 68–84. For more information on the tool set, see http://www.inrialpes.fr/vasy/pub/cadp.html.

[Gau95] M.-C. Gaudel. Testing can be formal, too. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1995.

[GFL+96] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal. Coverage-directed test generation using symbolic techniques. In M. Srivas and A. Camilleri, editors, *FMCAD'96,* Palo Alto, CA, USA, volume 1166 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 1996.

[GH93] J.V. Guttag and J.J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[GHM+98] Grundig, Hitachi, Matsushita, Philips, Sharp, Sony, Thomson, and Toshiba. The HAVi Specification – Specification of the Home Audio/Video Interoperability (HAVi) Architecture. Version 1.0 beta, November 19, 1998. Available from URL http://www.havi.org/.

[GLV97] S.J. Garland, N.A. Lynch, and M. Vaziri. IOA: A language for specifying, programming, and validating distributed systems, December 1997. Available through URL http://larch.lcs.mit.edu:8001/~garland/ ioaLanguage.html.

[God96] P. Godefroid. *Partial-order methods for the verification of concurrent systems – An approach to the state-explosion problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Got96] R. Gotzhein. Applying a temporal logic to the RPC-Memory specification problem. In Broy et al. [BMS96], pages 253–274.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer-Verlag, 1995.

[GS] J.F. Groote and J.G. Springintveld. Focus points and convergent process operators — a proof strategy for protocol verification. In A. Arnold, editor, *Proceedings 2nd AMAST Workshop on Real-Time Systems (ARTS'95)*. To appear. Report versions: Logic Group Preprint Series 142, Utrecht University, 1995, and Technical Report CS-R9566, CWI, 1995.

[GS97] V. Gyuris and A.P. Sistla. On-the-fly model checking under fairness that exploits symmetry. In O. Grumberg, editor, *Proceedings* 9th *International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 232–243. Springer-Verlag, 1997.

[GV98] W.O.D. Griffioen and F.W. Vaandrager. Normed simulations. In *Proceedings CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 332–344. Springer-Verlag, 1998.

[Hal90] J.A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[Hol98a] M. Hollenberg. Testen van een digitale TV. Presentation at the Fourth Dutch Testing Conference, 1998.

[Hol98b] G.J. Holzmann. Personal communication, November 1998.

[Hol99] G.J. Holzmann. Personal communication, June 1999.

[Hoo96] J. Hooman. Using PVS for an assertional verification of the RPC-Memory specification problem. In Broy et al. [BMS96], pages 275–304.

[HP94] G.J. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings Formal Description Techniques, FORTE94*, pages 197–211, Bern, Switzerland, October 1994. Chapman & Hall.

[Hun96] H. Hungar. Specification and verification using a visual formalism on top of temporal logic. In Broy et al. [BMS96], pages 305–337.

[HYHD95] R. Ho, C.H. Yang, M.A. Horowitz, and D. Dill. Architecture validation for processors. In *Proceedings of the International Symposium on Computer Architecture, Santa Margerita Ligure, Italy*, 1995.

[ID96] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, August 1996.

[IEE93] IEEE Computer Society. Standard VHDL language reference manual (ANSI). International standard 1076, 1993.

[IEE95] IEEE Computer Society. Standard description language based on the Verilog(TM) hardware description language. International standard 1364, 1995.

[IEE96] IEEE Computer Society. IEEE Standard for a High Performance Serial Bus. Std 1394-1995, August 1996.

[IEE99] IEEE Computer Society. Draft Standard for a High Performance Serial Bus (Supplement). P1394a Draft 3.0, June 1999.

[ISO89] ISO. Information processing systems – Open Systems Interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour. ISO/IEC 8807, 1989.

[ISO91] ISO. Information technology, open systems interconnection, conformance testing methodology and framework. International standard IS–9646, 1991.

[Jen99] H.E. Jensen. *Abstraction-Based Verification of Distributed Systems*. PhD thesis, Aalborg University, June 1999.

[Jon94] B. Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, March 1994.

[KHR97] L. Kühne, J. Hooman, and W.P. de Roever. Towards mechanical verification of parts of the IEEE P1394 serial bus. In I. Lovrek, editor, *Proceedings 2nd International Workshop on Applied Formal Methods in System Design,* Zagreb, pages 73–85, 1997.

[KK97] S. Kang and M. Kim. Interoperability test suite derivation for symmetric communication protocols. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE X/ PSTV XVII '97)*, pages 57–72. Chapman & Hall, 1997.

[Kni93] K.G. Knightson. *OSI Protocol Conformance Testing: IS 9646 Explained*. McGraw-Hill, 1993.

[KNS96] N. Klarlund, M. Nielsen, and K. Sunesen. A case study in verification based on trace abstractions. In Broy et al. [BMS96], pages 341–374.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KP98] Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to practical formal verification. In L. Brim, J. Gruska, and J. Zlatuska, editors, *The 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS 1998)*, volume 1450 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, 1998.

[KS96] R. Kurki-Suonio. Incremental specification with joint actions: The RPC-Memory specification problem. In Broy et al. [BMS96], pages 375–404.

[KV98] O. Kupferman and M.Y. Vardi. Relating linear and branching model checking. In D. Gries and W.-P. de Roever, editors, *IFIP Working Conference on Programming Concepts and Methods*, New York, June 1998. Chapman & Hall.

[KVZ98] H. Kahlouche, C. Viho, and M. Zendri. An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In A. Petrenko and N. Yevtushenko, editors, *Testing of Communicating Systems*, volume 11. Kluwer Academic Publishers, September 1998.

[KVZ99] H. Kahlouche, C. Viho, and M. Zendri. Hardware testing using a communication protocol conformance testing tool. In W.R. Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 315–329. Springer Verlag, March 1999.

[KWKK91] E. Kwast, H. Wilts, H. Kloosterman, and J. Kroon. User manual of the Conformance Kit, October 1991.

[Lam94a] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam94b] Leslie Lamport. How to write a long formula. *Formal Aspects of Computing*, 6:580–584, 1994. Also appeared as SRC Research Report 119.

[LSW96] K.G. Larsen, B. Steffen, and C. Weise. The methodology of modal constraints. In Broy et al. [BMS96], pages 405–436.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the* 6th *Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[Lut97] S.P. Luttik. Description and formal specification of the Link layer of P1394. In I. Lovrek, editor, *Proceedings of the 2nd International Workshop on Applied Formal Methods in System Design,* Zagreb, pages 43–56, 1997. Also available as Report SEN-R9706, CWI, Amsterdam. See URL http://www.cwi.nl/~luttik/.

[LV95] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations, I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[LV96] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations, II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[MAD96] F. Michel, P. Azema, and K. Drira. Selective generation of symmetrical test cases. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *Testing of Communicating Systems, Volume 9*, pages 191–206. Chapman & Hall, 1996.

[Mat98] R. Mateescu. *Vérification des propriétés temporelles des programmes parallèles*. Thèse de doctorat, Institut National Polytechnique de Grenoble, April 1998.

[MG98] R. Mateescu and H. Garavel. XTL: A meta-language and tool for temporal logic model-checking. In T. Margaria and B. Steffen, editors, *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98*, number NS-98-4 in BRICS Notes Series, July 1998.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[MP93] Z. Manna and A. Pnueli. Verifying hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 4–35. Springer-Verlag, 1993.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[MRS$^+$96] J.R. Moonen, J.M.T. Romijn, O. Sies, J.G. Springintveld, L.M.G. Feijs, and R.L.C. Koymans. A course in using Phact: Testing the conformance of a small protocol, November 1996.

[MRS$^+$97a] J.R. Moonen, J.M.T. Romijn, O. Sies, J.G. Springintveld, L.M.G. Feijs, and R.L.C. Koymans. A two-level approach to automated conformance testing of VHDL designs. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, volume 10, pages 432–447. Chapman and Hall, 1997.

[MRS$^+$97b] J.R. Moonen, J.M.T. Romijn, O. Sies, J.G. Springintveld, L.M.G. Feijs, and R.L.C. Koymans. Phact: A tool environment for automated conformance testing of VHDL designs – User manual and documentation, 1997.

[ORSH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Pec98] C. Pecheur. Advanced modelling and verification techniques applied to a cluster file system. Technical Report 3416, INRIA Rhône-Alpes, May 1998.

[PHK95] A. Petrenko, T. Higashino, and T. Kaji. Handling redundant and additional states in protocol testing. In A. Cavalli and S. Budkowski, editors, *Protocol Test Systems, Volume VIII*, pages 307–322. Chapman & Hall, 1995.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 46–57. IEEE, 1977.

[Pnu85] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proceedings of 12th International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, pages 15–32. Springer-Verlag, 1985.

[QS83] J.-P. Queille and J. Sifakis. Fairness and related properties in transition systems – a temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983.

[RAH98] E. Riccobene, M.M. Archer, and C.L. Heitmeyer. Applying TAME to I/O automata: A user's perspective, 1998. Draft.

[Rom96] J.M.T. Romijn. Tackling the RPC-Memory specification problem with I/O automata. In Broy et al. [BMS96], pages 437–476.

[Rom99a] J.M.T. Romijn. Model checking the HAVi leader election protocol. Technical Report SEN-R9915, CWI, Amsterdam, 1999. Submitted.

[Rom99b] J.M.T. Romijn. A timed verification of the IEEE 1394 leader election protocol. In S. Gnesi and D. Latella, editors, *Proceedings of the Fourth International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'99)*, pages pages 3–29, 1999. Full version available as CWI Report SEN-R9919.

[RS98] J.M.T. Romijn and J.G. Springintveld. Exploiting symmetry in protocol testing. In S. Budkowski, A. Cavalli, and E. Najm, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE XI/PSTV XVIII '98)*, pages 337–352. Kluwer Academic Publishers, 1998. Full version available as CWI Report SEN-R9918.

[Rus95] J. Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, March 1995.

[RV96] J.M.T. Romijn and F.W. Vaandrager. A note on fairness in I/O automata. *Information Processing Letters*, 59(5):245–250, September 1996.

[Sch97] S.A. Schneider. Timewise refinement for communicating processes. *Science of Computer Programming*, 28(1):43–90, 1997.

[SD88] K.K. Sabnani and A.T. Dahbura. A protocol testing procedure. *Computer Networks and ISDN Systems*, 15(4):285–297, 1988.

[Seg95] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1995. Available as Technical Report MIT/LCS/TR-676.

[SGSL98] R. Segala, R. Gawlick, J.F. Søgaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141(2):119–171, March 1998.

[Sie96] O. Sies. Automatic techniques for protocol conformance testing, 1996. Master's Thesis.

[SL95] R. Segala and N.A. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.

[SM98] M. Sighireanu and R. Mateescu. Verification of the link layer protocol of the ieee-1394 serial bus (firewire): an experiment with e-lotos. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2(1):68–88, December 1998.

[Ste98] B. Steffen, editor. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, volume 1384 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[Sti92] C. Stirling. Modal and temporal logics. In S. Abramsky, Dov M. Gabbay, and T. S.Ẽ. Maibaum, editors, *Handbook of Logic in Computer Science. Volume 2. Background: Computational Structures*, pages 477–563. Oxford University Press, 1992.

[Stø96] K. Stølen. Using relations on streams to solve the RPC-Memory specification problem. In Broy et al. [BMS96], pages 477–520.

[Sun99] Sun Microsystems, Inc. Jini Connection Technology, 1999. Specifications available from URL http://www.sun.com/jini/whitepapers/.

[SV99] M. I. A. Stoelinga and F. W. Vaandrager. Root contention in IEEE 1394. In J.-P. Katoen, editor, *Proceedings 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems (ARTS'99),* Bamberg, Germany, volume 1601 of *Lecture Notes in Computer Science*, pages 53–74. Springer-Verlag, 1999.

[SZ98] C. Shankland and M.B. van der Zwaag. The tree identify protocol of IEEE 1394 in μCRL. *Formal Aspects of Computing*, 10(5/6):509–531, 1998.

[TLH^{+}99] T. Trew, B. Lanaspre, M. Hollenberg, J.G. Springintveld, and T. Harosia. Delivering high definition TV to the USA – testing subcontracted embedded real-time software. In *Proceedings of 16th International Conference and Exposition on Testing Computer Software,* Washington D.C., June 1999.

[TPHT96] R. Terpstra, L. Ferreira Pires, L. Heerink, and J. Tretmans. Testing theory in practice: A simple experiment. In *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design*, 1996. Also published as Technical Report CTIT 96-21, University of Twente, The Netherlands.

[Tre89] J. Tretmans. A theory for the derivation of tests. In *Formal Description Techniques (FORTE II '89)*. North-Holland, 1989.

[Tre92] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, December 1992.

[UK96] R.T. Udink and J.N. Kok. The RPC-Memory specification problem: UNITY + refinement calculus. In Broy et al. [BMS96], pages 521–540.

[Use99] Y.S. Usenko. A comparison of Spin and the μCRL toolset on HAVi leader election protocol. Technical Report SEN-R9917, CWI, Amsterdam, 1999. Submitted.

[Vas73] M.P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9(4):653–665, 1973.

[WH96] P. Walsh and D. Hoffman. Automated behavioral testing of VHDL components. In *Canadian Conference on Electrical and Computer Engineering, Calgary, Alberta, Canada*, May 1996.

[WM97] H. Wupper and H. Meijer. A taxonomy for computer science. Technical Report CSI-R9713, Computing Science Institute, University of Nijmegen, August 1997. Also available via URL http://www.cs.kun.nl/ wupper/taxonomy/taxonomy-frame.html.

[WM98] H. Wupper and H. Meijer. Towards a taxonomy for computer science. In F. Mulder and T. van Weert, editors, *Informatics as a discipline and in other disciplines: what is in common? Informatics in Higher Education – IFIP WG 3.2 Working Conference*. Chapman & Hall, 1998.

[Yi90] Wang Yi. Real-time behaviour of asynchronous agents. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 502–520. Springer-Verlag, 1990.

Appendix A

I/O automata

In this appendix we review some basic definitions from [SGSL98, LV95, LV96, MP95], and we give some new sufficient conditions. The sufficient conditions for including invariants in refinement proofs, when the invariants at the refined level depend on invariants at the abstract level, are presented in Lemma A.2 and Lemma A.7. The sufficient conditions for feasibility are presented in Lemma A.4 and Theorem A.5. In Appendix A.4, we introduce the *fair timed I/O automaton*, which extends the timed I/O automaton in [LV96] with fairness properties.

A.1 Safe I/O automata

A *safe I/O automaton* B consists of the following components:

- A set $states(B)$ of *states* (possibly infinite).
- A nonempty set $start(B) \subseteq states(B)$ of *start states*.
- A set $acts(B)$ of *actions*, partitioned into three sets $in(B)$, $int(B)$ and $out(B)$ of *input*, *internal* and *output* actions, respectively.
  Actions in $local(B) \triangleq out(B) \cup int(B)$ are called *locally controlled*.
- A set $steps(B) \subseteq states(B) \times acts(B) \times states(B)$ of *transitions*, with the property that for every state s and input action $a \in in(B)$ there is a transition $(s, a, s') \in steps(B)$.

We let s, s',.. range over states, and a,.. over actions. We write $s \xrightarrow{a}_B s'$, or just $s \xrightarrow{a} s'$ if B is clear from the context, as a shorthand for $(s, a, s') \in steps(B)$.

Enabling of actions An action a of a safe I/O automaton B is *enabled* in a state s iff $s \xrightarrow{a} s'$ for some s'. Since every input action is enabled in every state, safe I/O automata are said to be *input enabled*. The intuition behind the input-enabling condition is that input actions are under control of the environment and that the system that is modeled by a safe I/O automaton cannot prevent the environment from doing these actions.

Executions An *execution fragment* of a safe I/O automaton B is a finite or infinite alternating sequence $s_0 a_1 s_1 a_2 s_2 \cdots$ of states and actions of B, beginning with a state, and if it is finite also ending with a state, such that for all i, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. An *execution* is an execution fragment that begins with a start state. We write $execs^*(B)$ for the set of finite executions of B, and $execs(B)$ for the set of all executions of B. A state s of B is *reachable* if it is the last state of some finite execution of B. We write $rstates(B)$ for the set of reachable states of B.

Traces Suppose $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ is an execution fragment of B. Let $\gamma = a_1 a_2 \cdots$. Then the *trace* of α is the sequence $(\gamma \lceil in(B) \cup out(B))$, denoted by $\hat{\gamma}$. With $traces(B)$ we denote the set of traces of executions of B. For s, s' states of B and β a finite sequence of input and output actions of B, we define $s \overset{\beta}{\Rightarrow}_B s'$ iff B has a finite execution fragment with first state s, last state s' and trace β.

Implementation relation Let A and B be safe I/O automata.
A implements B if $traces(A) \subseteq traces(B)$.

Invariants Let $P, Q \subseteq states(B)$. P is *invariant* for B if it is a superset of the reachable states of B, i.e. $rstates(B) \subseteq P$. P is *inductive relative to Q* if $start(B) \subseteq P$ and if for each $s' \in P \cap Q$: $s' \overset{a}{\rightarrow}_B s$ implies $s \in P$.

Refinements Let A and B be safe I/O automata. A *refinement* from A to B is a function $r : states(A) \rightarrow states(B)$ that satisfies:

1. If $s \in start(A)$ then $r(s) \in start(B)$.
2. If $s' \overset{a}{\rightarrow}_A s$ then $r(s') \overset{\beta}{\Rightarrow}_B r(s)$, where $\beta = \hat{a}$.

Let A and B be safe I/O automata with invariants P and Q, respectively. A *weak refinement* from A to B, with respect to P and Q, is a function $r : states(A) \rightarrow states(B)$ that satisfies:

1. If $s \in start(A)$ then $r(s) \in start(B)$.
2. If $s' \overset{a}{\rightarrow}_A s$, $s' \in P$, and $r(s') \in Q$, then $r(s') \overset{\beta}{\Rightarrow}_B r(s)$, where $\beta = \hat{a}$.

Theorem A.1 Let A and B be safe I/O automata. If there exists a (weak) refinement from A to B, then $traces(A) \subseteq traces(B)$.

Using abstract and refined invariants in a refinement Let A, B be safe I/O automata. The following lemma gives sufficient conditions for a weak refinement from A to B when one wants to use P_1, P_2, Q such that Q is invariant for B, P_1 is invariant for A depending on Q and the definition of the refinement function, and P_2 is invariant for A depending on P_1.

Lemma A.2 Let A, B be safe I/O automata. Let Q be invariant for B and P_2 be inductive relative to P_1 for A. Let $r : states(A) \rightarrow states(B)$ such that

1. $r(s) \in Q$ implies $s \in P_1$,
2. $s \in start(A)$ implies $r(s) \in start(B)$, and

3. $s' \in P_2$, $r(s') \in Q$ and $s' \xrightarrow{a}_A s$ implies $r(s') \overset{\beta}{\Rightarrow}_B r(s)$, where $\beta = \hat{a}$.

Then

1. P_1, P_2 are invariant for A.
2. r is a weak refinement from A to B with respect to P_2 and Q.

Proof

1. By induction.

$$\begin{aligned} \mathrm{IH}(n) \quad = \quad \forall s, \alpha : \quad & (s \in \mathit{rstates}(A) \wedge \alpha \in \mathit{execs}(A) \wedge \alpha = s_0 a_1 s_1 \ldots s_n \wedge s = s_n) \\ & \rightarrow (r(s) \in \mathit{rstates}(B) \wedge s \in (P_1 \cap P_2)) \end{aligned}$$

   - Base step: $n = 0$.
     By definition of α, $s \in \mathit{start}(A)$. By definition of r, $r(s) \in \mathit{start}(B)$ so certainly $r(s) \in \mathit{rstates}(B)$. Since Q is invariant for B, $r(s) \in Q$. By definition of r, $s \in P_1$. Since $s \in \mathit{start}(A)$, and since P_2 is inductive relative to P_1 for A, $s \in P_2$.
   - Induction step: $\forall n \leq n' : \mathrm{IH}(n)$.
     Let $s \in \mathit{rstates}(A) \wedge \alpha \in \mathit{execs}(A) \wedge \alpha = s_0 a_1 s_1 \ldots s_{n'} a_{n'+1} s_{n'+1} \wedge s = s_{n'+1}$. Since $s_0 a_1 s_1 \ldots s_{n'} \in \mathit{execs}(A)$, certainly $s_{n'} \in \mathit{rstates}(A)$. Combining this with $n' \leq n'$, we get $\mathrm{IH}(n')$. Since $\mathrm{IH}(n')$, $r(s_{n'}) \in \mathit{rstates}(B) \wedge s_{n'} \in (P_1 \cap P_2)$. Since $r(s_{n'}) \in \mathit{rstates}(B)$ and Q is invariant for B, $r(s_{n'}) \in Q$. Since $s_{n'} \xrightarrow{a_{n'+1}}_A s_{n'+1}$ and by definition of r, $r(s_{n'}) \overset{\beta}{\Rightarrow}_B r(s_{n'+1})$ with $\beta = \widehat{a_{n'+1}}$, hence $r(s_{n'+1}) \in \mathit{rstates}(B)$, hence $r(s_{n'+1}) \in Q$. By definition of r, $s_{n'+1} \in P_1$. Since $s_{n'} \in (P_1 \cap P_2)$ and $s_{n'} \xrightarrow{a_{n'+1}}_A s_{n'+1}$ and since P_2 is inductive relative to P_1 for A, $s_{n'+1} \in P_2$.

2. By Item 1, the assumption that Q is invariant for B and by definition of r.

⊠

Composition Two safe I/O automata B_1 and B_2 are *compatible* iff $\mathit{out}(B_1) \cap \mathit{out}(B_2) = \emptyset$, $\mathit{int}(B_1) \cap \mathit{acts}(B_2) = \emptyset$, and $\mathit{int}(B_2) \cap \mathit{acts}(B_1) = \emptyset$. The *composition* $B_1 \| B_2$ of compatible safe I/O automata B_1 and B_2 is the safe I/O automaton B defined by

- $\mathit{states}(B) = \mathit{states}(B_1) \times \mathit{states}(B_2)$,
- $\mathit{start}(B) = \mathit{start}(B_1) \times \mathit{start}(B_2)$,
- $\mathit{acts}(B) = \mathit{in}(B) \cup \mathit{out}(B) \cup \mathit{int}(B)$, where

$$\begin{aligned} \mathit{in}(B) &= (\mathit{in}(B_1) \cup \mathit{in}(B_2)) - (\mathit{out}(B_1) \cup \mathit{out}(B_2)), \\ \mathit{out}(B) &= \mathit{out}(B_1) \cup \mathit{out}(B_2), \\ \mathit{int}(B) &= \mathit{int}(B_1) \cup \mathit{int}(B_2), \end{aligned}$$

- $\mathit{steps}(B)$ is the set of triples $((s_1, s_2), a, (s_1', s_2'))$ in $\mathit{states}(B) \times \mathit{acts}(B) \times \mathit{states}(B)$ such that, for $i \in \{1, 2\}$, if $a \in \mathit{acts}(B_i)$ then $s_i \xrightarrow{a}_{B_i} s_i'$ else $s_i = s_i'$.

A.2 Live I/O automata

Intuitively, a *live I/O automaton* is a pair of a safe I/O automaton B and a set L of executions of B such that B can always generate an execution in L independently of the input provided by its environment. Formally, live I/O automata can be defined in terms of a two person game between a system player and an environment player. The goal of the system player is to construct an execution in L, and the goal of the environment player is to prevent this. The pair (B, L) is a live I/O automaton iff there exists a strategy by which the system player can always win the game, irrespective of the behaviour of the environment player.

A *strategy* defined on a safe I/O automaton B is a pair of functions (g, f) where $g : \mathit{execs}^*(B) \times \mathit{in}(B) \rightarrow \mathit{states}(B)$ and $f : \mathit{execs}^*(B) \rightarrow (\mathit{local}(B) \times \mathit{states}(B)) \cup \{\bot\}$ such that

1. $g(\alpha, a) = s \;\Rightarrow\; \alpha\, a\, s \in \mathit{execs}^*(B)$,
2. $f(\alpha) = (a, s) \;\Rightarrow\; \alpha\, a\, s \in \mathit{execs}^*(B)$.

An *environment sequence* for B is an infinite sequence of symbols from $\mathit{in}(B) \cup \{\lambda\}$ with infinitely many occurrences of λ. The symbol λ represents the points at which the system is allowed to move. The occurrence of infinitely many λ symbols in an environment sequence guarantees that each environment move consists of only finitely many input actions.

Let $\rho = (g, f)$ be a strategy defined on B, $\mathcal{I} = a_1 a_2 a_3 \cdots$ an environment sequence for B, and α a finite execution of B. Then the *outcome* $\mathcal{O}_\rho(\alpha, \mathcal{I})$ is the limit of the sequence $(\alpha_i)_{i \geq 0}$ of finite executions defined inductively by

- $\alpha_0 = \alpha$.
- If $i > 0$ then
  1. $a_i = \lambda \wedge f(\alpha_{i-1}) = (a, s) \;\Rightarrow\; \alpha_i = \alpha_{i-1}\, a\, s$,
  2. $a_i = \lambda \wedge f(\alpha_{i-1}) = \bot \;\Rightarrow\; \alpha_i = \alpha_{i-1}$,
  3. $a_i \in \mathit{in}(B) \wedge g(\alpha_{i-1}, a_i) = s \;\Rightarrow\; \alpha_i = \alpha_{i-1}\, a_i\, s$.

A *live I/O automaton* is a pair (B, L) with B a safe I/O automaton and $L \subseteq \mathit{execs}(B)$ such that there exists a strategy ρ defined on B with for any finite execution α of B and any environment sequence $\mathcal{I}$ for B, $\mathcal{O}_\rho(\alpha, \mathcal{I}) \in L$.

Composition Let (B_1, L_1) and (B_2, L_2) be live I/O automata. (B_1, L_1) and (B_2, L_2) are *compatible* iff B_1 and B_2 are compatible. The *composition* $(B_1, L_1) \| (B_2, L_2)$ of two compatible live I/O automata (B_1, L_1) and (B_2, L_2) is the pair (B, L) defined by

- $B = B_1 \| B_2$,
- $L = \{\alpha \in \mathit{execs}(B) \mid \alpha \lceil B_1 \in L_1 \text{ and } \alpha \lceil B_2 \in L_2\}$.
  Here $\alpha \lceil B_i$ is obtained by projecting each state in α on the i-th component and by removing each action that is not in $\mathit{acts}(B_i)$ together with the state that follows it.

A major result of [SGSL98] is that the class of live I/O automata is closed under composition.

Theorem A.3 Let (B_1, L_1) and (B_2, L_2) be compatible live I/O automata.
Then $(B_1, L_1) \| (B_2, L_2)$ is a live I/O automaton.

A.3 Timed I/O automata

A *timed I/O automaton A* is a safe I/O automaton whose set of actions includes R^+, the set of positive reals. Actions from R^+ are referred to as *time-passage actions*. Other actions are referred to as *discrete actions*. Performing one or more consecutive time-passage actions is called *idling*. We let $d, d', \ldots$ range over R^+ and, more generally, $t, t', \ldots$ over the set R of real numbers. The set of *visible* actions is defined by $vis(A) \triangleq (in(A) \cup out(A)) - \mathsf{R}^+$.
We assume that a timed I/O automaton satisfies the following axioms.

S1 If $s' \xrightarrow{d} s''$ and $s'' \xrightarrow{d'} s$, then $s' \xrightarrow{d+d'} s$.

For the second axiom, an auxiliary definition is needed. A *trajectory* for a step $s' \xrightarrow{d} s$ is a function $w : [0, d] \to states(A)$ such that $w(0) = s'$, $w(d) = s$, and

$$w(t) \xrightarrow{t'-t} w(t') \text{ for all } t, t' \in [0, d] \text{ with } t < t'.$$

Now we can state the second axiom.

S2 Each step $s \xrightarrow{d} s'$ has a trajectory.

Axiom **S1** gives a natural property of time, namely that if time can pass in two steps, then it can also pass in a single step. The *trajectory axiom* **S2** is a kind of converse to **S1**; it says that any time-passage step can be "filled in" with states for each intervening time, in a "consistent" way. Executions of timed I/O automata correspond to what are called *sampling computations* in [MP93].

Timed traces The full externally visible behaviour of a timed I/O automaton can be inferred from its executions as follows: suppose $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ is an execution fragment of a timed I/O automaton A. For each index j, let t_j be given by

$$\begin{aligned} t_0 &= 0, \\ t_{j+1} &= \text{if } a_{j+1} \in \mathsf{R}^+ \text{ then } t_j + a_{j+1} \text{ else } t_j. \end{aligned}$$

The *limit time* of α, notation $\alpha.ltime$, is the smallest element of $\mathsf{R}^{\geq 0} \cup \{\infty\}$ larger than or equal to all the t_j. We say α is *admissible* if $\alpha.ltime = \infty$, and *Zeno* if it is an infinite sequence but with a finite limit time. The *timed trace t-trace*(α) associated with α is defined by

$$t\text{-}trace(\alpha) \quad \triangleq \quad (((a_1, t_1)(a_2, t_2) \cdots)\lceil(vis(A) \times \mathsf{R}^{\geq 0}), \alpha.ltime).$$

Thus, *t-trace*(α) records the visible actions of α paired with their times of occurrence, as well as the limit time of the execution. A pair β is a *timed trace* of A if it is the timed trace of some finite or admissible execution of A. Thus, we explicitly exclude the timed traces that originate from Zeno executions. We write *t-traces*(A) for the set of all timed traces of A, *t-traces*$^*(A)$ for the set of *finite* timed traces (the timed traces derived from the finite executions), and *t-traces*$^\infty(A)$ for the set of *admissible* traces (the timed traces derived from the admissible executions).

Moves We say $s' \overset{p}{\leadsto}_A s$ is a *t-move* of A if A has a finite timed execution fragment $\alpha = s_0 a_1 s_1 \ldots s_n$ such that $s' = s_0$, $s = s_n$ and $p = t\text{-}trace(\alpha)$.

Feasibility Let A be a timed I/O automaton. We say A is *feasible* if each element of *t-traces*$^*(A)$ is the prefix of some element of *t-traces*$^\infty(A)$.

Giving the proof for feasibility can be hard or tiresome. However, in some cases it follows rather straightforwardly from the definition of the timed I/O automaton. We give the following sufficient conditions, divided over two results, of which the first is rather simple, and the second is a bit more involved.

Lemma A.4 Let A be a timed I/O automaton with clock variables X and discrete variables Y. If

1. The precondition of time action d is of the following form, in which $\phi, \psi_1, \ldots, \psi_n$ are Boolean expressions over variables in $Y, x_1, \ldots, x_n \in X$ and $c_1, \ldots, c_n \in \mathsf{R}^+$:

   $$\neg\phi \wedge (\psi_1 \rightarrow x_1 + d \leq c_1) \wedge \cdots \wedge (\psi_n \rightarrow x_n + d \leq c_n)$$

2. The effect of time action d is of the following form:

   $$\forall x \in X : x := x + d$$

3. For each $s \in$ reachable(A):

   $$(s \models \phi) \rightarrow \exists a : s \xrightarrow{a} \wedge\, a \text{ is discrete}$$

4. For each $s \in$ reachable(A) and $0 \leq i \leq n$:

   $$(s \models \psi_i \wedge x_i \geq c_i) \rightarrow \exists a : s \xrightarrow{a} \wedge\, a \text{ is discrete}$$

then for each $s \in$ reachable(A) and $d > 0$, the following holds:

$$\begin{array}{ll} \vee & s \xrightarrow{d} \\ \vee & \exists a : s \xrightarrow{a} \wedge\, a \text{ is discrete} \\ \vee & \exists d', a, s' : d' < d \wedge s \xrightarrow{d'} s' \xrightarrow{a} \wedge a \text{ is discrete} \end{array}$$

Proof Suppose $s \in$ reachable(A), $d > 0$ and s does not enable d. Then $s \models \neg(\neg\phi \wedge (\psi_1 \rightarrow x_1 + d \leq c_1) \wedge \cdots \wedge (\psi_n \rightarrow x_n + d \leq c_n))$, which can easily be rewritten to $s \models \phi \vee (\psi_1 \wedge x_1 + d > c_1) \vee \cdots \vee (\psi_n \wedge x_n + d > c_n)$.

Suppose $s \models \phi$. By Assumption 3, there is a discrete action a such that $s \xrightarrow{a}$, and the result follows.

Suppose $s \models \neg\phi$. Then $s \models (\psi_1 \wedge x_1 + d > c_1) \vee \cdots \vee (\psi_n \wedge x_n + d > c_n)$. Take J to be the set of indices for which the disjunct is true, that is, $J = \{i | 1 \leq i \leq n \wedge s \models \psi_i \wedge x_i + d > c_i)\}$.

Suppose that for some $i \in J$, $s \models x_i \geq c_i$. Then by Assumption 4, there is a discrete action a such that $s \xrightarrow{a}$, and the result follows.

Suppose that for all $i \in J$, $s \models x_i < c_i$. Take d' to be the smallest value such that for some $i \in J$, $s \models x_i + d' = c_i$. Fix i. We now have for each $j \in J$, $s \models x_j \leq c_j$. It is clear that for each $1 \leq j \leq n$ which is not in J, $s \models x_j + d' \leq c_j$. By assumption, $s \models \neg\phi$, so we now see that s enables d'. Let $s \xrightarrow{d'} s'$. By Assumption 2, and $s \models x_i + d' = c_i$, the effect of d' is such that $s' \models x_i = c_i$. Now by Assumption 4, there is a discrete action a such that $s \xrightarrow{a}$, and the result follows. ⊠

Theorem A.5 Let A be a timed I/O automaton.
If

1. For each $s \in \text{reachable}(A)$ and $d > 0$, the following holds:

$$\begin{array}{ll} \vee & s \overset{d}{\rightarrow} \\ \vee & \exists a : s \overset{a}{\rightarrow} \wedge\, a \text{ is discrete} \\ \vee & \exists d', a, s' : d' < d \wedge s \overset{d'}{\rightarrow} s' \overset{a}{\rightarrow} \wedge a \text{ is discrete} \end{array}$$

2. Function $M : \mathit{states}(A) \rightarrow D$ is a measure function, $\prec$ is a well-founded ordering on D, and $C \in \mathsf{R}^+$ is a constant such that for each $s, s' \in \text{reachable}(A)$: $s \overset{a}{\rightarrow} s'$ implies that if a is discrete and s does not enable C, then $M(s') \prec M(s)$, otherwise $M(s') \preceq M(s)$.

then A is feasible.

Proof Suppose $\alpha \in \mathit{t\text{-}traces}^{\infty}(A)$. We define the function f that recursively builds an admissible execution from any state, as follows:

$$f(s) = \begin{cases} s\ C\ f(s') & \mathit{if}\, s \overset{C}{\rightarrow} s' \\ s\ a\ f(s') & \mathit{if}\, s \overset{C}{\not\rightarrow} \wedge\, s \overset{a}{\rightarrow} s' \\ s\ d\ s'\ a\ f(s'') & \mathit{if}\, s \overset{C}{\not\rightarrow} \wedge (\forall a' :\ a \text{ is discrete } \ \rightarrow\ s \overset{a'}{\not\rightarrow}) \wedge\ s \overset{d}{\rightarrow} s' \overset{a}{\rightarrow} s'' \end{cases}$$

Note that $f(s)$ may pick a and d in an arbitrary way when s does not enable C. For the proof this has no consequence.

Let $\alpha = \alpha' a s$ and let β be the execution resulting from $\alpha' a f(s)$. By Assumption 1, β can be constructed.

Suppose β is not admissible. Then there is an infinite suffix in β in which each occurrence of a time step implies that the time passing is smaller than C. Without loss of generality we assume that the suffix starts after the prefix α', that is, in the part which is constructed by f. By definition of f, no state in this suffix enables C, so there are no two adjacent time steps in this suffix. We see that there are infinitely many occurrences of discrete actions in the suffix. Combining this with the fact that each state in the suffix does not enable C we have a contradiction with Assumption 2, our decreasing measure function. We conclude that β is admissible. ☒

Implementation relation Let A and B be timed I/O automata. A *implements* B if $\mathit{t\text{-}traces}(A) \subseteq \mathit{t\text{-}traces}(B)$.

Refinements Let A and B be timed I/O automata. A *timed refinement* from A to B is a function $r : \mathit{states}(A) \rightarrow \mathit{states}(B)$ that satisfies:

1. If $s \in \mathit{start}(A)$ then $r(s) \in \mathit{start}(B)$.

2. If $s' \overset{a}{\rightarrow}_A s$ then $r(s') \overset{p}{\rightsquigarrow}_B r(s)$, where $p = \mathit{t\text{-}trace}(s' a s)$.

Let A and B be timed I/O automata with invariants P and Q, respectively. A *weak timed refinement* from A to B, with respect to P and Q, is a function $r : states(A) \rightarrow states(B)$ that satisfies:

1. If $s \in start(A)$ then $r(s) \in start(B)$.

2. If $s' \stackrel{a}{\rightarrow}_A s$, $s' \in P$, and $r(s') \in Q$, then $r(s') \stackrel{p}{\rightsquigarrow}_B r(s)$, where $p = t\text{-}trace(s'as)$.

Theorem A.6 Let A and B be timed I/O automata. If there exists a (weak) timed refinement from A to B, then $t\text{-}traces(A) \subseteq t\text{-}traces(B)$.

Using abstract and refined invariants in a timed refinement We now present the timed version of Lemma A.2, since the timed version is used in the verification in Chapter 7.

Lemma A.7 Let A, B be timed I/O automata. Let Q be invariant for B and P_2 be inductive relative to P_1 for A. Let $r : states(A) \rightarrow states(B)$ such that

1. $r(s) \in Q$ implies $s \in P_1$,

2. $s \in start(A)$ implies $r(s) \in start(B)$, and

3. $s' \in P_2$, $r(s') \in Q$ and $s' \stackrel{a}{\rightarrow}_A s$ implies $r(s') \stackrel{p}{\rightsquigarrow}_B r(s)$, where $p = t\text{-}trace(s'as)$.

Then

1. P_1, P_2 are invariant for A.

2. r is a weak timed refinement from A to B with respect to P_2 and Q.

Proof Similar to the proof for Lemma A.2. ⊠

A.4 Fair Timed I/O automata

In Problem 5 in the RPC-Memory specification problem in Chapter 3, a timed implementation is required for an untimed specification. In our model, this means that we have to compare the admissible behaviour of a timed specification with the fair behaviour of an untimed specification. This may be solved by adding time to the untimed specification. However, the fairness restrictions are lost in this manner, and we may prove the wrong implementation relation. Our final solution is to consider the traces that are both admissible, and fair in the sense that we know from the untimed model. For this purpose, we define the *fair timed I/O automaton*, which is a timed I/O automaton with additional fairness requirements.

Although carrying fairness semantics over from the untimed model to a timed model is very tricky in general, we can get away with the same definition as for the untimed case as long as the discrete actions used in the fairness sets cannot be overruled by the passage of time. This property is known as *persistency* [Yi90] and can be summarised as follows:

> If a discrete action a is enabled in state s, then a is enabled in each state s' that can be reached from s by idling.

All fair timed I/O automata in Chapter 3 meet the persistency requirement.

We now list the basic definitions that enable us to use fairness for timed I/O automata. A *fair timed I/O automaton* A is a triple consisting of

- a timed I/O automaton $timed(A)$, and
- sets $wfair(A)$ and $sfair(A)$ of subsets of $local(timed(A))$, called the *weak fairness sets* and *strong fairness sets*, respectively.

Enabling of sets Let U be a set of locally controlled actions of a fair timed I/O automaton A. Then U is *enabled* in a state s iff an action from U is enabled in s. Set U is *input resistant* if and only if, for each pair of reachable states s, s' and for each input action a, s enables U and $s \xrightarrow{a} s'$ implies s' enables U. So once U is enabled, it can only be disabled by the occurrence of a locally controlled action.

Fair executions An execution α of a fair timed I/O automaton A is *weakly fair* if the following conditions hold for each $W \in wfair(A)$:

1. If α is finite then W is not enabled in the last state of α.
2. If α is infinite then either α contains infinitely many occurrences of actions from W, or α contains infinitely many occurrences of states in which W is not enabled.

Execution α is *strongly fair* if the following conditions hold for each $S \in sfair(A)$:

1. If α is finite then S is not enabled in the last state of α.
2. If α is infinite then either α contains infinitely many occurrences of actions from S, or α contains only finitely many occurrences of states in which S is enabled.

Execution α is *fair* if it is both weakly and strongly fair. In a fair execution each weak fairness set gets turns if enabled continuously, and each strong fairness set gets turns if enabled infinitely many times. We write $fairexecs(A)$ for the set of fair executions of A.

Fair timed traces We write $fair\text{-}t\text{-}traces(A)$ for the set of timed traces derived from the fair executions of fair timed I/O automaton A.

Implementation relation Let A and B be fair timed I/O automata. *A implements B* if $(t\text{-}traces^{\infty}(A) \cap fair\text{-}t\text{-}traces(A)) \subseteq (t\text{-}traces^{\infty}(B) \cap fair\text{-}t\text{-}traces(B))$.

Samenvatting

Dit proefschrift gaat over de analyse van industriële protocollen met behulp van formele methoden. De resultaten van een project, uitgevoerd op het Centrum voor Wiskunde en Informatica (CWI) in Amsterdam voor het Philips Natuurkundig Laboratorium (Eindhoven), worden gepresenteerd en beoordeeld.

Industrieel kader

Tegenwoordig bevatten huishoudelijke apparaten meer en meer elektronica, waardoor de functionaliteit van de apparaten groter en meer divers wordt. Een voorbeeld is een koffiezetapparaat met een ingebouwde klok waarin men kan vastleggen dat het apparaat op een bepaalde tijd uit zichzelf koffie moet gaan zetten. De volgende stap in de ontwikkeling van de systemen die dergelijke apparaten besturen, is om de elektronica in verschillende apparaten te laten communiceren. Het is bijvoorbeeld al mogelijk om een multifunctionele afstandsbediening te kopen die audio, video en andere apparatuur van verschillende merken bestuurt. In de zeer nabije toekomst zal het mogelijk worden om apparaten binnen een huishouden te koppelen tot een intelligent netwerk dat uiteenlopende diensten aanbiedt. Momenteel wordt gewerkt aan diverse technologieën om dit mogelijk te maken. Twee voorbeelden zijn HAVi [GHM+98] en Jini [Sun99]. HAVi, een initiatief van acht bedrijven, is gericht op de interoperabiliteit tussen audio- en videoapparatuur. Jini, een initiatief van een computerfabrikant, is gericht op het koppelen van willekeurige elektronische apparaten.

Het is de bedoeling dat binnen enkele jaren netwerken kunnen worden gerealiseerd die de volgende diensten leveren:

- Gebruikersprofielen. De voorkeuren van iedere persoon in het huishouden kunnen op één plaats worden bijgehouden, en door elk apparaat in het netwerk worden opgevraagd. Vervolgens gedraagt elk apparaat zich per gebruiker precies zoals die dat graag heeft.

- Dynamische netwerkstruktuur. Het is mogelijk om een nieuw apparaat aan te sluiten, waarna, zonder verdere interactie met de gebruiker, het apparaat zelf kennismaakt met het netwerk en alle benodigde informatie kan vinden. Het weghalen van een apparaat wordt gesignaleerd en automatisch doorgegeven aan alle applicaties die hiervan op de hoogte dienen te zijn.

- Dynamische dienstverlening. Wanneer een gebruiker een bepaalde dienst nodig heeft, kan deze dat aan een willekeurig aanspreekpunt in het netwerk doorgeven. Het aanspreekpunt zorgt vervolgens dat de juiste partijen aan het werk worden gezet.

- Ontwikkelingsbestendigheid. Nieuwe soorten van diensten of apparatuur die nu nog niet bestaan kunnen zonder problemen in het netwerk worden ingevoegd, omdat de apparaten op een van tevoren afgesproken manier hierover kunnen "leren".

Om dit soort intelligente netwerken mogelijk te maken, moeten veel technische vraagstukken worden opgelost. De vraag is of de ontwikkelde technologieën wel in alle situaties zullen werken. Hierbij kunnen formele methoden van nut zijn.

Formele methoden

Formele methoden zijn de wiskundige gereedschappen bij het ontwerpen van computersystemen. We gebruiken de term *artefact* om een systeem of een ontwerp (zowel apparatuur als programmatuur) aan te duiden. Formele methoden kunnen worden gebruikt:

- om een bepaalde relatie tussen een artefact en een verzameling vereisten vast te stellen,

- om uit een verzameling vereisten een artefact te onwikkelen,

- om uit een artefact de vereisten te reconstrueren.

In dit proefschrift ligt de nadruk op de eerstgenoemde mogelijkheid.

De mogelijkheden van formele methoden, zoals toegepast in dit onderzoek, kunnen worden opgesomd als formalisatie, validatie, verificatie en conformance-testen.

Formalisatie is het maken van een formele beschrijving uitgaande van een informele beschrijving. Een formele beschrijving van een artefact noemen we een *model*, een formele beschrijving van vereisten noemen we een *specificatie*. Enerzijds bevatten de informele beschrijvingen vaak onduidelijkheden, dubbelzinnigheden en erg veel details, anderzijds moet de formalisatie een precieze, eenduidige en niet te ingewikkelde of te grote beschrijving opleveren. Daarom wordt bij onduidelijkheden een aanname gemaakt, bij dubbelzinnigheden voor een interpretatie gekozen, en van bepaalde details geabstraheerd.

Validatie is het vaststellen of een formalisatie klopt. Veelal komt validatie neer op het handmatig vergelijken van de informele en formele beschrijving, of het raadplegen van deskundigen of de makers van de informele beschrijving. Als er tool-ondersteuning is voor de taal waarin de formele beschrijving gesteld is, kunnen controles op de syntactische of type-correctheid en simulatietechnieken helpen bij de validatie.

Verificatie is het vaststellen van een relatie tussen twee formele beschrijvingen met een wiskundig bewijs. In dit proefschrift gaat het om de relatie tussen de specificatie en het model. De relatie zegt iets over het gedrag dat het model vertoont, en het gedrag dat de specificatie toelaat. Er zijn grofweg twee verificatie-methoden te onderscheiden, namelijk *theorem proving* en *model checking*. Bij theorem proving wordt een bewijs geconstrueerd met bewijstechnieken zoals inductie of bewijs uit het ongerijmde. Bij model checking wordt voor ieder gedrag dat het model vertoont nagegaan of het voldoet aan wat de specificatie voorschrijft. Voor beide methoden zijn tools ontwikkeld. Theorem proving tools zijn onder te verdelen in tools die een gegeven bewijs controleren (checkers), tools die de gebruiker op een interactieve manier

helpen bij het construeren van een bewijs (assistants), en tools die autonoom proberen een bewijs te vinden (provers). Theorem proving met behulp van een assistant vereist meer tijd en inspanning van de gebruiker dan van de computer waarop het tool draait. Theorem proving met behulp van een prover en model checking met behulp van een tool, vereisen meer tijd en inspanning (geheugenruimte, opslagruimte) van de computer dan van de gebruiker.

Wanneer een relatie tussen de specificatie en het model is bewezen, moet nog worden vastgesteld of deze relatie ook op het informele niveau geldt, tussen de vereisten en het artefact. De eerder gemaakte validatie wordt vaak gebruikt als rechtvaardiging hiervoor.

Conformance-testen is het vaststellen van een relatie tussen vereisten en een artefact door het uitvoeren van experimenten op het artefact. Wanneer we formele methoden gebruiken voor conformance-testen, wordt de relatie vastgesteld tussen het artefact en de formele representatie van de vereisten, namelijk de specificatie. De formele testmethoden worden gebruikt voor het *genereren* en *uitvoeren* van tests en het *evalueren* van de uitkomsten. De test worden gegenereerd uit de specificatie. De evaluatie geeft aan of het artefact juist op de tests heeft gereageerd of niet. De meeste testmethoden zijn gebaseerd op de *test hypothese*, dit is de aanname dat het artefact waarop de experimenten worden uitgevoerd, gemodelleerd kan worden in de formele taal van de test methode.

Meestal is het niet mogelijk of wenselijk om een artefact volledig te testen, omdat de verzameling tests te groot of zelfs oneindig is. Dan wordt getest onder aannames die het optreden van fouten betreffen, en is de uitkomst altijd relatief ten opzichte van de aannames. Testen is dus veelal gericht op het vinden van fouten en het vergroten van het vertrouwen in de correctheid van het artefact, en kan de afwezigheid van fouten niet garanderen.

Protocollen

In dit proefschrift ligt de nadruk op de analyse van *protocollen*. Een protocol is een afgesproken methode om informatie tussen twee of meer entiteiten uit te wisselen, waarbij van een onderliggende dienst of medium gebruik wordt gemaakt. De drie basis-ingrediënten voor een protocol zijn: (1) de boodschappen en hun betekenis, (2) de volgorde waarin de boodschappen uitgewisseld worden, en (3) de manier waarop een onderliggende dienst of medium wordt gebruikt. Veel protocollen beginnen bijvoorbeeld met een kennismakingsfase, gevolgd door een fase waarin belangrijke informatie wordt uitgewisseld, gevolgd door een beëindigingsfase, waarbij iedere fase zijn eigen boodschappen gebruikt.

Een *open* protocol is gepubliceerd en daardoor beschikbaar voor publiek gebruik. In de meeste gevallen worden deze protocollen ontwikkeld door een gemeenschappelijke inspanning van een groep firma's en/of individuen. Een *standaard*-protocol is een open protocol dat vaak wereldwijd is geaccepteerd, en meestal is gepubliceerd door een standaardiserings-organisatie.

Het onderzoek

Er zijn zes deelonderzoeken gedaan, waarvan er vijf succesvol zijn afgerond. Hoofdstukken 2 t/m 7 zijn gebaseerd op de artikelen die uit de deelonderzoeken zijn voortgekomen.

Hoofdstuk 2 presenteert een uitbreiding van de theorie van I/O automaten, welke nodig was voor de verificatie die in Hoofdstuk 3 wordt gepresenteerd. De uitbreiding is een generalisatie

voor het gebruik van zwakke en sterke fairness in I/O automaten modellen, waarbij liveness kan worden afgeleid met twee condities die simpel te controleren zijn.

Hoofdstuk 3 presenteert de formalisatie en verificatie van een protocol uit de literatuur, bedoeld om ervaring op te doen. De formalisatie is gedaan met I/O automaten, en de verificatie met theorem proving zonder tools. De verificatie laat zien dat het protocol correct werkt.

Hoofdstuk 4 beschrijft de ervaringen opgedaan bij de constructie van een test-omgeving voor hardware-ontwerpen. Bij het testen wordt uitgegaan van een abstracte specificatie voor de test-afleiding, en een hardware-ontwerp waar de tests op worden uitgevoerd. Het centrale probleem is enerzijds de vertaling van de abstracte tests naar het niveau van het hardware-ontwerp, en anderzijds een generieke opzet voor het uitvoeren van tests op het hardware-ontwerp. De geconstrueerde test-omgeving bestaat uit een verzameling tools en is uitgeprobeerd op twee protocollen: een industrieel protocol en een klein protocol uit de literatuur.

Hoofdstuk 5 presenteert een uitbreiding van conformance-test-theorie, gebaseerd op symmetrie-eigenschappen in de specificatie en het te testen artefact. Een algoritme wordt gepresenteerd dat uit de toestandsruimte van een specificatie een zogenaamde kernel selecteert, dusdanig dat voor ieder gedrag van de specificatie een symmetrische variant van dit gedrag in de kernel bestaat. Testafleiding kan dan gebeuren op grond van de kernel in plaats van de hele specificatie. Wanneer de kernel aanzienlijk kleiner is dan de specificatie, zal men met veel minder tests toch een oordeel over het hele artefact kunnen geven. De methode voor testafleiding en een correctheidsbewijs worden in dit hoofdstuk gegeven.

Hoofdstuk 6 beschrijft de formalisatie en verificatie van een industrieel protocol uit de HAVi-specificatie, een architectuur die momenteel gestandaardiseerd wordt door acht verschillende electronicabedrijven. Het protocol is in twee talen geformaliseerd, en de verificatie is gedaan met behulp van twee model checking tools. Een aantal eigenschappen is geformaliseerd in temporele logica, de formele representaties zijn als specificatie gebruikt bij het model checken. Het is gebleken dat sommige eigenschappen niet gelden voor de modellen, en dat dit kan worden terugvertaald naar fouten in het protocol.

Hoofdstuk 7 presenteert de formalisatie en verificatie van een industrieel protocol uit de IEEE 1394-standaard (FireWire). Het protocol is geformaliseerd met I/O automaten, en een verificatie is gedaan door theorem proving zonder tools. Het model van het protocol is gedetailleerder dan andere modellen van dit protocol, en de verificatie bouwt verder op een serie van verificaties die steeds meer details uit de IEEE-standaard in acht nemen. Hierbij worden steeds resultaten van het meer abstract niveau hergebruikt op het niveau met meer details.

Conclusies

In Hoofdstuk 8 wordt geconcludeerd dat formele methoden zeker effectief kunnen worden toegepast, op grond van drie observaties, namelijk (1) de kracht van formele methoden zal ongetwijfeld significant toenemen in de komende jaren, (2) de toepassingen van formele methoden in dit project hebben bruikbare resultaten opgeleverd, en (3) formele methoden kunnen effectiever worden toegepast dan in dit project al is gebeurd. Er is sprake van beperkende factoren waar veel aan te verbeteren is, vanuit zowel de academische als de industriële wereld, en als de verbetering van die factoren serieus wordt aangepakt, zal de effectiviteit van het toepassen van formele methoden bij het ontwikkelen van industriële protocollen alleen maar groter worden.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1996-1

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms.* Faculty of Mathematics and Computer Science, KUN. 1996-2

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation.* Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. *Parallel Local Search.* Faculty of Mathematics and Computing Science, TUE. 1996-4

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory.* Faculty of Mathematics and Computer Science, KUN. 1996-5

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems.* Faculty of Mathematics and Computing Science, TUE. 1996-6

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance.* Faculty of Mathematics and Computer Science, UvA. 1996-7

H. Doornbos. *Reductivity Arguments and Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. *Functorial Operational Semantics and its Denotational Dual.* Faculty of Mathematics and Computer Science, VUA. 1996-9

A.M.G. Peeters. *Single-Rail Handshake Circuits.* Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism.* Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference.* Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics.* Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth.* Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context.* Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types.* Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics.* Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution.* Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing.* Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing.* Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering.* Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation.* Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller.* Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model.* Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing.* Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes.* Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search.* Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection.* Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases.* Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics.* Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems.* Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods.* Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems.* Faculty of Computer Science, UT. 1999-10