VRIJE UNIVERSITEIT

Formal Verification of Distributed Systems

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op dinsdag 26 oktober 2004 om 10.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Jun Pang

geboren te Jiangsu, China

promotor: prof.dr. W.J. Fokkink

Formal Verification of Distributed Systems

Jun Pang

August, 2004

Printed by Ponsen & Looijen B.V.
ISBN 90-6464-865-4

This research has been supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research (NWO), the Dutch Ministry of Economic Affairs and the Technology Foundation (STW), within the scope of the project CES.5008 “Improving the Quality of Embedded Systems using Formal Design and Systematic Testing”. It has been carried out under the auspices of the Institute for Programming Research and Algorithmics (IPA), at the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam.

Contents

Preface

This is the ending point of my journey of being an *onderzoeker in opleiding* at CWI. I am indebted to everybody who made it possible for me to write this thesis!

First of all, I would like to thank my supervisor and promotor Wan Fokkink, who directed my research in the last four years. Wan gave me many inspiring and valuable ideas. The door of his office at CWI was always open to me. No matter how busy he was, he would discuss any problem I encountered, and read all my drafts very carefully. I really owe much to him!

I am very grateful to all my co-authors for their pleasant cooperation. Apart from Wan, they are Bahareh Badban, Jan Friso Groote, Rutger Hofman, Jaap-Henk Hoepman, Bart Karstens, Jaco van de Pol, Miguel Valero Espada, Ronald Veldema and Arno Wouters.

I also thank my CWI roommate Simona Orzan for many pleasant conversations. Many thanks to all my former colleagues at CWI, in particular Bahareh Badban, Stefan Blom, Wan Fokkink, Izak van Langevelde, Bert Lisser, Natalia Ioustinova, Vincent van Oostrom, Jaco van de Pol, Yaroslav Usenko, Miguel Valero Espada, Anton Wijs and Yinwei Zhan. I am grateful to Wan Fokkink, Judi Romijn and Anton Wijs, who prepared the Dutch summary for me. Jos van der Werf designed the cover of this thesis, which I appreciated a lot. I also wish to extend my gratitude to the PAM speakers and participants during the last four years for their nice talks, discussions and comments.

I am grateful to the members of the reading committee, Maarten Boasson, Hubert Garavel, Jan Friso Groote, Jan Willem Klop and Jaco van de Pol for reviewing the manuscript and for their constructive criticism. I want to thank Gerard Tel for his insightful comments on the leader election algorithms in Chaper 8.

I thank the members of the user committee of my research project, Maarten Boasson, Frank Karelse, Ernst Kesseler, Anton Klip, Wim Pelt, Jan Tretmans and Berto Wanschers for their comments on my work and providing many ideas on how to organize this thesis.

I am grateful to Catuscia Palamidessi for helping me to find a new job at INRIA Futurs, which means a lot to me.

Many thanks to all my friends who always cheer me up and share many sides of life with me.

My family in China, especially my mother Zhenshan Ge, deserve my endless

thanks for their unconditional love and support.

I reserve my greatest thanks to Qin for her love, encouragement, support, patience, and many other things.

Jun Pang
Paris, August 2004

Chapter 1

Introduction

The last several decades have seen a rapid growth of information technology. Computer based systems, e.g., traffic control system for airlines, transaction systems for international banks, are used world-wide in our daily life. Clearly, the correctness of such systems is of crucial importance. Failures of those systems can be potentially disastrous and cause the loss of human life and a huge amount of money. However, the design and implementation of computer based systems, including both hardware and software systems, are error-prone and becoming extremely complex.

Mathematics can provide solid foundations for methods to describe and analyze systems. Formal methods are of this kind. Their mathematical underpinning allows formal methods to specify systems more precisely, more consistently and in a non-ambiguous way. Moreover, formal analysis techniques can be used to verify whether a system has desired properties. The research in this thesis is motivated by the conviction that the proper use of formal methods will lead to more reliable, dependable, and secure systems in the future.

This thesis concerns the application of formal verification to distributed systems, including industrial products, communication protocols, and distributed algorithms. The aim of this chapter is to give a broad view of the main topics studied (without being exhaustive) and results obtained in the embedded systems research program (PROGRESS) of the Dutch organization for Scientific Research (NWO), the Dutch Ministry of Economic Affairs and the Technology Foundation (STW) supported project CES.5008 – Improving the Quality of Embedded Systems by Formal Design and Systematic Testing.

1.1 The Title

First things first. According to the textbook [36] of Coulouris, Dollimore and Kindberg, *distributed systems* are defined as systems consisting of a collection of autonomous computers linked by a computer network and equipped with distributed system software. Computer networks provide the necessary means for communication between the components of a distributed systems. Distributed

systems have to combine desirable characteristics, such as *resource sharing*, *openness*, *concurrency*, *scalability*, *fault tolerance*, and *transparency*. This thesis focuses on the assurance of the correctness of distributed systems, with an emphasis on concurrency and fault tolerance.

Formal methods refer to a collection of notations and techniques for describing and analyzing systems. They can be used to improve the quality of (distributed) systems. A formal method generally consists of a formalism to model a system, a specification language to express the desired properties of the system, a formal semantics to interpret both the system and the properties, and verification techniques to check whether the properties are satisfied by the system. This thesis concentrates on the process of applying such verification techniques, which is called *formal verification*. The URL `http://vl.fmnet.info/` collects information on formal methods, available around the world on the World Wide Web (WWW).

There is a wide range of verification techniques to establish the correctness of a system, i.e. asserting that a system has desired properties and only those. *Process algebra*, such as ACP, CCS, CSP, and LOTOS, is defined as an algebraic approach to model the behavior of distributed systems. Their axiomatic theories provide an elegant way for the study of elementary behavioral properties of such systems. Both a system and its desired external behavior can be expressed in a process algebraic specification. Correctness of the system can be verified by proving that these two specifications are equivalent in terms of a chosen *bisimulation* relation, which respects the branching structure of systems and is a standard equivalence relation for a setting with concurrency. Verification techniques based on the axiomatic theories, such as methods for proving bisimulation, have been developed for process algebras.

A *manual proof* is only feasible for formal models of small systems, as the complexity of a system can make manual mathematical proofs infeasible. Computer support is necessary for the verification of most real-life systems. An alternative to manual proof is *automatic* or *mechanical verification*. *Proof checking* assumes the presence of a *proof checker* implemented on a computer. Both the manual proof and a set of proof rules are fed to the proof checker, which then automatically decides whether the proof contains flaws. A *theorem prover* provides automated support to aid the creation of proofs. Proofs are generated along strict lines, but this process requires human-computer interactions. The aim of proof checkers and theorem provers is obviously to increase the reliability of the correctness of the proofs. The problem with this approach is that it is highly time consuming and can be rather non-trivial.

Unlike theorem proving, model checking is usually restricted to finite-state systems. It first builds a finite state space of a formal model of a system, and then verifies a property, written in some temporal logic, through an explicit state space search. Due to the finiteness of the state space, the search always terminates. Model checking is largely automatic. It can produce an answer in a few minutes or even seconds for many models. A counter-example can be generated when the checked property fails to hold. This information can be used for debugging the model. Techniques such as partial order reduction, symmetry

reduction, abstract interpretation, have been developed to deal with the *state explosion* problem and enhance the scalability of model checking. Recently, attention in this area has been devoted to model checking infinite-state systems. Other challenges are probabilistic systems, timed systems, and so on.

The combination of manual proof, theorem proving and model checking is widely used nowadays in verification tasks. Note that both theorem proving and model checking require a formal model of the verified system. The model is achieved by abstracting away irrelevant information or ignoring some implementation details. This means that we verify distributed systems at a rather abstract level. Systems which have passed the verification can thus still contain errors in their real implementation. Thus, other techniques to check the correctness of systems, e.g. testing, remain necessary.

The strengths of formal methods are that they 1) force to reason at the conceptually clear level of a formal model, 2) can detect errors in the design, 3) are able to prove correctness of a system, and 4) are supported by automated techniques.

1.2 The Project

The research in this thesis is carried out within the PROGRESS supported project CES.5008 – Improving the Quality of Embedded Systems using Formal Design and Systematic Testing. It was co-proposed by the Embedded Systems Group at the Centrum voor Wiskunde en Informatica (CWI) and the Dutch company Weidmüller, later Add-Controls. Add-Controls builds embedded controllers for a large range of applications, such as a distributed system for lifting trucks and a steam unit used for steam baths and saunas. Add-Controls of course wants to deliver fault-free products, but experienced that this is almost unattainable with software. It happens too often that finalized software still contains bugs. Therefore, Add-Controls set up a project to automatically analyze the software in a rigorous manner, and to make this analysis reproducible.

The proposal of the project is intended to go beyond the ambitions of the company by making formal verification techniques applicable in the design process of embedded systems. The general goal of the project is:

> "to establish whether it is possible to achieve reliable quality of software for medium size embedded systems, and to better utilize formal methods in industry."

Formal methods have already proved their usefulness for several years, although mainly from an academic perspective. The project also proposed a major question:

> "whether the current technology developed in the past by the formal methods research community can indeed become an effective practical tool within a development environment."

There have been numerous case studies which suggest that this is the case. However, most of these case studies were quite remote from the actual product design process and generally only dealt with fractions of a total system.

I was recruited as a PhD student to work on this project for the duration of four years. According to the proposal, the first year was planned on describing and analyzing an existing system to get acquainted with formal techniques and the software development method used in Add-Controls. The second and the third year were used to completely and formally design a number of embedded systems, before implementation took place. In parallel with the design of these embedded systems, I was supposed to develop tools to facilitate the connection between the formal descriptions and the development environment used at Add-Controls. The fourth year was devoted to writing a thesis.

1.3 The Results

In this section, I give the list of case studies and the results that were achieved within the project.

A mechanical framework for protocol verification

Together with Wan Fokkink and Jaco van de Pol, I defined a cones and foci proof method [54], which rephrases the question whether two system specifications are branching bisimilar in terms of proof obligations on relations between data objects. Compared to the original cones and foci method from Groote and Springintveld [79], this method is more generally applicable, and does not require a preprocessing step to eliminate internal loops. We proved soundness of our approach. Furthermore, we designed a set of rules to support the reachability analysis of so-called focus points. We formalized the method and proved its correctness using the theorem prover PVS, and thus established a framework for mechanical protocol verification.

More recently, together with Wan Fokkink, I extended this cones and foci method for timed systems verification [55]. This work is not included in the current thesis.

A sliding window protocol

Together with Bahareh Badban, Wan Fokkink, Jan Friso Groote, and Jaco van de Pol, I applied the cones and foci method and the mechanical framework in PVS to the verification of one of the most complex sliding window protocols presented in Tanenbaum's *Computer Networks* textbook [165]. We proved the correctness of this sliding window protocol with an arbitrary finite window size n and sequence numbers modulo $2n$. We showed that the external behavior of this protocol is equivalent to a FIFO queue of capacity $2n$. This proof is entirely based on the axiomatic theory underlying μCRL and the axioms characterizing the data types. It implies both safety and liveness of the protocol.

Sliding window protocols have attracted much attention from the process algebra community, which has led to significant developments in the realm of process algebraic proof techniques for protocol verification. We therefore consider this work as a true milestone in process algebraic verification.

A distributed system for lifting trucks

A main product of Add-Controls is a distributed system for lifting heavy vehicles (e.g. trucks, railway carriages and buses). The system consists of a number of lifts; each lift supports one wheel of the truck that is being lifted and has its own micro controller. The controls of the different lifts are connected by means of a network. A special purpose protocol has been developed to let the lifts operate synchronously.

When testing the implementation the developers found problems. They solved these problems by trial and error, partly because the causes of problems were unclear. Together with Jan Friso Groote and Arno Wouters, I applied the process algebraic language μCRL in combination with the model checker CADP to the verification of this lift system [73]. The analysis in μCRL revealed the reasons for the problems. Another new problem was found in the model, which was indeed present in the implementation of the system. Solutions were proposed and included in the μCRL specification, and we showed by model checking that the problems were solved indeed.

The developers tried to solve the problems independently. They made a redesign of the lift system based on their own solutions, which Bart Karstens, Wan Fokkink and I checked using the real-time model checker UPPAAL [135]. We showed that the solutions of the developers do not solve the problems completely, while a refined version of our solutions contained in the μCRL specification does. Currently, the lift system is under revision, and our solutions to the problems are being implemented.

Together with Jaco van de Pol and Miguel Valero Espada, I developed a general framework for abstracting uniform parallel processes with data, and applied it to the verification of a simplified lift system [136]. This work is not included in the current thesis.

A cache coherence protocol for a Java DSM implementation

Jackal (developed at the Vrije Universiteit Amsterdam) is a fine-grained, distributed shared memory implementation of Java. Its goal is to run unmodified concurrent Java programs efficiently on a cluster of workstations. It is based upon a self-invalidation based, multiple-writer cache coherence protocol. Together with Wan Fokkink, Rutger Hofman, and Ronald Veldema, I developed a formal specification of this protocol in μCRL [134]. Three requirements were formulated for the protocol: deadlock freedom, relaxed cache coherency, and liveness of writing and flushing regions. The verification allowed the discovery of two errors in the design of the cache coherence protocol. Also, a large number of inconsistencies and misunderstandings were found, mostly caused by the

evolution of the implementation simultaneously with the formal analysis process. This case study benefited a lot from the μCRL distributed state space generation tool, and also pushed forward its development.

Distributed algorithms: self-stabilization and leader election

Together with Wan Fokkink and Jaap-Henk Hoepman, I showed that, contrary to common belief, Dijkstra's K-state mutual exclusion algorithm on a ring also stabilizes when the number K of states per process is one less than the number $N+1$ of processes in the ring [52]. We formalized the algorithm and verified the proof in the theorem prover PVS, based on Qadeer and Shankar's work [144].

Furthermore, together with Wan Fokkink, I designed two probabilistic leader election algorithms for anonymous unidirectional rings with FIFO channels [56], based on an algorithm from Itai and Rodeh. In contrast to the Itai-Rodeh algorithm, our algorithms are finite-state, so they can be analyzed using explicit state space exploration. We used the probabilistic model checker PRISM to verify that eventually a unique leader is elected with probability one. Furthermore, we gave a manual correctness proof for each algorithm, for arbitrary ring size.

Needham-Schroeder public key authentication protocol

I described the Needham-Schroeder public key authentication protocol in μCRL as a configuration containing an initiator, a responder, and an intruder [133]. It showed that the capabilities of the language (especially the data types) are well-adapted for describing this kind of protocols. This work is not included in the current thesis.

Two abandoned case studies

A small control system of Add-Controls, being a converter which measures the displacement of a hydraulic cylinder, was also studied. Some customer reported an error of the system. We made a start to analyze the system using the TorX tool. Due to the fact that only one of the 150 systems that had been sold so far exhibited an error, and the error could not even be reproduced with a simulator, the developers of TorX pointed out that it was very unlikely that this formal analysis would produce a useful result. It was therefore decided to abandon this case study.

Another challenging embedded system was proposed by Add-Controls. It concerns an embedded controller for a lift system for a staircase, including a SmartCard with minimal information on the topology of the staircase for which it is used. Adapting the speed and keeping the chair horizontal is the responsibility of the SmartCard, using information on the actual speed and position of the lift. Interestingly, the topology of the staircase lying ahead of the lift is taken into account when keeping the chair horizontal. Thus it is a truly hybrid system. But later on, Add-Controls lost the bidding to develop the system, and no more detailed design information could be given. We stopped this case study after building an experimental model using hybrid automata.

1.4 The Structure

The thesis is organized as follows. This chapter contains a short introduction to formal verification, the project and its scope, and the achieved results. Chapter 2 presents some preliminaries for this thesis.

Part I of this thesis is concerned with theorem proving. Chapter 3 presents the generalized cones and foci method for protocol verification. It is an extension of [54] with a formalization of the cones and foci method in the theorem prover PVS (mainly done by Jaco van de Pol). The verification of the sliding window protocol is presented in Chapter 4. It extends [51] by allowing the mediums of the sliding window protocol to have unbounded capacity. Chapter 5 reports the formal verification of a distributed algorithm for self-stabilization. It was previously published as a CWI technical report [52].

Part II presents applications of model checking. Chapter 6 presents the analysis of the distributed lift system of Add-Controls. It is a revised version of [73] and [135]. The cache coherence protocol for concurrent Java programs on a distributed shared memory implementation is analyzed in Chapter 7. It is a revised version of [134]. Chapter 8 presents two probabilistic leader election algorithms for anonymous rings and their verification results. It was previously published as a CWI technical report [56]. Chapter 9 contains the conclusions, from the perspective of the entire project.

Chapter 2

Preliminaries

2.1 μCRL

Process algebra, such as ACP [16, 9, 50], CCS [126, 128] and CSP [89, 90], is defined as an algebraic approach to model the behavior of distributed systems. The axiomatic theories of process algebra provide an elegant way for the study of elementary behavioral properties of such systems. However, when it comes to the study of more realistic systems, these languages turn out to lack the ability to handle data adequately. In order to solve this problem, formalisms such as LOTOS [46] and μCRL [75] were developed by enhancing process algebras with data types. They are suitable to describe realistic, interacting systems. μCRL is the main formalism used in this thesis. We briefly give an introduction to this language. The syntax and semantics of μCRL are given in [75].

μCRL is a language for specifying distributed systems and protocols in an algebraic style. This language combines the process algebra ACP with equational *abstract data types* [115]. In a μCRL specification, one part specifies the data types, while a second part specifies the process behavior. Each data type is declared using the keyword **sort**. Elements of a data type are declared by using the keywords **func** and **map**. Using **func** one can declare constructors with as target sort the data type in question; these constructors define the structure of the data type. E.g. by

sort Bool
func T, F: →Bool

one declares that T (true) and F (false) are the only elements of sort *Bool*. We say that T and F are the constructors of sort *Bool*. The keyword **map** is used to declare additional functions for a data type that are not constructors. Their meanings are defined by means of equations, which consist of a variable declaration (starting with the keyword **var**) followed by an equation section (starting with the keyword **rew**). For instance, conjunction ($\wedge$) and negation ($\neg$) on booleans are defined as follows:

map and: Bool×Bool→Bool

```
        not: Bool→Bool
var     b: Bool
rew     and(T,b)=b
        and(F,b)=F
        not(T)=F
        not(F)=T
```

Since booleans are used in the conditional construct of process descriptions (see below), the sort *Bool* must be included in every μCRL specification. Besides the declaration of the sort *Bool*, it is also obligatory that T and F are declared in every specification and that $\mathsf{T} \neq \mathsf{F}$. To reflect equality between terms, one needs to specify an equality function $eq : D \times D \rightarrow Bool$, such that $eq(s,t) = \mathsf{T}$ if and only if $s = t$. Actually, such an equality function is only needed for data types that are used as parameters of actions that occur in a communication (see below). For data types in this thesis, the specification of the equality function eq is mostly omitted, for the sake of presentation.

The specification of a process is constructed from actions, recursion variables and process algebraic operators (processes are declared by the keyword **proc**). Actions and recursion variables carry zero or more data parameters (actions are declared by means of the keyword **act**). Intuitively, an action can execute itself, after which it terminates successfully. There are two predefined processes in μCRL: δ represents deadlock, and τ a hidden action. These two processes never carry data parameters. $p{\cdot}q$ denotes sequential composition and $p + q$ non-deterministic choice, where p and q are processes. Summation $\sum_{d:D} p(d)$ provides the possibly infinite choice over a data type D, and the conditional construct $p \lhd b \rhd q$ with b a data term of sort *Bool* behaves as p if $b = \mathsf{T}$ and as q if $b = \mathsf{F}$. Parallel composition $p \parallel q$ interleaves the actions of p and q; moreover, actions from p and q may also synchronize to a communication action, when this is explicitly allowed by a predefined communication function using the keyword **comm**. Two actions can only synchronize if their data parameters are semantically the same, which means that communication can be used to represent data transfer from one system component to another. Encapsulation $\partial_H(p)$, which renames all occurrences in p of actions from the set H into δ, can be used to force actions into communication. Finally, hiding $\tau_I(p)$ renames all occurrences in p of actions from the set I into τ. The initial behavior of the system can be specified with the keyword **init**.

Example 2.1.1 A data buffer with size n can be modeled in μCRL as follows:

$$\begin{aligned}\mathsf{Buffer}(\lambda{:}\mathsf{List}) \quad &= \textstyle\sum_{\mathsf{d:Data}} \mathsf{receive}(\mathsf{d}).\mathsf{Buffer}(\mathsf{append}(\mathsf{d},\lambda)) \lhd \mathsf{length}(\lambda){<}\mathsf{n} \rhd \delta \\ &\quad + \mathsf{send}(\mathsf{top}(\lambda)).\mathsf{Buffer}(\mathsf{tail}(\lambda)) \lhd \mathsf{length}(\lambda){>}0 \rhd \delta\end{aligned}$$

This says whenever the list is not full ($length(\lambda){<}n$), the buffer can receive any datum d (modeled by action $receive(d)$) and append it to the end of the list ($append(d,\lambda)$); the buffer can also take the datum at the top of the list and send it outside (modeled by action $send(top(\lambda))$) if the list is not empty

($length(\lambda)>0$). In this case only the tail of the list ($tail(\lambda)$) remains. Initially, the list contains no data ($\lambda=\langle\rangle$), which can be expressed as follows:

init Buffer($\langle\rangle$)

2.2 Labeled Transition Systems and Behavioral Equivalences

Labeled transition systems (LTSs) [102] can capture the state space of distributed systems. An LTS consists of transitions $s \xrightarrow{a} s'$, denoting that the state s can evolve into the state s' by the execution of action a.

Definition 2.2.1 (Labeled transition system) A *labeled transition system* is a tuple $(S, Lab, \rightarrow, s_0)$, where S is a set of states, Lab a set of transition labels, $\rightarrow \subseteq S \times Lab \times S$ a transition relation, and s_0 the initial state. A transition (s, ℓ, s') is denoted by $s \xrightarrow{\ell} s'$.

To each μCRL specification there belongs an LTS, defined by the structural operational semantics for μCRL in [75], in which the states S consist of process terms and the edges Lab consist of actions from $Act \cup \{\tau\}$ parametrized by data. We define *strong bisimilarity* [12, 127, 137] and *branching bisimilarity* [64] between states in LTSs. Both are an equivalence relation (for branching bisimulation, see [13]).

Definition 2.2.2 (Strong bisimulation) Assume an LTS. A *strong bisimulation relation* $\mathcal{B}$ is a symmetric binary relation on states such that if $s \mathcal{B} t$ and $s \xrightarrow{\ell} s'$, then for some t', $t \xrightarrow{\ell} t'$ with $s' \mathcal{B} t'$.

Two states s and t are *strongly bisimilar*, denoted by $s \leftrightarrow t$, if there is a strong bisimulation relation $\mathcal{B}$ such that $s \mathcal{B} t$.

Definition 2.2.3 (Branching bisimulation) Assume an LTS. A *branching bisimulation relation* $\mathcal{B}$ is a symmetric binary relation on states such that if $s \mathcal{B} t$ and $s \xrightarrow{\ell} s'$, then

- either $\ell = \tau$ and $s' \mathcal{B} t$;
- or there is a sequence of (zero or more) τ-transitions $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_0$ such that $s \mathcal{B} t_0$ and $t_0 \xrightarrow{\ell} t'$ with $s' \mathcal{B} t'$ for some t'.

Two states s and t are *branching bisimilar*, denoted by $s \leftrightarrow_b t$, if there is a branching bisimulation relation $\mathcal{B}$ such that $s \mathcal{B} t$.

We defined bisimilarity of states in the same LTS. States of different LTSs are said to be strong/branching bisimilar, if they are strong/branching bisimilar in the disjoint union of the LTSs, which can be defined straightforwardly.

If the LTS belonging to a μCRL specification consists of finitely many states, then the μCRL tool set [21] can be used to support the generation of this LTS,[1] together with reduction modulo strong and branching bisimulation equivalence. More information on the μCRL tool set can be obtained at `http://www.cwi.nl/~mcrl/`.

2.3 Linear Process Equations

A *linear process equation* (LPE) [20] is a μCRL specification consisting of one recursion variable, actions, summations, sequential compositions and conditional constructs. In particular, an LPE does not contain any parallel operators, encapsulations or hidings. In essence an LPE is a vector of data parameters together with a list of condition, action and effect triples, describing when an action may happen and what is its effect on the vector of data parameters. Each μCRL specification that does not include successful termination can be transformed into an LPE [170].[2]

Definition 2.3.1 (Linear process equation) A *linear process equation* is a μCRL specification of the form

$$X(d{:}D) = \sum_{a \in Act \cup \{\tau\}} \sum_{e:E_a} a(f_a(d,e)) \cdot X(g_a(d,e)) \lhd h_a(d,e) \rhd \delta$$

where $f_a : D \times E_a \to D_i$, $g_a : D \times E_a \to D$, $h_a : D \times E_a \to \mathit{Bool}$ for each $a \in Act \cup \{\tau\}$, and a is an action label with data parameters of type D_i.

The LPE in Definition 2.3.1 has exactly one LTS as its solution (modulo strong bisimulation).[3] In this LTS, the states are data elements $d{:}D$ (where D may be a Cartesian product of n data types, meaning that d is a tuple $(d_1, ..., d_n)$) and the transition labels are actions parametrized with data. The LPE expresses that state d can perform $a(f_a(d,e))$ to end up in state $g_a(d,e)$, under the condition that $h_a(d,e)$ is true. The data types E_a give LPEs a more general form, as not only the data parameter $d{:}D$ but also the data parameter $e{:}E_a$ can influence the parameter of action a, the condition h_a and the resulting state g_a.

Definition 2.3.2 (Invariant) A mapping $\mathcal{I} : D \to \mathit{Bool}$ is an invariant for an LPE, written as in Definition 2.3.1, if for all $a \in Act \cup \{\tau\}$, $d{:}D$ and $e{:}E$,

$$\mathcal{I}(d) \wedge h_a(d,e) \;\Rightarrow\; \mathcal{I}(g_a(d,e)).$$

Intuitively, an invariant approximates the set of reachable states of an LPE. That is, if $\mathcal{I}(d)$, and if one can evolve from state d to state d' in zero or more

[1] Sometimes the finite LTS cannot be generated by the μCRL tool set, as it is too large.

[2] To cover μCRL specifications with successful termination, LPEs should include a summand $\sum_{a \in Act \cup \{\tau\}} \sum_{e:E_a} a(f_a(d,e)) \lhd h_a(d,e) \rhd \delta$.

[3] LPEs exclude "unguarded" recursive specifications such as $X = X$, which have multiple solutions.

transitions, then $\mathcal{I}(d')$. Namely, if $\mathcal{I}$ holds in state d and it is possible to execute $a(f_a(d,e))$ in this state (meaning that $h_a(d,e)$), then it is ensured that $\mathcal{I}$ holds in the resulting state $g_a(d,e)$. Invariants tend to play a crucial role in algebraic verifications of system correctness that involve data.

2.4 Regular Alternation-free μ-calculus

Model checking [35] is an automatic technique to determine which states in an LTS satisfy certain requirements. In order to check whether a certain requirement holds, it should be expressed as a temporal logic formula first.

A variety of so-called *modal logics* [94] have been developed to express properties of LTSs, such as *Hennessy-Milner logic* (HML) [85], *linear temporal logic* (LTL) [139], *computation tree logic* (CTL) [47] and *μ-calculus* [104]. We proceed to present a brief description of the μ-calculus, and then the *regular alternation-free* μ-calculus [122], which is the input language for the model checker Evaluator in the Construction and Analysis of Distributed Processes toolbox (see Section 2.5).

The μ-calculus is based on fixpoint computations [166]. Let D be a finite set with a partial ordering $\leq$ with a least and a greatest element. Given a mapping $\varphi : D \rightarrow D$, an element d of D is a *fixpoint* of φ if $\varphi(d) = d$. Moreover, d is a *least fixpoint* or *greatest fixpoint* if $d \leq e$ or $e \leq d$, respectively, for all fixpoints e of φ. The least and the greatest fixpoint of φ (if they exist) are denoted by $\mu Y.\varphi(Y)$ and $\nu Y.\varphi(Y)$, respectively. The mapping $\varphi : D \rightarrow D$ is called *monotonic* if $d \leq e$ implies $\varphi(d) \leq \varphi(e)$. If φ is monotonic, and D has a least element d_0 and a greatest element e_0 (i.e., $d_0 \leq d$ and $d \leq e_0$ for all $d \in D$), then φ has a least and a greatest fixpoint.

The formulas of μ-calculus, which express properties of states, are defined by the following BNF grammar:

$$\varphi ::= \mathsf{F} \mid \mathsf{T} \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \langle a \rangle \varphi \mid [a]\varphi \mid Y \mid \mu Y.\varphi \mid \nu Y.\varphi$$

where a ranges over $Act \cup \{\tau\}$ and Y ranges over some collection of recursion variables. We restrict to *closed* μ-calculus formulas, meaning that each occurrence of a recursion variable Y is within the scope of a minimal fixpoint μY or a maximal fixpoint νY.

The intuitive meaning of the formula $\langle a \rangle \varphi$ is "it is possible to make a-transition to a state where φ holds." Likewise, $[a]\varphi$ means that "φ holds in all states reachable by making a a-transition." The boolean operators have the usual meaning: a state of an LTS always satisfies T; it never satisfies F; it satisfies $\neg\varphi$ if and only if it does not satisfy φ; it satisfies $\varphi_1 \vee \varphi_2$ if and only if it satisfies φ_1 or it satisfies φ_2; it satisfies $\varphi_1 \wedge \varphi_2$ if and only if it satisfies both φ_1 and φ_2. The formulas $\mu Y.\varphi$ and $\nu Y.\varphi$ represent minimal and maximal fixpoints, respectively. Here, φ represents a mapping from sets of states to sets of states: a set S of states is mapped to those states where φ holds, under the assumption that the recursion variable Y evaluates to T for states in S and to F for states outside S. As partial ordering on sets of states we take set inclusion

(so the least and the greatest element are the empty set and the set of all states, respectively.). The mapping φ is monotonic, so $\mu Y.\varphi$ and $\nu Y.\varphi$ are well-defined.

The *alternation-free* μ-calculus [48] consists of μ-calculus formulas with no alternation between least and greatest fixpoint operators, which makes a good compromise between expressiveness and efficiency of model checking.

The *regular* μ-calculus [122] is an extension of the μ-calculus with action predicates and regular expressions over action sequences. One is allowed to use expressions $\langle\beta\rangle\varphi$ and $[\beta]\varphi$ where β is a so-called *regular expression*, which is defined by the following BNF grammar:

$$\alpha ::= \mathsf{T} \mid a \mid \neg\alpha \mid \alpha_1 \wedge \alpha_2$$

$$\beta ::= \alpha \mid \beta_1 \cdot \beta_2 \mid \beta_1 | \beta_2 \mid \beta^*$$

Action formulas α represent a set of actions: T denotes the set of all actions, a the set $\{a\}$, $\neg\alpha$ the complement of α, and $\alpha_1 \wedge \alpha_2$ the intersection of α_1 and α_2. Regular expressions β represent a set of traces: $\beta_1 \cdot \beta_2$ denotes the traces that can be obtained by concatenating a trace from β_1 and a trace from β_2, $\beta_1 | \beta_2$ the union of β_1 and β_2, and β^* the traces that can be obtained by concatenating finitely many traces from β.

$\langle\beta\rangle\,\phi$ means that ϕ holds after some trace from β, and $[\beta]\,\phi$ means that ϕ holds after all traces from β.

The regular alternation-free μ-calculus allows a simple, compact specification of safety and liveness properties [108], where safety properties require that "nothing bad ever happens" and liveness properties require that "something good will eventually happen".

Example 2.4.1 A safety property describing the absence of some *error* action is defined as follows:

$$[\mathsf{T}^* \cdot \mathsf{error}]\ \mathsf{F}$$

Example 2.4.2 A safety property detecting the absence of τ-cycles is defined as follows:

$$[\mathsf{T}^*]\ \mu Y.[\tau]\ Y$$

Example 2.4.3 A liveness property stating that there exists a path leading to some *move* action after performing zero or more transitions is defined as follows:

$$\langle \mathsf{T}^* \cdot \mathsf{move} \rangle\ \mathsf{T}$$

Fairness properties are similar to liveness properties, except that they express reachability of actions by considering only *fair* execution sequences. The notion of fairness encoded in the regular alternation-free μ-calculus is the "fair reachability of predicates" [145]: a sequence is fair if and only if it does not infinitely often enables the reachability of a certain state without infinitely often reaching it.

Example 2.4.4 A fairness property expressing that after sending a message (action *send*) all fair execution sequences will lead to the reception of the message (action *receive*) is defined as follows:

$$[\ \mathsf{T}^{*}\cdot\mathsf{send}\cdot(\neg\mathsf{receive})^{*}\ ]\ \langle(\neg\mathsf{receive})^{*}\cdot\mathsf{receive}\rangle\ \mathsf{T}$$

2.5 Construction and Analysis of Distributed Processes Toolbox

The μCRL tool set, in combination with the Construction and Analysis of Distributed Processes toolbox (CADP) [49, 63], formerly known as Cæsar Aldébaran Development Package, which acts as a back-end for the μCRL tool set, features visualization, simulation, state space generation, model checking, theorem proving and state bit hashing capabilities. This approach has been used to analyze a wide range of protocols and distributed systems (e.g., [6, 53, 93, 142]).

CADP is a tool set to support protocol engineering. CADP was jointly developed by the VASY team at INRIA Rhône-Alpes and the Verimag laboratory in France. It has a set of tools for compiling high-level protocol descriptions written in LOTOS [46], simulation, state space generation, minimization, comparison and model checking properties on LTSs, and testing. Cæsar is a compiler that translates a LOTOS specification into an LTS. Aldébaran allows the minimization of an LTS modulo for instance strong and branching bisimulation and compares LTSs. It has diagnosis capabilities that provide the user with explanations when two LTSs are found to be not equivalent. In the package, Evaluator [122] is an on-the-fly model checker for regular alternation-free μ-calculus formulas on LTSs. It is equipped with diagnostic generation algorithms, which construct both examples and counter-examples, i.e., portions of an LTS explaining why a formula is true or false. More information on CADP can be obtained at `http://www.inrialpes.fr/vasy/cadp/`.

Part I

Theorem Proving

Chapter 3

Cones and Foci: A Mechanical Proof Framework

3.1 Introduction

Protocol verification with the help of a theorem prover is often rather ad hoc, in the sense that one has to develop the entire proof structure from scratch. Inventing such a structure takes a lot of effort, and makes that in general such a proof cannot be readily adapted to other protocols. Groote and Springintveld [79] proposed a general proof framework for protocol verification, which they named the *cones and foci* method. In this chapter we introduce some significant improvements for this framework. Furthermore, we have cast the framework in the interactive theorem prover PVS [131].

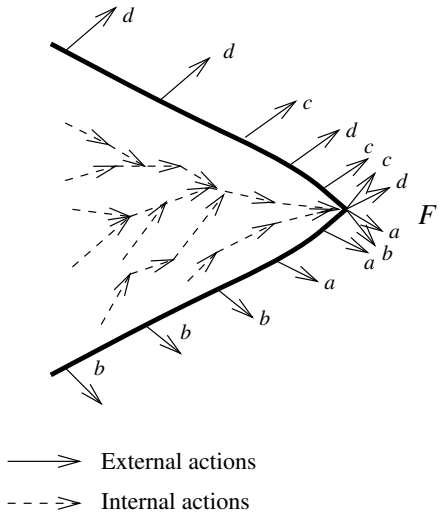
For finite labeled transition systems, checking whether two states are branching bisimilar can be performed efficiently [80]. The μCRL tool set [21] supports the generation of labeled transition systems, together with reduction modulo branching bisimulation equivalence, and allows model checking of temporal logic formulas [35] via a back-end to the CADP tool set [49]. This approach to verify system correctness has three important drawbacks. First, the labeled transition systems of the μCRL specifications involved must be generated; often the labeled transition system of the implementation of a system cannot be generated, as it is too large, or even infinite. Second, this generation usually requires a specific choice for one network or data domain; in other words, only the correctness of an instantiation of the system is proved. Third, support from and rigorous formalization by theorem provers and proof checkers is not readily available.

In this chapter we focus on analyzing protocols and distributed systems on the level of their symbolic specifications. *Linear process equations* [20] (also see Definition 2.3.1) constitute a restricted class of μCRL specifications in a so-called linear format. Algorithms have been developed to transform μCRL specifications into this linear format [76, 81, 170]. In a linear process equation, the states of the associated labeled transition system are data objects.

The cones and foci method from [79] rephrases the question whether two linear process equations are branching bisimilar in terms of proof obligations on

relations between data objects. These proof obligations can be derived by means of algebraic calculations, in general with the help of invariants (i.e., properties of the reachable states) that are proved separately. This method was used in the verification of a considerable number of real-life protocols (e.g., [60, 72, 157]), often with the support of a theorem prover or proof checker.

The main idea of the cones and foci method is that quite often in the implementation of a system, τ-transitions progress inertly towards a state in which no τ can be executed; such a state is declared to be a *focus point*. The *cone* of a focus point consists of the states that can reach this focus point by a string of τ-transitions. In the absence of infinite sequences of τ-transitions, each state belongs to some cone. This core idea is depicted below. Note that the external actions at the edge of the depicted cone can also be executed in the ultimate focus point F; this is essential for soundness of the cones and foci method, as otherwise τ-transitions in the cone would not be inert.



The starting point of the cones and foci method are two linear process equations, expressing the implementation and the desired external behavior of a system. A *state mapping* ϕ relates each state of the implementation to a state of the desired external behavior. Groote and Springintveld [79] formulated *matching criteria*, consisting of relations between data objects, which ensure that states s and $\phi(s)$ are branching bisimilar.

If an implementation, with all internal activity hidden, gives rise to infinite sequences of τ-actions, then Groote and Springintveld [79] distinguish between *progressing* and *non-progressing* τ's, where the latter are treated in the same way as external actions. They require that there is no infinite sequence of progressing τ's, and define focus points as the states that cannot perform progressing τ's. A *pre-abstraction function* divides occurrences of τ in the implementation into progressing and non-progressing ones; in many cases it is far from trivial to

define the proper pre-abstraction. Finally, a special *fair abstraction rule* [8] can be used to try and eliminate the remaining (non-progressing) τ's.

In this chapter, we propose an adaptation of the cones and foci method, in which the cumbersome treatment of infinite sequences of τ-transitions is no longer necessary. This improvement of the cones and foci method was conceived during the verification of a sliding window protocol [51] (also see Chapter 4), where the adaptation simplified matters considerably. As before, the method deals with linear process equations, requires the definition of a state mapping, and generates the same matching criteria. However, we allow the user to freely assign which states are focus points (instead of prescribing that they are the states in which no progressing τ-actions can be performed), as long as each state is in the cone of some focus point. We do allow infinite sequences of τ-transitions. No distinction between progressing and non-progressing τ's is needed, and τ-loops are eliminated without having to resort explicitly to a fair abstraction rule. We prove that our method is sound modulo branching bisimulation equivalence.

Compared to the original cones and foci method [79], our method is more generally applicable. As expected, some extra price may have to be paid for this generalization. Groote and Springintveld must prove strong termination of progressing τ-transitions. They use a standard approach to prove strong termination: provide a well-founded ordering on states such that for each progressing τ-transition $s \xrightarrow{\tau} s'$ one has $s > s'$. Here we must prove that each state can reach a focus point by a series of τ-transitions. This means that in principle we have a weaker proof obligation, but for a larger class of τ-transitions. We develop a set of rules to prove the reachability of focus points. These rules have been formalized and proved in PVS.

We formalize the cones and foci method in PVS. The intent is to provide a common framework for mechanical verification of protocols using our approach. PVS theories are developed to represent basic notions like labeled transition systems, branching bisimulation, linear process equations, and then the cones and foci method itself. The proof of soundness for the method has been mechanically checked by PVS within this framework. Once we had the linear process equations, the state mapping and the focus condition encoded in PVS, the PVS theorem prover and its type-checking condition system were then used to generate and verify all correctness conditions to ensure that the implementation and the external behavior of a system are branching bisimilar.

We apply our mechanical proof framework to the Concurrent Alternating Bit Protocol [105], which served as the main example in [79]. Our aims are to compare our method with the one from [79], and to illustrate our mechanical proof framework and our approach to the reachability analysis of focus points. While the old cones and foci method required a typical cumbersome treatment of τ-loops, here we can take these τ-loops in our stride. Thanks to the mechanical proof framework we detected a bug in one of the invariants of our original manual proof. The reachability analysis of focus points is quite crisp.

Related Work. In compiler correctness, advances have been made to validate programs at a symbolic level with respect to an underlying simulation notion (e.g., [34, 66, 129]). The methodology surrounding cones and foci incorporates well-known and useful concepts such as the precondition/effect notation [97, 117], invariants and simulations. Linear process equations resemble the UNITY format [31] and recursive applicative program schemes [37]; state mappings are comparable to refinement mappings [118, 140] and simulation [57]. Van der Zwaag [180] gave an adaptation of the cones and foci method from [79] to a timed setting, modulo timed branching bisimulation equivalence.

Outline of the chapter. This chapter is organized as follows. In Section 3.2, we present the main theorem and prove that our method is sound modulo branching bisimulation equivalence. A proof theory for reachability of focus points is also presented. In Section 3.3, the cones and foci method is formalized in PVS, and a mechanical proof framework is set up. In Section 3.4, we illustrate the method by verifying the Concurrent Alternating Bit Protocol. Part of the verification within the mechanical proof framework in PVS is presented in Section 3.4.4. We draw some conclusions in Section 3.5.

3.2 Cones and Foci

In this section, we present our version of the cones and foci method from [79] in the setting of μCRL. We do not describe the treatment of data types in μCRL in detail. For our purpose it is sufficient that processes can be parametrized with data. We assume the data sort of booleans *Bool* with constant T and F, and the usual connectives $\wedge$, $\vee$, $\neg$ and $\Rightarrow$. For a boolean b, we abbreviate $b = \mathsf{T}$ to b and $b = \mathsf{F}$ to $\neg b$.

Suppose that we have an LPE $X(d{:}D)$ specifying the implementation of a system, and an LPE $Y(d'{:}D')$ (without occurrences of τ) specifying the desired external behavior of this system. We want to prove that the implementation exhibits the desired external behavior.

We assume the presence of an invariant $\mathcal{I} : D \to \mathit{Bool}$ for X. In the cones and foci method, a *state mapping* $\phi : D \to D'$ relates each state of the implementation X to a state of the desired external behavior Y. Furthermore, some states in D are designated to be *focus points*. In contrast with the approach of [79], we allow to freely designate focus points, as long as each state $d{:}D$ of X with $\mathcal{I}(d)$ can reach a focus point by a sequence of τ-transitions. If a number of *matching criteria* for $d{:}D$ are fulfilled, consisting of relations between data objects, and if $\mathcal{I}(d)$, then the states d and $\phi(d)$ are guaranteed to be branching bisimilar. These matching criteria require that (A) all τ-transitions at d are inert, (B) each external transition of d can be mimicked by $\phi(d)$, and (C) if d is a focus point, then vice versa each transition of $\phi(d)$ can be mimicked by d.

In Section 3.2.1, we present the general theorem underlying our method. Then we introduce proof rules for the reachability of focus points in Section 3.2.2.

3.2.1 The general theorem

Let the LPE X be of the form

$$X(d{:}D) = \sum_{a \in Act \cup \{\tau\}} \sum_{e:E_a} a(f_a(d,e)) \cdot X(g_a(d,e)) \lhd h_a(d,e) \rhd \delta.$$

Furthermore, let the LPE Y be of the form

$$Y(d'{:}D') = \sum_{a \in Act} \sum_{e:E_a} a(f'_a(d',e)) \cdot Y(g'_a(d',e)) \lhd h'_a(d',e) \rhd \delta.$$

Note that Y is not allowed to have τ-steps. We start with defining the predicate FC, designating the focus points of X in D. Next we define the state mapping together with its matching criteria.

Definition 3.2.1 (Focus point) A *focus condition* is a mapping $FC : D \to Bool$. If $FC(d)$, then d is called a *focus point*.

Definition 3.2.2 (State mapping) A *state mapping* is of the form $\phi : D \to D'$.

Definition 3.2.3 (Matching criteria) A state mapping $\phi : D \to D'$ satisfies the *matching criteria* for $d{:}D$ if for all $a \in Act$:

I $\quad \forall e{:}E_a\,(h_\tau(d,e) \Rightarrow \phi(d) = \phi(g_\tau(d,e)))$;

II $\quad \forall e{:}E_a\,(h_a(d,e) \Rightarrow h'_a(\phi(d),e))$;

III $\quad FC(d) \Rightarrow \forall e{:}E_a\,(h'_a(\phi(d),e) \Rightarrow h_a(d,e))$;

IV $\quad \forall e{:}E_a\,(h_a(d,e) \Rightarrow f_a(d,e) = f'_a(\phi(d),e))$;

V $\quad \forall e{:}E_a\,(h_a(d,e) \Rightarrow \phi(g_a(d,e)) = g'_a(\phi(d),e))$.

Matching criterion I requires that the τ-transitions at d are inert, meaning that d and $g_\tau(d,e)$ are branching bisimilar. Criteria II, IV and V express that each external transition of d can be simulated by $\phi(d)$. Finally, criterion III expresses that if d is a focus point, then each external transition of $\phi(d)$ can be simulated by d.

Theorem 3.2.4 Assume LPEs $X(d{:}D)$ and $Y(d'{:}D')$ written as before (Definition 2.3.1). Let $\mathcal{I} : D \to Bool$ be an invariant for X. Suppose that for all $d{:}D$ with $\mathcal{I}(d)$,

1. $\phi : D \to D'$ satisfies the matching criteria for d, and
2. there is a $\hat{d}{:}D$ such that $FC(\hat{d})$ and $d \xrightarrow{\tau} \cdots \xrightarrow{\tau} \hat{d}$ in the LTS for X.

Then for all $d{:}D$ with $\mathcal{I}(d)$,

$$X(d) \leftrightarrows_b Y(\phi(d)).$$

Proof. We assume without loss of generality that D and D' are disjoint. Define $\mathcal{B} \subseteq D \cup D' \times D \cup D'$ as the smallest relation such that whenever $\mathcal{I}(d)$ for a $d{:}D$ then $d \mathcal{B} \phi(d)$ and $\phi(d) \mathcal{B} d$. Clearly, $\mathcal{B}$ is symmetric. We show that $\mathcal{B}$ is a branching bisimulation relation.

Let $s \mathcal{B} t$ and $s \xrightarrow{\ell} s'$. First consider that case where $\phi(s) = t$. By definition of $\mathcal{B}$ we have $\mathcal{I}(s)$.

- If $\ell = \tau$, then $h_\tau(s,e)$ and $s' = g_\tau(s,e)$ for some $e{:}E$. By matching criterion I, $\phi(g_\tau(s,e)) = t$. Moreover, $\mathcal{I}(s)$ and $h_\tau(s,e)$ together imply $\mathcal{I}(g_\tau(s,e))$. Hence, $g_\tau(s,e) \mathcal{B} t$.
- If $\ell \neq \tau$, then $h_a(s,e)$, $s' = g_a(s,e)$ and $\ell = a(f_a(s,e))$ for some $a \in Act$ and $e{:}E$. By matching criteria II and IV, $h'_a(t,e)$ and $f_a(s,e) = f'_a(t,e)$. Hence, $t \xrightarrow{a(f_a(s,e))} g'_a(t,e)$. Moreover, $\mathcal{I}(s)$ and $h_a(s,e)$ together imply $\mathcal{I}(g_a(s,e))$, and matching criterion V yields $\phi(g_a(s,e)) = g'_a(t,e)$, so $g_a(s,e) \mathcal{B} g'_a(t,e)$.

Next consider the case where $s = \phi(t)$. Since $s \xrightarrow{\ell} s'$, for some $a \in Act$ and $e{:}E$, $h'_a(s,e)$, $s' = g'_a(s,e)$ and $\ell = a(f'_a(s,e))$. By definition of $\mathcal{B}$ we have $\mathcal{I}(t)$. By assumption 2 of the Theorem, there is a $\hat{t}{:}D$ with $FC(\hat{t})$ such that $t \xrightarrow{\tau} ... \xrightarrow{\tau} \hat{t}$ in the LTS for X. Invariant $\mathcal{I}$, so also the matching criteria, hold for all states on this τ-path. Repeatedly applying matching criterion I we get $\phi(\hat{t}) = \phi(t) = s$. So matching criterion III together with $h'_a(s,e)$ yields $h_a(\hat{t},e)$. Then by matching criterion IV, $f_a(\hat{t},e) = f'_a(s,e)$, so $t \xrightarrow{\tau} ... \xrightarrow{\tau} \hat{t} \xrightarrow{a(f'_a(s,e))} g_a(\hat{t},e)$. Moreover, $\mathcal{I}(\hat{t})$ and $h_a(\hat{t},e)$ together imply $\mathcal{I}(g_a(\hat{t},e))$, and matching criterion V yields $\phi(g_a(\hat{t},e)) = g'_a(s,e)$, so $s \mathcal{B} \hat{t}$ and $g'_a(s,e) \mathcal{B} g_a(\hat{t},e)$.

Concluding, $\mathcal{B}$ is a branching bisimulation relation. ⊠

We note that Groote and Springintveld [79] proved for their version of the cones and foci method that it can be derived from the axioms of μCRL, which implies that their method is sound modulo branching bisimulation equivalence.

3.2.2 Proof rules for reachability

The cones and foci method requires as input a state mapping and a focus condition. It generates two kinds of proof obligations: matching criteria, and a reachability criterion. The latter states that from all reachable states, a state satisfying the focus condition must be reachable. Note that it suffices to prove that from any state satisfying a given set of invariants, a state satisfying the focus conditions is reachable. In this section we develop proof rules, in order to establish this condition. First we introduce some notation.

Definition 3.2.5 (τ-Reachability) Given an LTS $(S, Lab, \rightarrow, s_0)$ and $\phi, \psi \subseteq S$. ψ is τ-reachable from ϕ, written as $\phi \twoheadrightarrow \psi$, if and only if for all $x \in \phi$ there exists a $y \in \psi$ such that $x \xrightarrow{\tau} \cdots \xrightarrow{\tau} y$.

The above mentioned reachability criterion can now be expressed as $Inv \twoheadrightarrow FC$, where Inv denotes a set of invariants, and FC denotes the focus condition. Here and in the sequel, we use predicates with variables from the state vector to denote sets of states.

Definition 3.2.6 (Reachability in one τ-step) Let $X(d{:}D)$ be an LPE (see Definition 3.2.1). The set of states $Pre_X(\psi)$, that can reach the set of states ψ in one τ-step, is defined as:

$$Pre_X(\psi)(d) = \exists e{:}E(h_\tau(d,e) \wedge \psi(g_\tau(d,e)))$$

Lemma 3.2.7 (Proof rules for reachability) A list of rules for proving $\twoheadrightarrow$ with respect to an LPE X are given as follows:

- (precondition) $Pre_X(\phi) \twoheadrightarrow \phi$
- (implication) If $\phi \Rightarrow \psi$ then $\phi \twoheadrightarrow \psi$.
- (transitivity) If $\phi \twoheadrightarrow \psi$ and $\psi \twoheadrightarrow \chi$ then $\phi \twoheadrightarrow \chi$.
- (disjunction) If $\phi \twoheadrightarrow \chi$ and $\psi \twoheadrightarrow \chi$, then $\{\phi \vee \psi\} \twoheadrightarrow \chi$.
- (invariant) If $\phi \twoheadrightarrow \psi$ and $\mathcal{I}$ is an invariant, then $\{\phi \wedge \mathcal{I}\} \twoheadrightarrow \{\psi \wedge \mathcal{I}\}$.
- (induction) If for all $n > 0$, $\{\phi \wedge (t = n)\} \twoheadrightarrow \{\phi \wedge (t < n)\}$, then $\phi \twoheadrightarrow \{\phi \wedge (t = 0)\}$, where t is any term containing state variables from D.

Proof. These rules can be easily proved. In the precondition rule we obtain a one step reduction from the semantics of LPEs. The implication rule is obtained by an empty reduction sequence; for transitivity we can concatenate the reduction sequences. The disjunction rule can be proved by case distinction. For the invariant rule, assume that $\phi(d)$ and $\mathcal{I}(d)$ hold. By the assumption $\phi \twoheadrightarrow \psi$, we obtain a sequence $d \xrightarrow{\tau} \cdots \xrightarrow{\tau} d'$, such that $\psi(d')$. Because $\mathcal{I}$ is an invariant, we have $\mathcal{I}(d')$ (by induction on the length of that reduction). So indeed $\{\psi \wedge \mathcal{I}\}(d')$. Finally, for the induction rule we first prove with well-founded induction over n and using the transitivity rule that $\forall n.\{\phi \wedge (t = n)\} \twoheadrightarrow \{\phi \wedge (t = 0)\}$. Then observe that $\phi \Rightarrow \{\phi \wedge (t = t)\}$, and use the implication and transitivity rule to conclude that $\phi \twoheadrightarrow \{\phi \wedge (t = 0)\}$. ⊠

The proof rules for reachability were proved correct in PVS, and they were used in the verification of the reachability criterion for the CABP in PVS, which we will present in Section 3.4.4.

3.3 A Mechanical Proof Framework

In this section, our method is formalized in the language of the interactive theorem prover PVS [131]. This formalism enables computer aided protocol verification using the cones and foci method. PVS is chosen for the following

main reasons. First, the specification language of PVS is based on simply typed higher-order logics. PVS provides a rich set of types and the ability to define subtypes and dependent types. Second, PVS constitutes a powerful, extensible system for verifying obligations. It has a tool set consisting of a type checker, an interactive theorem prover, and a model checker. Third, PVS includes high level proof strategies and decision procedures that take care of many of the low level details associated with computer aided theorem proving. In addition, PVS has useful proof management facilities, such as a graphical display of the proof tree, and proof stepping and editing.

The type system of PVS contains basic types such as *boolean, natural, integer, real, et al.* and type constructors such as *set, tuple, record,* and *function.* Tuple, record, and type constructors are extensively used in the following sections to formalize the cones and foci method. Tuple types have the form `[T1,...,Tn]`, where the `Ti` are type expressions. A record is a finite list of fields of the form `R:TYPE=[# E1:T1, ...,En:Tn #]`, where the `Ei` are *record accessor* functions. Associated with every tuple type or record is a set of projection functions: `‘1,‘2,...,` (or `proj_1,proj_2,...`). A function constructor has the form `F:TYPE=[T1,...,Tn->R]`, where `F` is a function with domain `T1`×`T2`×...×`Tn` and range `R`.

A PVS specification can be structured through a hierarchy of *theories.* Each theory consists of a *signature* for the type names, constants introduced in the theory, axioms, definitions, and theorems associated with the signature. A PVS theory can be parametric in certain specified types and values, which are placed between `[ ]` after the theory name. A theory can build on other theories. To import a theory, PVS uses the notation `IMPORTING` followed by the theory name. For example, we can give part of the theory of abstract reduction systems [7] in PVS as follows:

```
ARS[A:TYPE]: THEORY BEGIN
      x,y,z:VAR A   n:VAR nat   R:VAR pred[[A,A]]
      iterate(R,n)(x,y):RECURSIVE bool=
         IF n=0 THEN x=y
         ELSE EXISTS z:iterate(R,n-1)(x,z) AND R(z,y)
         ENDIF MEASURE n
      star(R)(x,y):bool= EXISTS n:iterate(R,n)(x,y)
      ...
END ARS
```

Theory `ARS` contains the basic notations, like transitive closure of a relation, and theorems for abstract reduction systems. The rest of this section gives the main part of the PVS formalism of our approach. We will explain PVS notation throughout this section, when necessary.

3.3.1 LTSs and branching bisimulation

We formalize basic notions like labeled transition systems, branching bisimulation, linear process equations from Chapter 2 in PVS. An LTS (see Defini-

tion 2.2.1) is parameterized by a set of states `D`, a set of actions `Act` and a special action `tau`. The type `LTS` is then defined as a record containing an initial state, and a ternary step relation. The relation `step_01` extends `step` with the reflexive closure of the `tau`-steps. We also abbreviate the reflexive transitive closure of `tau`-steps `tau_star`. Finally, the set `reachable` of states reachable from the initial state can be easily characterized using an inductive definition.

```
LTS[D,Act:TYPE,tau:Act]:  THEORY BEGIN
      IMPORTING ARS[D]
      LTS: TYPE = [# init:D, step:[D,Act,D->bool] #]
      x,y:VAR D   a:VAR Act   lts:VAR LTS
      step(lts,a)(x,y):bool= lts`step(x,a,y)
      step_01(lts)(x,a,y):bool= lts`step(x,a,y) OR (a=tau AND x=y)
      tau_star(lts)(x,y):bool= star(step(lts,tau))(x,y)
      reachable(lts)(x):INDUCTIVE bool=
         x=lts`init OR EXISTS y,a:
            reachable(lts)(y) AND lts`step(y,a,x)
END LTS
```

To define a branching bisimulation relation (see Definition 2.2.3) between two labeled transition systems in PVS, we first introduce a formalization of a branching simulation relation in PVS. A relation is a branching bisimulation if and only if both itself and its inverse are a branching simulation relation.

```
BRANCHING_SIMULATION [D,E,Act:TYPE,tau:Act]:  THEORY BEGIN
      IMPORTING LTS[D,Act,tau], LTS[E,Act,tau]
      x1,y1,z1:VAR D   x2,y2,z2:VAR E
      lts1:VAR LTS[D,Act,tau]   lts2:VAR LTS[E,Act,tau]
      a:VAR Act   R:VAR [D,E->bool]
      brsim(lts1,lts2)(R):bool=
         FORALL x1,a,z1,x2:lts1`step(x1,a,z1) AND R(x1,x2) IMPLIES
         EXISTS y2,z2:tau_star(lts2)(x2,y2) AND
            step_01(lts2)(y2,a,z2) AND R(x1,y2) AND R(z1,z2)
END BRANCHING_SIMULATION

BRANCHING_BISIMULATION [D,E,Act:TYPE,tau:Act]:  THEORY BEGIN
      IMPORTING BRANCHING_SIMULATION[D,E,Act,tau],
         BRANCHING_SIMULATION[E,D,Act,tau]
      x1:VAR D   x2:VAR E
      lts1:VAR LTS[D,Act,tau]   lts2:VAR LTS[E,Act,tau]
      a:VAR Act   R:VAR [D,E->bool]
      brbisim(lts1,lts2)(R):bool=
         brsim(lts1,lts2)(R) AND brsim(lts2,lts1)(converse(R))
      brbisimilar(lts1,lts2)(x1,x2):bool=
         EXISTS R:brbisim(lts1,lts2)(R) AND R(x1,x2)
      brbisimilar(lts1,lts2):bool=
         brbisimilar(lts1,lts2)(lts1`init,lts2`init)
END BRANCHING_BISIMULATION
```

In our actual PVS theory of branching bisimulation, we also defined a semi-branching bisimulation relation [64]. In [13], this notion was used to show that branching bisimilarity is an equivalence. Basten showed that the relation composition of two branching bisimulation relations is not necessarily again a branching bisimulation relation, while the relation composition of two semi-branching bisimulation relations is again a semi-branching bisimulation relation. It is easy to see that semi-branching bisimilarity is reflexive and symmetric. Hence, semi-branching bisimilarity is an equivalence relation. Basten also proved that semi-branching bisimilarity and branching bisimilarity coincide, that means two states in an LTS are related by a branching bisimulation relation if and only if they are related by a semi-branching bisimulation relation. Thus, he proved that branching bisimilarity is an equivalence relation. We have checked these facts in PVS.

3.3.2 Representing LPEs and invariants

We now show how an LPE (see Definition 2.3.1) can be represented in PVS. The formal definitions will slightly deviate from the mathematical presentation before. A first decision was to represent μCRL abstract data types directly by PVS types. This enables one to reuse the PVS library for definitions and theorems of "standard" data types, and to focus on the behavioral part.

A second distinction will be that we assumed so far that LPEs are *clustered*. This means that each action name occurs in at most one summand, so that the set of summands can be indexed by the set of action names Act. This is no real limitation, because any LPE can be transformed into clustered form, basically by replacing $+$ by $\sum$ over finite types. Clustered LPEs enable a notationally smoother presentation of the theory. However, when working with concrete LPEs this restriction is not convenient, so we avoid it in the PVS framework: an arbitrarily sized index set $\{0, \ldots, n-1\}$ will be used, represented by the PVS type `below(n)`. A third deviation is that we will assume from now on that every summand has the same set E of local variables (instead of E_a before). Again this is no limitation, because void summations can always be added (i.e.: $p = \sum_{e:E} p$, when e doesn't occur in p). This restriction is needed to avoid the use of polymorphism, which doesn't exist in PVS. A fourth deviation is that we do not distinguish action names from action data parameters. We simply work with one type Act of expressions for actions. Note that this is a real extension. Namely, in our PVS formalization, each LPE summand is a function from $D \times E$ (with D the set of states) to $Act \times Bool \times D$, so one summand may now generate steps with various action names, possibly visible as well as invisible.

So an LPE is parameterized by a set of actions (`Act`), a global parameter (`State`) and a local variable (`Local`), and by the size of its index set (`n`) and the special action τ (`tau`). Note that the guard, action and next-state of a summand depend on the global parameter `d:State` and on the local variable `e:Local`. This dependency is represented in the definition `SUMMAND` by a PVS function type. An LPE consists of an initial state and a list of summands indexed by `below(n)`. Finally, the function `lpe2lts` provides the LTS semantics of an

LPE, `Step(L,a)` provides the corresponding binary relation on states, and the set of `Reachable` states is lifted from LTS to LPE level.

```
LPE[Act,State,Local:TYPE,n:nat,tau:Act]:  THEORY BEGIN
      IMPORTING LTS[State,Act,tau]
      SUMMAND:TYPE= [State,Local->[#act:Act,guard:bool,next:State#]]
      LPE:TYPE= [#init:State,sums:[below(n)->SUMMAND]#]
      L:VAR LPE   i:VAR below(n)   d,d1,d2:VAR State
      a:VAR Act   e:VAR Local   s:VAR SUMMAND
      step(s)(d1,a,d2):bool=
         EXISTS e:s(d1,e)`guard AND a=s(d1,e)`act
            AND d2=s(d1,e)`next
      lpe2lts(L):LTS= (#init:= init(L),
         step:= LAMBDA d1,a,d2:  EXISTS i:step(L`sums(i))(d1,a,d2)#)
      Step(L,a)(d1,d2):bool= step(lpe2lts(L),a)(d1,d2)
      Reachable(L)(d):bool= reachable(lpe2lts(L))(d)
END LPE
```

We define an invariant (see Definition 2.3.2) of an LPE in PVS by a theory `INVARIANT` as follows, where `p` is a predicate over states. `p` is an invariant of an LPE if and only if it holds initially and it is preserved by the execution of every summand. Note that we only require preservation for reachable states. This allows that previously proved invariants can be used in proving that `p` is invariant, which occurs frequently in practice. The abstract notion of reachability can itself be proved to be the strongest invariant (`reachable_inv1` and `reachable_inv2`).

```
INVARIANT[Act,State,Local:TYPE,n:nat,tau:Act]:  THEORY BEGIN
      IMPORTING LPE[Act,State,Local,n,tau]
      L:VAR LPE   p:VAR [State->bool]
      d:VAR State   a:VAR Act   e:VAR Local   i:VAR below(n)
      preserves(L,i)(p):bool=
         FORALL d,e:Reachable(L)(d) AND p(d) AND L`sums(i)(d,e)`guard
         IMPLIES p(L`sums(i)(d,e)`next)
      invariant(L)(p):bool= p(L`init) AND FORALL i:preserves(L,i)(p)
      reachable_inv1:  LEMMA invariant(L)(Reachable(L))
      reachable_inv2:  LEMMA invariant(L)(p)
         IMPLIES subset?(Reachable(L),p)
END INVARIANT
```

3.3.3 Formalizing the cones and foci method

In this section, we give the PVS development of the cones and foci method. Compared to the mathematical definitions in Section 3.2 we make two adaptations. First, we use the abstract reachability predicate instead of invariants; by the previous lemmas we can always switch back to invariants. Second, we have to reformulate the matching criteria in the setting of our slightly extended notion

of LPEs, allowing arbitrary index sets, and more action names per summand.

We start with two LPEs, for the implementation and the desired external behavior of a system, `X:LPE[Act,D,L,m,tau]` and `Y:LPE[Act,E,L,n,tau]` respectively. Both LPE `X` and LPE `Y` have the same set of actions and the same set of local variables. However, the type of global parameters (D and E, respectively) and the number of summands (`m` and `n`, respectively) may be different. Note that here we do not exclude the presence of `tau` in the LPE `Y`. For the correctness proof this restriction is not needed, and by lifting this restriction we avoid the use of subtypes in PVS. However it does not really extend the method, because the matching criteria enforce that all `tau`-steps in `Y` are `tau`-loops.

The next ingredients are the state mapping function `h:[D->E]` and a focus condition `fc:pred[D]`. But, as summands are no longer indexed by action names, we also need a mapping of the summands `k:[below(m)->below(n)]`. The idea is that summand `i:below(m)` of LPE `X` is mapped to summand `k(i):below(n)` of LPE `Y`. Having these ingredients, we can subsequently define the matching criteria (MC) and the reachability criterion (RC). The individual matching criteria (MC1–MC5) are displayed separately. The theorem `CONESFOCI` was proved in PVS along the lines of Section 3.2.

```
CONESFOCI_METHOD [D,E,L,Act:TYPE,tau:Act,m,n:nat]:  THEORY BEGIN
      IMPORTING BRANCHING_BISIMULATION [D,E,Act,tau],
         LPE[Act,D,L,m,tau], LPE [Act,E,L,n,tau]
      X:VAR LPE[Act,D,L,m,tau]   Y:VAR LPE[Act,E,L,n,tau]
      h:VAR [D->E]   fc:VAR pred[D]   k:VAR [below(m)->below(n)]
      d,d1:VAR D

      % The matching criteria:  MC1-MC5.
      ...
      MC(X,Y,k,h,fc)(d):bool=
         MC1(X,h)(d) AND MC2(X,Y,k,h)(d) AND MC3(X,Y,k,h,fc)(d)
         AND MC4(X,Y,k,h)(d) AND MC5(X,Y,k,h)(d)

      % The reachability criterion of focus points.
      RC(X,fc)(d):bool=
         EXISTS d1:fc(d1) AND tau_star(lpe2lts(X))(d,d1)

      % The main theorem.
      CONESFOCI: THEOREM
         h(X`init)=Y`init AND (FORALL d:Reachable(X)(d)
            IMPLIES MC(X,Y,k,h,fc)(d) AND RC(X,fc)(d))
         IMPLIES brbisimilar(lpe2lts(X),lpe2lts(Y))
END CONESFOCI_METHOD
```

```
x:VAR L   i:VAR [below(m)]   j:VAR [below(n)]
MC1(X,h)(d):bool= FORALL i:  FORALL x:
      X‘sums(i)(d,x)‘act=tau AND X‘sums(i)(d,x)‘guard
      IMPLIES h(d)=h(X‘sums(i)(d,x)‘next)
MC2(X,Y,k,h)(d):bool= FORALL i:  FORALL x:
      NOT X‘sums(i)(d,x)‘act=tau AND X‘sums(i)(d,x)‘guard
      IMPLIES Y‘sums(k(i))(h(d),x)‘guard
MC3(X,Y,k,h,fc)(d):bool= FORALL j:  FORALL x:
      fc(d) AND Y‘sums(j)(h(d),x)‘guard
      IMPLIES EXISTS i:
      k(i)=j AND X‘sums(i)(d,x)‘guard AND NOT X‘sums(i)(d,x)‘act=tau
MC4(X,Y,k,h)(d):bool= FORALL i:  FORALL x:
      NOT X‘sums(i)(d,x)‘act=tau AND X‘sums(i)(d,x)‘guard
      IMPLIES X‘sums(i)(d,x)‘act = Y‘sums(k(i))(h(d),x)‘act
MC5(X,Y,k,h)(d):bool= FORALL i:  FORALL x:
      NOT X‘sums(i)(d,x)‘act=tau AND X‘sums(i)(d,x)‘guard
      IMPLIES h(X‘sums(i)(d,x)‘next) = Y‘sums(k(i))(h(d),x)‘next
```

3.3.4 The symbolic reachability criterion

The last part of the formalization of the framework in PVS is on the proof rules for the reachability criterion. We start on the level of abstract reduction systems (`ARS[S]`), which talks about binary relations, formalized in PVS as `pred[S,S]`. First, we have to lift conjunction (`AND`) and disjunction (`OR`) to predicates on `S` (overloading is allowed in PVS). We use `Reach` to denote $\twoheadrightarrow$. Next, several proof rules can be expressed and proved in PVS. Here we only show the rules for disjunction and induction; the latter depends on a measure function `f:[S->nat]` (this rule is not used in the verification of Concurrent Alternating Bit Protocol later, but it was essential in the verification of the Sliding Window Protocol (see Chapter 4)).

```
REACH_CONDITION [S:TYPE]: THEORY BEGIN
      IMPORTING ARS[S]
      X,Y,Z:VAR pred[S]   x,y:VAR S   R:VAR pred[[S,S]]
      AND(X,Y)(x):bool= X(x) AND Y(x) ;
      OR(X,Y)(x) :bool= X(x) OR Y(x) ;
      Reach(R)(X,Y):bool= FORALL x:X(x)
         IMPLIES EXISTS y:Y(y) AND star(R)(x,y)
      reach_disjunction:  LEMMA % Disjunction rule
         Reach(R)(X,Z) AND Reach(R)(Y,Z) IMPLIES Reach(R)(X OR Y,Z)
      f:VAR [S->nat]   n:VAR nat
      reach_induction:  LEMMA % Induction rule
         (FORALL n:n>0 IMPLIES
            Reach(R)( X AND LAMBDA x:f(x)=n, X AND LAMBDA x:f(x)<n))
         IMPLIES Reach(R)( X, X AND LAMBDA x:f(x)=0 )
END REACH_CONDITION
```

Finally, the *precondition* and *invariant* rules depend on the LPE under scrutiny, so we define them in a separate theory:

```
PRECONDITION [Act,State,Local:TYPE,n:nat,tau:Act]:  THEORY BEGIN
      IMPORTING INVARIANT[Act,State,Local,n,tau],
         REACH_CONDITION[State]
      L:VAR LPE   X,Y:VAR pred[State]   i:VAR below(n)
      d:VAR State   e:VAR Local   I:VAR [State->bool]
      precondition(L,X)(d):bool=
         EXISTS i:  EXISTS e:L`sums(i)(d,e)`act=tau
         AND L`sums(i)(d,e)`guard AND X(L`sums(i)(d,e)`next)
      reach_precondition:  LEMMA % Precondition rule
         Reach(Step(L,tau))(precondition(L,X),X)
      reach_invariant:  LEMMA % Invariant rule
         Reach(Step(L,tau))(X,Y) AND invariant(L)(I)
         IMPLIES Reach(Step(L,tau))(X AND I, Y AND I)
END PRECONDITION
```

To connect the proof rules on the `Reach` predicate with the reachability condition of the previous section, we proved the following theorem in PVS:

```
reachability[D,E,L,Act:TYPE, tau:Act, m,n:nat]:  THEORY BEGIN
      IMPORTING CONESFOCI_METHOD[D,E,L,Act,tau,m,n],
         PRECONDITION[Act,D,L,m,tau]
      I,fc:  VAR [D->bool] X: VAR LPE[Act,D,L,m,tau] d:  VAR D
      REACH_CRIT: LEMMA invariant(L)(I) AND Reach(Step(L,tau))(I,fc)
         IMPLIES (FORALL d:Reachable(L)(d) IMPLIES RC(L,fc)(d))
END reachability
```

This finishes the formalization of the cones and foci method in PVS. We view this as an important step. First of all, this part is protocol independent, so it can be reused in different protocol verifications. Second, it provides a rigorous formalization of the meta-theory. For a concrete protocol specification and implementation, and given invariants, mapping functions and focus condition, all proof obligations can be generated automatically and proved with relatively little effort. The theorem `CONESFOCI` in Section 3.3.3 states that this is sufficient to prove that the implementation is correct w.r.t. the specification modulo branching bisimulation. No additional axioms are used besides the standard PVS library. The files of the PVS formalization of the cones and foci method can be found at `http://www.cwi.nl/~vdpol/conesfoci/`.

3.4 Application to the CABP

Groote and Springintveld [79] proved correctness of the Concurrent Alternating Bit Protocol (CABP) [105] as an application of their cones and foci method. Here we redo their correctness proof using our version of the cones and foci method, where in contrast to [79] we can take τ-loops in our stride. We also

illustrate our mechanical proof framework and our approach to the reachability analysis of focus points by this case study.

3.4.1 Informal description

In the CABP, data elements $d_1, d_2, \ldots$ are communicated from a data transmitter S to a data receiver R via a lossy channel, so that a message can be corrupted or lost. Therefore, acknowledgments are sent from R to S via a lossy channel. In the CABP, sending and receiving of acknowledgments is decoupled from R and S, in the form of separate components AS and AR, respectively, where AS autonomously sends acknowledgments to AR.

S attaches a bit 0 to data elements d_{2k-1} and a bit 1 to data elements d_{2k}, and AS sends back the attached bit to acknowledge reception. S keeps on sending a pair (d_i, b) until AR receives the bit b and succeeds in sending the message ac to S; then S starts sending the next pair $(d_{i+1}, 1-b)$. Alternation of the attached bit enables R to determine whether a received datum is really new, and alternation of the acknowledging bit enables AR to determine which datum is being acknowledged.

The CABP contains unbounded internal behavior, which occurs when a channel eternally corrupts or loses the same datum or acknowledgment. The fair abstraction paradigm [8], which underlies branching bisimulation, says that such infinite sequences of faulty behavior do not exist in reality, because the chance of a channel failing infinitely often is zero. Groote and Springintveld [79] defined a pre-abstraction function to hide all τ's except those that are executed in focus points, and used Koomen's fair abstraction rule [8] to eliminate the remaining τ-loops. In our adaptation of the cones and foci method, neither pre-abstraction nor Koomen's fair abstraction rule are needed.

The structure of the CABP is shown in Figure 3.1. The CABP system is built from six components.

S is a *data transmitter*, which reads a datum from port 1 and transmits such a datum repeatedly via channel K, until an acknowledgment ac regarding this datum is received from AR.

K is a lossy *data transmission channel*, which transfers data from S to R. Either it delivers the datum correctly, or it can make two sorts of mistakes: lose the datum or change it into a checksum error ce.

R is a *data receiver*, which receives data from K, sends freshly received data into port 2, and sends an acknowledgment to AS via port 5.

AS is an *acknowledgment transmitter*, which receives an acknowledgment from R and repeatedly transmits it via L to AR.

L is a lossy *acknowledgment transmission channel*, which transfers acknowledgments from AS to AR. Either it delivers the acknowledgment correctly, or it can make two sorts of mistakes: lose the acknowledgment or change it into an acknowledgment error ae.

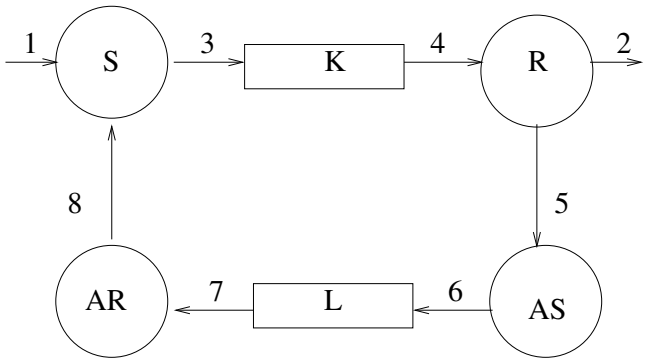


Figure 3.1: The structure of the CABP

AR is an *acknowledgment receiver*, which receives acknowledgments from L and passes them on to S.

The components can perform read $r_n(...)$ and send $s_n(...)$ actions to transport data through port n. A read and a send action over the same port n can synchronize into a communication action $c_n(...)$.

3.4.2 μCRL specification

We give descriptions of the data types and each component's specification in μCRL, which were originally presented in [79]. For convenience of notation, in each summand of the μCRL specifications below, we only present the parameters whose values are changed, e.g. d/d_s denotes that the new value of the parameter d_s is d.

We use the sort *Nat* of natural numbers, and the sort *Bit* with elements b_0 and b_1 with an inversion function $inv : Bit \to Bit$ to model the alternating bit. The sort D contains the data elements to be transferred. The sort *Frame* consists of pairs $\langle d, b\rangle$ with $d{:}D$ and $b{:}Bit$. *Frame* also contains two error messages, *ce* for checksum error and *ae* for acknowledgment error. $eq : S \times S \to Bool$ coincides with the equality relation between elements of the sort S.

The data transmitter S reads a datum at port 1 and repeatedly transmits the datum with a bit b_s attached at port 3 until it receives an acknowledgment *ac* through port 8; after that, the bit-to-be-attached is inverted. The parameter i_s is used to model the state of the data transmitter.

Definition 3.4.1 (Data transmitter)

$$
\begin{aligned}
& S(d_s{:}D, b_s{:}Bit, i_s{:}Nat) \\
= \;& \textstyle\sum_{d:D} r_1(d)\cdot S(d/d_s, 2/i_s) \lhd eq(i_s, 1) \rhd \delta \\
+ \;& (s_3(\langle d_s, b_s\rangle)\cdot S() + r_8(ac)\cdot S(inv(b_s)/b_s, 1/i_s)) \lhd eq(i_s, 2) \rhd \delta
\end{aligned}
$$

The data transmission channel K reads a datum at port 3. It can do one of three things: it can deliver the datum correctly via port 4, lose the datum, or corrupt the datum by changing it into *ce*. The non-deterministic choice between the three options is modeled by the action j. b_k is the attached alternating bit for K. And its state is modeled by the parameter i_k.

Definition 3.4.2 (Data transmission channel)

$$\begin{aligned}
& K(d_k{:}D, b_k{:}Bit, i_k{:}Nat) \\
= \;& \textstyle\sum_{d:D} \sum_{b:Bit} r_3(\langle d, b\rangle)\cdot K(d/d_k, b/b_k, 2/i_k) \lhd eq(i_k, 1) \rhd \delta \\
+ \;& (j\cdot K(1/i_k) + j\cdot K(3/i_k) + j\cdot K(4/i_k)) \lhd eq(i_k, 2) \rhd \delta \\
+ \;& s_4(\langle d_k, b_k\rangle)\cdot K(1/i_k) \lhd eq(i_k, 3) \rhd \delta \\
+ \;& s_4(ce)\cdot K(1/i_k) \lhd eq(i_k, 4) \rhd \delta
\end{aligned}$$

The data receiver R reads a datum at port 4. If the datum is not a checksum *ce* and if the bit attached is the expected bit, it sends the received datum into port 2, sends an acknowledgment *ac* via port 5, and inverts the bit-to-be-expected is inverted. If the datum is *ce* or the bit attached is not the expected one, the datum is simply ignored. The parameter i_r is used to model the state of the data receiver.

Definition 3.4.3 (Data receiver)

$$\begin{aligned}
& R(d_r{:}D, b_r{:}Bit, i_r{:}Nat) \\
= \;& \textstyle\sum_{d:D} r_4(\langle d, b_r\rangle)\cdot R(d/d_r, 2/i_r) \lhd eq(i_r, 1) \rhd \delta \\
+ \;& (r_4(ce) + \textstyle\sum_{d:D} r_4(\langle d, inv(b_r)\rangle))\cdot R() \lhd eq(i_r, 1) \rhd \delta \\
+ \;& s_2(d_r)\cdot R(3/i_r) \lhd eq(i_r, 2) \rhd \delta \\
+ \;& s_5(ac)\cdot R(inv(b_r)/b_r, 1/i_r) \lhd eq(i_r, 3) \rhd \delta
\end{aligned}$$

The acknowledgment transmitter AS repeats sending its acknowledgment bit b'_r via port 6, until it receives an acknowledgment *ac* from port 5, after which the acknowledgment bit is inverted.

Definition 3.4.4 (Acknowledgment transmitter)

$$AS(b'_r{:}Bit) = r_5(ac)\cdot AS(inv(b'_r)/b'_r) + s_6(b'_r)\cdot AS()$$

The acknowledgment transmission channel L reads an acknowledgment bit from port 6. It non-deterministically does one of three things: deliver it correctly via port 7, lose the acknowledgment, or corrupt the acknowledgment by changing it to *ae*. The non-deterministic choice between the three options is modeled by the action j. b_l is the attached alternating bit for L. And its state is modeled by the parameter i_l.

Definition 3.4.5 (Acknowledgment transmission channel)

$$\begin{aligned}
& L(b_l{:}Bit, i_l{:}Nat) \\
= \;& \textstyle\sum_{b:Bit} r_6(b)\cdot L(b/b_l, 2/i_l) \lhd eq(i_l,1) \rhd \delta \\
+ \;& (j\cdot L(1/i_l) + j\cdot L(3/i_l) + j\cdot L(4/i_l)) \lhd eq(i_l,2) \rhd \delta \\
+ \;& s_7(b_l)\cdot L(1/i_l) \lhd eq(i_l,3) \rhd \delta \\
+ \;& s_7(ae)\cdot L(1/i_l) \lhd eq(i_l,4) \rhd \delta
\end{aligned}$$

The acknowledgment receiver AR reads an acknowledgment bit from port 7. If the bit is the expected one, it sends an acknowledgment ac to the data transmitter S via port 8, after which the bit-to-be-expected is inverted. Acknowledgments errors ae or unexpected bits are ignored. And its state is modeled by the parameter i'_s.

Definition 3.4.6 (Acknowledgment receiver)

$$\begin{aligned}
& AR(b'_s{:}Bit, i'_s{:}Nat) \\
= \;& r_7(b'_s)\cdot AR(2/i'_s) \lhd eq(i'_s,1) \rhd \delta \\
+ \;& (r_7(ae) + r_7(inv(b'_s)))\cdot AR() \lhd eq(i'_s,1) \rhd \delta \\
+ \;& s_8(ac)\cdot AR(inv(b'_s)/b'_s, 1/i'_s) \lhd eq(i'_s,2) \rhd \delta
\end{aligned}$$

The μCRL specification of the CABP is obtained by putting the six components in parallel and encapsulating the internal actions at ports $\{3,4,5,6,7,8\}$. Synchronization between the components is modeled by communication actions at connecting ports.

Definition 3.4.7 Let H denote $\{s_3, r_3, s_4, r_4, s_5, r_5, s_6, r_6, s_7, r_7, s_8, r_8\}$, and I denote $\{c_3, c_4, c_5, c_6, c_7, c_8, j\}$.

$$\begin{aligned}
& CABP(d{:}D) \\
= \;& \tau_I(\partial_H(S(d,b_0,1) \parallel AR(b_0,1) \parallel K(d,b_1,1) \parallel L(b_1,1) \parallel R(d,b_0,1) \parallel AS(b_1)))
\end{aligned}$$

Next the CABP is expanded to an LPE Sys. Note that the parameters b'_s (of AR) and b'_r (of AS) are missing. The reason for this is that during the linearization the communications at ports 6 and 7 enforce $eq(b'_s, b_l)$ and $eq(b'_r, b_l)$.

Lemma 3.4.8 For all $d{:}D$ we have

$$CABP(d) = Sys(d, b_0, 1, 1, d, b_0, 1, d, b_1, 1, b_1, 1)$$

where

$$
\begin{aligned}
& Sys(d_s{:}D, b_s{:}Bit, i_s{:}Nat, i'_s{:}Nat, d_r{:}D, b_r{:}Bit, i_r{:}Nat, d_k{:}D, b_k{:}Bit, \\
& \quad i_k{:}Nat, b_l{:}Bit, i_l{:}Nat) \\
= \;& \textstyle\sum_{d:D} r_1(d){\cdot}Sys(d/d_s, 2/i_s) \lhd eq(i_s, 1) \rhd \delta && (1)\\
+ \;& \tau{\cdot}Sys(d_s/d_k, b_s/b_k, 2/i_k) \lhd eq(i_s, 2) \wedge eq(i_k, 1) \rhd \delta && (2)\\
+ \;& (\tau{\cdot}Sys(1/i_k) + \tau{\cdot}Sys(3/i_k) + \tau{\cdot}Sys(4/i_k)) \lhd eq(i_k, 2) \rhd \delta && (3)\\
+ \;& \tau{\cdot}Sys(d_k/d_r, 2/i_r, 1/i_k) \lhd eq(i_r, 1) \wedge eq(b_r, b_k) \wedge eq(i_k, 3) \rhd \delta && (4)\\
+ \;& \tau{\cdot}Sys(1/i_k) \lhd eq(i_r, 1) \wedge eq(b_r, inv(b_k)) \wedge eq(i_k, 3) \rhd \delta && (5)\\
+ \;& \tau{\cdot}Sys(1/i_k) \lhd eq(i_r, 1) \wedge eq(i_k, 4) \rhd \delta && (6)\\
+ \;& s_2(d_r){\cdot}Sys(3/i_r) \lhd eq(i_r, 2) \rhd \delta && (7)\\
+ \;& \tau{\cdot}Sys(inv(b_r)/b_r, 1/i_r) \lhd eq(i_r, 3) \rhd \delta && (8)\\
+ \;& \tau{\cdot}Sys(inv(b_r)/b_l, 2/i_l) \lhd eq(i_l, 1) \rhd \delta && (9)\\
+ \;& (\tau{\cdot}Sys(1/i_l) + \tau{\cdot}Sys(3/i_l) + \tau{\cdot}Sys(4/i_l)) \lhd eq(i_l, 2) \rhd \delta && (10)\\
+ \;& \tau{\cdot}Sys(1/i_l, 2/i'_s) \lhd eq(i'_s, 1) \wedge eq(b_l, b_s) \wedge eq(i_l, 3) \rhd \delta && (11)\\
+ \;& \tau{\cdot}Sys(1/i_l) \lhd eq(i'_s, 1) \wedge eq(b_l, inv(b_s)) \wedge eq(i_l, 3) \rhd \delta && (12)\\
+ \;& \tau{\cdot}Sys(1/i_l) \lhd eq(i'_s, 1) \wedge eq(i_l, 4) \rhd \delta && (13)\\
+ \;& \tau{\cdot}Sys(inv(b_s)/b_s, 1/i_s, 1/i'_s) \lhd eq(i_s, 2) \wedge eq(i'_s, 2) \rhd \delta && (14)
\end{aligned}
$$

Proof. See [79]. ⊠

The specification of the external behavior of the CABP is a one-datum buffer, which repeatedly reads a datum at port 1, and sends out this same datum at port 2.

Definition 3.4.9 The LPE of the external behavior of the CABP is

$$B(d{:}D, b{:}Bool) = \textstyle\sum_{d':D} r_1(d'){\cdot}B(d', \mathsf{F}) \lhd b \rhd \delta + s_2(d){\cdot}B(d, \mathsf{T}) \lhd \neg b \rhd \delta.$$

3.4.3 Verification using cones and foci

We apply our version of the cones and foci method to verify the CABP. Let Ξ abbreviate $D \times Bit \times Nat \times Nat \times D \times Bit \times Nat \times D \times Bit \times Nat \times Bit \times Nat$. Furthermore, let $\xi{:}\Xi$ denote $(d_s, b_s, i_s, i'_s, d_r, b_r, i_r, d_k, b_k, i_k, b_l, i_l)$. We list six invariants for the CABP, which are taken from [79].

Definition 3.4.10

$$
\begin{array}{lcl}
\mathcal{I}_1(\xi) & \equiv & eq(i_s,1) \vee eq(i_s,2) \\
\mathcal{I}_2(\xi) & \equiv & eq(i'_s,1) \vee eq(i'_s,2) \\
\mathcal{I}_3(\xi) & \equiv & eq(i_k,1) \vee eq(i_k,2) \vee eq(i_k,3) \vee eq(i_k,4) \\
\mathcal{I}_4(\xi) & \equiv & eq(i_r,1) \vee eq(i_r,2) \vee eq(i_r,3) \\
\mathcal{I}_5(\xi) & \equiv & eq(i_l,1) \vee eq(i_l,2) \vee eq(i_l,3) \vee eq(i_l,4) \\
\mathcal{I}_6(\xi) & \equiv & (eq(i_s,1) \Rightarrow eq(b_s,inv(b_k)) \wedge eq(b_s,b_r) \wedge eq(d_s,d_k) \\
 & & \qquad \wedge eq(d_s,d_r) \wedge eq(i'_s,1) \wedge eq(i_r,1)) \\
 & & \wedge\ (eq(b_s,b_k) \Rightarrow eq(d_s,d_k)) \\
 & & \wedge\ (eq(i_r,2) \vee eq(i_r,3) \Rightarrow eq(d_s,d_r) \wedge eq(b_s,b_r) \wedge eq(b_s,b_k)) \\
 & & \wedge\ (eq(b_s,inv(b_r)) \Rightarrow eq(d_s,d_r) \wedge eq(b_s,b_k)) \\
 & & \wedge\ (eq(b_s,b_l) \Rightarrow eq(b_s,inv(b_r))) \\
 & & \wedge\ (eq(i'_s,2) \Rightarrow eq(b_s,b_l)).
\end{array}
$$

$\mathcal{I}_1 \sim \mathcal{I}_5$ describe the range of the data parameters i_s, i'_s, i_k, i_r, and i_l, respectively. $\mathcal{I}_6$ expresses that each component in Figure 3.1 either has received information about the datum being transmitted which it must forward, or did not yet receive this information.

Lemma 3.4.11 $\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{I}_3$, $\mathcal{I}_4$, $\mathcal{I}_5$ and $\mathcal{I}_6$ are invariants of *Sys*.

Proof. We need to show that the invariants are preserved by each of the summands $(1) - (14)$ in the specification of *Sys*. Invariants $\mathcal{I}_1 - \mathcal{I}_5$ are trivial to prove. To prove $\mathcal{I}_6$, we divide $\mathcal{I}_6$ into its six parts:

$$
\begin{array}{lcl}
\mathcal{I}_{61}(\xi) & \equiv & (eq(i_s,1) \Rightarrow eq(b_s,inv(b_k)) \wedge eq(b_s,b_r) \wedge eq(d_s,d_k) \\
 & & \qquad \wedge eq(d_s,d_r) \wedge eq(i'_s,1) \wedge eq(i_r,1)) \\
\mathcal{I}_{62}(\xi) & \equiv & eq(b_s,b_k) \Rightarrow eq(d_s,d_k) \\
\mathcal{I}_{63}(\xi) & \equiv & eq(i_r,2) \vee eq(i_r,3) \Rightarrow eq(d_s,d_r) \wedge eq(b_s,b_r) \wedge eq(b_s,b_k) \\
\mathcal{I}_{64}(\xi) & \equiv & eq(b_s,inv(b_r)) \Rightarrow eq(d_s,d_r) \wedge eq(b_s,b_k) \\
\mathcal{I}_{65}(\xi) & \equiv & eq(b_s,b_l) \Rightarrow eq(b_s,inv(b_r)) \\
\mathcal{I}_{66}(\xi) & \equiv & eq(i'_s,2) \Rightarrow eq(b_s,b_l).
\end{array}
$$

We consider only seven summands in the specification of *Sys*; the other summands trivially preserve $\mathcal{I}_6$. For the sake of presentation, we represent $eq(b_1,inv(b_2))$ as $\neg eq(b_1,b_2)$, where b_1 and b_2 range over the sort *Bit*.

1. Summand (1): $\mathcal{I}_6 \wedge eq(i_s,1) \Rightarrow \mathcal{I}_6(d/d_s, 2/i_s)$.

   $\mathcal{I}_{61}(d/d_s, 2/i_s)$ is straightforward. By $eq(i_s,1)$ and $\mathcal{I}_{61}$, we have $eq(i_r,1)$, $\neg eq(b_s,b_k)$, and $eq(b_s,b_r)$. By $\neg eq(b_s,b_k)$, $\mathcal{I}_{62}(d/d_s, 2/i_s)$. By $eq(i_r,1)$, $\mathcal{I}_{63}(d/d_s, 2/i_s)$. $eq(b_s,b_r)$ implies $\mathcal{I}_{64}(d/d_s, 2/i_s)$. $\mathcal{I}_{65}$, $\mathcal{I}_{66}(d/d_s, 2/i_s)$ are trivial.

2. Summand (2): $\mathcal{I}_6 \wedge eq(i_s,2) \wedge eq(i_k,1) \Rightarrow \mathcal{I}_6(d_s/d_k, b_s/b_k, 2/i_k)$.

   $eq(i_s,2)$ implies $\mathcal{I}_{61}(d_s/d_k, b_s/b_k, 2/i_k)$, $\mathcal{I}_{62}(d_s/d_k, b_s/b_k, 2/i_k)$ is straightforward. $\mathcal{I}_{63}(d_s/d_k, b_s/b_k, 2/i_k)$ and $\mathcal{I}_{64}(d_s/d_k, b_s/b_k, 2/i_k)$ follows immediately from $\mathcal{I}_{63}$ and $\mathcal{I}_{64}$, respectively. $\mathcal{I}_{65}$, $\mathcal{I}_{66}(d_s/d_k, b_s/b_k, 2/i_k)$ are trivial.

3. Summand (4): $\mathcal{I}_6 \wedge eq(i_r,1) \wedge eq(b_r,b_k) \wedge eq(i_k,3) \Rightarrow \mathcal{I}_6(d_k/d_r, 2/i_r, 1/i_k)$.

   Assuming $eq(i_s,1)$, by $\mathcal{I}_{61}$, it follows that $\neg eq(b_s,b_k)$ and $eq(b_s,b_r)$. Hence, $\neg eq(b_r,b_k)$. This contradicts the condition $eq(b_r,b_k)$. $\neg eq(i_s,1)$ implies $\mathcal{I}_{61}(d_k/d_r, 2/i_r, 1/i_k)$. $\mathcal{I}_{64}$ implies $eq(b_s,b_r) \vee eq(b_s,b_k)$, which together with the condition $eq(b_r,b_k)$ yields $eq(b_s,b_r) \wedge eq(b_s,b_k)$. So $\mathcal{I}_{62}$ implies $eq(d_s,d_k)$. Hence, $\mathcal{I}_{63}(d_k/d_r, 2/i_r, 1/i_k)$. By $eq(b_s,b_r)$, it follows that $\mathcal{I}_{64}(d_k/d_r, 2/i_r, 1/i_k)$. $\mathcal{I}_{62}$, $\mathcal{I}_{65}$, $\mathcal{I}_{66}(d_k/d_r, 2/i_r, 1/i_k)$ are trivial.

4. Summand (8): $\mathcal{I}_6 \wedge eq(i_r,3) \Rightarrow \mathcal{I}_6(inv(b_r)/b_r, 1/i_r)$.

   Assuming $eq(i_s,1)$, by $\mathcal{I}_{61}$, we have $eq(i_r,1)$, which contradicts the condition $eq(i_r,3)$. So $\mathcal{I}_{61}(inv(b_r)/b_r, 1/i_r)$. $\mathcal{I}_{63}(inv(b_r)/b_r, 1/i_r)$ is straightforward. By $eq(i_r,3)$ and $\mathcal{I}_{63}$, we have $eq(d_s,d_r)$ and $eq(b_s,b_k)$. Hence, $\mathcal{I}_{64}(inv(b_r)/b_r, 1/i_r)$. By $eq(i_r,3)$ and $\mathcal{I}_{63}$, we have $eq(b_s,b_r)$, so $\mathcal{I}_{65}$ implies $\neg eq(b_s,b_l)$. Hence, $\mathcal{I}_{65}(inv(b_r)/b_r, 1/i_r)$. $\mathcal{I}_{62}$, $\mathcal{I}_{66}(inv(b_r)/b_r, 1/i_r)$ are trivial.

5. Summand (9): $\mathcal{I}_6 \wedge eq(i_l,1) \Rightarrow \mathcal{I}_6(inv(b_r)/b_l, 2/i_l)$,

   $\mathcal{I}_{65}(inv(b_r)/b_l, 2/i_l)$ is straightforward. If $eq(i'_s,2)$, it follows $\mathcal{I}_{66}$ that $eq(b_s,b_l)$, so by $\mathcal{I}_{65}$ we have $\neg eq(b_l,b_r)$. Hence, $\mathcal{I}_{66}(inv(b_r)/b_l, 2/i_l)$. $\mathcal{I}_{61} \sim \mathcal{I}_{64}(inv(b_r)/b_l, 2/i_l)$ are trivial.

6. Summand (11): $\mathcal{I}_6 \wedge eq(i'_s,1) \wedge eq(b_l,b_s) \wedge eq(i_l,3) \Rightarrow \mathcal{I}_6(1/i_l, 2/i'_s)$.

   By $eq(b_l,b_s)$ and $\mathcal{I}_{65}$, we have $\neg eq(b_s,b_r)$. So by $\mathcal{I}_{61}$, $\neg eq(i_s,1)$. Hence, $\mathcal{I}_{61}(1/i_l, 2/i'_s)$. $eq(b_l,b_s)$ implies $\mathcal{I}_{66}(1/i_l, 2/i'_s)$. $\mathcal{I}_{62} \sim \mathcal{I}_{65}(1/i_l, 2/i'_s)$ are trivial.

7. Summand (14): $\mathcal{I}_6 \wedge eq(i_s,2) \wedge eq(i'_s,2) \Rightarrow \mathcal{I}_6(inv(b_s)/b_s, 1/i_s, 1/i'_s)$.

   To prove $\mathcal{I}_{61}(inv(b_s)/b_s, 1/i_s, 1/i'_s)$, we need to show that $eq(b_s,b_k) \wedge \neg eq(b_r,b_s) \wedge eq(d_s,d_k) \wedge eq(d_s,d_r) \wedge eq(i_r,1)$. As $eq(i'_s,2)$, by $\mathcal{I}_{66}$ we have $eq(b_s,b_l)$, so by $\mathcal{I}_{65}$, we have $\neg eq(b_s,b_r)$. By $\mathcal{I}_{64}$, it follows that $eq(d_s,d_r) \wedge eq(b_s,b_k)$. As $eq(b_s,b_k)$, by $\mathcal{I}_{62}$, $eq(d_s,d_k)$. By $\mathcal{I}_{63}$ and $\mathcal{I}_4$, $\neg eq(b_s,b_r)$ implies $eq(i_r,1)$. Hence, $\mathcal{I}_{61}(inv(b_s)/b_s, 1/i_s, 1/i'_s)$. $\mathcal{I}_{62} \sim \mathcal{I}_{66}(inv(b_s)/b_s, 1/i_s, 1/i'_s)$ are trivial.

$\boxtimes$

We define the focus condition (see Definition 3.2.1) for *Sys* as the disjunction of the conditions of summands in the LPE in Definition 3.4.8 that deal with an external action; these summands are (1) and (7). (Note that this differs from the prescribed focus condition in [79], which would be the negation of the disjunction of conditions of the summands that deal with a τ.)

Definition 3.4.12 The focus condition for *Sys* is

$$FC(\xi) = eq(i_s,1) \vee eq(i_r,2).$$

We proceed to prove that each state satisfying the invariants $\mathcal{I}_1 - \mathcal{I}_6$ can reach a focus point (see Definition 3.2.1) by a sequence of τ-transitions.

Lemma 3.4.13 (Reachability of focus points) For each $\xi{:}\Xi$ together with $\bigwedge_{n=1}^{6} \mathcal{I}_n(\xi)$, there is a $\hat{\xi}{:}\Xi$ such that $FC(\hat{\xi})$ and $\xi \xrightarrow{\tau} \cdots \xrightarrow{\tau} \hat{\xi}$ in *Sys*.

Proof. The case $FC(\xi)$ is trivial. Let $\neg FC(\xi)$; in view of $\mathcal{I}_1$ and $\mathcal{I}_4$, this implies $eq(i_s, 2) \wedge (eq(i_r, 1) \vee eq(i_r, 3))$. In case $eq(i_s, 2) \wedge eq(i_r, 3)$, by summand (8) we can reach a state with $eq(i_s, 2) \wedge eq(i_r, 1)$. From a state with $eq(i_s, 2) \wedge eq(i_r, 1)$, by $\mathcal{I}_3$ and summands (2), (3) and (6), we can reach a state where $eq(i_s, 2) \wedge eq(i_r, 1) \wedge eq(i_k, 3)$. We distinguish two cases.

1. $eq(b_r, b_k)$.

   By summand (4) we can reach a focus point.

2. $eq(b_r, inv(b_k))$.

   If $i'_s = 2$, then by summand (14) we can reach a focus point. So by $\mathcal{I}_2$ we can assume that $i'_s = 1$. By summands (5), (2) and (3), we can reach a state where $eq(i_s, 2) \wedge eq(i'_s, 1) \wedge eq(i_r, 1) \wedge eq(i_k, 3) \wedge eq(b_r, inv(b_k)) \wedge eq(b_k, b_s)$. By $\mathcal{I}_5$ and summands (10), (9) and (13) we can reach a state where $eq(i_s, 2) \wedge eq(i'_s, 1) \wedge eq(i_r, 1) \wedge eq(i_k, 3) \wedge eq(b_r, inv(b_k)) \wedge eq(b_k, b_s) \wedge eq(i_l, 3)$. If $eq(b_l, b_s)$, then by summands (11) and (14) we can reach a focus point. Otherwise, $eq(b_l, inv(b_s))$. Since $eq(b_k, b_s)$ and $eq(b_r, inv(b_k))$, we have $eq(b_l, b_r)$. By summand (12), we can reach a state where $eq(i_s, 2) \wedge eq(i'_s, 1) \wedge eq(i_r, 1) \wedge eq(i_k, 3) \wedge eq(b_r, inv(b_k)) \wedge eq(b_k, b_s) \wedge eq(i_l, 1) \wedge eq(b_l, inv(b_s)) \wedge eq(b_l, b_r)$. Then by summand (9) we can reach a state where $eq(b_l, b_s)$, since b_l is replaced by $inv(b_r)$. Then by summands (10), (11) and (14), we can reach a focus point.

Our completely formal proof in PVS has many more steps. The main steps of the proof using the rules in Definition 3.2.7 can be found in Section 3.4.4. ⊠

We define the state mapping $\phi : \Xi \to D \times \mathit{Bool}$ (see Definition 3.2.2) by

$$\phi(\xi) = \langle d_s, eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r) \rangle.$$

Intuitively, ϕ maps those states to T in which R is awaiting a datum that still has to be received by S. This is the case if either S is awaiting a datum ($eq(i_s, 1)$), or R has sent out a datum that was not yet acknowledged to S ($eq(i_r, 3) \vee \neg eq(b_s, b_r)$). Note that ϕ is independent of $i'_s, d_r, d_k, b_k, i_k, b_l, i_l$; we write $\phi(d_s, b_s, i_s, b_r, i_r)$.

Theorem 3.4.14 For all $d{:}D$ and $b_0, b_1{:}\mathit{Bit}$,

$$\mathit{Sys}(d, b_0, 1, 1, d, b_0, 1, d, b_1, 1, b_1, 1) \leftrightarrow_b B(d, \mathsf{T}).$$

Proof. It is easy to check that $\wedge_{n=1}^{6}\mathcal{I}_n(d, b_0, 1, 1, d, b_0, 1, d, b_1, 1, b_1, 1)$.

We obtain the following matching criteria (see Definition 3.2.3). For class I, we only need to check the summands (4), (8) and (14), as the other nine summands that involve an initial action leave the values of the parameters in $\phi(d_s, b_s, i_s, b_r, i_r)$ unchanged.

1. $eq(i_r, 1) \wedge eq(b_r, b_k) \wedge eq(i_k, 3) \;\Rightarrow\; \phi(d_s, b_s, i_s, b_r, i_r) = \phi(d_s, b_s, i_s, b_r, 2/i_r)$
2. $eq(i_r, 3) \;\Rightarrow\; \phi(d_s, b_s, i_s, b_r, i_r) = \phi(d_s, b_s, i_s, inv(b_r)/b_r, 1/i_r)$
3. $eq(i_s, 2) \wedge eq(i'_s, 2) \;\Rightarrow\; \phi(d_s, b_s, i_s, b_r, i_r) = \phi(d_s, inv(b_s)/b_s, 1/i_s, b_r, i_r)$

The matching criteria for the other four classes are produced by summands (1) and (7). For class II we get:

1. $eq(i_s, 1) \;\Rightarrow\; eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r)$
2. $eq(i_r, 2) \;\Rightarrow\; \neg(eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r))$

For class III we get:

1. $(eq(i_s, 1) \vee eq(i_r, 2)) \wedge (eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r)) \;\Rightarrow\; eq(i_s, 1)$
2. $(eq(i_s, 1) \vee eq(i_r, 2)) \wedge \neg(eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r)) \;\Rightarrow\; eq(i_r, 2)$

For class IV we get:

1. $\forall d{:}D\,(eq(i_s, 1) \;\Rightarrow\; d = d)$
2. $eq(i_r, 2) \;\Rightarrow\; d_r = d_s$

Finally, for class V we get:

1. $\forall d{:}D\,(eq(i_s, 1) \;\Rightarrow\; \phi(d/d_s, b_s, 2/i_s, b_r, i_r) = \langle d, \mathsf{F}\rangle)$
2. $eq(i_r, 2) \;\Rightarrow\; \phi(d_s, b_s, i_s, b_r, 3/i_r) = \langle d_s, \mathsf{T}\rangle$

We proceed to prove the matching criteria.

I.1 Let $eq(i_r, 1)$. Then

$$\begin{aligned}\phi(d_s, b_s, i_s, b_r, i_r) &= \langle d_s, eq(i_s, 1) \vee eq(1, 3) \vee \neg eq(b_s, b_r)\rangle\\ &= \langle d_s, eq(i_s, 1) \vee eq(2, 3) \vee \neg eq(b_s, b_r)\rangle\\ &= \phi(d_s, b_s, i_s, b_r, 2/i_r).\end{aligned}$$

I.2 Let $eq(i_r, 3)$. Then by $\mathcal{I}_6$, $eq(b_s, b_r)$. Hence,

$$\begin{aligned}\phi(d_s, b_s, i_s, b_r, i_r) &= \langle d_s, eq(i_s, 1) \vee eq(3, 3) \vee \neg eq(b_s, b_r)\rangle\\ &= \langle d_s, \mathsf{T}\rangle\\ &= \langle d_s, eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, inv(b_r))\rangle\\ &= \phi(d_s, b_s, i_s, inv(b_r)/b_r, 1/i_r).\end{aligned}$$

I.3 Let $eq(i'_s, 2)$. $\mathcal{I}_6$, $eq(b_s, b_l)$ together with $\mathcal{I}_6$ yield $eq(b_s, inv(b_r))$. Hence,

$$\begin{aligned}\phi(d_s, b_s, i_s, b_r, i_r) &= \langle d_s, eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r)\rangle \\ &= \langle d_s, \mathsf{T}\rangle \\ &= \langle d_s, eq(1, 1) \vee eq(i_r, 3) \vee \neg eq(inv(b_s), b_r)\rangle \\ &= \phi(d_s, inv(b_s)/b_s, 1/i_s, b_r, i_r).\end{aligned}$$

II.1 Trivial.

II.2 Let $eq(i_r, 2)$. Then clearly $\neg eq(i_r, 3)$, and by $\mathcal{I}_6$, $eq(b_s, b_r)$. Furthermore, according to $\mathcal{I}_6$, $eq(i_s, 1) \Rightarrow eq(i_r, 1)$, so $eq(i_r, 2)$ also implies $\neg eq(i_s, 1)$.

III.1 If $\neg eq(i_r, 2)$, then $eq(i_s, 1) \vee eq(i_r, 2)$ implies $eq(i_s, 1)$. If $eq(i_r, 2)$, then by $\mathcal{I}_6$, $eq(b_s, b_r)$, so that $eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r)$ implies $eq(i_s, 1)$.

III.2 If $\neg eq(i_s, 1)$, then $eq(i_s, 1) \vee eq(i_r, 2)$ implies $eq(i_r, 2)$. If $eq(i_s, 1)$, then $\neg(eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r))$ is false, so that it implies $eq(i_r, 2)$.

IV.1 Trivial.

IV.2 Let $eq(i_r, 2)$. Then by $\mathcal{I}_6$, $eq(d_r, d_s)$.

V.1 Let $eq(i_s, 1)$. Then by $\mathcal{I}_6$, $eq(i_r, 1)$ and $eq(b_s, b_r)$. So for any $d{:}D$,

$$\begin{aligned}\phi(d/d_s, b_s, 2/i_s, b_r, i_r) &= \langle d, eq(2, 1) \vee eq(1, 3) \vee \neg eq(b_s, b_r)\rangle \\ &= \langle d, \mathsf{F}\rangle.\end{aligned}$$

V.2

$$\begin{aligned}\phi(d_s, b_s, i_s, b_r, 3/i_r) &= \langle d_s, eq(i_s, 1) \vee eq(3, 3) \vee \neg eq(b_s, b_r)\rangle \\ &= \langle d_s, \mathsf{T}\rangle.\end{aligned}$$

Note that $\phi(d, b_0, 1, b_0, 1) = \langle d, \mathsf{T}\rangle$. So by Theorem 3.2.4 and Lemma 3.4.13,

$$Sys(d, b_0, 1, 1, d, b_0, 1, d, b_1, 1, b_1, 1) \leftrightarrows_b B(d, \mathsf{T}).$$

⊠

3.4.4 Illustration of the proof framework

Let us illustrate the mechanical proof framework set up in Section 3.3 on the verification of the CABP as it was described in Section 3.4.3. The purpose of this section is to show how the mechanical framework can be instantiated with a concrete protocol. A second goal is to illustrate in more detail how we can use the proof rules (see Lemma 3.2.7) for reachability, to formally prove in PVS that focus points are always reachable.

To apply the generic theory, we use the PVS mechanism of theory instantiation. For instance, the theory LPE was parameterized by sets of actions, states, *et al.* This theory will be imported, using the set of actions, states *et al.* from the linearized version of CABP, which we have to define first. To this end we start a new theory, parameterized by an arbitrary type of data elements (D, with special element $d_0{:}D$).

Defining the LPEs. The starting point will be the linearized version of the CABP, represented by *Sys* in Lemma 3.4.8. The type `cabp_state` is defined as a record of all state parameters. Note that we use the predefined PVS-types `nat` and `bool` (`bool` is also used to represent sort *Bit*). The type `cabp_act` is defined as an abstract data type. The syntax below introduces constructors (`r1,s2:[D->cabp_act]` and `tau:cabp_act`), recognizer predicates (`r1?,s2?,tau?:[cabp_act->bool]`), and another destructors (`d:[(r1?)->D]` and `d:[(r2?)->D]`). Subsequently we import the theory LPE with the corresponding parameters. The LPE for the implementation of the CABP contains 18 summands (note that summands (3) and (10) in Lemma 3.4.8 each represent three summands). Note that the only local parameter in this LPE that is bound by $\sum$ has type `D`.

```
CABP[D:TYPE+,d0:D]: THEORY BEGIN
      cabp_state:TYPE= [#ds:D,bs:bool,is:nat,i1s:nat,dr:D,br:bool,
         ir:nat,dk:D,bk:bool,ik:nat,bl:bool,il:nat#]
      cabp_act:DATATYPE BEGIN
            r1(d:D):r1?  s2(d:D):s2?  tau:tau?
         END cabp_act
      IMPORTING LPE[cabp_act,cabp_state,D,18,tau]
```

The next step is to define the implementation of the CABP as an LPE in PVS. It consists of an initial vector, and a list of summands, indexed by `LAMBDA i`. The `LAMBDA (S,d)` indicates the dependency of each summand on the state and the local variables. Note that given state `S`, `S‘x` denotes the value of parameter `x` in `S`. The notation `S WITH [x := v]` denotes the same state as `S` except the value of field `x` which is set to `v`. We only display the summands corresponding to summand (1) and (14) of *Sys*.

```
      i:VAR below(18)   S:VAR cabp_state   d:VAR D
      cabp:  LPE= (#
         init:= (#ds:=d0,bs:=FALSE,is:=1,i1s:=1,dr:=d0,
            br:=FALSE,ir:=1,dk:=d0,bk:=TRUE,ik:=1,bl:=TRUE,il:=1#),
         sums:=LAMBDA i:  LAMBDA (S,d):COND
            i=0->(#act:=r1(d),guard:=S‘is=1,
               next:=S WITH [ds:=d,is:=2]#),
            ...
            i=17->(#act:=tau,guard:=S‘is=2 AND S‘i1s=2,
               next:=S WITH [bs:=NOT S‘bs,is:=1,i1s:=1]#)
            ENDCOND#)
```

In a similar way, the desired external behavior of the CABP is presented as a one-datum buffer. The representation of the LPE B from Definition 3.4.9 in PVS is:

```
buf_state:TYPE=[#d:D,b:bool#]
B:VAR buf_state d1:VAR D j:VAR below(2)
IMPORTING LPE[cabp_act,buf_state,D,2,tau]
buffer:  LPE=
   (#init:=(#d:=d0,b:=TRUE#),
   sums:=LAMBDA j:  LAMBDA (B,d1):COND
      j=0->(#act:=r1(d1),guard:=B`b,next:=(#d:=d1,b:=FALSE#)#),
      j=1->(#act:=s2(B`d),guard:=NOT B`b,next:=B
         WITH [b:=TRUE]#)
   ENDCOND#)
```

Invariants, state mapping, focus points. The next step is to define the ingredients for the cones and foci method. We need to define invariants, a state mapping and focus points. In PVS these are all functions that take state vectors as input. We only show a snapshot:

```
IMPORTING invariant[cabp_act,cabp_state,D,18]
      I1(S):bool= S`is=1 OR S`is=2
      ...
      I64(S):bool= (S`bs = NOT S`br) IMPLIES
         S`ds=S`dr AND S`bs=S`bk
      I6(S):bool=I61(S) AND ...  AND I66(S)
IMPORTING CONESFOCI_METHOD[cabp_state,buf_state,D,cabp_act,tau,18,2]
      FC(S):bool= S`is=1 OR S`ir=2
      h(S):buf_state=(#d:=S`ds,b:=S`is=1 OR S`ir=3 OR NOT S`bs=S`br#)
cabp_inv:LEMMA invariant(cabp)(I1 AND I2 AND I3 AND I4 AND I5 AND I6)
matching:LEMMA Reachable(cabp)(S) IMPLIES MC(cabp,buffer,k,h,FC)(S)
```

The proof of the reachability criterion will be discussed in the next paragraph. The correctness of the invariants and the matching criteria were proved already (see Section 3.4). These proofs could be formalized in PVS in a straightforward fashion. The proof script follows a fixed pattern: first we unfold the definitions of LPE and invariants or matching criteria. Then we use rewriting to generate a finite conjunction from the quantification `FORALL i:below(n)`. Subsequently (using the PVS tactic `THEN*`), we apply the powerful PVS tactic `(GRIND)` to the subgoals. Sometimes a few subgoals remain, which are then proved manually.

Reachability of focus points. We formally prove Lemma 3.4.13, which states that each reachable state of the CABP can reach a focus point by a sequence of τ-transitions using the rules in Lemma 3.2.7. This corresponds to the theorem `CABP_RC` in the PVS part below. Using the general theorems `CONESFOCI` and `REACH_CRIT`, we conclude from the specific theorems `cabp_inv`, `matching` and `CABP_RC` that `CABP` is indeed `CORRECT` w.r.t. the one-datum buffer specification.

```
        IMPORTING PRECONDITION[cabp_act,cabp_state,D,18]
           ...
           CABP_RC:LEMMA
              Reach(step(cabp,tau))(I1 AND I2 AND I3 AND I4 AND I5,FC)
           CABP_CORRECT:
              THEOREM brbisimilar(lpe2lts(cabp),lpe2lts(buffer))
END CABP
```

We now explain the structure of the proof of `CABP_RC`. This proof is based on the proof rules for reachability, introduced in Sections 3.2.2 and 3.3.4. It requires some manual work, viz. the identification of the intermediate predicates, and characterizing the reachable set of states after a number of steps. Each step corresponds to a separate lemma in PVS. The atomic steps are proved by the precondition rule (semi-automatically). An example of such a lemma in PVS is:

```
        Q2(S):bool = S`ir=1 AND S`is=2 AND S`ik=2 AND S`i1s=1
           AND S`bk = S`bs
        Q3(S):bool = S`ir=1 AND S`is=2 AND S`ik=3 AND S`i1s=1
           AND S`bk = S`bs
        Q2_to_Q3:  LEMMA Reach(Tau)(Q2,Q3)
```

These basic steps are combined by using mainly the transitivity rule and the disjunction rule. We now provide the complete list of the intermediate predicates, together with the used proof rules. We do not display the use of implication and invariant rules, but of course the PVS proofs contain all details. The fragment before corresponds to the third step of item (5) below, where summand (3) is used to increase i_k.

1. $\{i_r = 1 \wedge i_s = 2 \wedge i_k = 4\} \twoheadrightarrow \{i_r = 1 \wedge i_s = 2 \wedge i_k = 1\} \twoheadrightarrow$
   $\{i_r = 1 \wedge i_s = 2 \wedge i_k = 2\} \twoheadrightarrow \{i_r = 1 \wedge i_s = 2 \wedge i_k = 3\}$
   Using the precondition rule, on summands (6), (2) and (3), respectively.

2. $\{\mathcal{I}_3 \wedge i_r = 1 \wedge i_s = 2\} \twoheadrightarrow \{i_r = 1 \wedge i_s = 2 \wedge i_k = 3\}$
   Using the disjunction rule with $i_k = 1 \vee i_k = 2 \vee i_k = 3 \vee i_k = 4$, and the transitivity rule on the results of step 1.

3. $\{i_r = 1 \wedge i_s = 2 \wedge i_k = 3 \wedge b_r = b_k\} \twoheadrightarrow FC$
   Using the precondition rule on summand (4).

4. $\{i_r = 1 \wedge i_s = 2 \wedge i_k = 3 \wedge i'_s = 2\} \twoheadrightarrow FC$
   Using the precondition rule on summand (14).

5. $\{i_r = 1 \wedge i_s = 2 \wedge i_k = 3 \wedge i'_s = 1 \wedge b_r \neq b_k\} \twoheadrightarrow$
   $\{i_r = 1 \wedge i_s = 2 \wedge i_k = 1 \wedge i'_s = 1\} \twoheadrightarrow$
   $\{i_r = 1 \wedge i_s = 2 \wedge i_k = 2 \wedge i'_s = 1 \wedge b_k = b_s\} \twoheadrightarrow$
   $\{i_r = 1 \wedge i_s = 2 \wedge i_k = 3 \wedge i'_s = 1 \wedge b_k = b_s\} =: Q$
   Using the precondition rule on summands (5), (2) and (3).

6. $\{Q \wedge i_l = 2\} \twoheadrightarrow \{Q \wedge i_l = 1\}$;
   $\{Q \wedge i_l = 4\} \twoheadrightarrow \{Q \wedge i_l = 1\}$;
   $\{Q \wedge i_l = 3 \wedge b_l \neq b_s\} \twoheadrightarrow \{Q \wedge i_l = 1\} \twoheadrightarrow \{Q \wedge i_l = 2 \wedge b_l \neq b_r\} \twoheadrightarrow \{Q \wedge i_l = 3 \wedge b_l \neq b_r\}$
   Using the precondition rule on summands (10), (13), (12), (9) and (10), respectively.

7. $\{Q \wedge (i_l \in \{1,2,4\} \vee (i_l = 3 \wedge b_l \neq b_s))\} \twoheadrightarrow \{Q \wedge i_l = 3 \wedge b_l \neq b_r\}$.
   Using the disjunction rule and the transitivity rule on the results of step 6.

8. $\{Q \wedge i_l = 3 \wedge b_l = b_s\} \twoheadrightarrow \{i_r = 1 \wedge i_s = 2 \wedge i_k = 3 \wedge i'_s = 2\} \twoheadrightarrow FC$.
   Using the precondition rule on summand (11), and then the transitivity rule with step 4.

9. $\{Q \wedge \mathcal{I}_5\} \twoheadrightarrow FC$.
   By $\mathcal{I}_5$, $i_l \in \{1,2,3,4\}$. So we can distinguish the cases $i_l \in \{1,2,4\}$, $i_l = 3 \wedge b_l \neq b_s$ and $i_l = 3 \wedge b_l = b_s$. In all but the last case, we arrive at a situation where $b_k = b_s \wedge b_l \neq b_r$ (by step 7). Note that this implies $b_k = b_r \vee b_l = b_s$. So we can use case distinction again, and reach the focus condition via step 3 or step 8.

10. $\{i_r = 1 \wedge i_s = 2 \wedge i_k = 3 \wedge \mathcal{I}_2 \wedge \mathcal{I}_5\} \twoheadrightarrow FC$.
    From $\mathcal{I}_2$ and the disjunction rule we can distinguish the cases $b_r = b_k$, $i'_s = 2$ and $i'_s = 1 \wedge b_r \neq b_k$. We solve them by the results of step 3, step 4, and transitivity of 5 and 9, respectively.

11. $\{i_r = 3 \wedge i_s = 2\} \twoheadrightarrow \{i_r = 1 \wedge i_s = 2\}$.
    Using the precondition rule on summand (8).

12. $\mathcal{I}_1 \wedge \mathcal{I}_2 \wedge \mathcal{I}_3 \wedge \mathcal{I}_4 \wedge \mathcal{I}_5 \twoheadrightarrow FC$.
    Using the invariants $\mathcal{I}_1$ and $\mathcal{I}_4$, we can distinguish the following cases:

    - $i_s = 1$ or $i_s = 2 \wedge i_r = 2$ (both reach FC in zero steps);
    - $i_s = 2 \wedge i_r = 3$ (leads to the next case by step 11);
    - $i_s = 2 \wedge i_r = 1$. This leads to $i_s = 2 \wedge i_r = 1 \wedge i_k = 3$ by step 2 and then to FC by step 10.

This finishes the complete mechanical verification of the CABP in PVS using the cones and foci method. The files of the verification of the CABP in PVS can be found at `http://www.cwi.nl/~vdpol/conesfoci/`.

3.5 Conclusions

In this chapter, we have developed a mechanical framework for protocol verification, based on the cones and foci method. We summarize our main contribution as follows:

- We generalized the original cones and foci method [79]. Compared to the original one, our method is more generally applicable, in the sense that it can deal with τ-loops without requiring a cumbersome treatment to eliminate them.
- We presented a set of rules to support the reachability analysis of focus points. They have been proved to be quite powerful in two case studies.
- We formalized the complete cones and foci method in PVS.

The feasibility of this mechanical framework has been illustrated by the verification of the CABP. We are confident that the framework forms a solid basis for mechanical protocol verification. For instance, the same framework has been applied to the verification of a sliding window protocol in μCRL (see Chapter 4), which we consider a true milestone in verification efforts using process algebra.

The foci and cones method provides a systematic approach to protocol verification. It allows for fully rigorous correctness proofs in a general setting with possibly infinite state spaces (i.e. with arbitrary data, arbitrary window size, *et al.*). The method requires intelligent manual steps, such as the invention of invariants, a state mapping, and the focus criterion. However, the method is such that after these creative parts a number of verification conditions can be generated and proved (semi-)automatically. So the strength of the mechanical framework is that one can focus on the creative steps, and check the tedious parts by a theorem prover. Yet, a complete machine-checked proof is obtained, because the meta-theory has also been proof-checked in a generic manner. We experienced that many proofs and proof scripts can be reused after small changes in the protocol, or after a change in the invariants. Actually, in some cases the PVS theorem prover assisted in finding the correct invariants.

Chapter 4

Verifying a Sliding Window Protocol in μCRL

4.1 Introduction

Sliding window protocols [30] (SWPs) ensure successful transmission of messages from a sender to a receiver through a medium, in which messages may get lost. Their main characteristic is that the sender does not wait for an incoming acknowledgment before sending next messages, for optimal use of bandwidth. This is the reason why many data communication systems include the SWP, in one of its many variations.

In SWPs, both the sender and the receiver maintain a buffer. In practice the buffer at the receiver is often much smaller than at the sender, but here we make the simplifying assumption that both buffers can contain up to n messages ($n > 0$). By providing the messages with sequence numbers, reliable in-order delivery without duplications is guaranteed. The sequence numbers can be taken modulo $2n$ (and not less, see [165] for a nice argument). The messages at the sender are numbered from i to $i+n$ (modulo $2n$); this is called a *window*. When an acknowledgment reaches the sender, indicating that k messages have arrived correctly, the window *slides* forward, so that the sending buffer can contain messages with sequence numbers $i+k$ to $i+k+n$ (modulo $2n$). The window of the receiver slides forward when the first element in this window is passed on to the environment.

Within the process algebraic community, SWPs have attracted much attention, because their precise formal verification turned out to be surprisingly difficult. We provide a comparison with verifications of SWPs from the literature in Section 4.2, and restrict ourselves here to the context in which this chapter was written. After the advent of process algebra in the early eighties of last century, it was observed that simple protocols, such as the alternating bit protocol, could readily be verified. In an attempt to show that more difficult protocols could also be dealt with, SWPs were considered. Middeldorp [125] and Brunekreef [25] gave specifications in ACP [16] and PSF [123], respectively. Vaandrager [171], Groenveld [68], van Wamel [176] and Bezem and Groote [19]

manually verified one-bit SWPs, in which the size of the sending and receiving window is one.

Starting in 1990, we attempted to prove the most complex SWP from [165] (not taking into account additional features such as duplex message passing and piggybacking) correct using μCRL, which is a suitable process algebraic formalism for such purposes. This turned out to be unexpectedly hard, and has led to the development of new proof methods for protocol verification. We therefore consider the current chapter as a true milestone in process algebraic verification.

Our first observation was that the external behavior of the protocol, as given in [165], was unclear. We adapted the SWP such that it nicely behaves as a queue of capacity $2n$. The second observation was that the SWP of [165] contained a deadlock [69, Stelling 7], which could only occur after at least n messages were transmitted. This error was communicated to Tanenbaum, and has been repaired in more recent editions of [165]. Another bug in the μCRL specification of the SWP was detected by means of a model checking analysis. A first attempt to prove the resulting SWP correct led to the verification of a bakery protocol [71], and to the development of the *cones and foci* proof method [79, 54]. This method plays an essential role in the proof in the current chapter, and has been used to prove many other protocols and distributed algorithms correct. But the correctness proof required an additional idea, already put forward by Schoone [154], to first perform the proof with unbounded sequence numbers, and to separately eliminate modulo arithmetic.

We present a specification in μCRL of a SWP with buffer size $2n$ and window size n, for arbitrary n. The medium between the sender and the receiver is modeled as a lossy queue of unbounded capacity. We manually prove that the external behavior of this protocol is branching bisimilar [64] to a FIFO queue of capacity $2n$. This proof is entirely based on the axiomatic theory underlying μCRL and the axioms characterizing the data types. It implies both safety and liveness of the protocol (the latter under the assumption of fairness). First, we linearize the specification, meaning that we get rid of parallel operators. Moreover, communication actions are stripped from their data parameters. Then we eliminate modulo arithmetic, using the proof principle CL-RSP [20]. Finally, we apply the cones and foci technique, to prove that the linear specification without modulo arithmetic is branching bisimilar to a FIFO queue of capacity $2n$. All lemmas for the data types, all invariants and all correctness proofs have been checked using PVS. The PVS files are available via `http://www.cwi.nl/~pangjun/swp/`.

A concise overview of other verifications of SWPs is presented in Section 4.2. Many of these verifications deal with either unbounded sequence numbers, in which case the intricacies of modulo arithmetic disappear, or a fixed finite window size. The papers that do treat arbitrary finite window sizes in most cases restrict to safety properties.

Outline of the chapter. This chapter is set up as follows. Section 4.2 gives an overview of related work on verifying SWPs. Section 4.3 introduces the proof techniques of μCRL used in this chapter. In Section 4.4, the data types needed for specifying the SWP and its external behavior are presented. Section 4.5 features the μCRL specifications of the SWP and its external behavior. In Section 4.6, three consecutive transformations are applied to the specification of the SWP, to linearize the specification, eliminate arguments of communication actions, and get rid of modulo arithmetic. In Section 4.7, properties of the data types and invariants of the transformed specification are proved. In Section 4.8, it is proved that the three transformations preserve branching bisimulation, and that the transformed specification behaves like a FIFO queue. We conclude this chapter in Section 4.9.

4.2 Related Work

Sliding window protocols have attracted considerable interest from the formal verification community. In this section we present an overview. Many of these verifications deal with unbounded sequence numbers, in which case modulo arithmetic is avoided, or with a fixed finite window size. The papers that do treat arbitrary finite window sizes mostly restrict to safety properties.

Infinite window size Stenning [163] studied a SWP with unbounded sequence numbers and an infinite window size, in which messages can be lost, duplicated or reordered. A timeout mechanism is used to trigger retransmission. Stenning gave informal manual proofs of some safety properties. Knuth [103] examined more general principles behind Stenning's protocol, and manually verified some safety properties. Hailpern [82] used temporal logic to formulate safety and liveness properties for Stenning's protocol, and established their validity by informal reasoning. Jonsson [97] also verified both safety and liveness properties of the protocol, using temporal logic and a manual compositional verification technique.

Fixed finite window size Richier *et al.* [146] specified a SWP in a process algebra based language Estelle/R, and verified safety properties for window size up to eight using the model checker Xesar. Madelaine and Vergamini [119] specified a SWP in LOTOS, with the help of the simulation environment Lite, and proved some safety properties for window size six. Holzmann [91, 92] used the Spin model checker to verify both safety and liveness properties of a SWP with sequence numbers up to five. Kaivola [99] verified safety and liveness properties using model checking for a SWP with window size up to seven. Godefroid and Long [65] specified a full duplex SWP in a guarded command language, and verified the protocol for window size two using a model checker based on Queue BDDs. Stahl *et al.* [162] used a combination of abstraction, data independence, compositional reasoning and model checking to verify safety and liveness properties for a SWP with window size up to sixteen. The protocol

was specified in Promela, the input language for the Spin model checker. Smith and Klarlund [160] specified a SWP in the high-level language IOA, and used the theorem prover MONA to verify a safety property for unbounded sequence numbers with window size up to 256. Latvala [112] modeled a SWP using Colored Petri nets. A liveness property was model checked with fairness constraints for window size up to eleven.

Arbitrary finite window size Cardell-Oliver [29] specified a SWP using higher order logic, and manually proved and mechanically checked safety properties using HOL. (Van de Snepscheut [161] noted that what Cardell-Oliver claims to be a liveness property is in fact a safety property.) Schoone [154] manually proved safety properties for several SWPs using assertional verification. Van de Snepscheut [161] gave a correctness proof of a SWP as a sequence of correctness preserving transformations of a sequential program. Paliwoda and Sanders [132] specified a reduced version of what they call a SWP (but which is in fact very similar to the bakery protocol from [71]) in the process algebra CSP, and verified a safety property modulo trace semantics. Röckl and Esparza [148] verified the correctness of this bakery protocol modulo weak bisimulation using Isabelle/HOL, by explicitly checking a bisimulation relation. Jonsson and Nilsson [98] used an automated reachability analysis to verify safety properties for a SWP with arbitrary sending window size and receiving window size one. Rusu [152] used the theorem prover PVS to verify both safety and liveness properties for a SWP with unbounded sequence numbers. Chkliaev *et al.* [33] used a timed state machine in PVS to specify a SWP with a timeout mechanism and proved some safety properties with the mechanical support of PVS. Correctness is based on the timeout mechanism, which allows messages in the mediums to be reordered.

4.3 Proof Techniques

The goal of this chapter is to prove that the initial state of the forthcoming μCRL specification of a SWP is branching bisimilar to a FIFO queue. In the proof of this fact, we will use three proof principles from the literature to derive that two μCRL specifications are branching (or even strongly) bisimilar: sum elimination, CL-RSP, and cones and foci.

- *Sum elimination* [71] states that a summation over a data type from which only one element can be selected can be removed. To be more precise,

$$\sum_{d:D} p(d) \lhd d = e \wedge b \rhd \delta \;\underline{\leftrightarrow}\; p(e) \lhd b \rhd \delta.$$

- *CL-RSP* [20] states that the solutions of a linear μCRL specification that does not contain any infinite τ sequence are all strongly bisimilar. This proof principle basically extends RSP [18] to a setting with data. The reader is referred to [20] for more details regarding CL-RSP.

- The *cones and foci* method from [54, 79] rephrases the question whether two linear μCRL specifications $\tau_{\mathcal{I}}(S_1)$ and S_2 are branching bisimilar, where S_2 does not contain actions from some set $\mathcal{I}$ of internal actions, in terms of data equalities. The reader is referred to Chapter 3 for the technical details of the cones and foci technique.

4.4 Data Types

In this section, the data types used in the μCRL specification of the SWP are presented: booleans, natural numbers supplied with modulo arithmetic, and buffers. Furthermore, basic properties are given for the operations defined on these data types.

4.4.1 Booleans

We introduce the data type *Bool* of booleans.

$$\begin{array}{l} \mathsf{T}, \mathsf{F} :\rightarrow \mathit{Bool} \\ \wedge, \vee : \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool} \\ \neg : \mathit{Bool} \rightarrow \mathit{Bool} \\ \Rightarrow, \Leftrightarrow : \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool} \end{array}$$

T and F denote true and false, respectively. The infix operations $\wedge$ and $\vee$ represent conjunction and disjunction, respectively. Finally, $\neg$ denotes negation. The defining equations are:

$$\begin{array}{lcllcl} b \wedge \mathsf{T} & = & b & \neg\mathsf{T} & = & \mathsf{F} \\ b \wedge \mathsf{F} & = & \mathsf{F} & \neg\mathsf{F} & = & \mathsf{T} \\ b \vee \mathsf{T} & = & \mathsf{T} & b \Rightarrow b' & = & b' \vee \neg b \\ b \vee \mathsf{F} & = & b & b \Leftrightarrow b' & = & (b \Rightarrow b') \wedge (b' \Rightarrow b) \end{array}$$

4.4.2 If-then-else and equality

For each data type D in this chapter we assume the presence of an operation

$$\mathit{if} : \mathit{Bool} \times D \times D \rightarrow D$$

with as defining equations

$$\begin{array}{lcl} \mathit{if}(\mathsf{T}, d, e) & = & d \\ \mathit{if}(\mathsf{F}, d, e) & = & e \end{array}$$

Furthermore, for each data type D in this chapter one can easily define a mapping $eq : D \times D \rightarrow \mathit{Bool}$ such that $eq(d, e)$ holds if and only if $d = e$ can be derived. For notational convenience we take the liberty to write $d = e$ instead of $eq(d, e)$.

4.4.3 Natural numbers

We introduce the data type *Nat* of natural numbers.

$$\begin{array}{l} 0 :\rightarrow Nat \\ S : Nat \rightarrow Nat \\ +, \dot{-}, \cdot : Nat \times Nat \rightarrow Nat \\ \leq, <, \geq, >: Nat \times Nat \rightarrow Bool \end{array}$$

0 denotes zero and $S(n)$ the successor of n. The infix operations $+$, $\dot{-}$ and $\cdot$ represent addition, monus (also called proper subtraction) and multiplication, respectively. Finally, the infix operations $\leq$, $<$, $\geq$ and $>$ are the less-than(-or-equal) and greater-than(-or-equal) operations. Usually, the sign for multiplication is omitted, and $\neg(i = j)$ is abbreviated to $i \neq j$.

$$\begin{array}{lcllcl} i + 0 & = & i & 0 \leq i & = & \mathsf{T} \\ i + S(j) & = & S(i + j) & S(i) \leq 0 & = & \mathsf{F} \\ i \dot{-} 0 & = & i & S(i) \leq S(j) & = & i \leq j \\ 0 \dot{-} i & = & 0 & 0 < S(i) & = & \mathsf{T} \\ S(i) \dot{-} S(j) & = & i \dot{-} j & i < 0 & = & \mathsf{F} \\ i{\cdot}0 & = & 0 & S(i) < S(j) & = & i < j \\ i{\cdot}S(j) & = & (i{\cdot}j) + i & i \geq j & = & \neg(j < i) \\ & & & i > j & = & \neg(j \leq i) \end{array}$$

We take as binding convention:

$$\{=, \neq\} > \{\cdot\} > \{+, \dot{-}\} > \{\leq, <, \geq, >\} > \{\neg\} > \{\wedge, \vee\} > \{\Rightarrow, \Leftrightarrow\}.$$

4.4.4 Modulo arithmetic

Since the size of the buffers at the sender and the receiver in the sliding window are of size $2n$, calculations modulo $2n$ play an important role. We introduce the following notation for modulo calculations:

$$\begin{array}{l} |\, : Nat \times Nat \rightarrow Nat \\ div : Nat \times Nat \rightarrow Nat \end{array}$$

$i|_n$ denotes i modulo n, while $i \; div \; n$ denotes i integer divided by n. The modulo operations are defined by the following equations (for $n > 0$):

$$\begin{array}{lcl} i|_n & = & if(i < n, i, (i \dot{-} n)|_n) \\ i \; div \; n & = & if(i < n, 0, S((i \dot{-} n) \; div \; n)) \end{array}$$

4.4.5 Buffers

The sender and the receiver in the SWP both maintain a buffer containing the sending and the receiving window, respectively (outside these windows both buffers are empty). Let Δ be the set of data elements that can be communicated between sender and receiver. The buffers are modeled as a list of pairs (d, i)

with $d{:}\Delta$ and $i{:}Nat$, representing that position (or sequence number) i of the buffer is occupied by datum d. The data type Buf is specified as follows, where $[]$ denotes the empty buffer:

$$\begin{array}{l} [] :\rightarrow Buf \\ inb : \Delta \times Nat \times Buf \rightarrow Buf \end{array}$$

$q|_n$ denotes buffer q with all sequence numbers taken modulo n.

$$\begin{array}{lcl} []|_n & = & [] \\ inb(d,i,q)|_n & = & inb(d, i|_n, q|_n) \end{array}$$

$test(i,q)$ produces T if and only if position i in q is occupied, $retrieve(i,q)$ produces the datum that resides at position i in buffer q (if this position is occupied),[1] and $remove(i,q)$ is obtained by emptying position i in buffer q.

$$\begin{array}{lcl} test(i,[]) & = & \mathsf{F} \\ test(i, inb(d,j,q)) & = & i{=}j \vee test(i,q) \\ retrieve(i, inb(d,j,q)) & = & if(i{=}j, d, retrieve(i,q)) \\ remove(i,[]) & = & [] \\ remove(i, inb(d,j,q)) & = & if(i{=}j, remove(i,q), inb(d,j,remove(i,q))) \end{array}$$

$release(i,j,q)$ is obtained by emptying positions i up to j in q. $release|_n(i,j,q)$ does the same modulo n.

$$\begin{array}{lcl} release(i,j,q) & = & if(i \geq j, q, release(S(i), j, remove(i,q))) \\ release|_n(i,j,q) & = & if(i|_n{=}j|_n, q, release|_n(S(i), j, remove(i,q))) \end{array}$$

$next\text{-}empty(i,q)$ produces the first empty position in q, counting upwards from sequence number i onward. $next\text{-}empty|_n(i,q)$ does the same modulo n.

$$\begin{array}{lcl} next\text{-}empty(i,q) & = & if(test(i,q), next\text{-}empty(S(i),q), i) \\ next\text{-}empty|_n(i,q) & = & if(next\text{-}empty(i|_n, q|_n) < n, next\text{-}empty(i|_n, q|_n), \\ & & \quad next\text{-}empty(0, q|_n)) \end{array}$$

Intuitively, $in\text{-}window(i,j,k)$ produces T if and only if j lies in the range from i to $k \dot{-} 1$, modulo n, where n is greater than i, j and k.

$$in\text{-}window(i,j,k) \quad = \quad i \leq j < k \vee k < i \leq j \vee j < k < i$$

Finally, we define an operation on buffers that is only needed in the derivation of some data equalities in Section 4.7.1: $max(q)$ produces the greatest sequence number that is occupied in q.

$$\begin{array}{lcl} max([]) & = & 0 \\ max(inb(d,i,q)) & = & if(i \geq max(q), i, max(q)) \end{array}$$

[1] Note that $retrieve(i,[])$ is undefined. One could choose to equate it to a default value in Δ, or to a fresh error element in Δ. However, the first approach could cover up flaws in the μCRL specification of the SWP, and the second approach would needlessly complicate the data type Δ. We prefer to work with a partially defined version of *retrieve*, which is allowed in μCRL. All operations in μCRL models, however, are total; partially specified operations just lead to the existence of multiple models.

4.4.6 Mediums

The medium in the SWP between the sender and the receiver is modeled as a lossy channel of unbounded capacity with FIFO behavior. We model the medium containing frames from the sender to the receiver by a data type *MedK*. It represents a list of pairs (d, i) with a datum $d{:}\Delta$ and its sequence number $i{:}Nat$. Let $[]^K$ denote an empty medium.

$$\begin{aligned} &[]^K :\rightarrow MedK \\ &inm : \Delta \times Nat \times MedK \rightarrow MedK \end{aligned}$$

$g|_n$ denotes medium g with all sequence numbers taken modulo n.

$$\begin{array}{lcl} []^K|_n & = & []^K \\ inm(d,i,g)|_n & = & inm(d, i|_n, g|_n) \end{array}$$

$member(d, i, g)$ produces T if and only if the pair (d, i) is in g. $length(g)$ denotes the length of g. $return\text{-}dat(i, g)$ and $return\text{-}seq(i, g)$ produce the datum and the sequence number, respectively, that resides at position i in g (positions are counted from 0). For convenience, we use $last\text{-}dat(g)$ and $last\text{-}seq(g)$ to produce the datum and the sequence number, respectively, that resides at the end of g. $delete(i, g)$ is obtained by emptying position i in g. Similarly, $delete\text{-}last(g)$ is obtained by emptying the last position in g.

$$\begin{array}{lcl} member(d, i, []^K) & = & \mathsf{F} \\ member(d, i, inm(e, j, g)) & = & (d = e \wedge i = j) \vee member(d, i, g) \\ length([]^K) & = & 0 \\ length(inm(d, i, g)) & = & S(length(g)) \\ return\text{-}dat(i, inm(d, j, g)) & = & if\,(i = 0, d, return\text{-}dat(i-1, g)) \\ return\text{-}seq(i, inm(d, j, g)) & = & if\,(i = 0, j, return\text{-}seq(i-1, g)) \\ last\text{-}dat(inm(d, i, g)) & = & if\,(length(g) = 0, d, last\text{-}dat(g)) \\ last\text{-}seq(inm(d, i, g)) & = & if\,(length(g) = 0, i, last\text{-}dat(g)) \\ delete(i, inm(d, j, g)) & = & if\,(i = 0, g, inm(d, j, delete(i-1, g))) \\ delete\text{-}last(inm(d, i, g)) & = & if\,(length(g) = 0, g, \\ & & \quad inm(d, i, delete\text{-}last(g))) \end{array}$$

The medium containing the sequence numbers from the receiver to the sender by a data type *MedL*. Similarly, we have the following defining equations.

$$\begin{aligned} &[]^L :\rightarrow MedL \\ &inm : Nat \times MedL \rightarrow MedL \end{aligned}$$

$$\begin{array}{lcl} []^L|_n & = & []^L \\ inm(i, g')|_n & = & inm(i|_n, g'|_n) \end{array}$$

$$
\begin{array}{lcl}
member(i, []^L) & = & \mathsf{F} \\
member(i, inm(j, g)) & = & i = j \vee member(d, i, g) \\
length([]^L) & = & 0 \\
length(inm(i, g')) & = & S(length(g')) \\
return\text{-}seq(i, inm(j, g')) & = & if(i = 0, j, return\text{-}seq(i - 1, g')) \\
last\text{-}seq(inm(i, g')) & = & if(length(g') = 0, i, last\text{-}seq(g')) \\
delete(i, inm(j, g')) & = & if(i = 0, g', inm(j, delete(i - 1, g'))) \\
delete\text{-}last(inm(j, g')) & = & if(length(g') = 0, g', inm(j, delete\text{-}last(g')))
\end{array}
$$

4.4.7 Lists

We introduce the data type of *List* of lists, which are used in the specification of the desired external behavior of the SWP: a FIFO queue of size $2n$. Let $\langle\rangle$ denote the empty list.

$$
\begin{array}{l}
\langle\rangle :\rightarrow List \\
inl : \Delta \times List \rightarrow List
\end{array}
$$

$length(\lambda)$ denotes the length of λ, $top(\lambda)$ produces the datum that resides at the top of λ, $tail(\lambda)$ is obtained by removing the top position in λ, $append(d, \lambda)$ adds datum d at the end of λ, and $\lambda{+\!\!+}\lambda'$ represents list concatenation.

$$
\begin{array}{lcl}
length(\langle\rangle) & = & 0 \\
length(inl(d, \lambda)) & = & S(length(\lambda)) \\
top(inl(d, \lambda)) & = & d \\
tail(inl(d, \lambda)) & = & \lambda \\
append(d, \langle\rangle) & = & inl(d, \langle\rangle) \\
append(d, inl(e, \lambda)) & = & inl(e, append(d, \lambda)) \\
\langle\rangle{+\!\!+}\lambda & = & \lambda \\
inl(d, \lambda){+\!\!+}\lambda' & = & inl(d, \lambda{+\!\!+}\lambda')
\end{array}
$$

Furthermore, $q[i..j\rangle$ is the list containing the elements in buffer q at positions i up to but not including j.

$$q[i..j\rangle \quad = \quad if(i \geq j, \langle\rangle, inl(retrieve(i, q), q[S(i)..j\rangle))$$

4.5 Sliding Window Protocol

In this section, a μCRL specification of a SWP is presented, together with its desired external behavior.

4.5.1 Specification of a sliding window protocol

Figure 4.1 depicts the SWP. A sender **S** stores data elements that it receives via channel A in a buffer of size $2n$, in the order in which they are received. **S** can send a datum, together with its sequence number in the buffer, to a receiver **R** via a medium that behaves as lossy queue of unbounded capacity, represented by the medium **K** and the channels B and C. Upon reception, **R** may store the

datum in its buffer, where its position in the buffer is dictated by the attached sequence number. In order to avoid a possible overlap between the sequence numbers of different data elements in the buffers of **S** and **R**, no more than one half of the buffers of **S** and **R** may be occupied at any time; these halves are called the sending and the receiving window, respectively. **R** can pass on a datum that resides at the first position in its window via channel D; in that case the receiving window slides forward by one position. Furthermore, **R** can send the sequence number of the first empty position in (or just outside) its window as an acknowledgment to **S** via a medium that behaves as lossy queue of unbounded capacity, represented by the medium **L** and the channels E and F. If **S** receives this acknowledgment, its window slides forward accordingly.

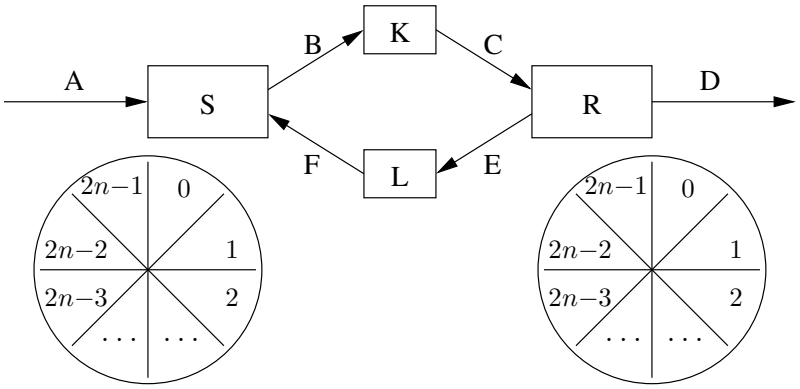


Figure 4.1: Sliding window protocol

The sender **S** is modeled by the process $\mathbf{S}(\ell, m, q)$, where q is a buffer of size $2n$, ℓ the first position in the sending window, and m the first empty position in (or just outside) the sending window. Data elements can be selected at random for transmission from (the filled part of) the sending window.

$$\begin{aligned}
\mathbf{S}(\ell{:}Nat, m{:}Nat, q{:}Buf) \;=\; & \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{S}(\ell, S(m)|_{2n}, inb(d, m, q)) \\
& \qquad \lhd\ in\text{-}window(\ell, m, (\ell+n)|_{2n}) \rhd \delta \\
+\; & \textstyle\sum_{k:Nat} s_{\mathrm{B}}(retrieve(k, q), k)\cdot\mathbf{S}(\ell, m, q) \\
& \qquad \lhd\ test(k, q) \rhd \delta \\
+\; & \textstyle\sum_{k:Nat} r_{\mathrm{F}}(k)\cdot\mathbf{S}(k, m, release|_{2n}(\ell, k, q))
\end{aligned}$$

The receiver **R** is modeled by the process $\mathbf{R}(\ell', q')$, where q' is a buffer of size $2n$ and ℓ' the first position in the receiving window.

$$\begin{aligned}\mathbf{R}(\ell'{:}Nat, q'{:}Buf) &= \textstyle\sum_{d:\Delta}\sum_{k:Nat} r_{\mathrm{C}}(d,k)\cdot(\mathbf{R}(\ell', inb(d,k,q'))\\ &\qquad \lhd\ in\text{-}window(\ell', k, (\ell'+n)|_{2n}) \rhd \mathbf{R}(\ell', q'))\\ &+ s_{\mathrm{D}}(retrieve(\ell', q'))\cdot\mathbf{R}(S(\ell')|_{2n}, remove(\ell', q'))\\ &\qquad \lhd\ test(\ell', q') \rhd \delta\\ &+ s_{\mathrm{E}}(next\text{-}empty|_{2n}(\ell', q'))\cdot\mathbf{R}(\ell', q')\end{aligned}$$

Finally, we specify the mediums $\mathbf{K}$ and $\mathbf{L}$, which have unbounded capacity and may lose frames between $\mathbf{S}$ and $\mathbf{R}$, and vice versa. We cannot allow reordering of messages in the medium, as this would violate the correctness of the protocol. The medium $\mathbf{K}$ (see Figure 4.2) is modeled by the process $\mathbf{K}(g,p)$, where $g{:}MedK$ is a buffer with unbounded capacity, and $p{:}Nat$ a pointer indicating that the frames in between position 0 and p (excluding p) can still be lost, and the frames beyond p cannot be lost any more.

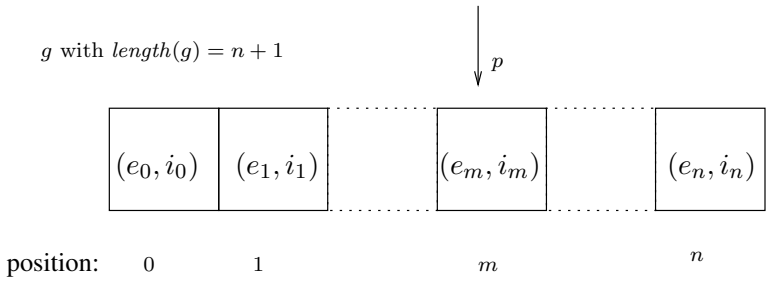


Figure 4.2: The medium $\mathbf{K}$

$\mathbf{K}$ receives a frame from $\mathbf{S}$, stores it at the front (position 0) of g, and accordingly increases p by one. It sends the last frame $(last\text{-}dat(g), last\text{-}seq(g))$ in g to $\mathbf{R}$. A frame at position k can be lost (if $k < p$), and p is then decreased by one. $\mathbf{K}$ can also make a choice that the frame at position p cannot be lost ($p{:=}p-1$). The action j expresses the nondeterministic choice whether or not a frame is lost. In a similar way, we model the medium $\mathbf{L}$ by the process $\mathbf{L}(g', p')$.

$$\begin{aligned}\mathbf{K}(g{:}MedK, p{:}Nat) &= \textstyle\sum_{d:\Delta}\sum_{k:Nat} r_{\mathrm{B}}(d,k)\cdot\mathbf{K}(inm(d,k,g), p+1)\\ &+ \textstyle\sum_{k:Nat} j\cdot\mathbf{K}(delete(k,g), p-1) \lhd\ k < p \rhd \delta\\ &+ s_{\mathrm{C}}(last\text{-}dat(g), last\text{-}seq(g))\cdot\mathbf{K}(delete\text{-}last(g), p)\\ &\qquad \lhd\ p < length(g) \rhd \delta\\ &+ j\cdot\mathbf{K}(g, p-1) \lhd\ p > 0 \rhd \delta\end{aligned}$$

$$\begin{aligned}\mathbf{L}(g'{:}MedL, p'{:}Nat) &= \textstyle\sum_{k:Nat} r_{\mathrm{E}}(k)\cdot\mathbf{L}(inm(k,g'),p'+1)\\ &+ \textstyle\sum_{k:Nat} j\cdot\mathbf{L}(delete(k,g'),p'-1) \lhd\ k<p' \rhd \delta\\ &+ s_{\mathrm{F}}(last\text{-}seq(g'))\cdot\mathbf{L}(delete\text{-}last(g'),p')\\ &\qquad \lhd\ p'<length(g') \rhd \delta\\ &+ j\cdot\mathbf{L}(g',p'-1) \lhd\ p'>0 \rhd \delta\end{aligned}$$

For each channel $i \in \{\mathrm{B},\mathrm{C},\mathrm{E},\mathrm{F}\}$, actions s_i and r_i can communicate, resulting in the action c_i. The initial state of the SWP is expressed by

$$\tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K}([]^K,0) \parallel \mathbf{L}([]^L,0)))$$

where the set $\mathcal{H}$ consists of the read and send actions over the internal channels B, C, E, and F, namely $\mathcal{H} = \{s_{\mathrm{B}}, r_{\mathrm{B}}, s_{\mathrm{C}}, r_{\mathrm{C}}, s_{\mathrm{E}}, r_{\mathrm{E}}, s_{\mathrm{F}}, r_{\mathrm{F}}\}$, while the set $\mathcal{I}$ consists of the communication actions over these internal channels together with j, namely $\mathcal{I} = \{c_{\mathrm{B}}, c_{\mathrm{C}}, c_{\mathrm{E}}, c_{\mathrm{F}}, j\}$.

4.5.2 External behavior

Data elements that are read from channel A should be sent into channel D in the same order, and no data elements should be lost. In other words, the SWP is intended to be a solution for the linear specification.

$$\begin{aligned}\mathbf{Z}(\lambda{:}List) &= \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{Z}(append(d,\lambda)) \lhd\ length(\lambda)<2n \rhd \delta\\ &+ s_{\mathrm{D}}(top(\lambda))\cdot\mathbf{Z}(tail(\lambda)) \lhd\ length(\lambda)>0 \rhd \delta\end{aligned}$$

Note that $r_{\mathrm{A}}(d)$ can be performed until the list λ contains $2n$ elements, because in that situation the sending and receiving windows will be filled. Furthermore, $s_{\mathrm{D}}(top(\lambda))$ can only be performed if λ is not empty.

The remainder of this chapter is devoted to proving the following theorem, expressing that the external behavior of our μCRL specification of a SWP corresponds to a FIFO queue of size $2n$.

Theorem 4.5.1 $\tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K}([]^K,0) \parallel \mathbf{L}([]^L,0))) \leftrightarrow_b \mathbf{Z}(\langle\rangle)$.

4.6 Transformations of the Specification

This section witnesses three transformations, one to eliminate parallel operators, one to eliminate arguments of communication actions, and one to eliminate modulo arithmetic.

4.6.1 Linearization

The starting point of our correctness proof is a linear specification $\mathbf{M}_{mod}$, in which no parallel operators occur. $\mathbf{M}_{mod}$ can be obtained from the μCRL

specification of the SWP without the hiding operator, i.e.,

$$\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K}([]^K,0) \parallel \mathbf{L}([]^L,0))$$

by means of a linearization algorithm presented in [76].

The linear specification $\mathbf{M}_{mod}$ of the SWP, with encapsulation but without hiding, takes the following form. For the sake of presentation, we only present parameters whose values are changed.

$$\begin{aligned}
& \mathbf{M}_{mod}(\ell{:}Nat, m{:}Nat, q{:}Buf, \ell'{:}Nat, q'{:}Buf, g{:}MedK, p{:}Nat, \\
& \qquad g'{:}MedL, p'{:}Nat) \\
= \;& \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{M}_{mod}(m{:=}S(m)|_{2n}, q{:=}inb(d,m,q)) \\
& \qquad \lhd \mathit{in\text{-}window}(\ell, m, (\ell+n)|_{2n}) \rhd \delta \\
+ \;& \textstyle\sum_{k:Nat} c_{\mathrm{B}}(\mathit{retrieve}(k,q),k)\cdot\mathbf{M}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k,q),k,g), p{:=}p+1) \\
& \qquad \lhd \mathit{test}(k,q) \rhd \delta \\
+ \;& \textstyle\sum_{k:Nat} j\cdot\mathbf{M}_{mod}(g{:=}\,\mathit{delete}(k,g), p{:=}p-1) \lhd\ k<p \rhd \delta \\
+ \;& j\cdot\mathbf{M}_{mod}(p{:=}p-1) \lhd\ p>0 \rhd \delta \\
+ \;& c_{\mathrm{C}}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g))\cdot \\
& \mathbf{M}_{mod}(q'{:=}\mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q'), g{:=}\mathit{delete\text{-}last}(g)) \\
& \qquad \lhd p<\mathit{length}(g) \wedge \mathit{in\text{-}window}(\ell', \mathit{last\text{-}seq}(g), (\ell'+n)|_{2n}) \rhd \delta \\
+ \;& c_{\mathrm{C}}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g))\cdot\mathbf{M}_{mod}(g{:=}\mathit{delete\text{-}last}(g)) \\
& \qquad \lhd p<\mathit{length}(g) \wedge \neg\mathit{in\text{-}window}(\ell', \mathit{last\text{-}seq}(g), (\ell'+n)|_{2n}) \rhd \delta \\
+ \;& s_{\mathrm{D}}(\mathit{retrieve}(\ell', q'))\cdot\mathbf{M}_{mod}(\ell'{:=}S(\ell')|_{2n}, q'{:=}\mathit{remove}(\ell', q')) \lhd\ \mathit{test}(\ell', q') \rhd \delta \\
+ \;& c_{\mathrm{E}}(\mathit{next\text{-}empty}|_{2n}(\ell', q'))\cdot \\
& \mathbf{M}_{mod}(g'{:=}\mathit{inm}(\mathit{next\text{-}empty}|_{2n}(\ell', q'), g'), p'{:=}p'+1) \\
+ \;& \textstyle\sum_{k:Nat} j\cdot\mathbf{M}_{mod}(g'{:=}\mathit{delete}(k,g'), p'{:=}p'-1) \lhd\ k<p' \rhd \delta \\
+ \;& j\cdot\mathbf{M}_{mod}(p'{:=}p'-1) \lhd\ p'>0 \rhd \delta \\
+ \;& c_{\mathrm{F}}(\mathit{last\text{-}seq}(g'))\cdot \\
& \mathbf{M}_{mod}(\ell{:=}\mathit{last\text{-}seq}(g'), q{:=}\mathit{release}|_{2n}(\ell, \mathit{last\text{-}seq}(g'), q), g'{:=}\mathit{delete\text{-}last}(g')) \\
& \qquad \lhd p'<\mathit{length}(g') \rhd \delta
\end{aligned}$$

Theorem 4.6.1

$$\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K}([]^K,0) \parallel \mathbf{L}([]^L,0)) \underline{\leftrightarrow} \mathbf{M}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0).$$

Proof. It is not hard to see that replacing $\mathbf{M}_{mod}(\ell,m,q,\ell',q',g,p,g',p')$ by $\partial_{\mathcal{H}}(\mathbf{S}(\ell,m,q) \parallel \mathbf{R}(\ell',q') \parallel \mathbf{K}(g,p) \parallel \mathbf{L}(g',p'))$ is a solution for the recursive equation above, using the axioms of μCRL [74]. (The details are left to the reader.) Hence, the theorem follows by CL-RSP [20]. ⊠

4.6.2 Eliminating arguments of communication actions

The linear specification $\mathbf{N}_{mod}$ is obtained from $\mathbf{M}_{mod}$ by stripping all arguments from communication actions, and renaming these actions to a fresh action c.

$$
\begin{aligned}
& \mathbf{N}_{mod}(\ell{:}Nat, m{:}Nat, q{:}Buf, \ell'{:}Nat, q'{:}Buf, g{:}MedK, p{:}Nat, \\
& \qquad g'{:}MedL, p'{:}Nat) \\
= \;& \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{N}_{mod}(m{:=}S(m)|_{2n}, q{:=}\mathit{inb}(d,m,q)) \\
& \qquad \lhd\ \mathit{in\text{-}window}(\ell, m, (\ell+n)|_{2n})\ \rhd\ \delta \\
+ \;& \textstyle\sum_{k:Nat} c\cdot\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k,q),k,g), p{:=}p+1)\ \lhd\ \mathit{test}(k,q)\ \rhd\ \delta \\
+ \;& \textstyle\sum_{k:Nat} j\cdot\mathbf{N}_{mod}(g{:=}\ \mathit{delete}(k,g), p{:=}p-1)\ \lhd\ k<p\ \rhd\ \delta \\
+ \;& j\cdot\mathbf{N}_{mod}(p{:=}p-1)\ \lhd\ p>0\ \rhd\ \delta \\
+ \;& c\cdot\mathbf{N}_{mod}(q'{:=}\mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q'), g{:=}\mathit{delete\text{-}last}(g)) \\
& \qquad \lhd\ p<\mathit{length}(g) \wedge \mathit{in\text{-}window}(\ell', \mathit{last\text{-}seq}(g), (\ell'+n)|_{2n})\ \rhd\ \delta \\
+ \;& c\cdot\mathbf{N}_{mod}(g{:=}\mathit{delete\text{-}last}(g)) \\
& \qquad \lhd\ p<\mathit{length}(g) \wedge \neg\mathit{in\text{-}window}(\ell', \mathit{last\text{-}seq}(g), (\ell'+n)|_{2n})\ \rhd\ \delta \\
+ \;& s_{\mathrm{D}}(\mathit{retrieve}(\ell',q'))\cdot\mathbf{N}_{mod}(\ell'{:=}S(\ell')|_{2n}, q'{:=}\mathit{remove}(\ell',q'))\ \lhd\ \mathit{test}(\ell',q')\ \rhd\ \delta \\
+ \;& c\cdot\mathbf{N}_{mod}(g'{:=}\mathit{inm}(\mathit{next\text{-}empty}|_{2n}(\ell',q'), g'), p'{:=}p'+1) \\
+ \;& \textstyle\sum_{k:Nat} j\cdot\mathbf{N}_{mod}(g'{:=}\mathit{delete}(k,g'), p'{:=}p'-1)\ \lhd\ k<p'\ \rhd\ \delta \\
+ \;& j\cdot\mathbf{N}_{mod}(p'{:=}p'-1)\ \lhd\ p'>0\ \rhd\ \delta \\
+ \;& c\cdot\mathbf{N}_{mod}(\ell{:=}\mathit{last\text{-}seq}(g'), q{:=}\mathit{release}|_{2n}(\ell, \mathit{last\text{-}seq}(g'), q), g'{:=}\mathit{delete\text{-}last}(g')) \\
& \qquad \lhd\ p'<\mathit{length}(g')\ \rhd\ \delta
\end{aligned}
$$

Theorem 4.6.2

$$\tau_{\mathcal{I}}(\mathbf{M}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0)) \leftrightarrow \tau_{\{c,j\}}(\mathbf{N}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0)).$$

Proof. By a simple renaming. ⊠

4.6.3 Getting rid of modulo arithmetic

The specification of $\mathbf{N}_{nonmod}$ is obtained by eliminating all occurrences of $|_{2n}$ from $\mathbf{N}_{mod}$, replacing $\mathit{in\text{-}window}(\ell, m, (\ell+n)|_{2n})$ by $m < \ell + n$, and replacing $\mathit{in\text{-}window}(\ell', \mathit{last\text{-}seq}(g), (\ell'+n)|_{2n})$ by $\ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n$.

$$\begin{aligned}
&\mathbf{N}_{nonmod}(\ell{:}Nat, m{:}Nat, q{:}Buf, \ell'{:}Nat, q'{:}Buf, g{:}MedK, p{:}Nat,\\
&\qquad g'{:}MedL, p'{:}Nat)\\
=\ & \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{N}_{nonmod}(m{:=}S(m), q{:=}inb(d,m,q)) \lhd\ m < \ell + n \rhd \delta && (A)\\
+\ & \textstyle\sum_{k:Nat} c\cdot\mathbf{N}_{nonmod}(g{:=}inm(retrieve(k,q),k,g), p{:=}p+1) \lhd\ test(k,q) \rhd \delta && (B)\\
+\ & \textstyle\sum_{k:Nat} j\cdot\mathbf{N}_{nonmod}(g{:=}\ delete(k,g), p{:=}p-1) \lhd\ k < p \rhd \delta && (C)\\
+\ & j\cdot\mathbf{N}_{nonmod}(p{:=}p-1) \lhd\ p > 0 \rhd \delta && (D)\\
+\ & c\cdot\mathbf{N}_{nonmod}(q'{:=}inb(last\text{-}dat(g), last\text{-}seq(g), q'), g{:=}delete\text{-}last(g))\\
&\qquad \lhd p < length(g) \wedge (\ell' \leq last\text{-}seq(g) < \ell' + n) \rhd \delta && (E)\\
+\ & c\cdot\mathbf{N}_{nonmod}(g{:=}delete\text{-}last(g))\\
&\qquad \lhd p < length(g) \wedge \neg(\ell' \leq last\text{-}seq(g) < \ell' + n) \rhd \delta && (F)\\
+\ & s_{\mathrm{D}}(retrieve(\ell', q'))\cdot\mathbf{N}_{nonmod}(\ell'{:=}S(\ell'), q'{:=}remove(\ell', q')) \lhd\ test(\ell', q') \rhd \delta && (G)\\
+\ & c\cdot\mathbf{N}_{nonmod}(g'{:=}inm(next\text{-}empty(\ell', q'), g'), p'{:=}p'+1) && (H)\\
+\ & \textstyle\sum_{k:Nat} j\cdot\mathbf{N}_{nonmod}(g'{:=}delete(k,g'), p'{:=}p'-1) \lhd\ k < p' \rhd \delta && (I)\\
+\ & j\cdot\mathbf{N}_{nonmod}(p'{:=}p'-1) \lhd\ p' > 0 \rhd \delta && (J)\\
+\ & c\cdot\mathbf{N}_{nonmod}(\ell{:=}last\text{-}seq(g'), q{:=}release(\ell, last\text{-}seq(g'), q), g'{:=}delete\text{-}last(g'))\\
&\qquad \lhd p' < length(g') \rhd \delta && (K)
\end{aligned}$$

Theorem 4.6.3

$$\mathbf{N}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0) \leftrightarrows \mathbf{N}_{nonmod}(0,0,[],0,[],[]^K,0,[]^L,0).$$

The proof of Theorem 4.6.3 is presented in Section 4.8.1. Next, in Section 4.8.2, we prove the correctness of $\mathbf{N}_{nonmod}$. In these proofs we will need a wide range of data equalities, which we proceed to prove in Section 4.7.

4.7 Properties of Data

4.7.1 Basic properties

In the correctness proof we will make use of basic properties of the operations on *Nat* and *Bool*, which are derivable from their axioms (using induction). Some typical examples of such properties are:

$$\begin{aligned}
\neg\neg b &= b\\
i + k < j + k &= i < j\\
i \geq j \Rightarrow (i \dot{-} j) + k &= (i + k) \dot{-} j
\end{aligned}$$

Lemmas 4.7.1 and 4.7.2 collect basic facts on modulo arithmetic and on buffers, respectively. Lemma 4.7.3 contains some results on modulo arithmetic related to buffers. Lemma 4.7.4 presents some facts on the *next-empty* operation, together

with one result on *max*, which is needed to derive those facts. Lemmas 4.7.5 and 4.7.6 collect some results on unbounded buffers. Finally, Lemma 4.7.7 contains basic facts on lists. Unless stated otherwise (this only happens in Lemmas 4.7.3.2-4.7.3.6, 4.7.3.9 and 4.7.5.12) all variables that occur in a data lemma are implicitly universally quantified at the outside of the equality.

Lemma 4.7.1 Let $n > 0$.

1. $(i|_n + j)|_n = (i + j)|_n$
2. $i|_n < n$
3. $(i{\cdot}n)|_n = 0$
4. $i = (i \mathbin{div} n){\cdot}n + i|_n$
5. $j \leq i \leq j + n$
   $\Rightarrow (i \mathbin{div} 2n = j \mathbin{div} 2n \wedge j|_{2n} \leq i|_{2n} \leq j|_{2n} + n) \vee (i \mathbin{div} 2n = S(j \mathbin{div} 2n) \wedge i|_{2n} + n \leq j|_{2n})$
6. $i \leq j \;\Rightarrow\; i \mathbin{div} n \leq j \mathbin{div} n$

Proof.

1. By induction on i.
   - $i < n$. Then $i|_n = i$.
   - $i \geq n$.

$$\begin{array}{lll} & (i|_n + j)|_n & \\ = & ((i \dot{-} n)|_n + j)|_n & \\ = & ((i \dot{-} n) + j)|_n & \text{(by induction, } i, n > 0) \\ = & ((i + j) \dot{-} n)|_n & (i \geq n) \\ = & (i + j)|_n & \end{array}$$

2. Trivial, by induction on i.
3. Trivial, by induction on i.
4. By induction on i.
   - $i < n$.
     Then $i \mathbin{div} n = 0$ and $i|_n = i$. Clearly, $i = 0{\cdot}n + i$.
   - $i \geq n$.
     Then $i \mathbin{div} n = S((i \dot{-} n) \mathbin{div} n)$ and $i|_n = (i \dot{-} n)|_n$. Hence,

$$\begin{array}{lll} & i & \\ = & (i \dot{-} n) + n & \text{(because } i \geq n) \\ = & ((i \dot{-} n) \mathbin{div} n){\cdot}n + (i \dot{-} n)|_n + n & \text{(by induction, } i, n > 0) \\ = & S((i \dot{-} n) \mathbin{div} n){\cdot}n + (i \dot{-} n)|_n & \\ = & (i \mathbin{div} n){\cdot}n + i|_n & \end{array}$$

5. Let $j \leq i \leq j + n$.
   Case 1: $i \mathit{\ div\ } 2n < j \mathit{\ div\ } 2n$.

$$\begin{array}{lll} & j - i & \\ = & (j \mathit{\ div\ } 2n)\cdot 2n + j|_{2n} - ((i \mathit{\ div\ } 2n)\cdot 2n + i|_{2n}) & \text{(Lem. 4.7.1.4)} \\ = & (j \mathit{\ div\ } 2n - i \mathit{\ div\ } 2n)\cdot 2n + (j|_{2n} - i|_{2n}) & \\ \geq & 2n + (j|_{2n} - i|_{2n}) & (i \mathit{\ div\ } 2n < j \mathit{\ div\ } 2n) \\ > & 2n - 2n & \text{(Lem. 4.7.1.2)} \\ = & 0 & \text{(contradict with } j \leq i) \end{array}$$

   Case 2: $i \mathit{\ div\ } 2n = j \mathit{\ div\ } 2n$. We need to show $j|_{2n} \leq i|_{2n} \leq j|_{2n} + n$.

$$\begin{array}{lll} & j \leq i \leq j + n & \\ = & (j \mathit{\ div\ } 2n)\cdot 2n + j|_{2n} \leq (i \mathit{\ div\ } 2n)\cdot 2n + i|_{2n} & \\ \leq & (j \mathit{\ div\ } 2n)\cdot 2n + j|_{2n} + n & \text{(Lem. 4.7.1.4)} \\ = & j|_{2n} \leq i|_{2n} \leq j|_{2n} + n & (i \mathit{\ div\ } 2n = j \mathit{\ div\ } 2n) \end{array}$$

   Case 3: $i \mathit{\ div\ } 2n = S(j \mathit{\ div\ } 2n)$. We need to show $i|_{2n} + n < j|_{2n}$.

$$\begin{array}{lll} & i \leq j + n & \\ = & (i \mathit{\ div\ } 2n)\cdot 2n + i|_{2n} & \\ \leq & (j \mathit{\ div\ } 2n)\cdot 2n + j|_{2n} + n & \text{(Lem. 4.7.1.4)} \\ = & (j \mathit{\ div\ } 2n)\cdot 2n + 2n + i|_{2n} & \\ \leq & (j \mathit{\ div\ } 2n)\cdot 2n + j|_{2n} + n & (i \mathit{\ div\ } 2n = S(j \mathit{\ div\ } 2n)) \\ = & i|_{2n} + n \leq j|_{2n} & \end{array}$$

   Case 4: $i \mathit{\ div\ } 2n > S(j \mathit{\ div\ } 2n)$.

$$\begin{array}{lll} & i - (j + n) & \\ = & (i \mathit{\ div\ } 2n)\cdot 2n + i|_{2n} & \\ & -((j \mathit{\ div\ } 2n)\cdot 2n + j|_{2n}) - n & \text{(Lem. 4.7.1.4)} \\ \geq & (j \mathit{\ div\ } 2n)\cdot 2n + 4n + i|_{2n} & \\ & -(j \mathit{\ div\ } 2n)\cdot 2n - j|_{2n} - n & (i \mathit{\ div\ } 2n > S(j \mathit{\ div\ } 2n)) \\ = & 3n + i|_{2n} - j|_{2n} & \\ > & 3n - 2n & \text{(Lem. 4.7.1.2)} \\ > & 0 & \text{(contradict with } i < j + n) \end{array}$$

6. By induction on i.

   - $i < n$. Then $i \mathit{\ div\ } n = 0$.
   - $i \geq n$.

$$\begin{array}{lll} & i \mathit{\ div\ } n & \\ = & S((i \dot{-} n) \mathit{\ div\ } n) & \\ \leq & S((j \dot{-} n) \mathit{\ div\ } n) & \text{(by induction, because } i \leq j, n > 0) \\ = & j \mathit{\ div\ } n & \text{(because } n \leq i \leq j) \end{array}$$

$\boxtimes$

Lemma 4.7.2 1. $test(i, remove(j, q)) = (test(i, q) \wedge i \neq j)$

2. $i \neq j \;\Rightarrow\; retrieve(i, remove(j, q)) = retrieve(i, q)$

3. $test(i, release(j, k, q)) = (test(i, q) \wedge \neg(j \leq i < k))$

4. $\neg(j \leq i < k) \;\Rightarrow\; retrieve(i, release(j, k, q)) = retrieve(i, q)$

5. $q \neq [] \;\Rightarrow\; test(max(q), q)$

Proof.

1. By induction on the structure of q.

   - $q = []$.
     $test(i, remove(j, [])) = test(i, []) = \mathsf{F} = test(i, []) \wedge i \neq j$.
   - $q = inb(d, k, q')$.
     Case 1: $j = k$.

     $$\begin{array}{lll} & test(i, remove(j, inb(d, k, q'))) & \\ = & test(i, remove(j, q')) & \\ = & test(i, q') \wedge i \neq j & \text{(by induction)} \\ = & test(i, inb(d, k, q')) \wedge i \neq j & \text{(because } j = k\text{)} \end{array}$$

     Case 2: $j \neq k$.
     Case 2.1: $i = k$. Then $i \neq j$.

     $$\begin{array}{lll} & test(i, remove(j, inb(d, k, q'))) & \\ = & test(i, inb(d, k, remove(j, q'))) & \\ = & \mathsf{T} & \\ = & test(i, inb(d, k, q')) \wedge i \neq j & \end{array}$$

     Case 2.2: $i \neq k$.

     $$\begin{array}{lll} & test(i, remove(j, inb(d, k, q'))) & \\ = & test(i, inb(d, k, remove(j, q'))) & \\ = & test(i, remove(j, q')) & \\ = & test(i, q') \wedge i \neq j & \text{(by induction)} \\ = & test(i, inb(d, k, q')) \wedge i \neq j & \end{array}$$

2. By induction on the structure of q.

   - $q = []$.
     Then $remove(j, []) = []$.
   - $q = inb(d, k, q')$.
     Case 1: $j = k$.

     $$\begin{array}{lll} & retrieve(i, remove(j, inb(d, k, q'))) & \\ = & retrieve(i, remove(j, q')) & \\ = & retrieve(i, q') & \text{(by induction)} \\ = & retrieve(i, inb(d, k, q')) & \end{array}$$

Case 2: $j \neq k$.
Case 2.1: $i = k$.

$$\begin{array}{rl} & \mathit{retrieve}(i, \mathit{remove}(j, \mathit{inb}(d, k, q'))) \\ = & \mathit{retrieve}(i, \mathit{inb}(d, k, \mathit{remove}(j, q'))) \\ = & d \\ = & \mathit{retrieve}(i, \mathit{inb}(d, k, q')) \end{array}$$

Case 2.2: $i \neq k$.

$$\begin{array}{rll} & \mathit{retrieve}(i, \mathit{remove}(j, \mathit{inb}(d, k, q'))) & \\ = & \mathit{retrieve}(i, \mathit{inb}(d, k, \mathit{remove}(j, q'))) & \\ = & \mathit{retrieve}(i, \mathit{remove}(j, q')) & \\ = & \mathit{retrieve}(i, q') & \text{(by induction)} \\ = & \mathit{retrieve}(i, \mathit{inb}(d, k, q')) & \end{array}$$

3. By induction on $k \dot{-} j$.

   - $j \geq k$.
     Then $\mathit{test}(i, \mathit{release}(j, k, q)) = \mathit{test}(i, q)$ and $\neg(j \leq i < k) = \mathsf{T}$.
   - $j < k$.

$$\begin{array}{rll} & \mathit{test}(i, \mathit{release}(j, k, q)) & \\ = & \mathit{test}(i, \mathit{release}(S(j), k, \mathit{remove}(j, q))) & \\ = & \mathit{test}(i, \mathit{remove}(j, q)) \wedge \neg(S(j) \leq i < k) & \text{(by induction)} \\ = & \mathit{test}(i, q) \wedge \neg(j \leq i < k) & \text{(Lem. 4.7.2.1)} \end{array}$$

4. By induction on $k \dot{-} j$.

   - $j \geq k$.
     Then $\mathit{retrieve}(i, \mathit{release}(j, k, q)) = \mathit{retrieve}(i, q)$.
   - $j < k$.
     Then $\neg(j \leq i < k)$ implies $i \neq j$. Hence,

$$\begin{array}{rll} & \mathit{retrieve}(i, \mathit{release}(j, k, q)) & \\ = & \mathit{retrieve}(i, \mathit{release}(S(j), k, \mathit{remove}(j, q))) & \\ = & \mathit{retrieve}(i, \mathit{remove}(j, q)) & \text{(by induction)} \\ = & \mathit{retrieve}(i, q) & \text{(Lem. 4.7.2.2, because } i \neq j) \end{array}$$

5. By induction on the structure of q.

   - $g = []$.
     This case is trivial.
   - $q = \mathit{inb}(d, k, q')$.
     By definition, $\mathit{max}(\mathit{inb}(d, k, q')) = \mathit{if}(k \geq \mathit{max}(q'), k, \mathit{max}(q'))$.
     Case 1: $k \geq \mathit{max}(q')$. Then $\mathit{max}(\mathit{inb}(d, k, q')) = k$.
     Clearly, $\mathit{test}(k, \mathit{inb}(d, k, q'))$.
     Case 2: $k < \mathit{max}(q')$. Then $\mathit{max}(\mathit{inb}(d, k, q')) = \mathit{max}(q')$.
     $\mathit{test}(\mathit{max}(q'), \mathit{inb}(d, k, q')) = \mathit{test}(\mathit{max}(q'), q')$. Hence, by induction, $\mathit{test}(\mathit{max}(q'), q')$.

⊠

Lemma 4.7.3 1. $test(k, q|_{2n}) \Rightarrow k = k|_{2n}$

2. $(\forall j{:}Nat(test(j,q) \Rightarrow i \leq j < i+n) \wedge i \leq k \leq i+n)$
   $\Rightarrow test(k,q) = test(k|_{2n}, q|_{2n})$

3. $(\forall j{:}Nat(test(j,q) \Rightarrow i \leq j < i+n) \wedge test(k,q))$
   $\Rightarrow retrieve(k,q) = retrieve(k|_{2n}, q|_{2n})$

4. $(\forall j{:}Nat(test(j,q) \Rightarrow i \leq j < i+n) \wedge i \leq k \leq i+n)$
   $\Rightarrow remove(k,q)|_{2n} = remove(k|_{2n}, q|_{2n})$

5. $(\forall j{:}Nat(test(j,q) \Rightarrow i \leq j < i+n) \wedge i \leq k \leq i+n)$
   $\Rightarrow release(i,k,q)|_{2n} = release|_{2n}(i,k,q|_{2n})$

6. $(\forall j{:}Nat(test(j,q) \Rightarrow i \leq j < i+n) \wedge i \leq k \leq i+n)$
   $\Rightarrow next\text{-}empty(k,q)|_{2n} = next\text{-}empty|_{2n}(k|_{2n}, q|_{2n})$

7. $i \leq k < i+n \Rightarrow in\text{-}window(i|_{2n}, k|_{2n}, (i+n)|_{2n})$

8. $in\text{-}window(i|_{2n}, k|_{2n}, (i+n)|_{2n})$
   $\Rightarrow k+n < i \vee i \leq k < i+n \vee k \geq i+2n$

9. $(\forall j{:}Nat(test(j,q) \Rightarrow i \leq j < i+n) \wedge test(k, q|_{2n}))$
   $\Rightarrow in\text{-}window(i|_{2n}, k, (i+n)|_{2n})$

Proof.

1. Trivial, by induction on the structure of q, using Lemma 4.7.1.2.

2. By induction on the structure of q.

   - $q = []$. Then $test(k, []) = \mathsf{F} = test(k|_{2n}, []|_{2n})$.
   - $q = inb(d, \ell, q')$.
     Let $test(j,q) \Rightarrow i \leq j < i+n$ and $i \leq k \leq i+n$.
     Case 1: $k|_{2n} = \ell|_{2n}$.
     $test(\ell, q)$, so $i \leq \ell < i+n$. In combination with $i \leq k \leq i+n$, $k|_{2n} = \ell|_{2n}$, Lemmas 4.7.1.4 and 4.7.1.5, this implies $k = \ell$. Hence, $test(k,q)$. Furthermore, $k|_{2n} = \ell|_{2n}$ implies $test(k|_{2n}, q|_{2n})$.
     Case 2: $k|_{2n} \neq \ell|_{2n}$. Then also $k \neq \ell$.
     $test(j,q') \Rightarrow test(j,q) \Rightarrow i \leq j < i+n$, so induction can be applied with respect to q'.

     $$\begin{array}{lll} & test(k, inb(d,\ell,q')) & \\ = & test(k,q') & \\ = & test(k|_{2n}, q'|_{2n}) & \text{(by induction)} \\ = & test(k|_{2n}, inb(d,\ell,q')|_{2n}) & \end{array}$$

3. By induction on the structure of q.

- $q = []$. Then $\mathit{test}(k, []) = \mathsf{F}$.
- $q = \mathit{inb}(d, \ell, q')$.
  Let $\mathit{test}(j, q) \Rightarrow i \leq j < i + n$ and $\mathit{test}(k, q)$.
  Case 1: $k = \ell$. Then also $k|_{2n} = \ell|_{2n}$.
  Hence, $\mathit{retrieve}(k, q) = d = \mathit{retrieve}(k|_{2n}, q|_{2n})$.
  Case 2: $k \neq \ell$.
  $\mathit{test}(j, q') \Rightarrow \mathit{test}(j, q) \Rightarrow i \leq j < i + n$, and $\mathit{test}(k, q)$ together with $k \neq \ell$ implies $\mathit{test}(k, q')$, so induction can be applied with respect to q'. $\mathit{test}(k, q)$ and $\mathit{test}(\ell, q)$, so $i \leq k < i + n$ and $i \leq \ell < i + n$. In combination with $k \neq \ell$, Lemmas 4.7.1.4 and 4.7.1.5, this implies $k|_{2n} \neq \ell|_{2n}$. Hence,

$$\begin{array}{lll} & \mathit{retrieve}(k, q) & \\ = & \mathit{retrieve}(k, q') & \\ = & \mathit{retrieve}(k|_{2n}, q'|_{2n}) & \text{(by induction)} \\ = & \mathit{retrieve}(k|_{2n}, q|_{2n}) & \end{array}$$

4. By induction on the structure of q.

- $q = []$.
  $\mathit{remove}(k, [])|_{2n} = [] = \mathit{remove}(k|_{2n}, []|_{2n})$.
- $q = \mathit{inb}(d, \ell, q')$.
  Let $\mathit{test}(j, q) \Rightarrow i \leq j < i + n$ and $i \leq k \leq i + n$.
  Case 1: $k = \ell$. Then also $k|_{2n} = \ell|_{2n}$.

$$\begin{array}{lll} & \mathit{remove}(k, q)|_{2n} & \\ = & \mathit{remove}(k, q')|_{2n} & \\ = & \mathit{remove}(k|_{2n}, q'|_{2n}) & \text{(by induction)} \\ = & \mathit{remove}(k|_{2n}, q|_{2n}) & \end{array}$$

  Case 2: $k \neq \ell$.
  $\mathit{test}(\ell, q)$, so $i \leq \ell < i + n$. In combination with $i \leq k \leq i + n$, $k \neq \ell$, Lemmas 4.7.1.4 and 4.7.1.5, this implies $k|_{2n} \neq \ell|_{2n}$. Hence,

$$\begin{array}{lll} & \mathit{remove}(k, q)|_{2n} & \\ = & \mathit{inb}(d, \ell, \mathit{remove}(k, q'))|_{2n} & \\ = & \mathit{inb}(d, \ell|_{2n}, \mathit{remove}(k, q')|_{2n}) & \\ = & \mathit{inb}(d, \ell|_{2n}, \mathit{remove}(k|_{2n}, q'|_{2n})) & \text{(by induction)} \\ = & \mathit{remove}(k|_{2n}, q|_{2n}) & \end{array}$$

5. By induction on $k \dot{-} i$. Let $\mathit{test}(j, q) \Rightarrow i \leq j < i + n$.

- $i = k$. Then also $i|_{2n} = k|_{2n}$.
  Hence, $\mathit{release}(i, k, q)|_{2n} = q|_{2n} = \mathit{release}|_{2n}(i, k, q|_{2n})$.

- $i < k \leq i + n$.
  By Lemmas 4.7.1.4 and 4.7.1.5, $i|_{2n} \neq k|_{2n}$. Hence,

$$\begin{array}{lll} & \mathit{release}(i,k,q)|_{2n} & \\ = & \mathit{release}(S(i),k,\mathit{remove}(i,q))|_{2n} & \\ = & \mathit{release}|_{2n}(S(i),k,\mathit{remove}(i,q)|_{2n}) & \text{(by induction)} \\ = & \mathit{release}|_{2n}(S(i),k,\mathit{remove}(i|_{2n},q|_{2n})) & \text{(Lem. 4.7.3.4)} \\ = & \mathit{release}|_{2n}(i,k,q|_{2n}) & \end{array}$$

6. By induction on $(i+n) \dot{-} k$. Let $\mathit{test}(j,q) \Rightarrow i \leq j < i+n$.

   - $k = i + n$.
     $\neg\mathit{test}(i+n,q)$, so by Lemma 4.7.3.2, $\neg\mathit{test}((i+n)|_{2n}, q|_{2n})$. Then by Lemma 4.7.1.2, $(i+n)|_{2n} < 2n$. Hence,

$$\begin{array}{ll} & \mathit{next\text{-}empty}(i+n,q)|_{2n} \\ = & (i+n)|_{2n} \\ = & \mathit{next\text{-}empty}((i+n)|_{2n}, q|_{2n}) \\ = & \mathit{next\text{-}empty}|_{2n}((i+n)|_{2n}, q|_{2n}) \end{array}$$

   - $i \leq k \leq i + n$.
     Case 1: $\neg\mathit{test}(k,q)$. By Lemma 4.7.3.2, also $\neg\mathit{test}(k|_{2n}, q|_{2n})$.
     By Lemma 4.7.1.2, $k|_{2n} < 2n$. Hence,

$$\begin{array}{ll} & \mathit{next\text{-}empty}(k,q)|_{2n} \\ = & k|_{2n} \\ = & \mathit{next\text{-}empty}(k|_{2n}, q|_{2n}) \\ = & \mathit{next\text{-}empty}|_{2n}(k|_{2n}, q|_{2n}) \end{array}$$

     Case 2: $\mathit{test}(k,q)$. By Lemma 4.7.3.2, also $\mathit{test}(k|_{2n}, q|_{2n})$.
     We prove $\mathit{next\text{-}empty}|_{2n}(k|_{2n}, q|_{2n}) = \mathit{next\text{-}empty}|_{2n}(S(k)|_{2n}, q|_{2n})$.
     Case 2.1: $k|_{2n} = 2n - 1$.
     By Lemma 4.7.4.3,

$$\begin{array}{ll} & \mathit{next\text{-}empty}(k|_{2n}, q|_{2n}) \\ = & \mathit{next\text{-}empty}(S(k|_{2n}), q|_{2n}) \\ = & \mathit{next\text{-}empty}(2n, q|_{2n}) \\ \geq & 2n \end{array}$$

     Hence,

$$\begin{array}{ll} & \mathit{next\text{-}empty}|_{2n}(k|_{2n}, q|_{2n}) \\ = & \mathit{next\text{-}empty}(0, q|_{2n}) \\ = & \mathit{next\text{-}empty}|_{2n}(S(k)|_{2n}, q|_{2n}) \end{array}$$

     Case 2.2: $k|_{2n} < 2n - 1$.

Using Lemma 4.7.1.1, we can derive $S(k)|_{2n} = S(k|_{2n})$. Since

$$\begin{array}{ll} & \mathit{next\text{-}empty}(k|_{2n}, q|_{2n}) \\ = & \mathit{next\text{-}empty}(S(k|_{2n}), q|_{2n}) \\ = & \mathit{next\text{-}empty}(S(k)|_{2n}, q|_{2n}) \end{array}$$

we have $\mathit{next\text{-}empty}|_{2n}(k|_{2n}, q|_{2n}) = \mathit{next\text{-}empty}|_{2n}(S(k)|_{2n}, q|_{2n})$. Concluding,

$$\begin{array}{ll} & \mathit{next\text{-}empty}(k, q)|_{2n} \\ = & \mathit{next\text{-}empty}(S(k), q)|_{2n} \\ = & \mathit{next\text{-}empty}|_{2n}(S(k)|_{2n}, q|_{2n}) \quad \text{(by induction)} \\ = & \mathit{next\text{-}empty}|_{2n}(k|_{2n}, q|_{2n}) \end{array}$$

7. Let $i \leq k < i + n$.

   Case 1: $S(i \mathit{\ div\ } 2n){\cdot}2n \leq k$.
   Then $S(i \mathit{\ div\ } 2n){\cdot}2n \leq k < i + n < S(i \mathit{\ div\ } 2n){\cdot}2n + n$ (by Lem. 4.7.1.4). Then by Lemmas 4.7.1.2, 4.7.1.5 and 4.7.1.6 it follows that $k \mathit{\ div\ } 2n = (i + n) \mathit{\ div\ } 2n = S(i \mathit{\ div\ } 2n)$. Hence, in view of Lemma 4.7.1.4, $k|_{2n} < (i + n)|_{2n} < i|_{2n}$.

   Case 2: $k < S(i \mathit{\ div\ } 2n){\cdot}2n \leq i + n$.
   Then $(i \mathit{\ div\ } 2n){\cdot}2n \leq i \leq k < (i \mathit{\ div\ } 2n){\cdot}2n + 2n$, so by Lemma 4.7.1.6 $k \mathit{\ div\ } 2n = i \mathit{\ div\ } 2n$. Furthermore, $S(i \mathit{\ div\ } 2n){\cdot}2n \leq i+n < S(i \mathit{\ div\ } 2n){\cdot}2n+n$, so $(i + n) \mathit{\ div\ } 2n = S(i \mathit{\ div\ } 2n)$. Hence, $(i + n)|_{2n} < i|_{2n} \leq k|_{2n}$.

   Case 3: $i + n < S(i \mathit{\ div\ } 2n){\cdot}2n$.
   Then $(i \mathit{\ div\ } 2n){\cdot}2n \leq i \leq k < i+n < (i \mathit{\ div\ } 2n){\cdot}2n+2n$. By Lemma 4.7.1.6, $k \mathit{\ div\ } 2n = (i + n) \mathit{\ div\ } 2n = i \mathit{\ div\ } 2n$. Hence, $i|_{2n} \leq k|_{2n} < (i + n)|_{2n}$.

   By definition,

$$\begin{array}{ll} & \mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i + n)|_{2n}) \\ = & i|_{2n} \leq k|_{2n} < (i + n)|_{2n} \vee \\ & (i + n)|_{2n} < i|_{2n} \leq k|_{2n} \vee \\ & k|_{2n} < (i + n)|_{2n} < i|_{2n} \end{array}$$

   so in all three cases we conclude $\mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i + n)|_{2n})$.

8. We prove

$$\begin{array}{l} (i + n \leq k < i + 2n \vee i \leq k + n < i + n) \\ \Rightarrow\ \neg \mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i + n)|_{2n}). \end{array}$$

   - $i + n \leq k < i + 2n$.
     Then $i \mathit{\ div\ } 2n \leq (i + n) \mathit{\ div\ } 2n \leq k \mathit{\ div\ } 2n \leq S(i \mathit{\ div\ } 2n)$. We distinguish three cases, in which we repeatedly apply Lemma 4.7.1.4.

     Case 1: $i \mathit{\ div\ } 2n = (i + n) \mathit{\ div\ } 2n = k \mathit{\ div\ } 2n$.
     Then $i < i+n$ yields $i|_{2n} < (i + n)|_{2n}$ and $i+n \leq k$ yields $(i + n)|_{2n} \leq k|_{2n}$.

CASE 2: $S(i \mathit{div} 2n) = S((i+n) \mathit{div} 2n) = k \mathit{div} 2n$.
Then $i < i+n$ yields $i|_{2n} < (i+n)|_{2n}$ and $k < i+2n$ yields $k|_{2n} < i|_{2n}$.
CASE 3: $S(i \mathit{div} 2n) = (i+n) \mathit{div} 2n = k \mathit{div} 2n$.
Then $i+n \leq k$ yields $(i+n)|_{2n} \leq k|_{2n}$ and $k < i+2n$ yields $k|_{2n} < i|_{2n}$.
In all three cases we can conclude $\neg \mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i+n)|_{2n})$.

- $i \leq k+n < i+n$.
  Then $i+n \leq k+2n < i+2n$, so by CASE 1,
  $\neg \mathit{in\text{-}window}(i|_{2n}, (k+2n)|_{2n}, (i+n)|_{2n})$.
  Hence, $\neg \mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i+n)|_{2n})$.

9. By induction on the structure of q.

   - $q = []$.
     This case follows from the fact that $\mathit{test}(k, []|_{2n}) = \mathsf{F}$.
   - $q = \mathit{inb}(d, \ell, q')$. Then $\mathit{test}(\ell, q)$, so $i \leq \ell < i+n$.
     Thus, by Lemma 4.7.3.7, $\mathit{in\text{-}window}(i|_{2n}, \ell|_{2n}, (i+n)|_{2n})$. Hence,

$$\begin{array}{ll} & \mathit{test}(k, \mathit{inb}(d, \ell, q')|_{2n}) \\ \Leftrightarrow & k = \ell|_{2n} \vee \mathit{test}(k, q'|_{2n}) \\ \Rightarrow & k = \ell|_{2n} \vee \mathit{in\text{-}window}(i|_{2n}, k, (i+n)|_{2n}) \\ \Leftrightarrow & \mathit{in\text{-}window}(i|_{2n}, k, (i+n)|_{2n}) \end{array}$$

⊠

Lemma 4.7.4 1. $\mathit{test}(i, q) \Rightarrow i \leq \mathit{max}(q)$

2. $i \leq j < \mathit{next\text{-}empty}(i, q) \Rightarrow \mathit{test}(j, q)$

3. $\mathit{next\text{-}empty}(i, q) \geq i$

4. $\mathit{next\text{-}empty}(i, \mathit{inb}(d, j, q)) \geq \mathit{next\text{-}empty}(i, q)$

5. $j \neq \mathit{next\text{-}empty}(i, q)$
   $\Rightarrow \mathit{next\text{-}empty}(i, \mathit{inb}(d, j, q)) = \mathit{next\text{-}empty}(i, q)$

6. $\mathit{next\text{-}empty}(i, \mathit{inb}(d, \mathit{next\text{-}empty}(i, q), q))$
   $= \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(i, q)), q)$

7. $\neg(i \leq j < \mathit{next\text{-}empty}(i, q))$
   $\Rightarrow \mathit{next\text{-}empty}(i, \mathit{remove}(j, q)) = \mathit{next\text{-}empty}(i, q)$

Proof.

1. By induction on the structure of q.

   - $q = []$.
     Then $\mathit{test}(i, []) = \mathsf{F}$.

- $q = inb(d, j, q')$.
  CASE 1: $i = j$. Then clearly $i \leq max(inb(d, j, q'))$.
  CASE 2: $i \neq j$. Then $test(i, inb(d, j, q'))$ implies $test(i, q')$, so

$$i \leq max(q') \text{ (by induction)} \leq max(inb(d, j, q')).$$

2. By induction on $j \dot{-} i$.

   - $i = j$.
     $\neg test(i, q)$ implies $next\text{-}empty(i, q) = i = j$.
   - $i < j$.
     CASE 1: $\neg test(i, q)$. Then $next\text{-}empty(i, q) = i < j$.
     CASE 2: $test(i, q)$.

$$\begin{array}{lll} & i < j < next\text{-}empty(i, q) & \\ \Leftrightarrow & S(i) \leq j < next\text{-}empty(S(i), q) & \\ \Rightarrow & test(j, q) & \text{(by induction)} \end{array}$$

3. By induction on $S(max(q)) \dot{-} i$.

   - $\neg test(i, q)$. (This includes the base case $S(max(q)) \leq i$.)
     Then $next\text{-}empty(i, q) = i$.
   - $test(i, q)$.
     By Lemma 4.7.4.1, $i \leq max(q)$, so $S(max(q)) \dot{-} S(i) < S(max(q)) \dot{-} i$. Hence, by induction, $next\text{-}empty(i, q) = next\text{-}empty(S(i), q) > i$.

4. By induction on $S(max(q)) \dot{-} i$.

   - $\neg test(i, q)$.
     Then $next\text{-}empty(i, inb(d, j, q)) \geq i$ (Lem. 4.7.4.3)$=next\text{-}empty(i, q)$.
   - $test(i, q)$. Then also $test(i, inb(d, j, q))$.
     By Lemma 4.7.4.1, $i \leq max(q)$, so $S(max(q)) \dot{-} S(i) < S(max(q)) \dot{-} i$. Hence,

$$\begin{array}{lll} & next\text{-}empty(i, inb(d, j, q)) & \\ = & next\text{-}empty(S(i), inb(d, j, q)) & \\ \geq & next\text{-}empty(S(i), q) & \text{(by induction)} \\ = & next\text{-}empty(i, q) & \end{array}$$

5. By induction on $S(max(q)) \dot{-} i$. Let $j \neq next\text{-}empty(i, q)$.

   - $\neg test(i, q)$.
     Then $next\text{-}empty(i, q) = i$. This implies $j \neq i$, so $\neg test(i, inb(d, j, q))$. Hence, $next\text{-}empty(i, inb(d, j, q)) = i$.

- $test(i, q)$. Then also $test(i, inb(d, j, q))$.
  By Lemma 4.7.4.1, $i \leq max(q)$, so $S(max(q)) \dot{-} S(i) < S(max(q)) \dot{-} i$. Furthermore, $test(i, q)$ implies $j \neq next\text{-}empty(S(i), q)$. Hence,

$$\begin{array}{lll} & next\text{-}empty(i, inb(d, j, q)) & \\ = & next\text{-}empty(S(i), inb(d, j, q)) & \\ = & next\text{-}empty(S(i), q) & \text{(by induction)} \\ = & next\text{-}empty(i, q) & \end{array}$$

6. By induction on $S(max(q)) \dot{-} i$.

   - $\neg test(i, q)$.
     Then $next\text{-}empty(i, q) = i$. By Lemma 4.7.4.3, $next\text{-}empty(S(i), q) \neq i$. Hence,

$$\begin{array}{lll} & next\text{-}empty(i, inb(d, next\text{-}empty(i, q), q)) & \\ = & next\text{-}empty(i, inb(d, i, q)) & \\ = & next\text{-}empty(S(i), inb(d, i, q)) & \\ = & next\text{-}empty(S(i), q) & \text{(Lem. 4.7.4.5)} \\ = & next\text{-}empty(S(next\text{-}empty(i, q)), q) & \end{array}$$

   - $test(i, q)$.
     By Lemma 4.7.4.1, $i \leq max(q)$, so the induction hypothesis can be applied with respect to $S(i)$.

$$\begin{array}{lll} & next\text{-}empty(i, inb(d, next\text{-}empty(i, q), q)) & \\ = & next\text{-}empty(S(i), inb(d, next\text{-}empty(S(i), q), q)) & \\ = & next\text{-}empty(S(next\text{-}empty(S(i), q)), q) & \text{(by induction)} \\ = & next\text{-}empty(S(next\text{-}empty(i, q)), q) & \end{array}$$

7. We apply induction on $S(max(q)) \dot{-} i$.

   - $\neg test(i, q)$.
     Then, by Lemma 4.7.2.1, $\neg test(i, remove(j, q))$. Hence, $next\text{-}empty(i, remove(j, q)) = i = next\text{-}empty(i, q)$.
   - $test(i, q)$.
     Let $\neg(i \leq j < next\text{-}empty(i, q))$. $test(i, q)$ implies $\neg(S(i) \leq j < next\text{-}empty(S(i), q))$. Furthermore, by Lemma 4.7.4.1, $i \leq max(q)$, so the induction hypothesis can be applied with respect to $S(i)$. Since $next\text{-}empty(i, q) = next\text{-}empty(S(i), q) \geq S(i)$ (Lem. 4.7.4.3), $\neg(i \leq j < next\text{-}empty(i, q))$ implies $j \neq i$. Then, by Lemma 4.7.2.1, $test(i, remove(j, q))$. Hence,

$$\begin{array}{lll} & next\text{-}empty(i, remove(j, q)) & \\ = & next\text{-}empty(S(i), remove(j, q)) & \\ = & next\text{-}empty(S(i), q) & \text{(by induction)} \\ = & next\text{-}empty(i, q) & \end{array}$$

⊠

Lemma 4.7.5 1. $\mathit{length}(g) = \mathit{length}(g|_{2n})$

2. $i < \mathit{length}(g) \;\Rightarrow\; \mathit{return\text{-}seq}(i, g)|_{2n} = \mathit{return\text{-}seq}(i, g|_{2n})$
3. $i < \mathit{length}(g) \;\Rightarrow\; \mathit{return\text{-}dat}(i, g) = \mathit{return\text{-}dat}(i, g|_{2n})$
4. $i < \mathit{length}(g) \;\Rightarrow\; \mathit{delete}(i, g)|_{2n} = \mathit{delete}(i, g|_{2n})$
5. $\mathit{length}(g) > 0 \;\Rightarrow\; \mathit{last\text{-}dat}(g) = \mathit{return\text{-}dat}(\mathit{length}(g) - 1, g)$
6. $\mathit{length}(g) > 0 \;\Rightarrow\; \mathit{last\text{-}seq}(g) = \mathit{return\text{-}seq}(\mathit{length}(g) - 1, g)$
7. $\mathit{length}(g) > 0 \;\Rightarrow\; \mathit{delete\text{-}last}(g) = \mathit{delete}(\mathit{length}(g) - 1, g)$
8. $(i < \mathit{length}(g) \wedge \mathit{member}(d, j, \mathit{delete}(i, g))) \;\Rightarrow\; \mathit{member}(d, j, g)$
9. $i < \mathit{length}(g) \;\Rightarrow\; \mathit{length}(\mathit{delete}(i, g)) = \mathit{length}(g) - 1$
10. $i < \mathit{length}(g) \;\Rightarrow\; \mathit{member}(\mathit{return\text{-}dat}(i, g), \mathit{return\text{-}seq}(i, g), g)$
11. $(i < \mathit{length}(g) - 1 \wedge j < \mathit{length}(g))$
    $\Rightarrow\; \mathit{return\text{-}seq}(i, \mathit{delete}(j, g)) = \mathit{if}\,(i < j, \mathit{return\text{-}seq}(i, g), \mathit{return\text{-}seq}(i + 1, g))$
12. $\mathit{member}(d, i, g)$
    $\Rightarrow\; \exists j{:}\mathit{Nat}\ (j < \mathit{length}(g) \wedge \mathit{return\text{-}seq}(j, g) = i \wedge \mathit{return\text{-}dat}(j, g) = d)$

Proof. We prove Lemma 4.7.5.11 by induction on the structure of g. The other lemmas are straightforward, by induction on g, and left to reader.

- $g = []^K$. Then $\mathit{length}(g) = 0$. This case is trivial.
- $g = \mathit{inm}(e, k, g_1)$.

  Let $i < \mathit{length}(g_1)$ and $j \leq \mathit{length}(g_1)$.

  Case 1: $j = 0$. Then $\neg(i < j)$ and

  $$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete}(j, g)) \\ = & \mathit{return\text{-}seq}(i, g_1) \\ = & \mathit{return\text{-}seq}(i + 1, g) \end{array}$$

  Case 2: $j > 0$.

  If $i = 0$, then $i < j$ and $\mathit{return\text{-}seq}(i, \mathit{delete}(j, g)) = k = \mathit{return\text{-}seq}(i, g)$.

  If $i > 0$, then

  $$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete}(j, g)) \\ = & \mathit{return\text{-}seq}(i - 1, \mathit{delete}(j - 1, g_1)) \\ = & \mathit{if}\,(i - 1 < j - 1, \mathit{return\text{-}seq}(i - 1, g_1), \mathit{return\text{-}seq}(i, g_1)) \text{ (by induction)} \\ = & \mathit{if}\,(i < j, \mathit{return\text{-}seq}(i, g), \mathit{return\text{-}seq}(i + 1, g)) \end{array}$$

⊠

Lemma 4.7.6 1. $length(g') = length(g'|_{2n})$

2. $i < length(g') \;\Rightarrow\; return\text{-}seq(i, g')|_{2n} = return\text{-}seq(i, g'|_{2n})$
3. $i < length(g') \;\Rightarrow\; delete(i, g')|_{2n} = delete(i, g'|_{2n})$
4. $length(g') > 0 \;\Rightarrow\; last\text{-}seq(g') = return\text{-}seq(length(g') - 1, g')$
5. $length(g') > 0 \;\Rightarrow\; delete\text{-}last(g') = delete(length(g') - 1, g')$
6. $(i < length(g') \wedge member(j, delete(i, g'))) \;\Rightarrow\; member(j, g')$
7. $i < length(g') \;\Rightarrow\; length(delete(i, g')) = length(g') - 1$
8. $i < length(g') \;\Rightarrow\; member(return\text{-}seq(i, g'), g')$
9. $(i < length(g') - 1 \wedge j < length(g'))$
   $\Rightarrow\; return\text{-}seq(i, delete(j, g')) = if(i<j, return\text{-}seq(i, g'), return\text{-}seq(i+1, g'))$

Proof. The proof of Lemma 4.7.6.9 is similar to the proof of Lemma 4.7.5.11. The other lemmas are straightforward by induction on g'. ⊠

Lemma 4.7.7 1. $(\lambda{+\!\!+}\lambda'){+\!\!+}\lambda'' = \lambda{+\!\!+}(\lambda'{+\!\!+}\lambda'')$

2. $length(\lambda{+\!\!+}\lambda') = length(\lambda) + length(\lambda')$
3. $append(d, \lambda{+\!\!+}\lambda') = \lambda{+\!\!+}append(d, \lambda')$
4. $length(q[i..j\rangle) = j \dot{-} i$
5. $i \leq k \leq j \;\Rightarrow\; q[i..j\rangle = q[i..k\rangle{+\!\!+}q[k..j\rangle$
6. $i \leq j \;\Rightarrow\; append(d, q[i..j\rangle) = inb(d, j, q)[i..S(j)\rangle$
7. $test(k, q) \;\Rightarrow\; inb(retrieve(k, q), k, q)[i..j\rangle = q[i..j\rangle$
8. $\neg(i \leq k < j) \;\Rightarrow\; remove(k, q)[i..j\rangle = q[i..j\rangle$
9. $\ell \leq i \;\Rightarrow\; release(k, \ell, q)[i..j\rangle = q[i..j\rangle$

Proof. The proofs of these nine facts are straightforward and left to the reader. We restrict to a listing of the induction bases.

1. By induction on the length of λ.
2. By induction on the length of λ.
3. By induction on the length of λ.
4. By induction on $j \dot{-} i$.
5. By induction on $k \dot{-} i$.

6. By induction on $j \dot{-} i$.
7. By induction on $j \dot{-} i$.
8. By induction on $j \dot{-} i$, together with Lemmas 4.7.2.1 and 4.7.2.2.
9. By induction on $j \dot{-} i$, together with Lemmas 4.7.2.3 and 4.7.2.4.

$\boxtimes$

4.7.2 Invariants

Invariants of a system are properties of data that are satisfied throughout the reachable state space of the system. Lemma 4.7.8 collects 27 invariants of $\mathbf{N}_{nonmod}$ that are needed in the correctness proof. Occurrences of variables i, j:*Nat* and d, e:Δ in an invariant are always implicitly universally quantified at the outside of the invariant.

Lemma 4.7.8 The invariants hold for $\mathbf{N}_{nonmod}(\ell, m, q, \ell', q', g, p, g', p')$.

1. $p \leq \mathit{length}(g)$
2. $p' \leq \mathit{length}(g')$
3. $\mathit{member}(i, g') \;\Rightarrow\; i \leq \mathit{next\text{-}empty}(\ell', q')$
4. $\ell \leq \mathit{next\text{-}empty}(\ell', q')$
5. $i < j < \mathit{length}(g') \;\Rightarrow\; \mathit{return\text{-}seq}(i, g') \geq \mathit{return\text{-}seq}(j, g')$
6. $\mathit{member}(i, g') \;\Rightarrow\; \ell \leq i$
7. $\mathit{test}(i, q) \;\Rightarrow\; i < m$
8. $\mathit{member}(d, i, g) \;\Rightarrow\; i < m$
9. $\mathit{test}(i, q') \;\Rightarrow\; i < m$
10. $\mathit{test}(i, q') \;\Rightarrow\; \ell' \leq i < \ell' + n$
11. $\ell' \leq m$
12. $\mathit{next\text{-}empty}(\ell', q') \leq m$
13. $\mathit{next\text{-}empty}(\ell', q') \leq \ell' + n$
14. $\ell \leq m$
15. $\mathit{test}(i, q) \;\Rightarrow\; \ell \leq i$
16. $\ell \leq i < m \;\Rightarrow\; \mathit{test}(i, q)$
17. $\ell \leq \ell' + n$

18. $m \leq \ell + n$
19. $i \leq j < \mathit{length}(g) \;\Rightarrow\; \mathit{return\text{-}seq}(i,g) + n > \mathit{return\text{-}seq}(j,g)$
20. $(\mathit{member}(d,i,g) \wedge \mathit{test}(j,q')) \;\Rightarrow\; i + n > j$
21. $\mathit{member}(d,i,g) \;\Rightarrow\; i + n \geq \ell'$
22. $\mathit{member}(d,i,g) \;\Rightarrow\; i + n \geq \mathit{next\text{-}empty}(\ell', q')$
23. $(\mathit{member}(d,i,g) \wedge \mathit{test}(i,q)) \;\Rightarrow\; \mathit{retrieve}(i,q) = d$
24. $(\mathit{test}(i,q) \wedge \mathit{test}(i,q')) \;\Rightarrow\; \mathit{retrieve}(i,q) = \mathit{retrieve}(i,q')$
25. $(\mathit{member}(d,i,g) \wedge \mathit{member}(e,i,g)) \;\Rightarrow\; d = e$
26. $(\mathit{member}(d,i,g) \wedge \mathit{test}(i,q')) \;\Rightarrow\; \mathit{retrieve}(i,q') = d$
27. $(\ell \leq i \leq m \wedge j \leq \mathit{next\text{-}empty}(i,q')) \;\Rightarrow\; q[i..j\rangle = q'[i..j\rangle$

Proof. It is easy to verify that all invariants hold in the initial state (where the buffers and mediums are empty, the parameters in the natural numbers equal zero). In case 1-27 we show that the invariant is preserved by each of the summands A-K in the specification of $\mathbf{N}_{nonmod}$. For each of these invariants we only treat the summands in which one or more values of parameters occurring in the invariant are updated. In each of these proof obligations, we list the new values of these parameters together with those conjuncts in the condition of the summand under consideration that play a role in the proof.

1. $p \leq \mathit{length}(g)$.
   Summands B, C, D, E and F need to be checked. F is the same as E.

   B: $g := \mathit{inm}(\mathit{retrieve}(k,q),k,g)$, $p := p + 1$;
   $\mathit{length}(\mathit{inm}(\mathit{retrieve}(k,q),k,g)) = \mathit{length}(g) + 1 \geq p + 1$.

   C: $g := \mathit{delete}(k,g)$, $p := p - 1$; under condition $k < p$;
   Since $k < p \leq \mathit{length}(g)$, by Lemma 4.7.5.9,
   $\mathit{length}(\mathit{delete}(k,g)) = \mathit{length}(g) - 1 \geq p - 1$.

   D: $p := p - 1$; under condition $p > 0$;
   $p - 1 < p \leq \mathit{length}(g)$.

   E: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
   Since $0 < \mathit{length}(g)$, by Lemmas 4.7.5.7 and 4.7.5.9,
   $\mathit{length}(\mathit{delete\text{-}last}(g)) = \mathit{length}(g) - 1 \geq p$.

2. $p' \leq \mathit{length}(g')$.
   Summands H, I, J and K need to be checked.

   H: $g' := \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')$, $p' := p' + 1$;
   $\mathit{length}(\mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) = \mathit{length}(g') + 1 \geq p' + 1$.

I: $g' := \mathit{delete}(k, g')$, $p' := p' - 1$; under condition $k < p'$;
Since $k < p' \leq \mathit{length}(g')$, by Lemma 4.7.6.7,
$\mathit{length}(\mathit{delete}(k, g')) = \mathit{length}(g') - 1 \geq p' - 1$.

J: $p' := p' - 1$; under condition $p' > 0$;
$p' - 1 < p' \leq \mathit{length}(g')$.

K: $g' := \mathit{delete\text{-}last}(g')$; under condition $p' < \mathit{length}(g')$;
Since $0 < \mathit{length}(g')$, by Lemmas 4.7.6.5 and 4.7.6.7,
$\mathit{length}(\mathit{delete\text{-}last}(g')) = \mathit{length}(g') - 1 \geq p$.

3. $\mathit{member}(i, g') \;\Rightarrow\; i \leq \mathit{next\text{-}empty}(\ell', q')$.
Summands E, G, H, I and K need to be checked.

E: $q' := \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')$;
Let $\mathit{member}(i, g')$. Then

$$\begin{array}{ll} & i \\ \leq & \mathit{next\text{-}empty}(\ell', q') \\ \leq & \mathit{next\text{-}empty}(\ell', \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) \quad \text{(Lem. 4.7.4.4)} \end{array}$$

G: $\ell' := S(\ell')$, $q' := \mathit{remove}(\ell', q')$; under condition $\mathit{test}(\ell', q')$;
Let $\mathit{member}(i, g')$. Then,

$$\begin{array}{ll} & i \\ \leq & \mathit{next\text{-}empty}(\ell', q') \\ = & \mathit{next\text{-}empty}(S(\ell'), q') \\ = & \mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q')) \quad \text{(Lem. 4.7.4.7)} \end{array}$$

H: $g' := \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')$;
Let $\mathit{member}(i, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g'))$.
Case 1: $i = \mathit{next\text{-}empty}(\ell', q')$.
$\mathit{next\text{-}empty}(\ell', q') \leq \mathit{next\text{-}empty}(\ell', q')$.

Case 2: $i \neq \mathit{next\text{-}empty}(\ell', q')$.
$\mathit{member}(i, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) = \mathit{member}(i, g') \Rightarrow$
$i \leq \mathit{next\text{-}empty}(\ell', q')$.

I: $g' := \mathit{delete}(k, g')$; under condition $k < p'$;
Let $\mathit{member}(i, \mathit{delete}(k, g'))$. By Invariant 4.7.8.2, $k < p' \leq \mathit{length}(g')$. By Lemma 4.7.6.6, $\mathit{member}(i, \mathit{delete}(k, g')) \Rightarrow \mathit{member}(i, g') \Rightarrow$
$i \leq \mathit{next\text{-}empty}(\ell', q')$.

K: $g' := \mathit{delete\text{-}last}(g')$; under condition $p' < \mathit{length}(g')$;
Let $\mathit{member}(i, \mathit{delete\text{-}last}(g'))$. By Lemmas 4.7.6.5 and 4.7.6.6,
$\mathit{member}(i, \mathit{delete\text{-}last}(g')) \Rightarrow \mathit{member}(i, g') \Rightarrow i \leq \mathit{next\text{-}empty}(\ell', q')$.

4. $\ell \leq \mathit{next\text{-}empty}(\ell', q')$.
Summands E, G and K need to be checked.

E: $q' := \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')$;
$\ell \leq \mathit{next\text{-}empty}(\ell', q') \leq \mathit{next\text{-}empty}(\ell', \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q'))$
(Lem. 4.7.4.4).

G: $\ell' := S(\ell')$, $q' := \mathit{remove}(\ell', q')$; under condition $\mathit{test}(\ell', q')$;

$$\begin{array}{ll} & \ell \\ \leq & \mathit{next\text{-}empty}(\ell', q') \\ = & \mathit{next\text{-}empty}(S(\ell'), q') \\ = & \mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q')) \quad \text{(Lem. 4.7.4.7)} \end{array}$$

K: $\ell := \mathit{last\text{-}seq}(g')$; under condition $p' < \mathit{length}(g')$.
$0 < \mathit{length}(g')$, so by Lemmas 4.7.6.4 and 4.7.6.8, $\mathit{member}(\mathit{last\text{-}seq}(g'), g)$.
Hence, by Invariant 4.7.8.3, $\mathit{last\text{-}seq}(g') \leq \mathit{next\text{-}empty}(\ell', q')$.

5. $i < j < \mathit{length}(g') \Rightarrow \mathit{return\text{-}seq}(i, g') \geq \mathit{return\text{-}seq}(j, g')$.
Summands H, I and K need to be checked.

H: $g' := \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')$;
Let $i < j < \mathit{length}(g') + 1$.
Case 1: $i > 0$. Then $i - 1 < j - 1 < \mathit{length}(g')$. So

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) \\ = & \mathit{return\text{-}seq}(i - 1, g') \\ \geq & \mathit{return\text{-}seq}(j - 1, g') \\ = & \mathit{return\text{-}seq}(j, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) \end{array}$$

Case 2: $i = 0$.
Since $j > 0$, $\mathit{return\text{-}seq}(j, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) = \mathit{return\text{-}seq}(j - 1, g')$. Since $j - 1 < \mathit{length}(g')$, by Lemma 4.7.6.8, $\mathit{member}(\mathit{return\text{-}seq}(j - 1, g'), g')$. By Invariant 4.7.8.3,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(j - 1, g') \\ \leq & \mathit{next\text{-}empty}(\ell', q') \\ = & \mathit{return\text{-}seq}(i, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) \quad \text{(because } i = 0\text{)} \end{array}$$

I: $g' := \mathit{delete}(k, g')$; under condition $k < p'$;
Let $i < j < \mathit{length}(\mathit{delete}(k, g'))$. By Invariant 4.7.8.2, $k < p' \leq \mathit{length}(g')$. So by Lemma 4.7.6.7, $\mathit{length}(\mathit{delete}(k, g')) = \mathit{length}(g') - 1$. Since $i < i + 1 \leq j < j + 1 < \mathit{length}(g')$, $\mathit{return\text{-}seq}(i, g') \geq \mathit{return\text{-}seq}(i + 1, g') \geq \mathit{return\text{-}seq}(j, g') \geq \mathit{return\text{-}seq}(j + 1, g')$. So by Lemma 4.7.6.9,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete}(k, g')) \\ \geq & \mathit{return\text{-}seq}(i + 1, g') \\ \geq & \mathit{return\text{-}seq}(j, g') \\ \geq & \mathit{return\text{-}seq}(j, \mathit{delete}(k, g')) \end{array}$$

K: $g' := \mathit{delete\text{-}last}(g')$; under condition $p' < \mathit{length}(g')$;
Let $i < j < \mathit{length}(\mathit{delete\text{-}last}(g'))$. Since $0 < \mathit{length}(g')$, Lemmas 4.7.6.5

and 4.7.6.7 imply $\mathit{length}(\mathit{delete\text{-}last}(g')) = \mathit{length}(g') - 1$. Hence, by Lemmas 4.7.6.5 and 4.7.6.9,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete\text{-}last}(g')) \\ = & \mathit{return\text{-}seq}(i, g') \\ \geq & \mathit{return\text{-}seq}(j, g') \\ = & \mathit{return\text{-}seq}(i, \mathit{delete\text{-}last}(g')) \end{array}$$

6. $\mathit{member}(i, g') \Rightarrow \ell \leq i$.
   Summands H, I and K need to be checked.

   H: $g' := \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')$;
   Let $\mathit{member}(i, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g'))$.
   Case 1: $i = \mathit{next\text{-}empty}(\ell', q')$.
   By Invariant 4.7.8.4, $\ell \leq \mathit{next\text{-}empty}(\ell', q')$.

   Case 2: $i \neq \mathit{next\text{-}empty}(\ell', q')$.
   $\mathit{member}(i, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) \Rightarrow \mathit{member}(i, g') \Rightarrow \ell \leq i$.

   I: $g' := \mathit{delete}(k, g')$; under condition $k < p'$;
   By Invariant 4.7.8.2, $k < p' \leq \mathit{length}(g')$. So by Lemma 4.7.6.6,
   $\mathit{member}(i, \mathit{delete}(k, g')) \Rightarrow \mathit{member}(i, g') \Rightarrow \ell \leq i$.

   K: $g' := \mathit{delete\text{-}last}(g')$; under condition $p' < \mathit{length}(g')$;
   Since $0 < \mathit{length}(g')$, by Lemmas 4.7.6.5 and 4.7.6.6,
   $\mathit{member}(i, \mathit{delete\text{-}last}(g')) \Rightarrow \mathit{member}(i, g') \Rightarrow \ell \leq i$.

7. $\mathit{test}(i, q) \Rightarrow i < m$.
   Summands A and K need to be checked.

   A: $m := S(m)$, $q := \mathit{inb}(d, m, q)$;
   $\mathit{test}(i, \mathit{inb}(d, m, q)) \Leftrightarrow (i = m \vee \mathit{test}(i, q)) \Rightarrow (i = m \vee i < m) \Leftrightarrow i < S(m)$.

   K: $q := \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)$;
   $\mathit{test}(i, \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)) \Rightarrow \mathit{test}(i, q)$ (Lem. 4.7.2.3) $\Rightarrow i < m$.

8. $\mathit{member}(d, i, g) \Rightarrow i < m$.
   Summands A, B, C, E and F need to be checked. F is the same as E.

   A: $m := S(m)$;
   $\mathit{member}(d, i, g) \Rightarrow i < m < S(m)$.

   B: $g := \mathit{inm}(\mathit{retrieve}(k, q), k, g)$; under condition $\mathit{test}(k, q)$;
   Let $\mathit{member}(d, i, \mathit{inm}(\mathit{retrieve}(k, q), k, g))$.
   Case 1: $i = k$. Since $\mathit{test}(k, q)$, by Invariant 4.7.8.7, $k < m$.
   Case 2: $i \neq k$.
   $\mathit{member}(d, i, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) = \mathit{member}(d, i, g) \Rightarrow i < m$.

   C: $g := \mathit{delete}(k, g)$; under condition $k < p$;
   By Invariant 4.7.8.1, $k < p \leq \mathit{length}(g)$. So by Lemma 4.7.5.8,
   $\mathit{member}(d, i, \mathit{delete}(k, g)) \Rightarrow \mathit{member}(d, i, g) \Rightarrow i < m$.

   E: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
   Since $0 < \mathit{length}(g)$, by Lemmas 4.7.5.7 and 4.7.5.8,
   $\mathit{member}(d, i, \mathit{delete\text{-}last}(g)) \Rightarrow \mathit{member}(d, i, g) \Rightarrow i < m$.

9. $\mathit{test}(i, q') \Rightarrow i < m$.
   Summands A, E and G need to be checked.

   A: $m := S(m)$;
   $\mathit{test}(i, q') \Rightarrow i < m < S(m)$.

   E: $q' := \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')$; under condition $p < \mathit{length}(g)$;
   Since $0 < \mathit{length}(g)$, by Lemmas 4.7.5.5, 4.7.5.6 and 4.7.5.10, $\mathit{member}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), g)$. By Invariant 4.7.8.8, $\mathit{last\text{-}seq}(g) < m$. Hence,

$$\begin{array}{ll} & \mathit{test}(i, \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) \\ \Leftrightarrow & (i = \mathit{last\text{-}seq}(g) \vee \mathit{test}(i, q')) \\ \Rightarrow & (i = \mathit{last\text{-}seq}(g) \vee i < m) \\ \Leftrightarrow & i < m \end{array}$$

   G: $q' := \mathit{remove}(\ell', q')$;
   $\mathit{test}(i, \mathit{remove}(\ell', q')) \Rightarrow \mathit{test}(i, q')$ (Lem. 4.7.2.1) $\Rightarrow i < m$.

10. $\mathit{test}(i, q') \Rightarrow \ell' \leq i < \ell' + n$.
    Summands E and G need to be checked.

    E: $q' := \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')$; under condition $\ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n$;

$$\begin{array}{ll} & \mathit{test}(i, \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) \\ \Leftrightarrow & (i = \mathit{last\text{-}seq}(g) \vee \mathit{test}(i, q')) \\ \Rightarrow & (i = \mathit{last\text{-}seq}(g) \vee \ell' \leq i < \ell' + n) \\ \Leftrightarrow & \ell' \leq i < \ell' + n \end{array}$$

    G: $\ell' := S(\ell')$, $q' := \mathit{remove}(\ell', q')$;

$$\begin{array}{lll} & \mathit{test}(i, \mathit{remove}(\ell', q')) & \\ \Leftrightarrow & (\mathit{test}(i, q') \wedge i \neq \ell') & \text{(Lem. 4.7.2.1)} \\ \Rightarrow & (\ell' \leq i < \ell' + n \wedge i \neq \ell') & \\ \Rightarrow & S(\ell') \leq i < S(\ell') + n & \end{array}$$

11. $\ell' \leq m$.
    Summands A and G need to be checked.

    A: $m := S(m)$;
    $\ell' \leq m < S(m)$.

    G: $\ell' := S(\ell')$; under condition $\mathit{test}(\ell', q')$;
    By Invariant 4.7.8.9, $\mathit{test}(\ell', q') \Rightarrow \ell' < m$. Hence, $S(\ell') \leq m$.

12. $\mathit{next\text{-}empty}(\ell', q') \leq m$.
    By Invariant 4.7.8.11, $\ell' \leq m$. By Invariant 4.7.8.9, $\neg \mathit{test}(m, q')$. Hence, by Lemma 4.7.4.2, $\mathit{next\text{-}empty}(\ell', q') \leq m$.

13. $\mathit{next\text{-}empty}(\ell', q') \leq \ell' + n$.
    By Invariant 4.7.8.10, $\neg \mathit{test}(\ell' + n, q')$. Hence, by Lemma 4.7.4.2, $\mathit{next\text{-}empty}(\ell', q') \leq \ell' + n$.

14. $\ell \leq m$.
    By Invariants 4.7.8.4 and 4.7.8.12.

15. $\mathit{test}(i, q) \Rightarrow \ell \leq i$.
    Summands A and K need to be checked.

    A: $q := \mathit{inb}(d, m, q)$;
    By Invariant 4.7.8.14, $\ell \leq m$. Hence,

$$\begin{array}{ll} & \mathit{test}(i, \mathit{inb}(d, m, q)) \\ \Leftrightarrow & (i = m \vee \mathit{test}(i, q)) \\ \Rightarrow & (i = m \vee \ell \leq i) \\ \Leftrightarrow & \ell \leq i \end{array}$$

    K: $\ell := \mathit{last\text{-}seq}(g')$, $q := \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)$;

$$\begin{array}{lll} & \mathit{test}(i, \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)) & \\ \Leftrightarrow & (\mathit{test}(i, q) \wedge \neg(\ell \leq i < \mathit{last\text{-}seq}(g'))) & \text{(Lem. 4.7.2.3)} \\ \Rightarrow & (\ell \leq i \wedge \neg(\ell \leq i < \mathit{last\text{-}seq}(g'))) & \\ \Rightarrow & \mathit{last\text{-}seq}(g') \leq i & \end{array}$$

16. $\ell \leq i < m \Rightarrow \mathit{test}(i, q)$.
    Summands A and K need to be checked.

    A: $m := S(m)$, $q := \mathit{inb}(d, m, q)$;

$$\begin{array}{ll} & \ell \leq i < S(m) \\ \Rightarrow & (i = m \vee \ell \leq i < m) \\ \Rightarrow & (i = m \vee \mathit{test}(i, q)) \\ \Leftrightarrow & \mathit{test}(i, \mathit{inb}(d, m, q)) \end{array}$$

    K: $\ell := \mathit{last\text{-}seq}(g')$, $q := \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)$; under condition $p' < \mathit{length}(g')$;
    Since $0 < \mathit{length}(g')$, by Lemmas 4.7.6.4 and 4.7.6.8, $\mathit{member}(\mathit{last\text{-}seq}(g'), g')$. Then by Invariant 4.7.8.6, $\ell \leq \mathit{last\text{-}seq}(g')$. So,

$$\begin{array}{lll} & \mathit{last\text{-}seq}(g') \leq i < m & \\ \Leftrightarrow & (\ell \leq i < m \wedge \neg(\ell \leq i < \mathit{last\text{-}seq}(g'))) & \\ \Rightarrow & (\mathit{test}(i, q) \wedge \neg(\ell \leq i < \mathit{last\text{-}seq}(g'))) & \\ \Leftrightarrow & \mathit{test}(i, \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)) & \text{(Lem. 4.7.2.3)} \end{array}$$

17. $\ell \leq \ell' + n$.
    By Invariants 4.7.8.4 and 4.7.8.13.

18. $m \leq \ell + n$.
    Summands A and K need to be checked.

    A: $m := S(m)$; under condition $m < \ell + n$;
    Then $S(m) \leq \ell + n$.

K: $\ell := \mathit{last\text{-}seq}(g')$; under condition $p' < \mathit{length}(g')$;
Since $0 < \mathit{length}(g')$, by Lemmas 4.7.6.4 and 4.7.6.8,
$\mathit{member}(\mathit{last\text{-}seq}(g'), g')$. Then by Invariant 4.7.8.6, $\ell \leq \mathit{last\text{-}seq}(g')$.
Hence, $m \leq \ell + n \leq \mathit{last\text{-}seq}(g') + n$.

19. $i \leq j < \mathit{length}(g) \;\Rightarrow\; \mathit{return\text{-}seq}(i, g) + n > \mathit{return\text{-}seq}(j, g)$.
Summands B, C, E and F need to be checked. F is the same as E.

B: $g := \mathit{inm}(\mathit{retrieve}(k, q), k, g)$; under condition $\mathit{test}(k, q)$;
Case 1: $i > 0$. Let $i \leq j < \mathit{length}(g) + 1$.

$$\begin{array}{ll} & \mathit{return\text{-}seq}(j, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) \\ = & \mathit{return\text{-}seq}(j - 1, g) \\ < & \mathit{return\text{-}seq}(i - 1, g) + n \\ = & \mathit{return\text{-}seq}(i, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) + n \end{array}$$

Case 2: $i = 0$.
Case 2.1: $j = 0$. This case is trivial.

Case 2.2: $j \neq 0$.
Lemma 4.7.5.10 yields $\mathit{member}(\mathit{return\text{-}dat}(j - 1, g), \mathit{return\text{-}seq}(j - 1, g), g)$.
By Invariant 4.7.8.8, $\mathit{return\text{-}seq}(j - 1, g) < m$. By Invariant 4.7.8.15, $\mathit{test}(k, q) \Rightarrow \ell \leq k$.

$$\begin{array}{lll} & \mathit{return\text{-}seq}(j, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) & \\ = & \mathit{return\text{-}seq}(j - 1, g) & \\ < & m & \\ \leq & \ell + n & \text{(Lem. 4.7.8.18)} \\ \leq & k + n & \\ = & \mathit{return\text{-}seq}(i, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) + n & \text{(because } i = 0\text{)} \end{array}$$

C: $g := \mathit{delete}(k, g)$; under condition $k < p$;
Let $i \leq j < \mathit{length}(\mathit{delete}(k, g))$. By Invariant 4.7.8.1, $k < p \leq \mathit{length}(g)$.
By Lemma 4.7.5.9, $\mathit{length}(\mathit{delete}(k, g)) = \mathit{length}(g) - 1$.

Case 1: $k \leq i$.
Since $i + 1 \leq j + 1 < \mathit{length}(g)$, by Lemma 4.7.5.11,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete}(k, g)) + n \\ = & \mathit{return\text{-}seq}(i + 1, g) + n \\ > & \mathit{return\text{-}seq}(j + 1, g) \\ = & \mathit{return\text{-}seq}(j, \mathit{delete}(k, g)) \end{array}$$

Case 2: $i < k \leq j$.
Since $i < j + 1 < \mathit{length}(g)$, by Lemma 4.7.5.11,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete}(k, g)) + n \\ = & \mathit{return\text{-}seq}(i, g) + n \\ > & \mathit{return\text{-}seq}(j + 1, g) \\ = & \mathit{return\text{-}seq}(j, \mathit{delete}(k, g)) \end{array}$$

CASE 3: $j < k$.
Since $i \leq j < \mathit{length}(g)$, by Lemma 4.7.5.11,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete}(k, g)) + n \\ = & \mathit{return\text{-}seq}(i, g) + n \\ > & \mathit{return\text{-}seq}(j, g) \\ = & \mathit{return\text{-}seq}(j, \mathit{delete}(k, g)) \end{array}$$

E: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
Let $i \leq j < \mathit{length}(\mathit{delete\text{-}last}(g))$. By Lemmas 4.7.5.6 and 4.7.5.9, $0 < \mathit{length}(g)$ implies $\mathit{length}(\mathit{delete\text{-}last}(g)) = \mathit{length}(g) - 1$. Since $i \leq j < \mathit{length}(g)$, by Lemma 4.7.5.11,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete\text{-}last}(g)) + n \\ = & \mathit{return\text{-}seq}(i, g) + n \\ > & \mathit{return\text{-}seq}(j, g) \\ = & \mathit{return\text{-}seq}(j, \mathit{delete\text{-}last}(g)) \end{array}$$

20. $(\mathit{member}(d, i, g) \wedge \mathit{test}(j, q')) \Rightarrow i + n > j$.
Summands B, C, E, F and G need to be checked.

B: $g := \mathit{inm}(\mathit{retrieve}(k, q), k, g)$; under condition $\mathit{test}(k, q)$;
Let $\mathit{member}(d, i, \mathit{inm}(\mathit{retrieve}(k, q), k, g))$ and $\mathit{test}(j, q')$.
CASE 1: $i = k$.
By Invariant 4.7.8.15, $\mathit{test}(k, q)$ yields $\ell \leq k$, and by Invariant 4.7.8.9, $\mathit{test}(j, q')$ yields $j < m$. Hence, $k + n \geq \ell + n \geq m$ (Inv. 4.7.8.18) $> j$.

CASE 2: $i \neq k$.
$\mathit{member}(d, i, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) = \mathit{member}(d, i, g)$. Hence, $i + n > j$.

C: $g := \mathit{delete}(k, g)$; under condition $k < p$;
Let $\mathit{member}(d, i, \mathit{delete}(k, g))$ and $\mathit{test}(j, q')$. By Invariant 4.7.8.1, $k < p \leq \mathit{length}(g)$. In view of Lemma 4.7.5.8, $\mathit{member}(d, i, \mathit{delete}(k, g)) \Rightarrow \mathit{member}(d, i, g)$. Hence, $i + n > j$.

E: $q' := \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')$, $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$ and $\ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n$.
Let $\mathit{member}(d, i, \mathit{delete\text{-}last}(g))$ and $\mathit{test}(j, \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q'))$.
Since $0 < \mathit{length}(g)$, by Lemmas 4.7.5.7 and 4.7.5.8,
$\mathit{member}(d, i, \mathit{delete\text{-}last}(g)) \Rightarrow \mathit{member}(d, i, g)$.

CASE 1: $j = \mathit{last\text{-}seq}(g)$.
CASE 1.1: $i = \mathit{last\text{-}seq}(g)$. This case is trivial.

CASE 1.2: $i \neq \mathit{last\text{-}seq}(g)$.
Since $0 < \mathit{length}(g)$, by Lemma 4.7.5.6, $\mathit{last\text{-}seq}(g) = \mathit{return\text{-}seq}(\mathit{length}(g) - 1, g)$. Since $\mathit{member}(d, i, g)$, by Lemma 4.7.5.12, there exists a k such that $k < \mathit{length}(g)$ and $\mathit{return\text{-}seq}(k, g) = i$. By Invariant 4.7.8.19,
$i + n > \mathit{return\text{-}seq}(\mathit{length}(g) - 1, g) = \mathit{last\text{-}seq}(g)$.

CASE 2: $j \neq \mathit{last\text{-}seq}(g)$.
$\mathit{test}(j, \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) = \mathit{test}(j, q')$. Hence, $i + n > j$.

F: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
Let $\mathit{member}(d, i, \mathit{delete\text{-}last}(g))$ and $\mathit{test}(j, q')$. Since $0 < \mathit{length}(g)$, by Lemmas 4.7.5.7 and 4.7.5.8, $\mathit{member}(d, i, \mathit{delete\text{-}last}(g)) \Rightarrow \mathit{member}(d, i, g)$. Hence, $i + n > j$.

G: $q' := \mathit{remove}(\ell', q')$;
Let $\mathit{member}(d, i, g)$ and $\mathit{test}(j, \mathit{remove}(\ell', q'))$. By Lemma 4.7.2.1, $\mathit{test}(j, \mathit{remove}(\ell', q')) \Rightarrow \mathit{test}(j, q')$. Hence, $i + n > j$.

21. $\mathit{member}(d, i, g) \;\Rightarrow\; i + n \geq \ell'$.
Summands B, C, E, F and G need to be checked. F is the same as E.

B: $g := \mathit{inm}(\mathit{retrieve}(k, q), k, g)$; under condition $\mathit{test}(k, q)$;
Let $\mathit{member}(d, i, \mathit{inm}(\mathit{retrieve}(k, q), k, g))$.

CASE 1: $i = k$.
By Invariant 4.7.8.15, $\mathit{test}(k, q)$ yields $\ell \leq k$. Hence, $k + n \geq \ell + n \geq m$ (Inv. 4.7.8.18) $\geq \ell'$ (Inv. 4.7.8.11).

CASE 2: $i \neq k$.
$\mathit{member}(d, i, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) = \mathit{member}(d, i, g) \Rightarrow i + n \geq \ell'$.

C: $g := \mathit{delete}(k, g)$; under condition $k < p$;
Let $\mathit{member}(d, i, \mathit{delete}(k, g))$. By Invariant 4.7.8.1, $k < p \leq \mathit{length}(g)$. By Lemma 4.7.5.8, we have
$\mathit{member}(d, i, \mathit{delete}(k, g)) \Rightarrow \mathit{member}(d, i, g) \Rightarrow i + n \geq \ell'$.

E: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
Let $\mathit{member}(d, i, \mathit{delete\text{-}last}(g))$. Since $0 < \mathit{length}(g)$, by Lemmas 4.7.5.7 and 4.7.5.8, we have
$\mathit{member}(d, i, \mathit{delete\text{-}last}(g)) \Rightarrow \mathit{member}(d, i, g) \Rightarrow i + n \geq \ell'$.

G: $\ell' = S(\ell')$; under condition $\mathit{test}(\ell', q')$;
Let $\mathit{member}(d, i, g)$. By Invariant 4.7.8.20, $\mathit{test}(\ell', q')$ implies $i + n > \ell'$. Hence, $i + n \geq S(\ell')$.

22. $\mathit{member}(d, i, g) \;\Rightarrow\; i + n \geq \mathit{next\text{-}empty}(\ell', q')$.
We distinguish two cases.
CASE 1: $q' = []$. Then $\mathit{next\text{-}empty}(\ell', q') = \ell'$.
By Invariant 4.7.8.21, $\mathit{member}(d, i, g) \Rightarrow i + n \geq \ell'$.

CASE 2: $q' \neq []$.
By Lemma 4.7.2.5, $\mathit{test}(\mathit{max}(q'), q')$. So Invariant 4.7.8.20 yields $\mathit{member}(d, i, g) \Rightarrow i + n > \mathit{max}(q')$. By Lemmas 4.7.4.1 and 4.7.4.2, $\mathit{next\text{-}empty}(\ell', q') \leq \mathit{max}(q') + 1$. Hence, $\mathit{member}(d, i, g) \Rightarrow i + n \geq \mathit{next\text{-}empty}(\ell', q')$.

23. $(\mathit{member}(d, i, g) \wedge \mathit{test}(i, q)) \;\Rightarrow\; \mathit{retrieve}(i, q) = d$.
Summands A, B, C, E, F and K need to be checked. F is the same as E.

A: $q := \mathit{inb}(e, m, q)$;
By Invariant 4.7.8.8, $\mathit{member}(d, i, g) \Rightarrow i < m$.
So $\mathit{retrieve}(i, \mathit{inb}(e, m, q)) = \mathit{retrieve}(i, q) = d$.

B: $g := \mathit{inm}(\mathit{retrieve}(k, q), k, g)$;
Let $\mathit{member}(d, i, \mathit{inm}(\mathit{retrieve}(k, q), k, g))$ and $\mathit{test}(i, q)$.
Case 1: $d = \mathit{retrieve}(k, q)$ and $i = k$. This case is trivial.

Case 2: Otherwise. $\mathit{member}(d, i, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) = \mathit{member}(d, i, g)$.
Since $\mathit{test}(i, q)$, $\mathit{retrieve}(i, q) = d$.

C: $g := \mathit{delete}(k, g)$; under condition $k < p$;
Let $\mathit{member}(d, i, \mathit{delete}(k, g))$ and $\mathit{test}(i, q)$. By Invariant 4.7.8.1, $k < p \leq \mathit{length}(g)$. Then by Lemma 4.7.5.8, $\mathit{member}(d, i, \mathit{delete}(k, g)) \Rightarrow \mathit{member}(d, i, g)$. Since $\mathit{test}(i, q)$, $\mathit{retrieve}(i, q) = d$.

E: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
Let $\mathit{member}(d, i, \mathit{delete\text{-}last}(g))$ and $\mathit{test}(i, q)$. Since $0 < \mathit{length}(g)$, by Lemmas 4.7.5.7 and 4.7.5.8, $\mathit{member}(d, i, \mathit{delete\text{-}last}(g)) \Rightarrow \mathit{member}(d, i, g)$.
Since $\mathit{test}(i, q)$, $\mathit{retrieve}(i, q) = d$.

K: $q := \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)$;
Let $\mathit{member}(d, i, \mathit{delete\text{-}last}(g))$ and $\mathit{test}(i, \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q))$. By Lemma 4.7.2.3, $\mathit{test}(i, q)$ and $\neg(\ell \leq i < \mathit{last\text{-}seq}(g'))$. By Lemma 4.7.2.4, $\mathit{retrieve}(i, \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)) = \mathit{retrieve}(i, q) = d$.

24. $(\mathit{test}(i, q) \wedge \mathit{test}(i, q')) \Rightarrow \mathit{retrieve}(i, q) = \mathit{retrieve}(i, q')$.
Summands A, E, G and K must be checked.

A: $q := \mathit{inb}(d, m, q)$;
By Invariant 4.7.8.9, $\mathit{test}(i, q')$ implies $i \neq m$. So

$$\begin{array}{ll} & \mathit{test}(i, \mathit{inb}(d, m, q)) \wedge \mathit{test}(i, q') \\ \Leftrightarrow & \mathit{test}(i, q) \wedge \mathit{test}(i, q') \\ \Rightarrow & \mathit{retrieve}(i, \mathit{inb}(d, m, q)) = \mathit{retrieve}(i, q) = \mathit{retrieve}(i, q') \end{array}$$

E: $q' := \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')$; under condition $p < \mathit{length}(g)$;
Let $\mathit{test}(i, q)$ and $\mathit{test}(i, \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q'))$.

Case 1: $i \neq \mathit{last\text{-}seq}(g)$.

$$\begin{array}{ll} & \mathit{test}(i, q) \wedge \mathit{test}(i, \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) \\ \Rightarrow & \mathit{test}(i, q) \wedge \mathit{test}(i, q') \\ \Rightarrow & \mathit{retrieve}(i, q) = \mathit{retrieve}(i, q') \\ & = \mathit{retrieve}(i, \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) \end{array}$$

Case 2: $i = \mathit{last\text{-}seq}(g)$.
Since $0 < \mathit{length}(g)$, by Lemmas 4.7.5.5, 4.7.5.6 and 4.7.5.10, $\mathit{member}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), g)$. Since $\mathit{test}(\mathit{last\text{-}seq}(g), q)$,

$$\begin{array}{lll} & \mathit{retrieve}(\mathit{last\text{-}seq}(g), q) & \\ = & \mathit{last\text{-}dat}(g) & \text{(Inv. 4.7.8.23)} \\ = & \mathit{retrieve}(\mathit{last\text{-}dat}(g), \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) & \end{array}$$

G: $q' := \mathit{remove}(\ell', q')$;

$$\begin{array}{lll} & \mathit{test}(i,q) \wedge \mathit{test}(i, \mathit{remove}(\ell', q')) & \\ \Leftrightarrow & \mathit{test}(i,q) \wedge \mathit{test}(i,q') \wedge i \neq \ell' & \text{(Lem. 4.7.2.1)} \\ \Rightarrow & \mathit{retrieve}(i,q) = \mathit{retrieve}(i,q') & \\ & = \mathit{retrieve}(i, \mathit{remove}(\ell', q')) & \text{(Lem. 4.7.2.2)} \end{array}$$

K: $q := \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)$;

$$\begin{array}{lll} & \mathit{test}(i, \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)) \wedge \mathit{test}(i,q') & \\ \Leftrightarrow & \mathit{test}(i,q) \wedge \mathit{test}(i,q') \wedge \neg(\ell \leq i < \mathit{last\text{-}seq}(g')) & \text{(Lem. 4.7.2.3)} \\ \Rightarrow & \mathit{retrieve}(i,q') = \mathit{retrieve}(i,q) & \\ & = \mathit{retrieve}(i, \mathit{release}(\ell, h', q)) & \text{(Lem. 4.7.2.4)} \end{array}$$

25. $(\mathit{member}(d,i,g) \wedge \mathit{member}(e,i,g)) \Rightarrow d = e$.
Summands B, C, E and F need to be checked. F is the same as E.

B: $g := \mathit{inm}(\mathit{retrieve}(k,q), k, g)$; under condition $\mathit{test}(k,q)$;
Let $\mathit{member}(d,i,\mathit{inm}(\mathit{retrieve}(k,q),k,g))$ and
$\mathit{member}(e,i,\mathit{inm}(\mathit{retrieve}(k,q),k,g))$.

Case 1: $i = k$.
By Invariant 4.7.8.23, $\mathit{test}(k,q)$ implies $d = \mathit{retrieve}(k,q) = e$.

Case 2: $i \neq k$.
$\mathit{member}(d,i,\mathit{inm}(\mathit{retrieve}(k,q),k,g)) \Rightarrow \mathit{member}(d,i,g)$ and
$\mathit{member}(e,i,\mathit{inm}(\mathit{retrieve}(k,q),k,g)) \Rightarrow \mathit{member}(e,i,g)$. Hence, $d = e$.

C: $g := \mathit{delete}(k,g)$; under condition $k < p$;
By Invariant 4.7.8.1, $k < p \leq \mathit{length}(g)$. By Lemma 4.7.5.8,

$$\begin{array}{ll} & \mathit{member}(d,i,\mathit{delete}(k,g)) \wedge \mathit{member}(e,i,\mathit{delete}(k,g)) \\ \Rightarrow & \mathit{member}(d,i,g) \wedge \mathit{member}(e,i,g) \\ \Rightarrow & d = e \end{array}$$

E: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
Since $0 < \mathit{length}(g)$, by Lemmas 4.7.5.7 and 4.7.5.8,

$$\begin{array}{ll} & \mathit{member}(d,i,\mathit{delete\text{-}last}(g)) \wedge \mathit{member}(e,i,\mathit{delete\text{-}last}(g)) \\ \Rightarrow & \mathit{member}(d,i,g) \wedge \mathit{member}(e,i,g) \\ \Rightarrow & d = e \end{array}$$

26. $(\mathit{member}(d,i,g) \wedge \mathit{test}(i,q')) \Rightarrow \mathit{retrieve}(i,q') = d$.
Summands B, C, E, F and G need to be checked.

B: $g := \mathit{inm}(\mathit{retrieve}(k,q), k, g)$; under condition $\mathit{test}(k,q)$;
Let $\mathit{member}(d,i,\mathit{inm}(\mathit{retrieve}(k,q),k,g))$ and $\mathit{test}(i,q')$.

Case 1: $d = \mathit{retrieve}(k,q)$ and $i = k$.
Since $\mathit{test}(k,q)$ and $\mathit{test}(k,q')$, by Invariant 4.7.8.24, $\mathit{retrieve}(k,q') = d = \mathit{retrieve}(k,q)$.

Case 2: Otherwise.
$member(d, i, inm(retrieve(k, q), k, g)) = member(d, i, g)$. Since $test(i, q')$, $retrieve(i, q') = d$.

C: $g := delete(k, g)$; under condition $k < p$;
Let $member(d, i, delete(k, g))$ and $test(i, q')$. By Invariant 4.7.8.1, $k < p \leq length(g)$. By Lemma 4.7.5.8, $member(d, i, delete(k, g)) \Rightarrow member(d, i, g)$. Since $test(i, q')$, $retrieve(i, q') = d$.

E: $q' := inb(last\text{-}dat(g), last\text{-}seq(g), q')$, $g := delete\text{-}last(g)$; under condition $p < length(g)$;
Let $member(d, i, delete\text{-}last(g))$ and $test(i, inb(last\text{-}dat(g), last\text{-}seq(g), q'))$. Since $0 < length(g)$, by Lemmas 4.7.5.7 and 4.7.5.8,
$member(d, i, delete\text{-}last(g)) \Rightarrow member(d, i, g)$.

Case 1: $i = last\text{-}seq(g)$.
Since $0 < length(g)$, by Lemmas 4.7.5.5, 4.7.5.6 and 4.7.5.10, we have $member(last\text{-}dat(g), last\text{-}seq(g), g)$.
Since $member(d, last\text{-}seq(g), delete\text{-}last(g))$, by Invariant 4.7.8.25,
$d = last\text{-}dat(g) = retrieve(last\text{-}seq(g), inb(last\text{-}dat(g), last\text{-}seq(g), q'))$.

Case 2: $i \neq last\text{-}seq(g)$.
Then $test(i, inb(last\text{-}dat(g), last\text{-}seq(g), q')) \Rightarrow test(i, q')$.
By $member(d, i, g)$, $retrieve(i, q') = d$.

F: $g := delete\text{-}last(g)$; under condition $p < length(g)$;
Let $member(d, i, delete\text{-}last(g))$ and $test(i, q')$. Since $0 < length(g)$, by Lemmas 4.7.5.7 and 4.7.5.8, $member(d, i, delete\text{-}last(g)) \Rightarrow member(d, i, g)$. Since $test(i, q')$, $retrieve(i, q') = d$

G: $q' := remove(\ell', q')$;
By Lemma 4.7.2.1, $test(i, remove(\ell', q'))$ implies $test(i, q')$ and $i \neq \ell'$. Hence, $member(d, i, g) \Rightarrow retrieve(i, remove(\ell', q')) = retrieve(i, q')$ (Lem. 4.7.2.2)$= d$.

27. $(\ell \leq i \leq m \wedge j \leq next\text{-}empty(i, q')) \Rightarrow q[i..j\rangle = q'[i..j\rangle$.
Let $\ell \leq i \leq m$ and $j \leq next\text{-}empty(i, q'))$. We apply induction on $j \dot{-} i$.

If $i \geq j$, then $q[i..j\rangle = \langle\rangle = q'[i..j\rangle$.

Let $i < j$.
Case 1: $i = m$.
By Invariant 4.7.8.9, $j \leq next\text{-}empty(i, q') = m$. Hence, $q[i..j\rangle = \langle\rangle = q'[i..j\rangle$.

Case 2: $\ell \leq i < m$.
Then by Invariant 4.7.8.16, $test(i, q)$. Furthermore, by Lemma 4.7.4.2,

$i < j \leq \mathit{next\text{-}empty}(i, q')$ implies $\mathit{test}(i, q')$. Hence,

$$\begin{array}{ll} & q[i..j\rangle \\ = & \mathit{inb}(\mathit{retrieve}(i, q), q[S(i)..j\rangle) \\ = & \mathit{inb}(\mathit{retrieve}(i, q), q'[S(i)..j\rangle) \quad \text{(by induction)} \\ = & \mathit{inb}(\mathit{retrieve}(i, q'), q'[S(i)..j\rangle) \quad \text{(Inv. 4.7.8.24)} \\ = & q'[i..j\rangle. \end{array}$$

$\boxtimes$

4.8 Correctness of $\mathbf{N}_{mod}$

In Section 4.8.1, we prove Theorem 4.6.3, which states that $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ are strongly bisimilar. Next, in Section 4.8.2 we prove that $\mathbf{N}_{nonmod}$ behaves like a FIFO queue of size $2n$. Theorem 4.5.1 is proved in Section 4.8.3.

4.8.1 Equality of $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$

In this section we present a proof of Theorem 4.6.3. It suffices to prove that for all $\ell, m, \ell'{:}\mathit{Nat}$, $q, q'{:}\mathit{Buf}$, $g{:}\mathit{MedK}$ and $g'{:}\mathit{MedL}$,

$$\begin{array}{ll} & \mathbf{N}_{mod}(\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, q'|_{2n}, g|_{2n}, p, g'|_{2n}, p') \\ \underline{\leftrightarrow} & \mathbf{N}_{nonmod}(\ell, m, q, \ell', q', g, p, g', p') \end{array}$$

Proof. We show that $\mathbf{N}_{mod}(\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, q'|_{2n}, g|_{2n}, p, g'|_{2n}, p')$ is a solution for the defining equation of $\mathbf{N}_{nonmod}(\ell, m, q, \ell', q', g, p, g', p')$. Hence, we must derive the following equation.[2]

$$\begin{array}{lll} & \mathbf{N}_{mod}(\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, q'|_{2n}, g|_{2n}, p, g'|_{2n}, p') & \\ = & \sum_{d:\Delta} r_{\mathrm{A}}(d) \cdot \mathbf{N}_{mod}(m{:=}S(m)|_{2n}, q{:=}\mathit{inb}(d, m, q)|_{2n}) & \\ & \quad \lhd\ m < \ell + n \ \rhd\ \delta & (A) \\ + & \sum_{k:\mathit{Nat}} c \cdot \mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k, q), k, g)|_{2n}, p{:=}p + 1) & \\ & \quad \lhd\ \mathit{test}(k, q) \ \rhd\ \delta & (B) \\ + & \sum_{k:\mathit{Nat}} j \cdot \mathbf{N}_{mod}(g{:=}\ \mathit{delete}(k, g)|_{2n}, p{:=}p - 1) \lhd\ k < p \ \rhd\ \delta & (C) \\ + & j \cdot \mathbf{N}_{mod}(p{:=}p - 1) \lhd\ p > 0 \ \rhd\ \delta & (D) \\ + & c \cdot \mathbf{N}_{mod}(q'{:=}\mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')|_{2n}, g{:=}\mathit{delete\text{-}last}(g)|_{2n}) & \\ & \quad \lhd\ p < \mathit{length}(g) \wedge (\ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n) \ \rhd\ \delta & (E) \\ + & c \cdot \mathbf{N}_{mod}(g{:=}\mathit{delete\text{-}last}(g)|_{2n}) & \\ & \quad \lhd\ p < \mathit{length}(g) \wedge \neg(\ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n) \ \rhd\ \delta & (F) \end{array}$$

[2] By abuse of notation, we use the parameters ℓ, m, q, ℓ', q', g, g' in an ambiguous way. For example, m refers both to the second parameter of $\mathbf{N}_{mod}$ and to the value of this parameter.

$$
\begin{array}{lll}
 & + & s_{\mathrm{D}}(\mathit{retrieve}(\ell',q'))\cdot\mathbf{N}_{mod}(\ell':=S(\ell')|_{2n}, q':=\mathit{remove}(\ell',q')|_{2n}) \\
 & & \qquad \lhd\ \mathit{test}(\ell',q')\ \rhd\ \delta \qquad (G) \\
 & + & c\cdot\mathbf{N}_{mod}(g':=\mathit{inm}(\mathit{next\text{-}empty}(\ell',q'),g')|_{2n}, p':=p'+1) \qquad (H) \\
 & + & \sum_{k:Nat} j\cdot\mathbf{N}_{mod}(g':=\mathit{delete}(k,g')|_{2n}, p':=p'-1) \lhd\ k<p'\ \rhd\ \delta \qquad (I) \\
 & + & j\cdot\mathbf{N}_{mod}(p':=p'-1) \lhd\ p'>0\ \rhd\ \delta \qquad (J) \\
 & + & c\cdot\mathbf{N}_{mod}(\ell:=\mathit{last\text{-}seq}(g')|_{2n}, q:=\mathit{release}(\ell,\mathit{last\text{-}seq}(g'),q)|_{2n}, \\
 & & \qquad g':=\mathit{delete\text{-}last}(g')|_{2n}) \lhd\ p'<\mathit{length}(g')\ \rhd\ \delta \qquad (K)
\end{array}
$$

In order to prove this, we instantiate the parameters in the defining equation of $\mathbf{N}_{mod}$ with $\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, g|_{2n}, p, g'|_{2n}, p'$.

$$
\begin{array}{lll}
 & & \mathbf{N}_{mod}(\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, q'|_{2n}, g|_{2n}, p, g'|_{2n}, p') \\
 & = & \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{N}_{mod}(m:=S(m|_{2n})|_{2n}, q:=\mathit{inb}(d,m|_{2n},q|_{2n})) \\
 & & \qquad \lhd\ \mathit{in\text{-}window}(\ell|_{2n}, m|_{2n}, (\ell|_{2n}+n)|_{2n})\ \rhd\ \delta \qquad (A) \\
 & + & \sum_{k:Nat} c\cdot\mathbf{N}_{mod}(g:=\mathit{inm}(\mathit{retrieve}(k,q|_{2n}),k,g|_{2n}), p:=p+1) \\
 & & \qquad \lhd\ \mathit{test}(k,q|_{2n})\ \rhd\ \delta \qquad (B) \\
 & + & \sum_{k:Nat} j\cdot\mathbf{N}_{mod}(g:=\mathit{delete}(k,g|_{2n}), p:=p-1) \lhd\ k<p\ \rhd\ \delta \qquad (C) \\
 & + & j\cdot\mathbf{N}_{mod}(p:=p-1) \lhd\ p>0\ \rhd\ \delta \qquad (D) \\
 & + & c\cdot\mathbf{N}_{mod}(q':=\mathit{inb}(\mathit{last\text{-}dat}(g|_{2n}),\mathit{last\text{-}seq}(g|_{2n}),q'|_{2n}), \\
 & & \qquad g:=\mathit{delete\text{-}last}(g|_{2n})) \lhd\ p<\mathit{length}(g|_{2n})\wedge \\
 & & \qquad \mathit{in\text{-}window}(\ell'|_{2n}, \mathit{last\text{-}seq}(g|_{2n}), (\ell'|_{2n}+n)|_{2n})\ \rhd\ \delta \qquad (E) \\
 & + & c\cdot\mathbf{N}_{mod}(g:=\mathit{delete\text{-}last}(g|_{2n})) \lhd\ p<\mathit{length}(g|_{2n})\wedge \\
 & & \qquad \neg\mathit{in\text{-}window}(\ell'|_{2n}, \mathit{last\text{-}seq}(g|_{2n}), (\ell'|_{2n}+n)|_{2n})\ \rhd\ \delta \qquad (F) \\
 & + & s_{\mathrm{D}}(\mathit{retrieve}(\ell'|_{2n},q'|_{2n}))\cdot\mathbf{N}_{mod}(\ell':=S(\ell'|_{2n})|_{2n}, q':=\mathit{remove}(\ell'|_{2n},q'|_{2n})) \\
 & & \qquad \lhd\ \mathit{test}(\ell'|_{2n},q'|_{2n})\ \rhd\ \delta \qquad (G) \\
 & + & c\cdot\mathbf{N}_{mod}(g':=\mathit{inm}(\mathit{next\text{-}empty}|_{2n}(\ell'|_{2n},q'|_{2n}),g'|_{2n}), p':=p'+1) \qquad (H) \\
 & + & \sum_{k:Nat} j\cdot\mathbf{N}_{mod}(g':=\mathit{delete}(k,g'|_{2n}), p':=p'-1) \lhd\ k<p'\ \rhd\ \delta \qquad (I) \\
 & + & j\cdot\mathbf{N}_{mod}(p':=p'-1) \lhd\ p'>0\ \rhd\ \delta \qquad (J) \\
 & + & c\cdot\mathbf{N}_{mod}(\ell:=\mathit{last\text{-}seq}(g'|_{2n})|_{2n}, q:=\mathit{release}|_{2n}(\ell|_{2n},\mathit{last\text{-}seq}(g'|_{2n})|_{2n},q|_{2n}), \\
 & & \qquad g':=\mathit{delete\text{-}last}(g'|_{2n})) \lhd\ p'<\mathit{length}(g'|_{2n})\ \rhd\ \delta \qquad (K)
\end{array}
$$

In order to equate the eleven summands in both specifications, we obtain the following proof obligations. Cases for summands that are syntactically the same are omitted.

A

- $m < \ell + n \Leftrightarrow \mathit{in\text{-}window}(\ell|_{2n}, m|_{2n}, (\ell|_{2n} + n)|_{2n})$.

$$\begin{array}{lll} & m < \ell + n & \\ \Leftrightarrow & \ell \leq m < \ell + n & \text{(Inv. 4.7.8.14)} \\ \Rightarrow & \mathit{in\text{-}window}(\ell|_{2n}, m|_{2n}, (\ell + n)|_{2n}) & \text{(Lem. 4.7.3.7)} \end{array}$$

Reversely,

$$\begin{array}{lll} & \mathit{in\text{-}window}(\ell|_{2n}, m|_{2n}, (\ell + n)|_{2n}) & \\ \Rightarrow & m + n < \ell \vee \ell \leq m < \ell + n \vee m \geq \ell + 2n & \text{(Lem. 4.7.3.8)} \\ \Leftrightarrow & m < \ell + n & \text{(Inv. 4.7.8.14, 4.7.8.18)} \end{array}$$

Moreover, by Lemma 4.7.1.1, $(\ell + n)|_{2n} = (\ell|_{2n} + n)|_{2n}$.

- $S(m)|_{2n} = S(m|_{2n})|_{2n}$.
  This follows from Lemma 4.7.1.1.
- $\mathit{inb}(d, m, q)|_{2n} = \mathit{inb}(d, m|_{2n}, q|_{2n})$.
  This follows from the definition of buffers modulo $2n$.

B Below we equate the entire summand B of the two specifications. The argument $p := p+1$ is omitted, because it is irrelevant for this derivation.

$$\begin{array}{lll} & \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k, q), k, g)|_{2n}) & \\ & \lhd\ \mathit{test}(k, q) \rhd \delta & \\ = & \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k, q), k|_{2n}, g|_{2n})) & \\ & \lhd\ \mathit{test}(k, q) \wedge \ell \leq k < \ell + n \rhd \delta & \text{(Inv. 4.7.8.7, 4.7.8.15, 4.7.8.18)} \\ = & \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k|_{2n}, q|_{2n}), k|_{2n}, g|_{2n})) & \\ & \lhd\ \mathit{test}(k|_{2n}, q|_{2n}) \wedge \ell \leq k < \ell + n \rhd \delta & \text{(Lem. 4.7.3.2, 4.7.3.3)} \\ = & \sum_{k':Nat} \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k', q|_{2n}), k', g|_{2n})) & \\ & \lhd\ \mathit{test}(k', q|_{2n}) \wedge \ell \leq k < \ell + n \wedge k' = k|_{2n} \rhd \delta & \text{(sum elim.)} \\ = & \sum_{k':Nat} \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k', q|_{2n}), k', g|_{2n})) & \\ & \lhd\ \mathit{test}(k', q|_{2n}) \wedge k = (\ell\ \mathit{div}\ 2n)2n + k' \wedge & \\ & \ell|_{2n} \leq k' < \ell|_{2n} + n \wedge k' = k|_{2n} \rhd \delta & \\ + & \sum_{k':Nat} \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k', q|_{2n}), k', g|_{2n})) & \\ & \lhd\ \mathit{test}(k', q|_{2n}) \wedge k = S(\ell\ \mathit{div}\ 2n)2n + k' \wedge & \\ & k' + n < \ell|_{2n} \wedge k' = k|_{2n} \rhd \delta & \text{(Lem. 4.7.1.4, 4.7.1.5)} \\ = & \sum_{k':Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k', q|_{2n}), k', g|_{2n})) & \\ & \lhd\ \mathit{test}(k', q|_{2n}) \wedge \ell|_{2n} \leq k' < \ell|_{2n} + n \wedge k' = k' \rhd \delta & \\ + & \sum_{k':Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k', q|_{2n}), k', g|_{2n})) & \\ & \lhd\ \mathit{test}(k', q|_{2n}) \wedge k' + n < \ell|_{2n} \wedge k' = k' \rhd \delta & \text{(sum elim., Lem. 4.7.1.3)} \\ = & \sum_{k':Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k', q|_{2n}), k', g|_{2n})) & \\ & \lhd\ \mathit{test}(k', q|_{2n}) \rhd \delta & \text{(see below)} \end{array}$$

The last equality follows from the following derivation:

$$\begin{array}{lll} & \mathit{test}(k', q|_{2n}) & \\ \Rightarrow & \mathit{test}(k'|_{2n}, q|_{2n}) & \text{(Lem. 4.7.3.1)} \\ \Rightarrow & \ell \leq k'|_{2n} < \ell + n & \text{(Inv. 4.7.8.7, 4.7.8.15, 4.7.8.18)} \\ \Rightarrow & \mathit{in\text{-}window}(\ell|_{2n}, k'|_{2n}, (\ell+n)|_{2n}) & \text{(Lem. 4.7.3.9)} \\ \Rightarrow & k' + n < \ell|_{2n} \vee \ell|_{2n} \leq k' < \ell|_{2n} + n & \\ & \vee k' \geq \ell|_{2n} + 2n & \text{(Lem. 4.7.1.1, 4.7.3.8)} \\ \Leftrightarrow & k' + n < \ell|_{2n} \vee \ell|_{2n} \leq k' < \ell|_{2n} + n & \text{(Lem. 4.7.1.2, 4.7.3.1)} \end{array}$$

C $k < p \Rightarrow \mathit{delete}(k, g)|_{2n} = \mathit{delete}(k, g|_{2n})$.

By Invariant 4.7.8.1, $k<p \leq \mathit{length}(g)$. So this follows from Lemma 4.7.5.4.

E

- $\mathit{length}(g) = \mathit{length}(g|_{2n})$.
  This follows from Lemma 4.7.5.1.
- $p < \mathit{length}(g) \Rightarrow (\ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n = \mathit{in\text{-}window}(\ell'|_{2n}, \mathit{last\text{-}seq}(g)|_{2n}, (\ell'|_{2n} + n)|_{2n}))$.
  Since $0 < \mathit{length}(g)$, Lemmas 4.7.5.5, 4.7.5.6, and 4.7.5.10 yield $\mathit{member}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), g)$. So in combination with Invariant 4.7.8.22, this implies $\mathit{next\text{-}empty}(\ell', q') \leq \mathit{last\text{-}seq}(g) + n$. Hence, by Lemma 4.7.4.3, $\ell' \leq \mathit{last\text{-}seq}(g) + n$. Furthermore, by Invariant 4.7.8.8, $\mathit{last\text{-}seq}(g) < m$, by Invariant 4.7.8.18, $m \leq \ell + n$, and by Invariant 4.7.8.17, $\ell \leq \ell' + n$. Hence, $\mathit{last\text{-}seq}(g) < \ell' + 2n$. So by Lemmas 4.7.3.7 and 4.7.3.8,
  $\ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n = \mathit{in\text{-}window}(\ell'|_{2n}, \mathit{last\text{-}seq}(g)|_{2n}, (\ell' + n)|_{2n})$.
  And by Lemma 4.7.1.1, $(\ell' + n)|_{2n} = (\ell'|_{2n} + n)|_{2n}$.
- $p < \mathit{length}(g) \Rightarrow \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')|_{2n} = \mathit{inb}(\mathit{last\text{-}dat}(g|_{2n}), \mathit{last\text{-}seq}(g|_{2n}), q'|_{2n})$.
  This follows from the definitions of buffers modulo $2n$, and Lemmas 4.7.5.5, 4.7.5.6, 4.7.5.2 and 4.7.5.3.
- $p < \mathit{length}(g) \Rightarrow \mathit{delete\text{-}last}(g)|_{2n} = \mathit{delete\text{-}last}(g|_{2n})$.
  This follows from Lemmas 4.7.5.7 and 4.7.5.4.

F

- $\neg(\ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n)$
  $\Leftrightarrow \neg \mathit{in\text{-}window}(\ell'|_{2n}, \mathit{last\text{-}seq}(g)|_{2n}, (\ell'|_{2n} + n)|_{2n})$.
  This follows immediately from the second item of the previous case.
- $p < \mathit{length}(g) \Rightarrow \mathit{delete\text{-}last}(g)|_{2n} = \mathit{delete\text{-}last}(g|_{2n})$.
  This follows immediately from the fourth item of the previous case.

G

- $\mathit{test}(\ell', q') = \mathit{test}(\ell'|_{2n}, q'|_{2n})$.
  This follows from Lemma 4.7.3.2 together with Invariant 4.7.8.10.
- $\mathit{test}(\ell', q') \Rightarrow (\mathit{retrieve}(\ell', q') = \mathit{retrieve}(\ell'|_{2n}, q'|_{2n}))$.
  This follows from Lemma 4.7.3.3 together with Invariant 4.7.8.10.

- $S(\ell')|_{2n} = S(\ell'|_{2n})|_{2n}$.
  This follows from Lemma 4.7.1.1.
- $remove(\ell', q')|_{2n} = remove(\ell'|_{2n}, q'|_{2n})$.
  This follows from Lemma 4.7.3.4 together with Invariant 4.7.8.10.

H $inm(next\text{-}empty(\ell', q')|_{2n}, g')|_{2n} = inm(next\text{-}empty|_{2n}(\ell'|_{2n}, q'|_{2n}), g'|_{2n})$.
By Lemma 4.7.3.6 and Invariant 4.7.8.10, $next\text{-}empty(\ell', q')|_{2n} = next\text{-}empty|_{2n}(\ell'|_{2n}, q'|_{2n})$. So the desired equality follows the definition of mediums modulo $2n$.

I $k < p' \Rightarrow delete(k, g')|_{2n} = delete(k, g'|_{2n})$.
By Invariant 4.7.8.2, $k < p' \leq length(g')$. So the desired equality follows from Lemma 4.7.6.3.

K
- $length(g') = length(g'|_{2n})$.
  This follows from Lemma 4.7.6.1.
- $p' < length(g') \Rightarrow last\text{-}seq(g')|_{2n} = last\text{-}seq(g'|_{2n})|_{2n}$.
  This follows from Lemmas 4.7.6.4, 4.7.6.2 and 4.7.1.1.
- $release(\ell, last\text{-}seq(g'), q)|_{2n} = release|_{2n}(\ell|_{2n}, last\text{-}seq(g')|_{2n}, q|_{2n})$.
  By Lemmas 4.7.6.4 and 4.7.6.8, the condition $p' < length(g')$ implies $member(last\text{-}seq(g'), g')$. So by Invariant 4.7.8.6, $\ell \leq last\text{-}seq(g')$. By Invariants 4.7.8.3 and 4.7.8.12, $last\text{-}seq(g') \leq next\text{-}empty(\ell', q') \leq m$. And by Invariant 4.7.8.18, $m \leq \ell + n$. So $\ell \leq last\text{-}seq(g') \leq \ell + n$. Furthermore, by Invariants 4.7.8.7, 4.7.8.15 and 4.7.8.18, $test(i, q) \Rightarrow \ell \leq i < \ell + n$. Hence, the desired equation follows from Lemma 4.7.3.5.
- $p' < length(g') \Rightarrow delete\text{-}last(g')|_{2n} = delete\text{-}last(g'|_{2n})$.
  This follows from Lemmas 4.7.6.3 and 4.7.6.5.

Hence, $\mathbf{N}_{mod}(\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, q'|_{2n}, g|_{2n}, p, g'|_{2n}, p')$ is a solution for the defining equation of $\mathbf{N}_{nonmod}(\ell, m, q, \ell', q', g, p, g', p')$. So by CL-RSP, they are strongly (and thus branching) bisimilar. ⊠

4.8.2 Correctness of $\mathbf{N}_{nonmod}$

We prove that $\mathbf{N}_{nonmod}$ is branching bisimilar to the FIFO queue $\mathbf{Z}$ of size $2n$ (see Section 4.5.2), using the cones and foci method [54].

Let Ξ abbreviate $Nat \times Nat \times Buf \times Nat \times Buf \times MedK \times Nat \times MedL \times Nat$. Furthermore, let $\xi{:}\Xi$ denote $(\ell, m, q, \ell', q', g, p, g', p')$. The state mapping $\phi : \Xi \Rightarrow List$, which maps states of $\mathbf{N}_{nonmod}$ to states of $\mathbf{Z}$, is defined by:

$$\phi(\xi) = q'[\ell'..next\text{-}empty(\ell', q')\rangle ++ q[next\text{-}empty(\ell', q')..m\rangle$$

Intuitively, ϕ collects the data elements in the sending and receiving windows, starting at the first position of the receiving window (i.e., ℓ') until the first empty

position in this window, and then continuing in the sending window until the first empty position in that window (i.e., m). Note that ϕ is independent of ℓ, g, p, g', p'; we therefore write $\phi(m, q, \ell', q')$.

The focus points are those states where either the sending window is empty (meaning that $\ell = m$), or the receiving window is full and all data elements in the receiving window have been acknowledged, meaning that $\ell = \ell' + n$. That is, the focus condition for $\mathbf{N}_{nonmod}(\ell, m, q, \ell', q', g, p, g', p')$ is

$$FC(\ell, m, q, \ell', q', g, p, g', p') \ := \ \ell = m \vee \ell = \ell' + n$$

Lemma 4.8.1 For each $\xi{:}\Xi$ where the invariants in Lemma 4.7.8 hold, there is a $\hat{\xi}{:}\Xi$ with $\mathrm{FC}(\hat{\xi})$ such that $\mathbf{N}_{nonmod}(\xi) \stackrel{c_1}{\rightarrow} \cdots \stackrel{c_n}{\rightarrow} \mathbf{N}_{nonmod}(\hat{\xi})$, where $c_1, \ldots, c_n \in \mathcal{I}$.

Proof. By Invariants 4.7.8.12 and 4.7.8.13, *next-empty*$(\ell', q') \leq \min\{m, \ell' + n\}$. We prove by induction on $\min\{m, \ell' + n\} - $ *next-empty*(ℓ', q') that for each state ξ where the invariants in Lemma 4.7.8 hold, a focus point can be reached by a sequence of internal actions.
Basis: *next-empty*$(\ell', q') = \min\{m, \ell' + n\}$.
Let $y =$ *length*(g') and $x =$ *next-empty*(ℓ', q') at state ξ. By summand H, we reach a state ξ' with $g' :=$ *inm*(x, g'). Hence, at state ξ' there exists a $0 \leq k < y$ such that *return-seq*$(k, g') = x$ and *return-seq*$(i, g') \neq x$ for any $k < i < y$. In view of Invariant 4.7.8.5, $k < i < y \Rightarrow x >$ *return-seq*(i, g'). Then, by repeating summand J (p' times), we reach a state ξ'' with $p' = 0$. Then, by repeating summand K ($y - (k + 1)$ times), we reach a state ξ''' such that *last-seq*$(g') = x$. During these executions of H, J and K the values of m, ℓ', q' remain the same. By again performing summand K, we reach a state $\hat{\xi}$ where $\ell =$ *last-seq*$(g') = x = \min\{m, \ell' + n\}$. Then $\ell = m$ or $\ell = \ell' + n$, so $FC(\hat{\xi})$.
Induction step: *next-empty*$(\ell', q') < \min\{m, \ell' + n\}$.
Let $y =$ *length*(g) and $x =$ *next-empty*(ℓ', q') at state ξ. By Invariants 4.7.8.4 and 4.7.8.12, $\ell \leq x < m$. So by Invariant 4.7.8.16, *test*(x, q). Furthermore, in view of Lemma 4.7.4.3, $\ell' \leq x < \ell' + n$. By summand B, we perform an internal action to a state ξ' with $g{:=}$*inm*(d, x, g) (where d denotes *retrieve*(x, q)). Hence, at state ξ' there exists a $0 \leq k < y$ such that *return-seq*$(k, g) = x$ and *return-seq*$(i, g) \neq x$ for any $k < i < y$. Then, by repeating summand D (p times), we reach a state ξ'' with $p = 0$. Then, by repeating summands E and F ($y - (k + 1)$ times), we reach a state ξ''' with *last-dat*$(g) = d$ and *last-seq*$(g) = x$. During these executions of B, D, E and F, the values of m, ℓ' remain the same; and since during the executions of E and F *last-seq*$(g) \neq x$, in view of Lemma 4.7.4.5, the value of *next-empty*(ℓ', q') remains the same. By again performing summand E, we reach a state ξ'''' where $q' :=$ *inb*(d, x, q'). Recall that $x =$ *next-empty*(ℓ', q').

$$\begin{array}{lll} & \textit{next-empty}(\ell', \textit{in}(d, \textit{next-empty}(\ell', q'), q')) & \\ = & \textit{next-empty}(S(\textit{next-empty}(\ell', q')), q') & \text{(Lem. 4.7.4.6)} \\ > & \textit{next-empty}(\ell', q') & \text{(Lem. 4.7.4.3)} \end{array}$$

So we can apply the induction hypothesis to conclude that from ξ'''' a focus point $\hat{\xi}$ can be reached by a sequence of internal actions.

⊠

Theorem 4.8.2 For all $e{:}\Delta$,

$$\tau_{\{c,j\}}(\mathbf{N}_{nonmod}(0,0,[],0,[],[]^K,0,[]^L,0)) \leftrightarrow_b \mathbf{Z}(\langle\rangle).$$

Proof. By the cones and foci method we obtain the following matching criteria (see Definition 3.2.3). Trivial matching criteria are left out.
Class I:

$$(p < \mathit{length}(g) \wedge \ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n)$$
$$\Rightarrow\ \phi(m,q,\ell',q') = \phi(m,q,\ell',\mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q'))$$
$$p' < \mathit{length}(g')\ \Rightarrow\ \phi(m,q,\ell',q') = \phi(m,\mathit{release}(\ell,\mathit{last\text{-}seq}(g'),q),\ell',q')$$

Class II:

$$m < \ell + n\ \Rightarrow\ \mathit{length}(\phi(m,q,\ell',q')) < 2n$$
$$\mathit{test}(\ell',q')\ \Rightarrow\ \mathit{length}(\phi(m,q,\ell',q')) > 0$$

Class III:

$$((\ell = m \vee \ell = \ell' + n) \wedge \mathit{length}(\phi(m,q,\ell',q')) < 2n)\ \Rightarrow\ m < \ell + n$$
$$((\ell = m \vee \ell = \ell' + n) \wedge \mathit{length}(\phi(m,q,\ell',q')) > 0)\ \Rightarrow\ \mathit{test}(\ell',q')$$

Class IV:

$$\mathit{test}(\ell',q')\ \Rightarrow\ \mathit{retrieve}(\ell',q') = \mathit{top}(\phi(m,q,\ell',q'))$$

Class V:

$$m < \ell + n\ \Rightarrow\ \phi(S(m),\mathit{inb}(d,m,q),\ell',q') = \mathit{append}(d,\phi(m,q,\ell',q'))$$
$$\mathit{test}(\ell',q')\ \Rightarrow\ \phi(m,q,S(\ell'),\mathit{remove}(\ell',q')) = \mathit{tail}(\phi(m,q,\ell',q'))$$

I.1 $(p < \mathit{length}(g) \wedge \ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n)$
$\Rightarrow \phi(m,q,\ell',q') = \phi(m,q,\ell',\mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q'))$.

Let $p < \mathit{length}(g)$. By Lemmas 4.7.5.5, 4.7.5.6 and 4.7.5.10, $\mathit{member}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),g)$.

Case 1: $\mathit{last\text{-}seq}(g) \neq \mathit{next\text{-}empty}(\ell',q')$. By Lemma 4.7.4.5, $\mathit{next\text{-}empty}(\ell',\mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q')) = \mathit{next\text{-}empty}(\ell',q')$. Hence,

$$\begin{array}{ll} & \phi(m,q,\ell',\mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q')) \\ = & \mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q')[\ell'..\mathit{next\text{-}empty}(\ell',q')\rangle \\ & +\!\!+ q[\mathit{next\text{-}empty}(\ell',q')..m\rangle \end{array}$$

Case 1.1: $\ell' \leq \mathit{last\text{-}seq}(g) < \mathit{next\text{-}empty}(\ell',q')$.
By Lemma 4.7.4.2, $\mathit{test}(\mathit{last\text{-}seq}(g),q')$, so by Invariant 4.7.8.26 and $\mathit{member}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),g)$, $\mathit{retrieve}(\mathit{last\text{-}seq}(g),q') = \mathit{last\text{-}dat}(g)$.

So by Lemma 4.7.7.7, $inb(last\text{-}dat(g), last\text{-}seq(g), q')[\ell'..next\text{-}empty(\ell', q')\rangle = q'[\ell'..next\text{-}empty(\ell', q')\rangle$.

Case 1.2: $\neg(\ell' \leq last\text{-}seq(g) < next\text{-}empty(\ell', q'))$. Using Lemma 4.7.7.8, it follows that

$$\begin{array}{ll} & inb(last\text{-}dat(g), last\text{-}seq(g), q')[\ell'..next\text{-}empty(\ell', q')\rangle \\ = & remove(last\text{-}seq(g), inb(last\text{-}dat(g), last\text{-}seq(g), q')) \\ & [\ell'..next\text{-}empty(\ell', q')\rangle \\ = & remove(last\text{-}seq(g), q')[\ell'..next\text{-}empty(\ell', q')\rangle \\ = & q'[\ell'..next\text{-}empty(\ell', q')\rangle \end{array}$$

Case 2: $last\text{-}seq(g) = next\text{-}empty(\ell', q')$.
The derivation splits into two parts.

(1) Using Lemma 4.7.7.8, it follows that

$$\begin{array}{ll} & inb(last\text{-}dat(g), last\text{-}seq(g), q')[\ell'..last\text{-}seq(g)\rangle \\ = & remove(last\text{-}dat(g), inb(last\text{-}dat(g), last\text{-}seq(g), q'))[\ell'..last\text{-}seq(g)\rangle \\ = & remove(last\text{-}dat(g), q')[\ell'..last\text{-}seq(g)\rangle \\ = & q'[\ell'..last\text{-}seq(g)\rangle \end{array}$$

(2) By Invariant 4.7.8.4, $\ell \leq last\text{-}seq(g)$.
By Invariant 4.7.8.8 and $member(last\text{-}dat(g), last\text{-}seq(g), g)$, $last\text{-}seq(g) < m$. Thus, by Invariant 4.7.8.16, $test(last\text{-}seq(g), q)$. So by Invariant 4.7.8.23 together with $member(last\text{-}dat(g), last\text{-}seq(g), g)$, $retrieve(last\text{-}seq(g), q) = last\text{-}dat(g)$. Since $\ell \leq S(last\text{-}seq(g)) \leq m$, by Invariant 4.7.8.27,

$$\begin{array}{ll} & q'[S(last\text{-}seq(g))..next\text{-}empty(S(last\text{-}seq(g)), q')\rangle \\ = & q[S(last\text{-}seq(g))..next\text{-}empty(S(last\text{-}seq(g)), q')\rangle \end{array}$$

Hence,

$$\begin{array}{ll} & inb(last\text{-}dat(g), last\text{-}seq(g), q') \\ & [last\text{-}seq(g)..next\text{-}empty(S(last\text{-}seq(g)), q')\rangle \\ = & inl(last\text{-}dat(g), inb(last\text{-}dat(g), last\text{-}seq(g), q') \\ & [S(last\text{-}seq(g))..next\text{-}empty(S(last\text{-}seq(g)), q')\rangle) \\ = & inl(last\text{-}dat(g), remove(last\text{-}seq(g), inb(last\text{-}dat(g), last\text{-}seq(g), q')) \\ & [S(last\text{-}seq(g))..next\text{-}empty(S(last\text{-}seq(g)), q')\rangle) \quad \text{(Lem. 4.7.7.8)} \\ = & inl(last\text{-}dat(g), remove(last\text{-}seq(g), q') \\ & [S(last\text{-}seq(g))..next\text{-}empty(S(last\text{-}seq(g)), q')\rangle) \\ = & inl(last\text{-}dat(g), q'[S(last\text{-}seq(g))..next\text{-}empty(S(last\text{-}seq(g)), q')\rangle) \\ & \qquad \text{(Lem. 4.7.7.8)} \\ = & inl(last\text{-}dat(g), q[S(last\text{-}seq(g))..next\text{-}empty(S(last\text{-}seq(g)), q')\rangle) \\ & \qquad \text{(see above)} \\ = & q[last\text{-}seq(g)..next\text{-}empty(S(last\text{-}seq(g)), q')\rangle \end{array}$$

We combine (1) and (2). Recall that $\mathit{last\text{-}seq}(g) = \mathit{next\text{-}empty}(\ell', q')$.

$$\begin{array}{ll} & (\mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q') \\ & [\ell'..\mathit{next\text{-}empty}(\ell', \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q'))\rangle) \\ & +\!\!+ q[\mathit{next\text{-}empty}(\ell', \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q'))..m\rangle \\ = & \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')[\ell'..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle \\ & +\!\!+ q[\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')..m\rangle & \text{(Lem. 4.7.4.6)} \\ = & (\mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')[\ell'..\mathit{last\text{-}seq}(g)\rangle \\ & +\!\!+ \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q') \\ & [\mathit{last\text{-}seq}(g)..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle) \\ & +\!\!+ q[\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')..m\rangle & \text{(Lem. 4.7.4.3, 4.7.7.5)} \\ = & (q'[\ell'..\mathit{last\text{-}seq}(g)\rangle +\!\!+ q[\mathit{last\text{-}seq}(g)..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle \\ & +\!\!+ q[\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')..m\rangle & \text{((1), (2))} \\ = & q'[\ell'..\mathit{last\text{-}seq}(g)\rangle +\!\!+ q[\mathit{last\text{-}seq}(g)..m\rangle & \text{(Lem. 4.7.7.1, 4.7.4.2, 4.7.7.5)} \end{array}$$

I.2 $p' < \mathit{length}(g') \Rightarrow \phi(m, q, \ell', q') = \phi(m, \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q), \ell', q')$.

$p'<\mathit{length}(g')$, so by Lemmas 4.7.6.4 and 4.7.6.8, $\mathit{member}(\mathit{last\text{-}seq}(g'), g')$. By Invariant 4.7.8.3, $\mathit{last\text{-}seq}(g')\leq\mathit{next\text{-}empty}(\ell', q')$. So by Lemma 4.7.7.9,

$$\mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)[\mathit{next\text{-}empty}(\ell', q')..m\rangle = q[\mathit{next\text{-}empty}(\ell', q')..m\rangle$$

II.1 $m < \ell + n \Rightarrow \mathit{length}(\phi(m, q, \ell', q')) < 2n$.

Let $m < \ell + n$.

$$\begin{array}{ll} & \mathit{length}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) \\ = & \mathit{length}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle) \\ & +\mathit{length}(q[\mathit{next\text{-}empty}(\ell', q')..m\rangle)) & \text{(Lem. 4.7.7.2)} \\ = & (\mathit{next\text{-}empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \mathit{next\text{-}empty}(\ell', q')) & \text{(Lem. 4.7.7.4)} \\ \leq & n + (m \dot{-} \ell) & \text{(Inv. 4.7.8.13, 4.7.8.4)} \\ < & 2n \end{array}$$

II.2 $\mathit{test}(\ell', q') \Rightarrow \mathit{length}(\phi(m, q, \ell', q')) > 0$.

$\mathit{test}(\ell', q')$ together with Lemma 4.7.4.3 yields $\mathit{next\text{-}empty}(\ell', q') = \mathit{next\text{-}empty}(S(\ell'), q') \geq S(\ell')$. Hence, by Lemmas 4.7.7.2 and 4.7.7.4,

$$\begin{array}{ll} & \mathit{length}(\phi(m, q, \ell', q')) \\ = & (\mathit{next\text{-}empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \mathit{next\text{-}empty}(\ell', q')) \\ > & 0 \end{array}$$

III.1 $((\ell = m \vee \ell = \ell' + n) \wedge \mathit{length}(\phi(m, q, \ell', q')) < 2n) \Rightarrow m < \ell + n$.

Case 1: $\ell = m$. Then $m < \ell + n$ holds trivially.

CASE 2: $\ell = \ell' + n$.

$$\begin{array}{lll} & \mathit{length}(\phi(m,q,\ell',q')) & \\ = & (\mathit{next\text{-}empty}(\ell',q') \dot{-} \ell') & \\ & +(m \dot{-} \mathit{next\text{-}empty}(\ell',q')) & \text{(Lem. 4.7.7.2, 4.7.7.4)} \\ \leq & ((\ell'+n) \dot{-} \ell') + (m \dot{-} \ell) & \text{(Inv. 4.7.8.13, 4.7.8.4)} \\ = & n + (m \dot{-} \ell) & \end{array}$$

So $\mathit{length}(\phi(m,q,\ell',q')) < 2n$ implies $m < \ell + n$.

III.2 $((\ell = m \vee \ell = \ell' + n) \wedge \mathit{length}(\phi(m,q,\ell',q')) > 0) \Rightarrow \mathit{test}(\ell',q')$.

CASE 1: $\ell = m$. Then $m \dot{-} \mathit{next\text{-}empty}(\ell',q') \leq m \dot{-} \ell$ (Inv. 4.7.8.4) $= 0$. Hence,

$$\begin{array}{lll} & \mathit{length}(\phi(m,q,\ell',q')) & \\ = & (\mathit{next\text{-}empty}(\ell',q') \dot{-} \ell') & \\ & +(m \dot{-} \mathit{next\text{-}empty}(\ell',q')) & \text{(Lem. 4.7.7.2, 4.7.7.4)} \\ = & \mathit{next\text{-}empty}(\ell',q') \dot{-} \ell' & \end{array}$$

Hence, $\mathit{length}(\phi(m,q,\ell',q')) > 0$ yields $\mathit{next\text{-}empty}(\ell',q') > \ell'$, which implies $\mathit{test}(\ell',q')$.

CASE 2: $\ell = \ell' + n$. Then by Invariant 4.7.8.4, $\mathit{next\text{-}empty}(\ell',q') \geq \ell' + n$, which implies $\mathit{test}(\ell',q')$.

IV $\mathit{test}(\ell',q') \;\Rightarrow\; \mathit{retrieve}(\ell',q') = \mathit{top}(\phi(m,q,\ell',q'))$.

$\mathit{test}(\ell',q')$ implies $\mathit{next\text{-}empty}(\ell',q') = \mathit{next\text{-}empty}(S(\ell'),q') \geq S(\ell')$ (Lem. 4.7.4.3). Hence,

$$\begin{array}{rl} & q'[\ell'..\mathit{next\text{-}empty}(\ell',q')\rangle \\ = & \mathit{inl}(\mathit{retrieve}(\ell',q'), q'[S(\ell')..\mathit{next\text{-}empty}(\ell',q')\rangle) \end{array}$$

So

$$\begin{array}{rl} & \mathit{top}(\phi(m,q,\ell',q')) \\ = & \mathit{top}(\mathit{inl}(\mathit{retrieve}(\ell',q'), q'[S(\ell')..\mathit{next\text{-}empty}(\ell',q')\rangle \\ & +\!\!+ q[\mathit{next\text{-}empty}(\ell',q')..m\rangle)) \\ = & \mathit{retrieve}(\ell',q') \end{array}$$

V.1 $m < \ell + n \Rightarrow \phi(S(m), \mathit{inb}(d,m,q), \ell', q') = \mathit{append}(d, \phi(m,q,\ell',q'))$.

$$\begin{array}{lll} & q'[\ell'..\mathit{next\text{-}empty}(\ell',q')\rangle +\!\!+ & \\ & \mathit{inb}(d,m,q)[\mathit{next\text{-}empty}(\ell',q')..S(m)\rangle & \\ = & q'[\ell'..\mathit{next\text{-}empty}(\ell',q')\rangle +\!\!+ & \\ & \mathit{append}(d, q[\mathit{next\text{-}empty}(\ell',q')..m\rangle) & \text{(Lem. 4.7.7.6, Inv. 4.7.8.12)} \\ = & \mathit{append}(d, q'[\ell'..\mathit{next\text{-}empty}(\ell',q')\rangle +\!\!+ & \\ & q[\mathit{next\text{-}empty}(\ell',q')..m\rangle) & \text{(Lem. 4.7.7.3)} \end{array}$$

V.2 $\mathit{test}(\ell', q') \Rightarrow \phi(m, q, S(\ell'), \mathit{remove}(\ell', q')) = \mathit{tail}(\phi(m, q, \ell', q'))$.

$\mathit{test}(\ell', q')$, together with Lemma 4.7.4.3 implies $\mathit{next\text{-}empty}(\ell', q') = \mathit{next\text{-}empty}(S(\ell'), q') \geq S(\ell')$. Hence,

$$
\begin{array}{ll}
 & \mathit{remove}(\ell', q')[S(\ell')..\mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q'))\rangle \\
 & +\!\!+ q[\mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q'))..m\rangle \\
= & \mathit{remove}(\ell', q')[S(\ell')..\mathit{next\text{-}empty}(S(\ell'), q')\rangle \\
 & +\!\!+ q[\mathit{next\text{-}empty}(S(\ell'), q')..m\rangle \qquad \text{(Lem. 4.7.4.7)} \\
= & \mathit{remove}(\ell', q')[S(\ell')..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle \\
= & q'[S(\ell')..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle \text{ (Lem. 4.7.7.8)} \\
= & \mathit{tail}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle)
\end{array}
$$

⊠

4.8.3 Correctness of the sliding window protocol

Finally, we can prove Theorem 4.5.1.

Proof.

$$
\begin{array}{lll}
 & \tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K}([]^K,0) \parallel \mathbf{L}([]^L,0))) & \\
\underline{\leftrightarrow} & \tau_{\mathcal{I}}(\mathbf{M}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0)) & \text{(Thm. 4.6.1)} \\
\underline{\leftrightarrow} & \tau_{\{c,j\}}(\mathbf{N}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0)) & \text{(Thm. 4.6.2)} \\
\underline{\leftrightarrow} & \tau_{\{c,j\}}(\mathbf{N}_{nonmod}(0,0,[],0,[],[]^K,0,[]^L,0)) & \text{(Thm. 4.6.3)} \\
\underline{\leftrightarrow_b} & \mathbf{Z}(\langle\rangle) & \text{(Thm. 4.8.2)}
\end{array}
$$

⊠

4.9 Conclusions

In this chapter, we have proved the correctness of a sliding window protocol with an arbitrary finite window size n and sequence numbers modulo $2n$. We showed that the sliding window protocol is branching bisimilar to a queue of capacity $2n$. This proof is entirely based on the axiomatic theory underlying μCRL and the axioms characterizing the data types, and was checked with the help of PVS. It implies both safety and liveness of the protocol.

Chapter 5

A Note on K-state Self-Stabilization in a Ring with $K = N$

5.1 Introduction

In his seminal paper [40], Dijkstra introduced the notion of self-stabilization. A distributed system is said to be self-stabilizing if it satisfies the following two properties:

1. *convergence*: starting from an arbitrary state, the system is guaranteed to reach a stable state;

2. *closure*: once the system reaches a stable state, it cannot become unstable anymore.

A system with the property of self-stabilization can have the advantages of fault tolerance, robustness for dynamic topologies, and straightforward initialization.

Consider a system with a number of processes sharing a common resource (usually called critical section). Given an arbitrary initial state of the system, there might be more than one process enabled to access the common resource. The problem of mutual exclusion is to guarantee that the common resource cannot be accessed by more than one process simultaneously. Self-stabilizing algorithms for mutual exclusion make sure that each infinite run of the system reaches a stable state where exactly one process is enabled; and from then on, mutual exclusion of the common resource is guaranteed.

In [40], Dijkstra presented three elementary self-stabilizing algorithms for mutual exclusion on a ring network: an algorithm with K-state processes, an algorithm with four-state processes, and an algorithm with three-state processes. Regarding their correctness, he wrote:

> "For brevity's sake most of the heuristics that led me to find them, together with the proofs that they satisfy the requirements, have been omitted, [...]".

After more than ten years, Dijkstra [42] published a proof of self-stabilization of his algorithm with three-state processes, and acknowledged that the verification was actually not trivial.

In this chapter, we focus on Dijkstra's algorithm with K-state processes. We consider a system of $N+1$ processes, numbered from 0 through N, arranged in a unidirectional ring. Each process p_i has a counter $v(i)$ that can hold a value from 0 to $K-1$. Each process can observe its own counter value and the counter value of its anti-clockwise neighbor. p_0 is a distinguished process that is enabled when $v(0) = v(N)$, and when enabled, it can increment its counter by 1 modulo K. Each process p_i for $i = 1, \ldots, N$ is enabled when $v(i) \neq v(i-1)$, and when enabled, it can update its counter value so that $v(i) = v(i-1)$. Thus the behavior of the system can be presented as follows:

Dijkstra's K-state algorithm for mutual exclusion.
Assume that processes $p_0, \ldots, p_N$ form a unidirectional ring, where the counter for each process p_i holds a value $v(i) \in \{0, \ldots, K-1\}$.

- if $v(0) = v(N)$, then $v(0) := (v(0) + 1) \bmod K$;
- if $v(i) \neq v(i-1)$ for $i = 1, \ldots, N$, then $v(i) := v(i-1)$.

The system is said to be in a *stable* state if it contains exactly one enabled process, which can be interpreted as holding a token. This token can be passed along the ring network; a process can access the common resource only when it holds the token.

This algorithm has been proved correct by different proof methods for self-stabilization, e.g. [172, 167, 168]. It attracted much attention from the formal verification community. There are two distinct traditions in automatic verification: theorem proving and model checking. Merz [124] formalized the algorithm and proved it correct in Isabelle/HOL [130]. Qadeer and Shankar [144] applied PVS [131] to prove its correctness. Later on, Kulkarni *et al.* [106] also proved its correctness using PVS in a different fashion. Model checking techniques were applied to this algorithm in [159, 169]. Due to the state explosion problem, this approach has some restrictions: it cannot be directly used for any possible initial state, and/or it can only prove the algorithm correct with a limited number of processes and states.

However, all these proofs only showed correctness of the algorithm under a weaker condition, namely the algorithm is correct if $K > N$. This also happened in Schneider's survey paper on self-stabilization [153]. The only exception we could find is [106]. Although they proved the algorithm correct for $K > N$, almost at the end of the paper, they stated:

> "it is possible to prove stabilization when $K \geq N$– we will need to redo only the proofs that depend on this assumption, namely Lemmas 6.4, 6.6, 6.8."

However, the validity of this claim is not clear, especially their formulation of Lemma 6.4 is false when $K = N$.

Judging on the literature, it seems to be a common belief that Dijkstra's K-state mutual exclusion algorithm on a ring only stabilizes when $K > N$. But in fact, Dijkstra gave a note after presenting the solution with K-state machines in [40] as follows:

> "Note 1. [...] the relation $K \geq N$ is sufficient."

A brief informal proof sketch was given by himself in [41]. In addition, he said:

> "(and for smaller values of K counter examples kill the assumption of self-stabilization.)"

We note that, if $K = N$, there should be at least three processes in the ring; namely, if $K = N = 1$, then clearly p_0 is always enabled and p_1 is never enabled. If $K > N$, then the algorithm also works for a ring with two processes.

In this chapter, we formally prove that if $N > 1$, then $K \geq N$ is sufficient for the stabilization of Dijkstra's K-state mutual exclusion algorithm. For the condition $K > N$, the proofs in [172, 167, 144, 124, 106] used the classic pigeonhole principle. The proof for $K = N$ becomes considerably more complicated, since the pigeonhole principle cannot be simply applied for any state of the algorithm. This will be explained in detail in Section 5.3. Our proof, which is different from the proof sketch in [41], has been checked in PVS.

Outline of the chapter. In Section 5.2, we show that Dijkstra's K-state mutual exclusion algorithm on a ring also stabilizes when the number of states per process is one less than the number of processes on the ring, namely $K \geq N$. We formalized the algorithm and checked our proof in PVS. Our verification in PVS is based on [144], we reused their formalization of the algorithm and most of their lemmas. We present the crucial lemmas of our PVS verification in Section 5.3. In Section 5.4, we show that $K \geq N$ is sharp by a counter-example, which was missing in [41]. Section 5.5 contains some conclusions.

5.2 Proof of Self-Stabilization

We give the proof that Dijkstra's K-state mutual exclusion algorithm on a ring stabilizes when $K \geq N$. First we prove the closure property for self-stabilization (see Proposition 5.2.2).

Lemma 5.2.1 In each state of the algorithm, there is at least one enabled process.

Proof. We distinguish two cases:

- for all $i \in \{1, \ldots, N\}$, $v(i) = v(0)$. In particular, $v(0) = v(N)$, which implies p_0 is enabled;
- otherwise, there exists a $j \in \{1, \ldots, N\}$ such that $v(j) \neq v(0)$, and for all $i \in \{1, \ldots, j-1\}$, $v(i) = v(0)$. Since $v(j) \neq v(j-1)$, p_j is enabled.

⊠

Lemma 5.2.1 implies that no run of the algorithm ever deadlocks, as in each state the enabled process(es) can "fire", meaning that the counter value is updated.

Proposition 5.2.2 Once in a stable state, the system will remain in stable states.

Proof. We assume p_i is the only enabled process in some stable state. It is easy to see that when p_i fires, it makes itself disabled, and it makes at most p_i's clockwise neighbor enabled. By Lemma 5.2.1, in each state of the algorithm, there exists at least one enabled process. Therefore, after the firing of p_i, the clockwise neighbor of p_i is the only enabled process, so the system remains in a stable state. ⊠

We proceed to prove the convergence property for self-stabilization (see Theorem 5.2.5).

Lemma 5.2.3 In each infinite run of the algorithm, p_0 fires infinitely often.

Proof. Given a state, consider the sum over all elements in the set $\{N - i \mid i \in \{1, \ldots, N\} \wedge p_i \textit{ is}$ enabled$\}$. Clearly, when a nonzero process fires, this sum strictly decreases. Furthermore, for each state, this sum is at least 0. Hence, in each infinite run, p_0 must fire infinitely often. ⊠

Definition 5.2.4 The *legitimate states* are those states that satisfy $v(i) = x$ for all $i < j$ and $v(i) = (x - 1) \bmod K$ for all $j \leq i \leq N$, for some choice of $x < K$ and $j \leq N$.

Note that a legitimate state is stable, as only p_j is enabled.

Theorem 5.2.5 Let $N > 1$. Even if $K = N$, Dijkstra's K-state mutual exclusion algorithm for $N + 1$ processes stabilizes.

Proof. By Lemma 5.2.1, no run of the algorithm deadlocks. By Lemma 5.2.3, in each infinite run of the algorithm p_0 fires infinitely often.

Let $N > 1$. We prove that each infinite run of the algorithm visits a legitimate state. Consider the case where p_0 fires for the first time. Then just before that, $v(0) = v(N) = y$ for some y, and the new value of $v(0)$ becomes $(y+1) \bmod K$. Now consider the case when p_0 fires again. Then just before that, $v(0) = v(N) = (y + 1) \bmod K$. In order for p_N to change its counter value from y to $(y + 1) \bmod K$, it must have copied $(y + 1) \bmod K$ from its anti-clockwise neighbor p_{N-1}. This moment must have occurred after p_0 changed its counter value to $v(0) = (y + 1) \bmod K$. But then, just after p_N copies $(y + 1) \bmod K$ from p_{N-1}, we actually have $v(N - 1) = v(N) = (y + 1) \bmod K$. In other words, since $N > 1$ implies that $p_{N-1} \neq p_0$, two different nonzero processes hold the same counter value $(y+1) \bmod K$. Then the N nonzero processes hold

at most $N-1$ different counter values from $\{0,\ldots,K-1\}$. When $K \geq N$ (so in particular when $K = N$), then at this point in time there is an $x < K$ that does not occur as the counter value of any nonzero process in the ring.

Since p_0 fires infinitely often, eventually $v(0)$ becomes x. The other processes merely copy counter values from their anti-clockwise neighbors, so at this point no other process holds x. The next time p_0 fires, $v(N) = v(0) = x$. The only way that p_N gets the counter value x is if all intermediate processes have copied x from p_0. We conclude that all processes have the counter value x, which is a legitimate state. ⊠

Dijkstra [41] gave a specific scenario to show that the system will definitely reach a legitimate state, after p_0 has been enabled for N times. In most cases, a legitimate state can be detected earlier than in that scenario, as shown in the above proof.

5.3 Mechanical Verification in PVS

In [144], Qadeer and Shankar presented a detailed description of a mechanical verification in PVS of stabilization of Dijkstra's K-state mutual exclusion algorithm. Although they only checked the correctness of the algorithm under the condition $K > N$, their PVS formalism and proof could for a large part be reused,[1] which saved us much effort and gave us many insightful thoughts on the verification in PVS.

First, we present Qadeer and Shankar's claims to sketch their proof skeleton. Then we show the lemma that we had to adapt for our proof. The algorithm satisfies the following properties, for each state of the system, and each infinite run from this state:

I. there is always at least one enabled process;

II. the number of enabled processes never increases;

III. the enabledness of each process is eventually toggled;

IV. p_0 eventually takes on any counter value below K (follows by Property III);

These properties require no restriction on the relation between N and K. Property I corresponds to Lemma 5.2.1. Property II follows the fact that when a process fires, it makes itself disabled, and it makes at most its clockwise neighbor enabled. Property III is a more general version of Lemma 5.2.3. Qadeer and Shankar's PVS proof of these first four properties could be (more or less) reused by us directly.

V. there is some value x below K such that $v(i) \neq x$ for all $i \in \{1,\ldots,N\}$ (follows by Property IV, and the proof of Theorem 5.2.5);

[1] The URL `http://www.csl.sri.com/pvs/examples/self-stability/` contains their PVS formalization and proofs.

VI. eventually $v(0) = x$, and $v(i) \neq x$ for all $i \in \{1, \ldots, N\}$; then p_0 is disabled until $v(i) = v(0)$ for all $i \in \{1, \ldots, N\}$ (follows by Property V);

VII. the system is self-stabilizing (follows by properties VI, I, and II).

The proof of Property V uses the pigeonhole principle, which states that if each of $n+1$ pigeons is assigned to one of n pigeonholes, then some hole must contain at least two pigeons. This principle was also formulated and proved in [144].

Let $S(v)$ denote the set $\{x < K \mid \exists i \in \{1, \ldots, N\}(v(i) = x)\}$. The following lemma corresponds to Property V. It states that the nonzero processes do not contain all the possible counter values.

Lemma 5.3.1 (Lemma 4.13 in [144]) If $K > N$, then $\exists x < K(x \notin S(v))$.

Under the condition $K > N$, this can be informally proved as follows [144]: there are N nonzero processes, and hence at most N distinct counter values at these processes; if there are K $(K > N)$ possible counter values, then there must be some $x < K$ that is not the counter value at any nonzero process.

If we relax the condition to $K \geq N$, the above proof fails, because the pigeonhole principle does not apply when the number of pigeons equals the number of pigeonholes.

Starting from this point, we assume that $K \geq N$. We define $T(v)$ to denote the set $\{x < K \mid \exists i \in \{1, \ldots, N-1\}(v(i) = x)\}$. In the following lemma the pigeonhole principle does apply.

Lemma 5.3.2 $\exists x < K(x \notin T(v))$.

Proof. $T(v)$ contains at most $N-1$ distinct counter values at processes from p_1 to p_{N-1}. If there are K $(K \geq N)$ possible counter values, then there must be some $x < K$ with $x \notin T(v)$. ⊠

To check the proof of Lemma 5.3.2 in PVS, we could simply follow the PVS proof steps of Lemma 5.3.1 in [144]. Now we introduce an extra lemma.

Lemma 5.3.3 $v(N) \in T(v) \Rightarrow S(v) = T(v)$.

Proof. This is straightforward by the definitions of $S(v)$ and $T(v)$. ⊠

In PVS, Lemma 5.3.3 could be proved by using existing PVS libraries for the finite cardinalities. Now we present the main lemma for our PVS proof, corresponding to Lemma 5.3.1 in [144] (Property VI).

Lemma 5.3.4 Each infinite run of the algorithm eventually reaches a state where the nonzero processes do not contain all the possible counter values.

Proof. We know from Property III that p_N will eventually fire. By the algorithm, we then have $v(N) = v(N-1)$, so that $v(N) \in T(v)$. By Lemma 5.3.3, $S(v) = T(v)$. By Lemma 5.3.2, we can find an $x < K$ with $x \notin T(v)$, so $x \notin S(v)$. ⊠

After proving Lemma 5.3.4, and reusing (more or less) the lemmas and the PVS proof steps for properties VI and VII in [144], we could mechanically prove self-stabilization of Dijkstra's K-state algorithm in PVS.

5.4 $K = N$ is Sharp

In this section, we give a counter-example showing that a smaller value of K would kill self-stabilization. For example, in Figure 5.1 (which assumes that $N \geq 3$), we have a system with $K = N - 1$, meaning that each process can have a counter value $\{0, \ldots, N-2\}$. Consider the initial state shown at the top left-hand side of Figure 5.1, in which $p_0, \ldots, p_{N-2}$ hold counter values from 0 to $N-2$, p_{N-1} holds counter value 0, and p_N holds counter value 1. By the algorithm, $p_1, \ldots, p_N$ are enabled, so the number of enabled processes is N. (In Figure 5.1, black processes are enabled.)

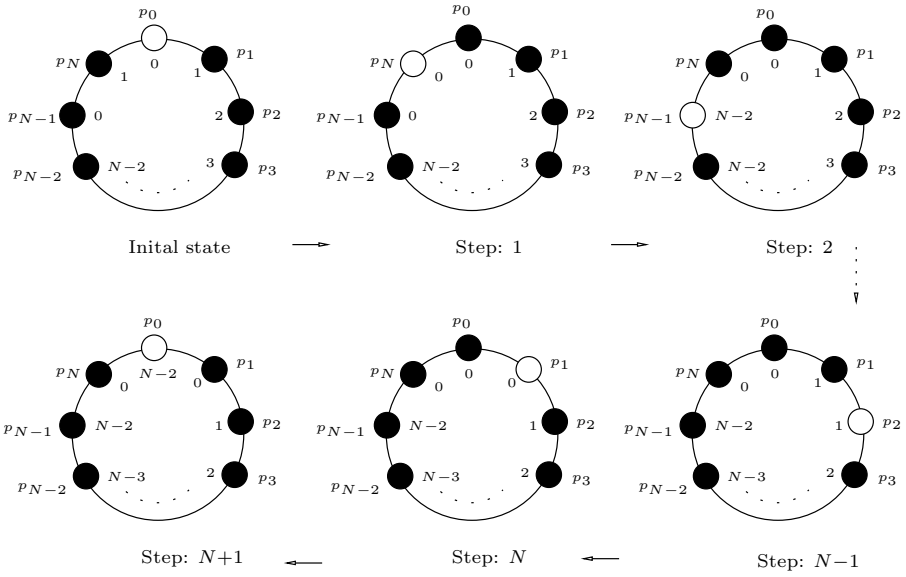


Figure 5.1: A counter-example: a ring with $K = N - 1$

We have a run as follows:

Step 1: p_N fires and makes p_0 enabled;

Step 2: p_{N-1} fires and makes p_N enabled;

.....

Step $N-1$: p_2 fires and makes p_3 enabled;

Step N: p_1 fires and makes p_2 enabled;

Step $N+1$: p_0 fires and makes p_1 enabled.

From the initial state, after the above $N+1$ steps (all processes have fired only once), the system ends in a state where the counter values of the processes are symmetric (modulo $N-1$) to the initial state, so it still has N enabled processes. This scenario can be executed infinitely often without breaking the symmetry. So the system will never reach a legitimate state. Thus $K = N$ is sharp!

5.5 Conclusions

Judging on the literature on self-stabilization, it seems to be common belief that Dijkstra's K-state algorithm on a ring stabilizes when $K > N$. In this chapter we show that, contrary to this common belief, the algorithm also stabilizes when the number of states per process is one less than the number of processes on the ring (namely $K = N$). Our proof was formalized and checked in PVS, based on [144]. We have given a counter-example showing that $K = N$ is indeed sharp.

One important fact (Lemma 5.3.4) used in our proof is that the nonzero processes do not contain all the possible counter values. By this observation, together with the fact that each process is infinitely often enabled, we can prove that each infinite run of the algorithm will reach a legitimate state. For the case $K > N$, this fact can be proved using the pigeonhole principle, as is done in [172, 167, 144, 124, 106]. For the case $K = N$ in this chapter, we choose the moment that p_N is enabled and fires, which makes $v(N) = v(N-1)$. After that we can apply the pigeonhole principle. Another important fact (Lemma 5.2.2) is that whenever the system reaches a stable state, it will remain in stable states. Thus we have proved the properties for self-stabilization.

Regarding the verification in PVS, we downloaded the PVS code and proof by Qadeer and Shankar. Following their proof steps in PVS, we simply added a new definition of $T(v)$, proved two new lemmas (Lemma 5.3.2 and Lemma 5.3.3), and adapted one lemma as Lemma 5.3.4. The whole verification did not take too much effort. First, we spent a few days to understand the formalism and proof in [144]. Since the PVS system, including PVS libraries, has been updated after 1998, the downloaded PVS proof could not be simply rerun. We made some adaptions to make their PVS proof work again. After that, when we had the idea to prove (as shown in Section 5.2) the algorithm correct under the condition $K = N$, the proof was completely checked in PVS within one day. The files containing our PVS formalization and proofs can be found at the URL `http://www.cwi.nl/~pangjun/stabilization/`.

Part II

Model Checking

Chapter 6

Analysis of a Distributed System for Lifting Trucks

6.1 Introduction

This chapter reports on the analysis of a real-life system for lifting trucks (lorries, railway carriages, buses and other vehicles). The system consists of a number of lifts; each lift supports one wheel of the truck that is being lifted and has its own micro-controller. The controls of the different lifts are connected by means of a network. A special purpose protocol has been developed to let the lifts operate synchronously.

This system has been designed and implemented by a Dutch company, that is specialized in the design of embedded systems. When testing the implementation the developers found three problems. They solved these problems by trial and error, partly because the causes of two of the three problems were unclear. In close cooperation with the developers, we specified the lift system in μCRL. Next, we analyzed the resulting specification with the μCRL tool set and CADP. The three known problems turned up in our specification (which adds to our confidence that the specification is close to the actual implementation). In addition we found a fourth error. This error was unknown and found its way into the implementation of the lift system. We incorporated solutions for these problems in the specification. We have analyzed the μCRL specification that results from the incorporation of the proposed solutions, showing that this specification meets the requirements of the developers.

However, this happened independently of the developers, who decided not to wait for the results of the formal analysis in μCRL and to redesign their implementation based on their own solutions. To distinguish between the two lift systems, we call the first lift system 'original design' and the one with the solutions of the developers 'redesign'.

The developers experienced a new problem in the redesign. Again the reason was unclear. Since the error traces displayed a regular pattern in time, the developers thought modeling exact timing might reveal the reason for this problem. In the μCRL specification, time is abstracted away. We could extend the μCRL

model with exact timing information, but there is no automated verification tool set for timed process algebras. Therefore it was decided to use UPPAAL [111], which is a tool set for validation and model checking of real-time systems.

The UPPAAL model of the redesign was achieved in several steps. First the μCRL model was translated into UPPAAL. Then the UPPAAL model was refined to move it closer to the real system; each lift is split into two components, where one component communicates with the other lifts and the other component can receive input from the environment. The developers' solutions for the aforementioned problems were adopted. After discussions with the developers, exact timing information was added. The requirements for the lift system were formulated in UPPAAL, using its requirement specification language and test automata, and model checked. Using the graphic simulation tool in UPPAAL, we detected the reason for the new problem, which the developers encountered in the redesign. We propose a new solution, which is based on the solution that was already put forward in the analysis of the original design. The UPPAAL model with the new solution satisfies all the requirements.

The developers acknowledge the efficiency and usefulness of formal verification for their redesign. Our solution is being implemented in the new release of the lift system; they are now more confident in the correct functioning of the redesigned lift system.

Outline of the chapter. This chapter is organized as follows. After this introduction, we give an informal specification of the lift system in Section 6.2. Next we discuss the requirements which the system should satisfy in Section 6.3. From Section 6.4 to Section 6.6, we present the analysis of the original design of the lift system in μCRL. From Section 6.7 to Section 6.9, we present the analysis of the redesign of the lift system in UPPAAL. We show that the solutions of the developers do not solve these problems found in the original design completely, while a refined version of our solution in the μCRL specification does. We conclude this chapter in Section 6.10.

6.2 Description of the Lift System

First, we explain the general layout of the lift system (Section 6.2.1). Then we explain the manner in which lift movement is controlled (Section 6.2.2).

6.2.1 Layout of the lift system

The system studied in this chapter consists of an arbitrary number of lifts. Each lift supports one wheel of a vehicle being lifted. The system is operated by means of buttons on the lifts. There are four such buttons on each lift: UP, DOWN, SETREF and AXLE. The system knows three kinds of movements. If the UP or DOWN button of a certain lift is pressed, all the lifts of the system should go up, respectively down. If the UP (or DOWN) button is pressed together with SETREF, only one lift (the one of which the buttons are pressed) should go up

(or down). This allows the operator to adjust the height of a lift to inequalities in the surface of the floor. If the UP or DOWN button is pressed together with the AXLE button, the opposite lifts (and only those) are supposed to move up or down, respectively. This is needed to replace the axle of a truck. As different trucks may have different numbers of wheels, the operator may add or remove lifts to or from the system. We have only studied the first kind of movement.

Normally, the lifts contain a locking pin which is intended to prevent the lift from moving down when motors fail, or oil is leaking from the hydraulic pumps or valves. This pin restricts the movement of the lifts. If one wants to move the lifts over a larger distance this pin has to be retracted. This detail is not taken into account in our specification.

Lift movement is controlled by means of a micro-controller. In real life, the lift controller can adopt eight different states. For our study the following states are important: STARTUP, STANDBY, UP, and DOWN. The meaning of these states will become clear in the course of the discussion.

The controllers of the different lifts belonging to a system are connected to a CAN (Controller Area Network) bus [147] which is interrupted by relays (see Figure 6.1). These relays do not exist in real systems, they are part of the protocol developed by the developers. The different controllers connected to the bus are called *stations*. There is a relay between every pair of adjacent stations and each relay is controlled by the station at its left side.

The CAN bus is a simple, low-cost, multi-master serial bus with error detection capabilities. Multi-master means that all stations can claim the bus at each bus cycle and several stations can claim the bus simultaneously, in which case a non-destructive arbitration mechanism determines which message is transmitted by the bus. A message on the bus is immediately received by all other stations connected to the sending station via closed relays. The CAN protocol does not use addresses.

In the lift system, the user data field of the messages transferred over the bus contains three pieces of information: the position of the sender station, the type of the message, and the (measured) height of the sender's lift. There are two kinds of messages: *state* messages and SYNC messages. State messages report the state of the sender station (e.g. STARTUP, STANDBY, UP, DOWN). SYNC messages initiate physical movement. In response to a SYNC message each station will immediately report its state to the motor of its lift. This means that if the station is in the UP state after a SYNC message, the lift will move up a fixed distance; if the station is in the DOWN state, the lift will move down a fixed distance; and if the station is in STANDBY it will not move.

The system continuously checks the heights broadcast in the messages to determine if they do not differ too much. If there is something wrong an emergency stop is brought about. This is not modeled in our specification as this would increase the number of states of the system tremendously.

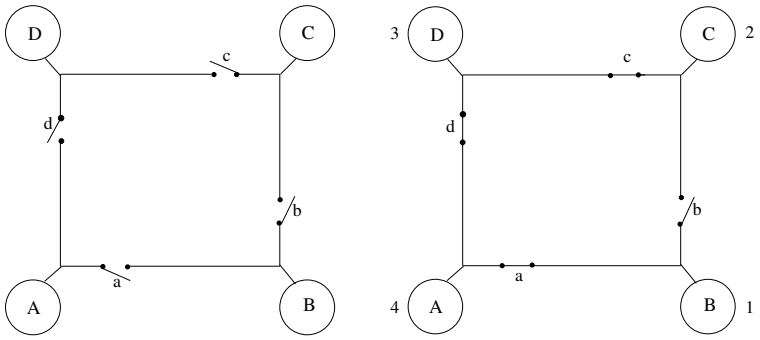


Figure 6.1: State of the relays before (left) and after (right) initialization

6.2.2 Control of lift movement

To assure that all lifts move simultaneously in the same direction, the station initiating a certain movement must verify whether all stations are in the appropriate state before it sends the SYNC message.

The CAN protocol allows several stations to claim the bus at the same time. However, in the lift system, the stations are programmed in such a way that (during normal operation) the stations take turns claiming the bus. They claim the bus in a fixed order (clockwise in Figure 6.1).

To achieve this orderly usage of the bus, each station must know its position in the network. Furthermore, in order to be able to find out whether all stations are in the same state, each station must know how many stations there are in the network. This is achieved by means of a startup phase in which all the stations come to know their position in the network as well as the total number of stations in the network. This startup phase is discussed below:

Startup

When the system is switched on, all the relays are open (see the left part of Figure 6.1).

In the startup phase two things might happen to a station:

- The SETREF button of that station might be pressed. In this case the station will initiate the startup phase as follows:

  1. it stores that it has position 1;
  2. it adopts the STARTUP state;
  3. it closes its relay;
  4. it broadcasts a STARTUP message;
  5. it opens its relay, this guarantees that this station will only receive a STARTUP message when all stations have determined their positions;

6. it waits for a STARTUP message;
7. it stores the position of the sender of that message as the number of stations in the network;
8. it adopts the STANDBY state;
9. it broadcasts this state.

- The station might receive a STARTUP message from another station. In this case:

1. it adds 1 to the position of the sender of that message and stores this as its own position;
2. it stores its own position as the number of stations in the network;
3. it adopts the STARTUP state;
4. it closes its relay;
5. it sends a STARTUP message (note that unlike the previous part the station does not open its relay, it will receive all subsequent STARTUP messages);
6. – if it receives another STARTUP message it stores the position of the sender of that message as the number of stations in the network;
   – if it receives a STANDBY message it adopts the STANDBY state (if the station has position 2 it will in addition initiate normal operation by broadcasting a STANDBY message).

Assume, for example that in the system of Figure 6.1 the SETREF button of station B is pressed. The station of this lift gets position 1. It closes the relay between B and C, broadcasts a STARTUP message, and opens this relay again. The STARTUP message from B is received by only one station (C). This station draws the conclusion that it has position 2. It subsequently closes the relay to D and broadcasts a STARTUP message. This message is received by only one station (D). This station draws the conclusion that it has position 3, closes the relay to A and sends a STARTUP message. This message is received by A and C. C draws the conclusion that now there are three stations in the network. A draws the conclusion that it has position 4, closes the relay to B and broadcasts a STARTUP message. This message is received by B, C, and D. C and D draw the conclusion that now there are four stations in the network. Station B draws the conclusion that the circle is completed. It stores the position of the sender of that message (4) as the number of stations in the network, adopts the STANDBY state and initiates normal operation by sending a STANDBY message. This message is received by C, D, and A which adopt the STANDBY state in response.

The result is that all stations are connected in the manner pictured in the right part of Figure 6.1, that all stations know how many stations there are in the network and what their position is, and that all stations are in STANDBY. Normal operation starts when station 2 broadcasts its state.

Normal operation

During normal operation, the first station (with position 1) broadcasts its state and height, then the next station broadcasts its state and height and so on, until the last station has broadcast its state and height, after which the first station starts again.

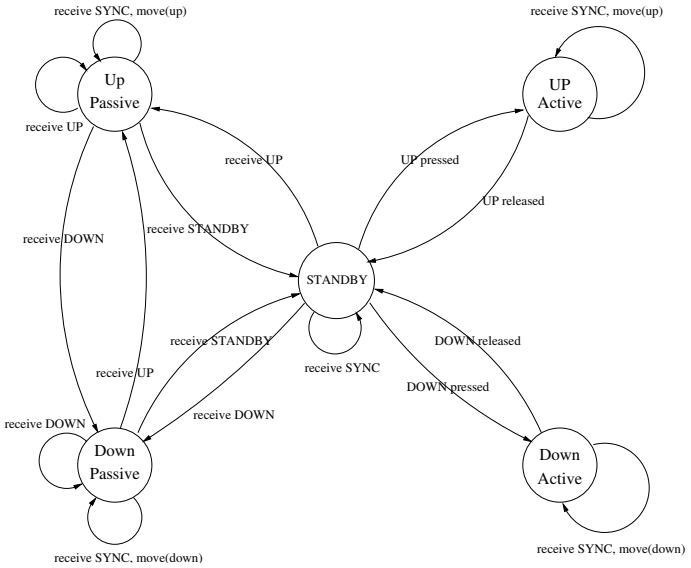


Figure 6.2: State transitions of an individual lift during normal operation

The transition diagram of each lift during normal operation is sketched in Figure 6.2.[1] Initially all stations are in STANDBY. A station in STANDBY changes to another state if one of its buttons is pressed or if it receives a message with another state. The station that is initiating a certain change (i.e. when it is in STANDBY and a button is pressed) is called the active station. All other stations are passive. If the UP or DOWN button of a certain lift is pressed and its station is in STANDBY, that station becomes active and changes its state to UP or DOWN, respectively. When a passive station receives a state message, it adopts the state in that message. An active station does not change its state in response to state messages. The state of an active station changes only if the pressed button is released. In that case its state changes to STANDBY and the station becomes passive again.

As said, physical movement is initiated by a SYNC message. In order to assure that all lifts move in the same direction, the active station will count the number of messages that contain the intended state. The active station will send a SYNC message if and only if it has counted enough messages with the

[1]Some actions of pressing or releasing a button are not represented in this figure, since those actions do not make any state transition of a lift during normal operation phase.

right state (i.e. all the other stations are in the same state as itself), when it is its turn to use the bus.

Assume, for example, that all stations are in STANDBY and that the UP button of station 4 (in the right part of Figure 6.2) is pressed. This station adopts the UP state. When it is this station's turn to use the bus (getting a message from its predecessor), it will broadcast its state; in response the other stations will adopt the UP state too. Next, it is station 1's turn to use the bus. This station will broadcast its state (which is UP). The message from station 1 is received by all other stations, among which the active station 4. As the state in the message is the same as that of the active station 4, this latter station will count this message. In the next two cycles station 2 and station 3 claim the bus in turn and broadcast their states (UP), both messages are counted by station 4. So, station 4 will have received the right number of UP messages when it is its turn to use the bus again and it will send a SYNC message to initiate physical movement.

6.3 Requirements

There are five requirements for the lift system, that have been formulated in cooperation with the developers. Each requirement describes a different aspect of the system's behavior.

1. *Deadlock freeness*: the lift system never ends up in a state where it cannot perform any action.
2. *Liveness I*: it is always possible for the system to get to a state in which pressing the UP or DOWN button of any lift will yield the appropriate response.
3. *Liveness II*: if exactly one UP or exactly one DOWN button is pressed and not released, then all the lifts will (eventually) move up or down, respectively.
4. *Safety I*: if one of the lifts moves, all the other lifts should simultaneously move in the same direction.
5. *Safety II*: if the lifts move, an appropriate button was pressed. In other words, the lifts will not move if no one has pressed a button.

The two liveness requirements make sure that buttons can always be pressed and in response the lifts will always move. The two safety requirements make sure that the system will move properly.

6.4 μCRL Model of the Original Design

We specified the lift system in μCRL. As is demonstrated by this case study, this language is useful as a tool to analyze embedded controllers.

As we described in Section 6.2, our specification is an abstraction of the real system. Such details as the locking pins, the parameter of height containing in the messages, and the checking of the height broadcast in messages are not modeled in our specification. And we only studied two kinds of movement of the lift system: If the UP or DOWN button of one lift is pressed, all the lifts of the system should go up, respectively down. The initial specification for system with three lifts is given at `http://www.cwi.nl/~pangjun/lift/`. Here we only highlight some parts of this specification. The part on data types is discussed in Section 6.4.1, and the part on processes in Section 6.4.2.

6.4.1 Data types

Obviously we need to represent the physical structure of the lift system. This is done by means of the sort *Address*. The constructors of this data type consist of identifiers (one for each station). The functions *suc* and *pre* yield the identifiers of the neighbors in the circle. *suc* yields the one at the right-hand side, *pre* yields the one at the left-hand side (see Figure 6.1). Because of the similarity in structure, we use this data type also to represent the position of a station. We specify the sort *Address* with three elements below:

sort Address
func 1, 2, 3: →Address
map suc: Address→Address
pre: Address→Address
eq: Address×Address→Bool
rew suc(1)=2 pre(1)=3
suc(2)=3 pre(2)=1
suc(3)=1 pre(3)=2
eq(1,1)=T eq(1,2)=F eq(1,3)=F
eq(2,2)=T eq(2,1)=F eq(2,3)=F
eq(3,3)=T eq(3,1)=F eq(3,2)=F

This data type is also used to identify the position of relays. Relay n is the one between the station with address n and the station with address $suc(n)$; it is controlled by the station at the left side (addressed as n).

To model the bus, we must record which relays are closed. This is done by means of the sort *Alist*, which is a list of addresses. The constructors of this sort are *ema* and *insert*. *ema* stands for an empty list. *insert* constructs a new list by inserting an address into a list. The function $remove(a, A)$ removes all the occurrences of the address a from the list A. Function $test(a, A)$ tells us whether the address a is in list A. The function $empty(A)$ is used to judge whether a list is empty, or not. $if(b, A, A')$ is an auxiliary function to specify *test* and *reset*, where b is a data term of sort *Bool*. It is used to simulate conditional equations, meaning that if b holds then A is selected, otherwise A'. And the concatenation of two lists is represented by the function $conc(A, A')$. The function $Addresses(A, a)$ is used to get the list of all stations connected to the station a via list A of closed relays. a is excluded in the result. Functions

Addresses-up, *Addresses-down*, *Addresses-up-aux* and *Addresses-down-aux* are used to help the specification of *Addresses*.

sort Alist
func ema: →Alist
insert: Address×Alist→Alist
map remove: Address×Alist→Alist
test: Address×Alist→Bool
empty: Alist→Bool
if: Bool×Alist×Alist→Alist
conc: Alist×Alist→Alist
Addresses: Alist×Address→Alist
Addresses-up: Alist×Address×Address→Alist
Addresses-down: Alist×Address×Address→Alist
Addresses-up-aux: Bool×Bool×Alist×Address×Address→Alist
Addresses-down-aux: Bool×Bool×Alist×Address×Address→Alist
var a, a': Address
A, A': Alist
b: Bool
rew remove(a,ema)=ema
remove(a,insert(a',A))=if(eq(a,a'),remove(a,A),insert(a',remove(a,A)))
test(a,ema)=F
test(a,insert(a',A))=if(eq(a,a'),f,test(a,A))
empty(ema)=T
empty(insert(a,A))=F
if(T,A,A')=A
if(F,A,A')=A'
conc(ema,A)=A
conc(insert(a,A),A')=insert(a,conc(A,A'))
Addresses(A,a)=conc(Address-up(A,a,a),Address-down(A,a,a))
Addresses-up(A,a,a')=Addresses-up-aux(test(a,A),eq(suc(a),a'),A,a,a')
Addresses-down(A,a,a')=
Addresses-down-aux(eq(pre(a),a'),test(pre(a),A),A,a,a')
Addresses-up-aux(T,T,A,a,a')=insert(suc(a),ema)
Addresses-up-aux(T,F,A,a,a')=insert(suc(a),Address-up(A,suc(a),a'))
Addresses-up-aux(F,b,A,a,a')=ema
Addresses-down-aux(T,b,A,a,a')=ema
Addresses-down-aux(F,T,A,a,a')=
insert(pre(a),Addresses-down(A,pre(a),a'))
Addresses-down-aux(F,F,A,a,a')=ema

In our model, the following states of stations are specified by a sort *State*: STANDBY, UP, DOWN, STARTUP and SYNC. The state SYNC is not really a state, but it can be broadcast in a message instead of the states. This kind of message is used to synchronize the physical movement of all the lifts.

sort State

func STANDBY, UP, DOWN, STARTUP, SYNC: →State

The messages traveling in the network are specified by a sort *Message*. A message has the form $mes(a, s)$: a is the position of the station sending the message and s is the state of the sending station. By using the functions *getaddress* and *getstate*, we can get the position, respectively the state of the station.

sort Message
func mes: Address×State→Message
map getaddress: Message→Address
getstate: Message→State
var a: Address s: State
rew getaddress(mes(a,s))=a
getstate(mes(a,s))=s

6.4.2 Processes

In this section, we focus on the process part of our specification. The bus and the stations are both modeled as separate processes.

The specification of the bus poses two problems. First, we must represent which relays are open and which ones are closed. This is done by parameterizing the bus process with an *Alist* R of identifiers of all closed relays. If a station closes a relay, the identifier of the relay is added to this list. If it opens a relay, the identifier of the relay is removed from this list. This is achieved with the help of two actions $r\_open\text{-}relay(n)$ and $r\_close\text{-}relay(n)$.

Second, we must represent the transportation of messages over the bus. In the system, a message put on the bus by one station is received by all the other stations connected to the sending station via closed relays. This is modeled by means of a delivery process (*Deliver*) parameterized with an Alist A of stations that have yet to receive the message. After accepting a message from a station with the action $r\_stob(m, a)$ (receive message m from station a to the bus), the bus process moves to the delivery phase, provided that the list R is not empty. This phase consists of a number of cycles. In each cycle, the message is delivered to one station in list A by the action $s\_btos(m, a)$ (send message m from the bus to station a) and then the next cycle is entered with the station a removed from list A. If the last station is removed, the bus process returns to the *Bus* phase. The *Deliver* process has R as one of its parameters; this is needed to restart the *Bus* process after the delivery phase with the correct list of closed relays. In the delivery phase, the bus does not accept messages from the stations, which ensures that a message broadcast by a station is received by all stations connected to it before the next station can send a message.

act r_stob, s_btos: Message×Address
r_open-relay, r_close-relay: Address

proc Bus(R:Alist) =
$\sum_{\text{mes:Message}} \sum_{\text{a:Address}}$ r_stob(mes,a)·

```
        (Bus(R)◁empty(Addresses(R,a))▷Deliver(mes,R,Addresses(R,a)))
        +∑a:Address r_open-relay(a)·Bus(remove(a,R))
        +∑a:Address r_close-relay(a)·Bus(insert(a,R))

proc    Deliver(mes:Message, R:Alist, A:Alist) =
        ∑a:Address s_btos(mes,a)·
        (Bus(R)◁empty(remove(a,A))▷Deliver(mes,R,remove(a,A)))
        ◁test(a,A)▷δ
        +∑a:Address r_open-relay(a)·Deliver(mes,remove(a,R),A)
        +∑a:Address r_close-relay(a)·Deliver(mes,insert(a,R),A)
```

The actions *r_stob* and *s_btos* are intended to communicate with the actions *s_stob* (send a message from a station to the bus) and *r_btos* (receive a message from the bus to a station) into *c_stob* and *c_btos*, respectively. Likewise, the actions *r_open-relay* and *r_close-relay* are synchronized with the actions *s_open-relay* and *s_close-relay*.

```
comm    s_stob | r_stob = c_stob
        s_btos | r_btos = c_btos
        s_open-relay | r_open-relay = c_open-relay
        s_close-relay | r_close-relay = c_close-relay
```

After modeling the bus process, we come to the specification of the lift controller. The following actions are associated with the buttons of a lift. They do not represent all physical actions of pressing a button of the real system. Only those actions of pressing a button which have effect on the behavior of the system are modeled in our specification (see Figure 6.2). For example, in the normal operation phase, a SETREF button can be physically pressed. Since in this phase a station does not respond to this action, the action *setref* cannot occur according to our specification of the normal operation phase (see the specification of *Lift2*). Leaving out these actions does not affect our verification. The action of outputting state s of station n to the motor input is represented as the action $move(n, s)$.

```
act     setref, up, down, released: Address
        move: Address×State
```

The control of the lift system movement is divided into two phases. Initially, all relays are open. In the first phase (startup phase), the network connection is set up, and each station gets to know its position and the number of stations in the network. In the second phase (normal operation phase), the stations claim the bus in a fixed order and the physical movement of the system can be initiated. Each lift process is parameterized with an address n, which identifies the station.

The behavior of a station in the startup phase is modeled by two processes, *Lift0* and *Lift1*. Initially, all stations are in *Lift0*. *Lift0* specifies the initial behavior of a station. In this phase, the SETREF button of a station can be pressed, or a station can receive a STARTUP message from another one. *Lift1*

models how the stations with a position greater than 1 get to know the number of stations in the network. The parameter m is added to *Lift1* to record the position of a station. The parameter *nos* is used to remember the number of the stations.

The station of which the SETREF button is pressed gets position 1. It closes its relay with the action $s\_close\text{-}relay(n)$ and broadcasts a STARTUP message. Next, it opens its relay with the action $s\_open\text{-}relay(n)$ and waits for a STARTUP message. When it gets the STARTUP message, it responds by changing its state to STANDBY and broadcasting its state, then it goes into the normal operation phase, which is modeled as *Lift2*. If a station (not the one on which the SETREF button is pressed) gets a STARTUP message, it adds 1 to the position of the message's sender and stores this both as its m and as its *nos*. It adapts the STARTUP state, closes its relay and broadcast its own state. Next, it moves into *Lift1*, where it can change its own *nos* according to the position of the STARTUP messages it receives. In the phase of *Lift1*, each station gets to know the number of stations in the network by the position of the last STARTUP message. When a station with a position greater than 1 gets a STANDBY message, it adopts its states to STANDBY and goes into process *Lift2*. If it is its turn to claim the bus (when it receives a message from its predecessor), it also broadcasts a STANDBY message. In this way, the startup phase is finished and all stations are connected to one bus. The processes *Lift0* and *Lift1* are specified as follows:

proc Lift0(n:Address)=
setref(n)·s_close-relay(n)· s_stob(mes(1,STARTUP),n)· s_open-relay(n)·
$\sum_{\text{mes:Message}}$r_btos(mes,n)·
(s_stob(mes(1,STANDBY),n)·Lift2(n,1,getaddress(mes),STANDBY)
◁eq(getstate(mes),STARTUP)▷δ)
+$\sum_{\text{mes:Message}}$ r_btos(mes,n)·
(s_close-relay(n)·s_stob(mes(suc(getaddress(mes)),STARTUP),n)·
Lift1(n,suc(getaddress(mes)),suc(getaddress(mes)))
◁eq(getstate(mes),STARTUP)▷δ)

proc Lift1(n:Address, m:Address, nos:Address)=
$\sum_{\text{mes:Message}}$r_btos(mes,n)·
(Lift1(n,m,getaddress(mes))
◁eq(getstate(mes),STARTUP)▷
((s_stob(mes(1,STANDBY),n)·Lift2(n,m,nos,STANDBY)
◁eq(getaddress(mes),pre(m))▷
Lift2(n,m,nos,STANDBY))
◁eq(getstate(mes),STANDBY)▷δ))

Note that during the startup phase, all the stations expect to receive either a STARTUP message or a STANDBY message, otherwise it will result into a deadlock. This can be model checked later on.

The behavior of a station during normal operation is specified by means of two processes (*Lift2* and *Lift3*). The parameter s is used to record the state

of the station. In this phase, the stations broadcast their messages in a fixed order. A station knows that it is its turn to claim the bus when it receives a message from its predecessor. In both *Lift2* and *Lift3*, a station responds to an incoming SYNC message by immediately outputting its state to the motor input with the action $move(n, s)$. *Lift2* models the behavior of a station that is passive or in STANDBY. In this phase, a station will respond to a state message by adopting the state in the message. When a station gets the turn to claim the bus, it adopts the state in the received message and broadcasts it. In addition, a station in STANDBY will respond to an action of pressing a button. It adopts the corresponding state and becomes active (*Lift3*). *Lift3* models the behavior of an active station. The parameter *count* is used to count the number of stations that are in the same state as this active one. This counter is initiated with the number of stations in the network. Each time the active station receives a message with the same state as itself, the counter is decreased. When the active station gets the turn to use the bus, it will determine whether it has received enough messages of the right type (i.e. whether its counter equals 2 and the state of the message of its predecessor is the same as the state of itself). If so, it will send a SYNC message, output its state to the motor, broadcast its own state and reset the counter to the number of the stations in the network. If not, it will broadcast its state and reset its counter. When the pressed button on the lift is released (modeled by *released*(n)), the active station returns to STANDBY.

proc Lift2(n:Address, m:Address, nos:Address, s:State)=
(up(n)·Lift3(n,m,nos,UP,nos)+down(n)·Lift3(n,m,nos,DOWN,nos))
◁eq(s,STANDBY)▷δ
+$\sum_{\text{mes:Message}}$r_btos(mes,n)·
(move(n,s)·Lift2(n,m,nos,s)
◁eq(getstate(mes),SYNC)▷
(s_stob(mes(m,getstate(mes)),n)·Lift2(n,m,nos,getstate(mes))
◁eq(getaddress(mes),pre(m))▷
Lift2(n,m,nos,getstate(mes))))

proc Lift3(n:Address, m:Address, nos:Address, s:State, count:Address)=
released(n)·Lift2(n,m,nos,STANDBY)
◁not(eq(s,STANDBY))▷δ
+$\sum_{\text{mes:Message}}$r_btos(mes,n)·
(move(n,s)·Lift3(n,m,nos,s,count)
◁eq(getstate(mes),SYNC)▷
((s_stob(mes(m,SYNC),n)·move(n,s)·
s_stob(mes(m,s),n)·Lift3(n,m,nos,s,nos)
◁eq((getstate(mes),s)∧eq(count,2)▷
s_stob(mes(m,s),n)·Lift3(n,m,nos,s,nos))
◁eq(getaddress(mes),pre(m))▷
(Lift3(n,m,nos,s,pre(count))
◁eq(getstate(mes),s)▷
Lift3(n,m,nos,s,count))))

By putting n *Lift0* processes and one *Bus* process in parallel, we model a system with n lifts ($n \geq 2$) as follows:

init $\tau_I \partial_H$ (Bus(ema) ‖ Lift0(1) ‖ Lift0(2) ‖ ... Lift0(n))

where I denotes the set {*c_stob, c_btos, c_open-relay, c_close-relay*} and H denotes the set {*s_open-relay, r_open-relay, s_close-relay, r_close-relay, s_stob, r_stob, s_btos, r_btos*}. Initially, the list of identifiers of closed relays is empty.

The encapsulation operator ∂ enforces the actions *s_open-relay, s_close-relay, s_btos and s_stob* to occur in communication with the actions *r_open-relay, r_close-relay, r_btos and r_btos*, respectively. To analyze the specification, all internal actions like the communication between bus and stations can be abstracted away, which is achieved by converting them into the τ action with the help of the τ operator.

6.5 Analysis the Original Design

In our study, the μCRL tool set was used to generate a labeled transition system from the μCRL specification. This LTS was analyzed with the CADP tool set. When an error was found the specification was modified and the modified specification was analyzed again.

It is interesting to see that the problems were being detected in a rather unordered fashion. For instance problem 1 showed itself by visualizing the system behavior for a system with 3 lifts after hiding all communications to and from the bus and reducing the resulting LTS modulo branching bisimulation. The first sign of the problem was that not all internal actions had been removed. Trying to understand the reason for this uncovered the precise problem quickly.

Four errors were found in the original design. We discuss these problems separately and propose solutions (Sections 6.5.1–6.5.4). The modified specification resulting from the incorporation of our suggestions was shown to meet the requirements (Section 6.6).

6.5.1 Problem 1

The first problem occurs if in the startup phase station 2 sends a STARTUP message before the relay between station 1 and 2 is opened (see Figure 6.1 and the example in Section 6.2.2). This STARTUP message is received by station 1, which will draw the erroneous conclusion that the circle is completed. From this all sorts of errors may occur (depending on the exact timing). For example, station 1 sends the STANDBY message, which initiates normal operation, while the relay between station 1 and station 2 is opened, no station will receive this message. The startup phase will continue as intended until station 1 receives the STARTUP message from the last station in the system. As this is unexpected it will result in a deadlock.

The developers had spotted this problem in the testing phase, but they were unaware of its cause. They had solved the problem by adding delays before sending a STARTUP message.

In our revised specification, the delay is modeled by the communication of two actions, *s_sync* and *r_sync*. This is enough to make sure that station 2 waits till the relay between station 1 and station 2 is closed, before it sends a STARTUP message.[2]

Our experiments have indicated that this solves the problem adequately (if the delay is long enough to make sure that the relay between station 1 and station 2 is opened before station 2 sends the STARTUP message). The developers implemented our solution and confirmed that it suffices to delay only the second STARTUP message. The main modification is made in the definition of process *Lift0*. It is shown together with the solution to the second problem at the end of Section 6.5.2.

6.5.2 Problem 2

The second problem occurs if the SETREF buttons of two lifts are pressed at almost the same time. This may result in different lifts moving in different directions. Assume that the system consists of four lifts (A, B, C, D) and that the SETREF buttons of A and C are pressed at the same time (see Figure 6.1). Both A and C send a STARTUP message, which is received by respectively B and D. The relays between A and B, and between C and D are opened again. Next B closes the relay between B and C and then B broadcasts a STARTUP message. This message is received by C. Station C draws the conclusion that the circle is completed and initiates normal operation. At the same time D closes the relay between D and A and sends a STARTUP message that is received by A, after which A initiates normal operation. The result is that there are two independently operating networks, one consisting of A and D; the other of B and C. There is no way in which the stations or the bus can prevent or detect this situation.

A similar situation may occur if the SETREF buttons of two neighboring lifts (say A and B) are pressed. Assume that B sends a STARTUP message before A does so. The message from B is received by C. Assume that next the relay between B and C is opened again and that A subsequently sends its startup message. Station B receives it, draws the conclusion that the circle is completed, and initiates normal operation. Station A opens the relay between A and B, and after receiving a STARTUP message from D it finishes the startup phase. The result is that B is isolated from the rest of the network. Again the system will not detect this error.

We have modified the specification in such way that it is impossible to initiate the system by pressing the SETREF button of several lifts at once. The process *Setref_monitor* is defined to prevent that in the startup phase more than one SETREF button is pressed at different lifts at the same time. The action $setref(n)$ in *Lift0* is replaced by the action $s\_init(n)$, which applies a lock on the monitor. After station 1 gets a STARTUP message, it releases the lock by the action *s_stable*. During the period when the monitor is locked, pressing

[2]The operator ∂ can enforce the two actions *s_sync* and *r_sync* to occur in communication with each other, and not on their own.

the SETREF button at another station does not have an effect on the whole lift system.

comm s_init | r_init = c_init
s_stable | r_stable = c_stable
s_sync | r_sync = c_sync

proc Setref_monitor =
$\sum_{n:Address}$r_init(n)·r_stable·Setref_monitor

proc Lift0(n:Address)=
s_init(n)·s_close-relay(n)·s_stob(mes(1,STARTUP),n)·
s_open-relay(n)·s_sync·
$\sum_{mes:Message}$r_btos(mes,n)·
(s_stable·s_stob(mes(1,STANDBY),n)·
Lift2(n,1,getaddress(mes),STANDBY)
$\lhd$eq(getstate(mes),STARTUP)$\rhd\delta$)
+$\sum_{mes:Message}$r_btos(mes,n)·
(s_close-relay(n)·
(r_sync·s_stob(mes(2,STARTUP),n)·Lift1(n,2,STARTUP)
$\lhd$eq(getaddress(mes),1)$\rhd$
s_stob(mes(suc(getaddress(mes)),STARTUP),n)·
Lift1(n,suc(getaddress(mes)),STARTUP))
$\lhd$eq(getstate(mes),STARTUP)$\rhd\delta$)

The developers did not implement this solution, but chose to emphasize in the manual that it is important to make sure that in the initial phase the SETREF button of only one lift is pressed. We also took this assumption into our μCRL model. Given the chosen bus it seems impossible to solve this problem satisfactorily. As a result of our analysis, the implementation of the lift system was adapted. At initialization of the system, a random identifier is created to minimize the risk that more than one independent network comes into existence.

6.5.3 Problem 3

The third problem occurs if during normal operation a button is pressed and released at an inappropriate moment. Suppose that in a network of four stations all stations are STANDBY, and that the DOWN button of station 1 is pressed, as a result of which it acquires the DOWN state. When it is the turn of station 1 to use the bus it broadcasts the DOWN state, and all other stations adopt this state in response. Suppose that the DOWN button of station 1 is released after station 3 sends its DOWN message, but before station 4 has done this. As a result station 1 returns to the STANDBY state. In this state it adopts the state of all state messages it receives, so when station 4 sends its state message it adopts the DOWN state. We now have the situation that all stations are in DOWN state, but there is no active station. This means that they will remain in that state until the system is shut down.

This problem was independently discovered by the developers when testing the system. They tried to use flags to solve this problem, more discussion can be found in the analysis of the redesign. Our solution to this problem is simple. We let the station wait to become passive after the button is released, until it is that station's turn to use the bus. This is the solution incorporated in our modified specification. The main modification is made in the definition of process *Lift3*. It is shown together with the solution to the fourth problem at the end of Section 6.5.4.

6.5.4 Problem 4

The fourth problem occurs when during normal operation two (UP or DOWN) buttons on different lifts are pressed at almost the same time. Suppose there are four stations in the network and that the DOWN buttons of station 1 and station 2 are pressed at the same moment, as a result of which both stations become active. Assume that it is station 1's turn to use the bus. It sends a DOWN message, and in response station 3 and station 4 adopt the DOWN state. In turn stations 2, 3 and 4 send a DOWN message. When it is the turn of station 1 to use the bus again, it has counted three DOWN messages, so it sends SYNC (after which all lifts move down), and as the DOWN button is still pressed it then sends DOWN. Now it is station 2's turn and as this station is active and has counted three DOWN messages it sends a SYNC message. Suppose (and now comes the problem) that the DOWN button of station 1 is released after station 1 has sent the DOWN message and before station 2 sends the SYNC message. As a result station 1 is in STANDBY when it receives the SYNC message, and its lift remains at the same height while the others move down.

A similar problem occurs if the DOWN button of station 2 is released just after station 3 has sent its DOWN message but before station 1 sends its SYNC message. In this case lift 2 will remain at the same height while the others move down.

This problem was not known to the developers and found its way into the implementation. We propose to solve this problem by allowing a station to become active only when it is its turn to use the bus and only when at that moment there is no other station active. In the revised specification, a *Bool* parameter is added into the definition of process *Lift2* to mark the station that wants to be active. It is set true when one button of the station is pressed. When it is the marked station's turn to use the bus, but it finds there is already an active station in the system, the marked station fails to be active. It adopts the state of the received message and broadcasts the message. Our experiments indicate that this solves the problem adequately.

proc Lift2(n:Address, m:Address, nos:Address, s:State, c:Bool)=
(up(n)·Lift2(n,m,nos,UP,nos,T)+down(n)·Lift2(n,m,nos,DOWN,nos,T))
◁eq(s,STANDBY)▷δ
+$\sum_{\text{mes:Message}}$r_btos(mes,n)·
(move(n,s)·Lift2(n,m,nos,s,c)

```
    ◁eq(getstate(mes),SYNC)▷
        (((s_stob(mes(m,s),n)·Lift3(n,m,nos,s,nos)
        ◁ eq(getstate(mes),STANDBY)▷
        s_stob(mes(m,getstate(mes)),n)·Lift2(n,m,nos,getstate(mes),F))
      ◁c▷
      s_stob(mes(m,getstate(mes)),n)·Lift2(n,m,nos,getstate(mes),F))
    ◁eq(getaddress(mes),pre(m))▷
    (Lift2(n,m,nos,s,c)◁c▷Lift2(n,m,nos,getstate(mes),c))))

proc Lift3(n:Address, m:Address, nos:Address, s:State, count:Address)=
    released(n)·Lift3(n,m,nos,STANDBY,nos)
    ◁not(eq(s,STANDBY))▷δ
    +∑mes:Message r_btos(mes,n)·
      ((s_stob(mes(m,STANDBY),n)·Lift2(n,m,nos,STANDBY,F)
      ◁eq(s,STANDBY)▷
        (s_stob(mes(m,SYNC),n)·move(n,s)·
        s_stob(mes(m,s),n)·Lift3(n,m,nos,s,nos)
        ◁eq(getstate(mes),s)∧eq(count,2)▷
        s_stob(mes(m,s),n)·Lift3(n,m,nos,s,nos)))
    ◁eq(getaddress(mes),pre(m))▷
      (Lift3(n,m,nos,s,pre(count))
      ◁eq(getstate(mes),s)▷
      Lift3(n,m,nos,s,count)))
```

After these four problems were all repaired, no more problems have been found. We showed by means of model checking that that this modified specification meets the requirements in the next section. The specification that was model checked is given at `http://www.cwi.nl/~pangjun/lift/`.

6.6 Verification with CADP

6.6.1 Expressing the requirements

There are five requirements for the lift system. The first property is a universal one: *deadlock freeness*. In the regular alternation-free μ-calculus syntax (see Section 2.4) this is specified as follows:

P1 $[\mathsf{T}^*]\,\langle\mathsf{T}\rangle\,\mathsf{T}$

stating that every reachable state has at least one successor.

The second property is that of *Liveness I*, which means that buttons on the stations can eventually be pressed. The regular alternation-free μ-calculus code is given below,[3] where '$\star$' is universally quantified on the sort *Address*:

P2.1 $[(\neg \mathsf{up}(.))^*]\,\langle(\neg \mathsf{up}(.))^*\cdot \mathsf{up}(\star)\rangle\,\mathsf{T}$

P2.2 $[(\neg \mathsf{down}(.))^*]\,\langle(\neg \mathsf{down}(.))^*\cdot \mathsf{down}(\star)\rangle\,\mathsf{T}$

[3] "." is used to match any character in regular expressions.

It states that all fair execution sequences leading to an UP or DOWN action after zero or more transitions.

The property of *Liveness II* is expressed below. We use '$\star$' to indicate the address of the lift on which the UP (or DOWN) button is pressed, it is universally quantified on the sort *Address*.

P3.1 $[(\neg(\mathsf{up}(.)|\mathsf{down}(.)))^*\cdot\mathsf{up}(\star)]$
$\mu Y.\langle\mathsf{T}\rangle\ \mathsf{T} \wedge [(\neg(\mathsf{up}(.)|\mathsf{down}(.)|\mathsf{released}(\star)|\mathsf{move}(.,\text{UP})))]\ Y$

P3.2 $[(\neg(\mathsf{up}(.)|\mathsf{down}(.)))^*\cdot\mathsf{down}(\star)]$
$\mu Y.\langle\mathsf{T}\rangle\ \mathsf{T} \wedge [(\neg(\mathsf{up}(.)|\mathsf{down}(.)|\mathsf{released}(\star)|\mathsf{move}(.,\text{DOWN})))]\ Y$

It says that in any execution sequence containing only one button-pressed action, and containing no button-released action of the pressed button, the system always begins to move.

The fourth property of our specification is *Safety I*. It says that if one of the lifts moves, all the other lifts should not move in the opposite direction. What is more, to keep the trucks in balance, all lifts have to move in the same direction. Note that *Safety I* also requires that all lifts should move at (almost) the same moment, the CAN bus can guarantee that a SYNC message on the bus is immediately received by all other stations connected to the sending station via closed relays, we did not take this into account. To formalize this property, any order of the lifts' movements must be dealt with carefully. This means that the size of the formula grows in a factorial fashion with respect to the number of lifts. To solve this problem, we split the formula into pieces which can be checked by the model checker Evaluator. Taking a lift system with three stations as an example, one piece of this property is specified as follows:

P4 $[\neg(\mathsf{move}(1,\text{UP})|\mathsf{move}(2,\text{UP})|\mathsf{move}(3,\text{UP}))^*\cdot$
$\mathsf{move}(1,\text{UP})\cdot$
$\neg(\mathsf{move}(1,\text{UP})|\mathsf{move}(2,\text{UP})|\mathsf{move}(3,\text{UP}))^*\cdot$
$\mathsf{move}(2,\text{UP})\cdot$
$\neg(\mathsf{move}(1,\text{UP})|\mathsf{move}(2,\text{UP})|\mathsf{move}(3,\text{UP}))^*\cdot$
$(\mathsf{move}(3,\text{DOWN})|\mathsf{move}(3,\text{STANDBY}))]\ \mathsf{F}$

The above code says that in all paths, lift 1 is the first to move up, after that, no movement of the other stations, and then lift 2 moves up, also no movements of other stations following; moreover, the action of lift 3 moving down (or not moving) always results in a state where F holds. Equivalently, as long as lift 1 and lift 2 move up, lift 3 cannot move down or remain at the same height. The other possibilities of the movement of stations can be specified similarly.

The fifth property of *Safety II* states that if no UP or DOWN button is pressed, then the system cannot move UP or DOWN. The following shows the code in the regular alternation-free μ-calculus.

P5.1 $[(\neg\mathsf{up}(.))^*\cdot\mathsf{move}(.,\text{UP})]\ \mathsf{F}$

P5.2 $[(\neg\mathsf{down}(.))^*\cdot\mathsf{move}(.,\text{DOWN})]\ \mathsf{F}$

This should be read as follows: if an execution sequence does not contain button-pressed action, then in the resulting state the stations cannot move up or down.

Number of lifts	States	Transitions
2	383	636
3	7,282	18,957
4	128,901	390,948
5	2,155,576	8,287,715

Table 6.1: Labeled transition system dimensions

6.6.2 Verifying the modified specification

All five requirements stated in Section 6.3 were shown to be satisfied by our modified μCRL specification of the lift system with respectively 2, 3, 4 and 5 lifts. The dimensions of the generated LTSs are summarized in Table 6.1. For each of the lift systems, the numbers of states and transitions of the generated LTS are given. The size of the generated LTS quickly increases with the number of the lifts. This is due to the fact that buttons on each lift can be pressed in any arbitrary order. Generation and model checking were performed on a 1.4 GHz AMD Athlon$^{\text{TM}}$ Processor with 512 Mb memory.

Owing to a distributed state space generation algorithm [22], we can generate the LTS for a system with six lifts on a cluster at CWI. The generated LTS has around $33,900,000$ states and $165,000,000$ transitions, which is too large to serve as an input to the model checker. Hence, the five requirements were not checked on this LTS.

6.7 UPPAAL Model of the Redesign

The developers of the original design in μCRL decided not to wait for the results of the formal analysis and redesigned their implementation based on their own solutions.

The developers experienced a new problem in the redesign. Again the reason was unclear. Since the error traces displayed a regular pattern in time, the developers thought modeling exact timing might reveal the reason for this problem. In the μCRL specification, time is abstracted away. We could extend the μCRL model with exact timing information, but there is no automated verification tool set for timed process algebras. Therefore it was decided to use UPPAAL [111], which is a tool set for validation and model checking of real-time systems.

UPPAAL is a tool set for validation and model checking of real-time systems, which are modeled as networks of timed automata [3] extended with global shared variables. It consists of a number of tools including a graphic editor for system description, a simulator and a model checker. The idea of the UPPAAL tool set is to model a system using timed automata, simulate it and then verify properties of the system. During the design phase, the graphic simulator is used intensively to validate the dynamic behavior of each design sketch, in particular for fault detection, and later on for debugging the generated diagnostic traces. The verifier mainly checks for invariants and reachability properties. It does so

by exploring the state space of a system using 'on-the-fly' searching techniques. It uses symbolic techniques to reduce the verification of modal logic formulas to solving simple reachability constraints. Some notable recent case studies with UPPAAL are [84, 114, 15].

The UPPAAL model presented in this section is the result of a few steps. First the μCRL model of the original design was translated into UPPAAL. This model was then changed into a representation of the redesign by adding the developers' solutions to the problems, that were found in the original design. The UPPAAL model of the redesign is also more specific, since interactions between the environment and the lift system are added that were abstracted away in the μCRL model of the original design. Furthermore, the model was extended with exact timing information. With respect to the explanation of the original design in Section 6.2, the redesign can be viewed as a refinement of the μCRL model. However, the desired behavior of the lift is basically the same as explained in Section 6.2. The redesign should therefore meet the same requirements as the original design.

The UPPAAL model contains four components. They are automata: *Station*, *Bus*, *Interface* and *Timer*. In UPPAAL, an automaton can be instantiated an arbitrary number of times. As explained in Section 6.2, the lift system consists of one bus and an arbitrary number of lifts. The automaton *Bus* models the CAN bus. For each lift in the system, we create two automata: *Station* and *Interface*. The automaton *Station* models the micro controller. In automaton *Interface*, the pressing and releasing of buttons on the lift is modeled. The automaton *Timer* is used to model time delay. In this section we walk through the model. Pictures of these automata are presented with only necessary explanation.

6.7.1 Transforming the μCRL model

To analyze the redesign of this system, we first transformed the μCRL model into UPPAAL. In this section, we discuss some model choices that were made.

Value passing

In μCRL, two actions can only synchronize if they occur in parallel, and if their data parameters are semantically the same, which means that communication can be used to represent data transfer from one process to another. The communication function was used heavily in the μCRL specification of the original design, to model the communications between the bus and stations. However, in UPPAAL, data transfer (or value passing) between processes (or automata) cannot be modeled in this way.

We define two channels between the bus and stations: *bustolift* and *lifttobus*, and declare several global variables for data transfer when communication happens. When a station wants to send a message to the bus, it has to instantiate the values for some global variables in the message, for instance the state and the sender's position. When communication takes place, the values of those global

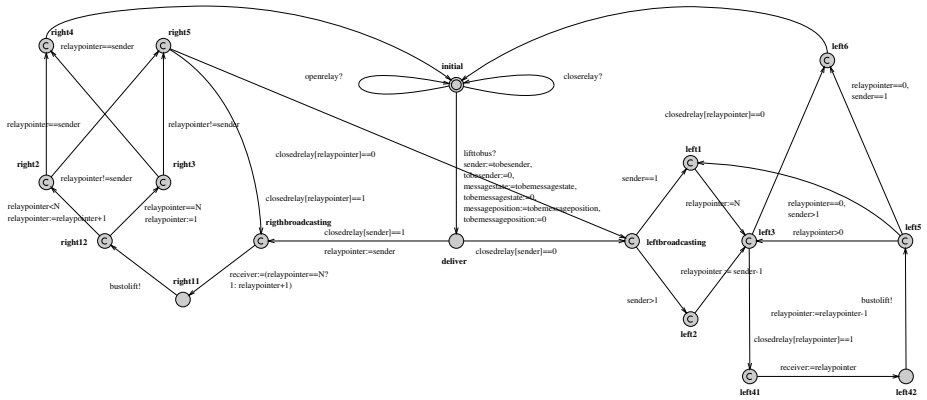


Figure 6.3: The automaton *Bus*

variables are saved to the variables used by the bus. After communication, those global variables are provided with default values. In a similar fashion, messages are sent from the bus to stations. Detailed information can be found in the automata *Station* and *Bus* (see Figure 6.3, Figure 6.5 and Figure 6.6).

Messages broadcasting

In μCRL, summation $\sum_{d:D} p(d)$ provides the possibly infinite choice over a data type D. In the μCRL specification of the bus, when the bus gets a message from a station, it can compute the set of stations who can get this message via closed relays. Then the bus can choose one station from the set nondeterministically, and send it the message. In this way, we can model the broadcasting of a message. In UPPAAL, the summation operator is absent. We set a kind of fixed order for the bus to broadcast a message. The relay controlled by a station is modeled as a flag. When the relay is closed, the flag is set to 1; otherwise it is 0. When a bus broadcasts a message, it starts to check the flag at the position of the message sender. If the flag is 1, it sends a message to the station connected by this relay, and continues to check the flag of this station. As soon as it reaches a flag with value 0, it continues at the station preceding the message sender. If the flag at this station is 1, the message is sent to the station, and the bus continues to check the flag at the preceding station. This procedure moves on until the bus reaches another flag with value 0. Recall that in both phases of the lift system, there is at least one open relay, which guarantees that the broadcasting procedure terminates. In the automaton *Bus* (see Figure 6.3), when a bus gets a message at the initial node, it starts broadcasting the message from the left part of the picture, then continues at the right part, and finally

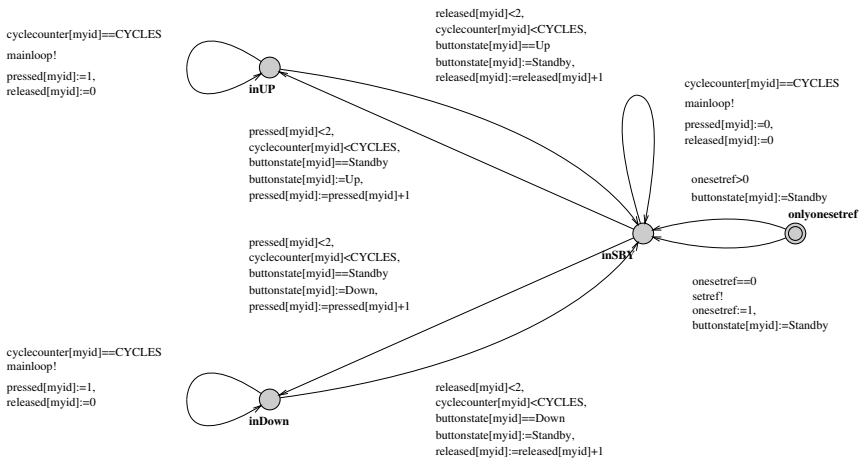


Figure 6.4: The automaton *Interface*

goes back to the initial node.

One SETREF button pressed

In Section 6.5, the second problem of the original design was found during the startup phase. It occurs if the SETREF buttons at two lifts are pressed. The result of the problem is that after the startup phase there will be two lift systems instead of one. The situation may lead to the violation of all the requirements. Given the chosen bus it seems impossible to solve this problem satisfactorily. The developers chose to emphasize in the manual that it is important to make sure that in the startup phase the SETREF button of only one lift is pressed. We also take this assumption into our analysis of the redesign.

In the UPPAAL model it is impossible to press another SETREF button after one is pressed. We use guards on transitions to block pressing of SETREF buttons after one SETREF button has been pressed. In the automaton *Interface* (see Figure 6.4), a variable *onesetref* is used as a guard on both transitions from the initial state. Initially the variable is zero, so one *Interface* can take the transition with the guard onesetref==0, if the SETREF button on the lift is pressed. The variable *onesetref* is now set to 1. In order to leave their initial state, the other *Interface* automata have to take the other transition with the guard onesetref>0. Therefore it is simply made impossible to press more than one SETREF button in our UPPAAL model.

6.7.2 Adding the solutions

In the automaton *Station*, the two phases of the lift system as explained in Section 6.2 are clearly distinguishable.

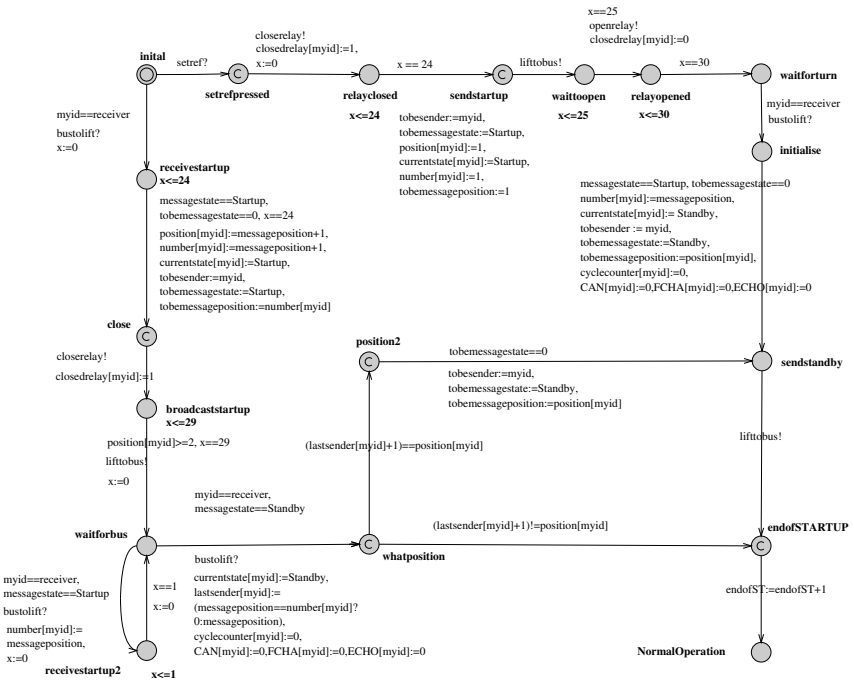


Figure 6.5: The automaton *Station*: Startup phase

Startup

Until all the stations have reached the node normaloperation, they are in the startup phase. The main role of the startup phase is to find out which position a lift has in the network and how many lifts there are in the network. The variables *position* and *number* are assigned to each lift to store this information.

The station where the SETREF button is pressed will move clockwise in Figure 6.5 from the initial node. It gets position 1, closes its relay, and sends a STARTUP message to the bus. After that it opens its relay and waits for a STARTUP message. When it gets the STARTUP message, it adopts the value of the variable *number* in this message; iy this way it gets to know how many lifts there are in the system. Then, it sends a STANDBY message and reaches the normaloperation node. The other stations will move anti-clockwise in Figure 6.5 from the initial node. They first get a STARTUP message, increase the sender of the message by one, and save it as their own *position*. They close their own relay and send a STARTUP message. There is a small loop in Figure 6.5, to indicate

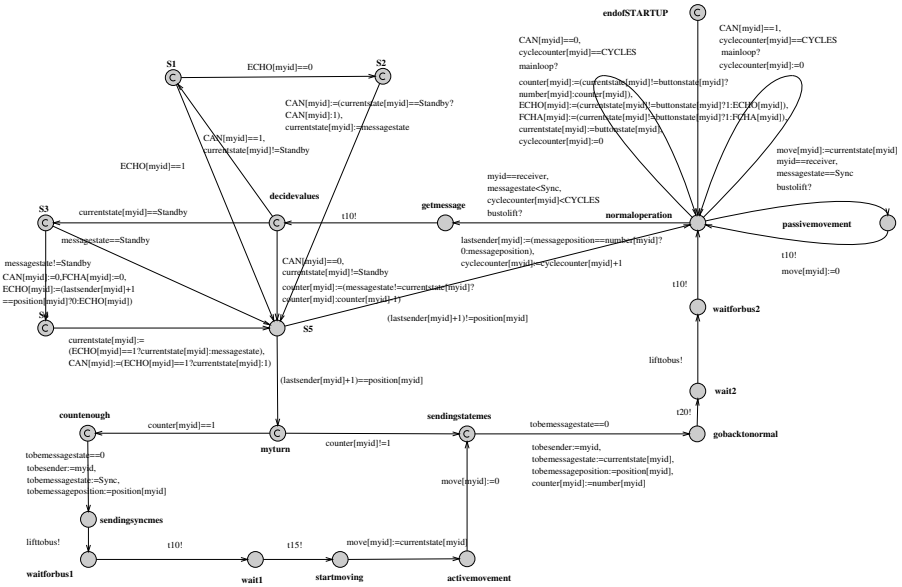


Figure 6.6: The automaton *Station*: Normal operation

that the stations keep getting STARTUP messages and changing the knowledge of the number of lifts in the system. In the end, they will get a STANDBY message, and end up in the normaloperation node. When all the stations have reached the normaloperation node, all the stations are STANDBY. They all have a unique value for *position*, and the value of *number* of all the lifts is equal to the total number of lifts in the network.

Some time delays are added into the startup phase to solve one problem found during testing. The timing information will be discussed in Section 6.7.3.

Normal operation

At node normaloperation, a station enters the normal operation phase, which is depicted in Figure 6.6. In the normal operation phase, a distinction is made between two loops which a station can perform. One is the main loop, which takes place at the node normaloperation in Figure 6.6; and the other one we will call internal loop, which is the other part of Figure 6.6. The difference between the main loop and the internal loop can be stated as follows: in a main loop the station receives state messages from its *Interface* and can change its state accordingly, and in an internal loop the station exchanges state messages with *Bus* and changes its state accordingly.

The main loop is a short loop in which the automaton *Station* synchronizes with its *Interface*. Executing the main loop is the only way the station can get

information about which button on the lift (if any) is pressed or released. This main loop takes place after a fixed number of internal loops, which is modeled as a constant *CYCLES* in the UPPAAL model. And a counter *cyclecounter* is used to record the number of internal loops that have happened after the last main loop. When cyclecounter==CYCLES, the main loop takes place and *cyclecounter* is reset to 0. If the station detects a difference between its current state (modeled by variable *currentstate*) and the state of the *Interface* (modeled by variable *buttonstate*), the station may change its state and adopt the one from the *Interface*. The main loop is also part of the original design, but it was abstracted away in the μCRL model in Section 6.4. In the UPPAAL model of the redesign it could not be left out, because as we will see the solutions from the developers interact in a critical way with the main loop.

In an internal loop, a station can do several things. First a station can get messages from the bus. Second, a station can send a message to the other stations, if it gets the turn to use the bus. Third, the active station can count state messages and initiate a movement of the whole system. In that case the active station will enter the node activemovement, while the other stations get a SYNC message and enter the node passivemovement. A variable *move* is associated to each station to indicate the direction of the current movement.

Flags

Problem three and four found in Section 6.5 occur in the normal operation phase. The third problem happens when an UP or DOWN button is pressed and released at an inappropriate moment. The lift system will end up in the situation that all stations are in UP or DOWN state, but there is no active station. This means that all the lifts will remain in that state until the system is shut down. This problem violates property *Liveness II* in Section 6.3. The reason for this problem is that in the original system a station becomes passive as soon as the pressed button on this lift is released. This problem was discovered by the developers when testing the system, and they solved it by means of flags.

The fourth problem occurs when two UP or DOWN buttons on different lifts are pressed at the same time and one of them is released at an inappropriate moment. As a result, some lifts will move, and one lift (where the button is released) remains at the same height. This violates property *Safety I* in Section 6.3. The reason for this problem is that a station becomes active as soon as a button on this lift is pressed. This problem was unknown to the developers and found its way into the final implementation of the original system. The detailed description of each problem can be found in Section 6.5. We proposed to solve this problem by allowing a station to decide to be active or passive only when it is its turn to use the bus. In the analysis of the redesign, we focus on the solutions from the developers, and explain how they fail to solve the problems in Section 6.8. Furthermore, in Section 6.9 we refine our solution from Section 6.5, and show that it does solve the problems.

The developers attempted to solve the third problem with flags. When they are set their value is 1, and when they are reset their value is 0. The flags serve as

blocks: they can prevent state changes when they are set. Two type of flags are used in the redesign, i.e. CAN, ECHO. Every station has its own flags. Initially all flags are 0. The CAN flag is set when a station receives a state message from the bus. An exception is the STANDBY message. If a station receives this message, the opposite happens: CAN is reset, but only when the current state of the station is also STANDBY; otherwise CAN is left unchanged. The idea of the developers was to use the CAN flag to block state changes by the main loop. If CAN is set, the main loop cannot change the state of the station. In Figure 6.6, we have two main loops with different guards. One is CAN==1, and the other CAN==0. If CAN==0 the main loop is taken. The current state of the station is compared with the *Interface*. In Figure 6.4, *Interface* can communicate with *Station* when it is in the nodes inUp (the UP button is pressed), inDown (the DOWN button is pressed) or inSby (no button is pressed). If CAN==1, some counters such as *cyclecounter* are reset, but nothing else happens.

The ECHO flag can only be set via the main loop with guard CAN==0. When the station detects a difference between its current state and the state of the button, ECHO is set. When ECHO is set, the state of the station cannot change by messages it receives from the bus. Like CAN, ECHO can only be reset when the state of the station is STANDBY and a STANDBY message is received from the bus. But for ECHO, there is an extra requirement that has to be fulfilled before it can be reset: it has to be the station's turn to use the bus.

6.7.3 Adding timing information

The time model in UPPAAL is continuous or dense. Clocks are used to capture time in UPPAAL. They can be associated with a transition or a node. In a transition, clock variables can be reset or used as a guard. In a node, clock variables can be used as a hold up to let the process stay in that node for a certain amount of time. Such nodes are said to be labeled with an invariant.

The way we modeled the time information of the lift system is influenced by the developers' solution to solve one problem found in the startup phase. It is also influenced by the fact that during normal operation the stations take fixed turns to use the bus. During the startup phase there is no such order. This difference has led to a different treatment of the timing information in the two phases. We first discuss the startup phase and then normal operation.

Startup

The first problem found in Section 6.5 occurs in the startup phase. It has to do with the re-opening of the relay between the first and second lift at the wrong moment. Consider Figure 6.1 in Section 6.2 again. The SETREF button is pressed on station B, which closes its relay and sends a STARTUP message to station C. If station C sends a STARTUP message before the relay between station B and station C is opened, this message is received by station B, which draws the incorrect conclusion that there are only two lifts in the network.

The solution to this problem is to let station C (or in general the station

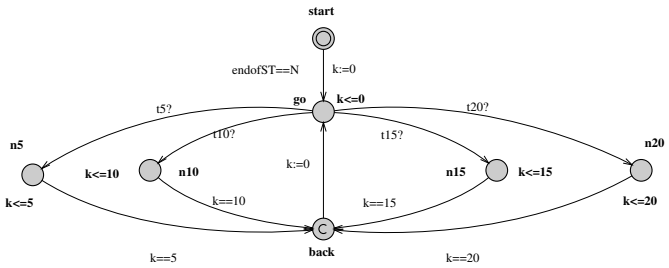


Figure 6.7: The automaton *Timer*

with position 2) wait until the relay between the first station and the second station is opened, before sending the STARTUP message. The developers added delays to the original design to make sure this happens.

In the redesign, during the startup phase, a local clock x is assigned to each station. The local clock is reset when a station gets a STARTUP message, or a SETREF button is pressed. This is used to capture the moment when the stations join the network. Receiving a message from the bus or sending a message to the bus costs 1 millisecond. The opening and closing of a relay cost 5 milliseconds. There is a delay of 24 milliseconds before sending a STARTUP message. This is all the timing information in the startup phase.

Normal operation

During normal operation, the local clocks used during the startup phase are not used anymore. Instead we use one global clock. We create an extra automaton *Timer* depicted in Figure 6.7.

Transitions normally don't take time in UPPAAL, but this does happen in the lift system. Each main loop consumes 1 millisecond. After each main loop, the station waits 0.5 millisecond to get messages from the bus. During the internal loop, the receiving and sending messages take 1 millisecond. Before sending a SYNC message, stations delay 1.5 milliseconds. Before sending a state message, stations delay 2 milliseconds. This is all the timing information in the normal operation phase. We use *Timer* to express time consumption by transitions; this idea is borrowed from [84]. The guard endofST==N makes sure that the *Timer* is only used in normal operation, where N is the number of lifts in the system. In node go, time is constrained to not progress at all. This means that in order for time to progress, one of the edges tn? must be taken; where $n \in \{5, 10, 15, 20\}$ expresses the amount time of delay. These edges then lead to nodes where time can progress with the corresponding number of time units, where after control returns immediately to the go node.

6.8 Analysis of the Redesign

Since the redesign does not change the desired external behavior of lifts, the UPPAAL model of the redesign should satisfy all the requirements in Section 6.3. We formulate those requirements in the UPPAAL requirement specification language, and verify them, sometimes with the help of test automata, to check whether the redesign solves problems 3 and 4. We give the definition and explanation of the UPPAAL requirement specification language [111] briefly as follows:

- A[] P: for all paths p always holds;
- E⟨⟩ P: there exists a path where p eventually hold;
- A⟨⟩ P: for all paths p will eventually hold;
- E[] P: there exists a path where p always holds;
- p→q: whenever p holds q will eventually hold.

where p and q are state formulas.

6.8.1 Expressing the requirements

We first check deadlock freeness. This can be translated into the UPPAAL requirement specification language directly:

- A[] not deadlock

The redesign satisfies this property, which indicates that the solution from the developers solves the first problem found in Section 6.5. In the implementation of the lift system, the delay for each STARTUP message is 24 milliseconds. In the UPPAAL model, a delay of 6 milliseconds for each STARTUP message is already enough to solve this problem.

Liveness I says that buttons on a lift can be pressed and released whenever the user wants, and that the system will respond to this. After implementing the main loop in the UPPAAL model, it is always possible to press or release buttons. So for the redesign, *Liveness I* becomes trivial.

Liveness II says that if an UP or DOWN button is pressed and not released and no other button is pressed, all lifts will move. In the UPPAAL requirement specification language, it is impossible to express this property. Fortunately, according to [2], we can transform this property into a test automaton, in which an approach is developed to model-checking of timed automata via reachability testing. The idea is to create a *bad* state in the test automaton and let the verifier check whether the system can reach this state. If it does, the system violates a certain property.

The test automaton may need some extra *decorations* for the verification purpose. In principle, with the test automaton we can express all scenarios we

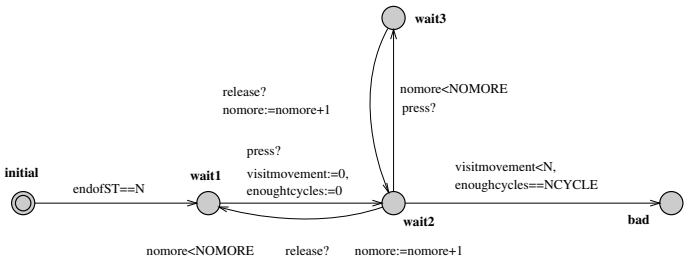


Figure 6.8: The test automaton for *Liveness II*

want to check. As this would lead to a possibly infinite state space, some scenarios which are not interesting can be abstracted away. For example, in the lift system, the buttons can be pressed and released many times. We consider only those scenarios where a button on one lift is pressed and released at most once. The test automaton for the requirement *Liveness II* is depicted in Figure 6.8 below.

We add new synchronizations between the *Interface* automata and the test automaton via *press* and *release* channels, to model the number of pressing and releasing actions. In the test automaton only one pressing and releasing per lift can take place. *nomore* is a variable that is used to block more pressing and releasing actions. This test automaton is used to express that if a button is pressed and not released any more, after some period of time (modeled by variable *enoughcycles*) all the lifts will move. We now check whether the test automaton can reach the node bad. If the test automaton reaches the node bad, it means that not all the lifts have moved and the system violates property *Liveness II*.

- A[] not testautomaton.bad

Test automata are also used to model and check the other two safety properties.

With *Liveness II*, we could check that if one button is pressed, all the lifts reach their activemovement or passivemovement node within a certain amount of time. What we do not check is whether they move in the same direction.

Safety I demands that whenever a lift moves, all the other lifts move simultaneously in the same direction. The corresponding test automaton is depicted in Figure 6.9. This test automaton waits for one lift to reach the activemovement node, which is detected by a synchronization on channel go? between *Station* and this test automaton. The test automaton then checks whether the other lifts move in the same direction (modeled by guard visitmovement<N) within a certain amount of time (modeled by enoughcycles==NCYCLES).

Safety II states that there will be no movement when no button has been pressed. The corresponding test automaton is depicted in Figure 6.10. The variable *noupdown* (meaning no UP or DOWN button pressed) is used to block

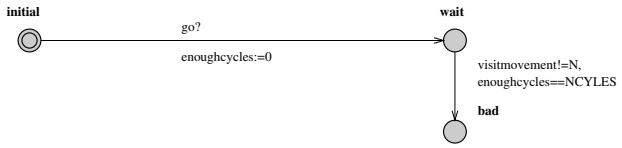


Figure 6.9: The test automaton for *Safety I*

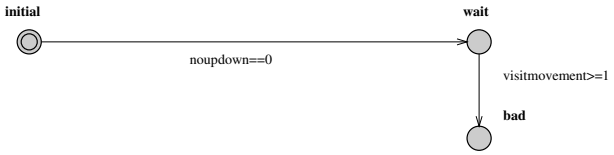


Figure 6.10: The test automaton for *Safety II*

all pressings of buttons in the *Interface*s. Now we can check whether it is still possible for the lifts to reach movement nodes (modeled by visitmovement>=1).

The redesign satisfies requirement *Safety II*, and violates requirements *Liveness II* and *Safety I*. We will discuss the diagnostic traces and the reasons in the next section.

6.8.2 Problems

The developers invented flags to solve the third problem found in Section 6.5. These flags seem to solve the error scenario described in Section 6.5. But during the testing phase, the developers encountered a new error; again the cause for this error was not clear to them. We have built a UPPAAL model (see Section 6.7) for the redesign and checked it. *Liveness II* turned out to be violated. We first investigated the diagnostic trace generated by the model checker in UPPAAL, and then gave the reason why the solution from the developers failed. The generated diagnostic trace contains 256 transitions; we used the graphic simulation tool in UPPAAL to analyze it.

Initially all the flags are 0. When an UP button is pressed on one station (A), ECHO will be set and the state of station A will change to UP. Station A sends an UP message. The other stations will set the CAN flag and change their state to UP. Suppose the button is released again. The flag of station A does not change, but its state will change to STANDBY (see the main loop in Figure 6.6). Station A will send a STANDBY message which the others will adopt. When they have adopted this state, and if they receive another STANDBY message, the CAN flags of the other stations will be reset. After a short while all CAN flags in the network are 0, ECHO of station A is 1, and all the states of

the stations are STANDBY. Suppose now that an UP button of another station (B) is pressed. Station B will send an UP message. Station A will receive this but cannot change its state because ECHO is set. When it is station A's turn to use the bus it will therefore send a STANDBY message. Station B will receive this STANDBY message, and it will not count enough UP messages. The whole counting procedure has to start over again. Station B will send an UP message. The other stations will adopt this state and send a UP message. But when it is station A's turn, again since ECHO is set, it will send a STANDBY message and station B will again not count enough UP messages. It is clear that the ECHO of station A should be reset to get out of this situation, but that can only happen when the state of the station is STANDBY, a STANDBY message is received, and it is this station's turn to use the bus. For station A this never happens. As a result, the whole system will never move, even when an UP button is pressed.

The test automaton detects this problem. Even though the solution of the developers has some virtue, they seem not to have taken into account that the main reason for the third problem lies in the fact that the active station immediately changes its state to STANDBY after a button is released. Their solution was directed to block state changes to the active station after its state has changed to STANDBY. This is not the heart of the problem and therefore the problem remains in the redesign.

The fourth problem found in Section 6.5 is also still in the redesign. The redesign violates property *Safety I*. The reason resembles what is already explained in Section 6.5. This is not very surprising, since the fourth problem was unknown to the developers at the time of the redesign.

6.9 A New Solution

In this section, we refine the solution proposed in Section 6.5 in such a way that it corresponds with UPPAAL and resemble to the solution from the developers. The key point why our solution differs from the flags added into the redesign is that our solution creates a link between the state change of a station and the turn of the station to use the bus. This idea was already mentioned in the μCRL model in Section 6.5, but it was not further specified. With the more exact model of the redesign, including the main loop, and using the idea of the flags the developers came up with, now we work out the idea in detail.

The new flags are called CHANGE and ACTIVE. They are assigned to each station. CAN and ECHO are no longer a part of the new solution. When ACTIVE is 1, the corresponding station is active; otherwise, the station is passive. CHANGE of a station is set when there is a button pressed or released at this station (through the main loop). This is used to remember that the ACTIVE flag at this station must change from active to passive, or vice versa. Only when the station gets its turn to use the bus, this change will actually happen. If one station wants to become active, it has to make sure that there are no other active stations in the system, by checking whether the state of the message from the bus is STANDBY. If the CHANGE of a station is set, this station does not

change its state until it is its turn to use the bus to make a decision. CHANGE is reset together with a setting or resetting of ACTIVE.

Changing the new flags has no effect on the automata *Interface*, *Bus* and *Timer*. They are exactly the same as in the redesign. Only the automaton *Station* has undergone crucial changes. We will not explain the new *Station* automaton in detail, more information can be found in [100]. All requirements have been checked successfully on the model with this new solution. In particular, problem three and four are resolved.

Remarks. Since more details of the lift system are taken into account in the UPPAAL model, the state space of the redesign increases dramatically. For the UPPAAL model of the redesign, we could only manage the analysis for systems with three lifts (UPPAAL version 3.2.4). The requirements were checked on a 1.4 GHz AMD Athlon^{TM} Processor with 512 Mb memory. When it turned out that the error traces that were discovered by the developers had nothing to do with exact timing properties, we made a translation of the UPPAAL model into μCRL, and repeated the verification with μCRL and CADP.

6.10 Conclusions

In this chapter, we have reported an industrial case study on applying formal techniques for the design and analysis of a distributed system for lifting trucks. Our work can be considered as one piece of evidence that formal verification techniques are mature enough to be applied in industrial projects.

First, we have described a model of the original design of a distributed lift system in μCRL. Our primary finding is that such a model is an efficient tool to understand the behavior of embedded distributed systems, in the sense that it helped us to find errors and understand their nature using the available technology. We also find confirmation of our previous findings that the possibility to describe interactions in a process algebraic way, and data using equational abstract data types provide exactly the required means for this specification and its validation. The four problems found in the original design are summarized as follows:

1. During the startup phase, the relay between the first and second lift is reopened at the wrong moment; it results in a deadlock in the lift system.

2. During the startup phase, the SETREF buttons at two lifts are pressed; the system will have two independent networks instead of one.

3. During normal operation, an UP or DOWN button is pressed and released at an inappropriate moment; this problem violates property *Liveness II*.

4. During normal operation, two UP or DOWN buttons at different lifts are pressed at almost the same time, and one is released at an inappropriate moment; this problem violates property *Safety I*.

The first three problems were also found by the developers, and the fourth was new and unknown to them.

Second, the redesign was then modeled in UPPAAL. The analysis in Section 6.8 has produced some interesting results. It shows that the redesign does not satisfy all the requirements, meaning that the redesign by the developers does not solve all the problems found in the original design. Only one problem is solved by adding time delays. The third problem, for which those flags were developed, and the fourth problem are not solved.

We could analyze the μCRL model of the original design with up to five lifts. For increased certainty, it would be nice to increase this number, preferably up to 32, as this is the maximal allowed configuration. It is clear that more advanced techniques are needed, and much work into these is going on. It leads too far to mention all of them but work on parametric reduction of state spaces [70], confluence reduction [78] and parametric composition of parallel processes [81] are all activities striving to enable the analysis of systems with many more up to possibly unbounded parallel components.

The developers of the system have fully acknowledged that these techniques have increased their understanding and are planning to release a new version of the product including the improvements we suggest.

Chapter 7

Model Checking a Cache Coherence Protocol for Jackal

7.1 Introduction

Shared memory is an attractive programming model for interprocess communication and synchronization in multiprocess computations. In the past decade, a popular research topic has been the design of systems to provide a shared memory abstraction of physically distributed memory machines. This abstraction, known as Distributed Shared Memory (DSM), has been implemented both in software (e.g., to provide the shared memory programming model on networks of workstations) and in hardware (e.g., using cache coherence protocols to support shared memory across physically distributed main memories).

Multithreading is a programming paradigm for implementing parallel applications on shared memory multiprocessors. The Java memory model (JMM) [67] prescribes certain abstract rules that any implementation of Java multithreading must follow. Jackal [174] is a fine-grained DSM implementation of the Java programming language. It aims to implement the JMM and allows multithreaded Java programs to run unmodified on DSM. It employs a self-invalidation based, multiple-writer cache coherence protocol, which allows processors to cache a region (which is either an object or a fixed-size partition of an array) created on another processor (i.e., the region's home). All threads on one process share one copy of a cached region. The home node and the caching processors store this copy at the same virtual address. A cached region copy remains valid for a particular thread until that thread reaches a synchronization point. In Jackal, several optimizations [173, 174] improve both sequential and parallel application performance. Among them, the *automatic home node migration* reduces the amount of synchronization, by automatically appointing the processor that is likely to access a region most often as the region's home.

In this chapter, we present our formal analysis of a cache coherence protocol for Jackal using the μCRL toolset and CADP. A μCRL specification of the protocol (including automatic home node migration) was extracted from an informal (C language-like) description of the protocol. To avoid state explosion,

we made certain abstractions with respect to the protocol's implementation. Requirements were verified by the μCRL toolset together with CADP. Our analysis revealed many inconsistencies between the description and the implementation. We found two errors were in the description. The developers of the protocol checked the two errors and found their way in the implementation. Both errors can happen when a thread is writing a region from remote (i.e., the thread does not run on the home of the region). During the thread's waiting for a proper protocol lock or an up-to-date copy of the region, the home node may migrate to the thread's processor, so that the thread actually accesses the region at home. The first error resulted into a deadlock. The second error was found when model checking the property of only one home for each region. After updating our formal specification, the requirements were successfully checked on several configurations. Our solutions to the errors were adapted in the implementation of the protocol.

Outline of the chapter. The remainder of this chapter is structured as follows. In Section 7.2, we discuss related work on analyzing the JMM or its replacement proposal and verifying cache coherence protocols using formal techniques. An informal description of the JMM is given in Section 7.3. Section 7.4 presents the Jackal system and its cache coherence protocol. Section 7.5 focuses on our formal analysis in μCRL. The μCRL specifications for each component of the protocol and the verification results are given. Discussions and future work are mentioned in Section 7.6.

7.2 Related Work

The use of formal methods to analyze the JMM is an active research topic. In [151], the authors developed a formal executable specification of the JMM [67]. Their specification is operational and uses guarded commands. This model can be used to verify popular software construction idioms for multithreaded Java. In [177], the Murϕ verification system [43] was applied to study the CRF memory mode [120]. A suite of test programs was designed to reveal pivotal properties of the model. This approach was also applied to Manson and Pugh's proposal [121] by the same authors [178]. Two proofs of the correctness for Cachet [158], an adaptive cache coherence protocol, were presented in [164]. Each proof demonstrates soundness (conformance to the CRF memory model) and liveness. One proof is manual, based on a term-rewriting system definition; the other is machine-assisted, based on a TLA [110] formulation and using the theorem prover PVS. Similar to [177, 178], we use formal specification and model checking techniques. A major difference is that we analyzed a cache coherence protocol within a Java DSM system that is already implemented and far more complicated than the abstract memory models analyzed in [151, 164, 177, 178]. Our analysis helped to improve the actual design and implementation of the protocol.

Our work is also related to the verification of cache coherence protocols.

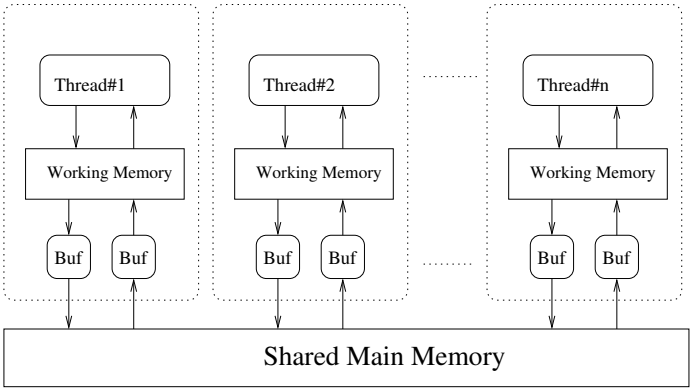


Figure 7.1: JMM memory system

Formal methods have been successfully applied in the automatic verification of cache coherence on sequentially consistent systems [109], e.g. [24, 38, 87]. Coherence in shared memory multiprocessors is much more difficult to verify. Recently, Pong and Dubois [143] used their state-based tool for the verification of a delayed protocol [45], which is an aggressive protocol for relaxed memory models. We encountered the same difficulties as [143], such as that the hardware to model is complex, and that the properties of the protocol are hard to formulate. Differences between our work and [143] are: we analyzed a protocol designed for *distributed* shared memory machines; and the protocol supports *multithreaded* Java programs, which makes matters more complicated.

7.3 Java Memory Model

The Java language supports multithreaded programming, where threads can interact among themselves via read/write of shared data. The JMM prescribes certain abstract rules that any implementation of Java multithreading must follow. We briefly present the current JMM as given in [67].

The JMM allows each thread to cache variables in its working memory, which keeps its own working copy of the variables. A thread can only manipulate the values in its working memory, which is inaccessible to other threads. The working memories are caches of a single main memory, which is shared by all threads. Main memory keeps the main copy of every variable. A thread's working memory must be flushed to main memory at each synchronization point. A synchronization point is a lock or unlock operation corresponding to the entry or exit of a synchronized block. The memory structure is depicted in Figure 7.1.

The JMM defines a set of actions that threads may use to interact with memory. A thread invokes four actions: *use*, *assign*, *lock* and *unlock*. The other actions: *read*, *load*, *store* and *write*, are invoked by a multithreaded implementation following the temporal ordering constraints in the current JMM ([67,

Chapter 17]). The meaning of each action is as follows:

1. *use*: Read from the working memory of a variable by a thread.
2. *assign*: Write into the working memory of a variable by a thread.
3. *read*: Initiate reading from the main memory of a variable by a thread.
4. *load*: Complete reading from the main memory of a variable by a thread.
5. *store*: Initiate writing the working memory into the main memory of a variable by a thread.
6. *write*: Complete writing the working memory into the main memory of a variable by a thread.
7. *lock*: Get the values in the main memory transferred to a thread's working memory through *read* and *load* action.
8. *unlock*: Put the values a thread holds in its working memory back to the main memory through *store* and *write* action.

There are many problems in the current JMM [67], such as that semantics for *final* variable operations is omitted and that *Volatile* variable operations do not have synchronization effects for normal variable operations. In view of these problems, the Java Specification Request (JSR) 133 is under development. Two replacement semantics have been proposed to improve the JMM, one by Manson and Pugh [121], the other by Maessen, Arind and Shen [120]. A detailed discussion of the various problems in the current JMM can be found at `http://www.cs.umd.edu/~pugh/java/memoryModel/`. Jackal is intended to implement the memory model in JSR, which will be released soon.

7.4 Jackal DSM System

Jackal [174] is a fine-grained DSM implementation of the Java programming language. Its runtime system implements a self-invalidation based, multiple-writer cache coherence protocol for regions.

The Jackal memory model allows processors to cache a region created on another processor (i.e., the region's home). All threads on one processor share one copy of a cached region. The home node and the caching processors all store this copy at the same virtual address. The protocol is based on self-invalidation, which means the cached copy of a region remains valid until the thread itself invalidates the copy, which occurs whenever it reaches a synchronization point. Jackal combines features of HLRC [179] and TreadMarks [101]. As in HLRC, modifications are flushed to a home node; as in TreadMarks, twinning and diffing are used to allow concurrent writes to shared data. Unlike TreadMarks, Jackal uses software access checks inserted before each object usage to detect non-local or stable data.

The implementation of the Jackal memory model contains three components: address space management, access checks and synchronization. Several optimizations were made to improve both sequential and parallel application performance [173, 174].

7.4.1 Address space management

Jackal stores all regions in a single, shared virtual address space. Each region occupies the same virtual address range on all processors that store a copy of the region. Regions are named and accessed through their virtual address. Each processor owns part of the physical memory and creates objects and arrays in its own part. In this way, each processor can allocate objects without synchronizing with other processors. When a thread wishes to access a region created by another processor, it must potentially allocate physical memory for the virtual memory pages in which the object is stored, and retrieve an up-to-date copy of the region from its home node. If a processor runs out of free physical memory, it initiates a global garbage collection that frees both Java objects and physical memory pages.

To implement self-invalidation, each thread keeps track of the regions it accessed and cached since its last synchronization point. The data structure storing this information is called the *flush list*. At synchronization points, all regions on the thread's flush list are invalidated for that thread, by writing *diffs* back to their home nodes. A diff contains the difference between a region's object data and its twin data.

7.4.2 Access check

Jackal's compiler generates a software access check for every use of a region. The access check determines whether the region referenced by a given pointer contains a valid local copy. Whenever an access check detects an invalid local copy, the runtime system contacts the region's home. It asks the home node for a copy of the region and stores this copy at the same virtual address as at the home node. The thread requesting the region receives a pointer to that region and adds it to its flush list. This flush list is similar to the working memory in the current JMM [67].

7.4.3 Synchronization

Logically, each Java object contains an object lock and a condition variable. Since threads can access objects from different processors, Jackal provides distributed synchronization protocols. Briefly, an object's home node acts as the object's object lock manager. *lock*, *unlock*, *wait* and *notify* calls are implemented as control messages to the lock's home node. To acquire an object lock, a thread sends a lock request message to the object lock manager and waits. When the lock is available, the manager replies with a notify message; otherwise, the thread needs to wait for the lock to be released. To unlock, the lock

holder sends an unlock message to the home node. We did not model object locks, since they are not relevant to the requirements that we formulated for the protocol (see Section 7.5.2).

7.4.4 Automatic home node migration

Java programs do not indicate which object locks protect which data items. This make it difficult to combine data and synchronization traffic. Jackal may have to communicate multiple times to acquire an object lock, to access the data protected by the lock and to release the lock. Recall that the home of a region acts as the manager of the object lock. To decrease synchronization traffic, automatic home node migration has been implemented in Jackal. It means that Jackal may automatically appoint the processor that is likely to access a region most often as the region's home. This optimization is triggered during the following two cases.

1. A thread writes to a region, and an access check detects an invalid local copy; the runtime system contacts the region's home, and finds that the thread's processor is the only one from which threads are writing to this region. Then the home of this region migrates to the thread's processor.

2. A thread flushes at a synchronization point, and there is only one processor left from which threads are writing to some region. Then the home of this region migrates to this processor.

Jackal can detect these situations at runtime, and thus reduce synchronization traffic. Automatic home node migration complicates meeting the requirements in Section 7.5.2.

7.4.5 Other features

To improve performance, a source-level global optimization *object-graph aggregation*, and runtime optimization *adaptive lazy flushing*, are implemented in Jackal.

The Jackal compiler can detect situations where an access to some object (called *root object*) is always followed by accesses to subobjects. In that case, the system views the root object and the subobjects as an object graph. Jackal attempts to aggregate all access checks on objects in such a graph into a single access check on the graph's root object. If this check fails, the entire object graph is fetched, which can reduce the number of network round-trips. We did not model object-graph aggregation since we modeled memory at a rather abstract level.

The Jackal cache coherence protocol invalidates all data in a thread's working memory at each synchronization point. That is, the protocol exactly follows the specification of the JMM, which potentially leads to much interprocessor communication. Due to adaptive lazy flushing, it is not necessary to invalidate and flush a region that is accessed by only a single processor or that is only read

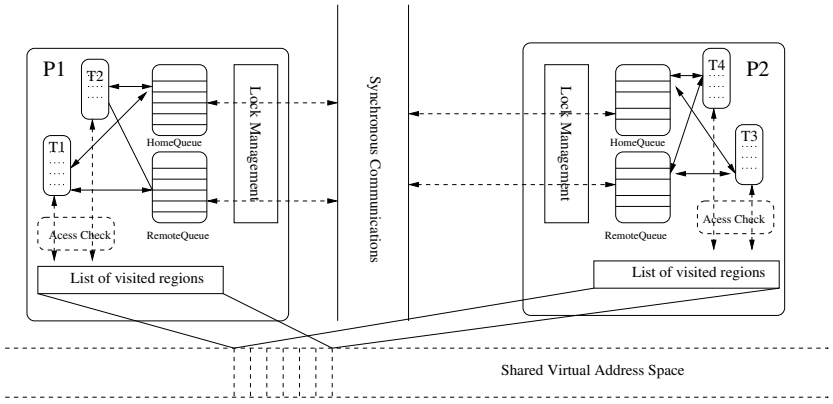


Figure 7.2: Components in the Jackal architecture

by its accessing threads. We did not model adaptive lazy flushing, since it is not relevant to the requirements that we formulated.

7.5 Specification and Analysis in μCRL

In this section, we present a formal specification of Jackal's cache coherence protocol in μCRL and verify some requirements at the behavioral level.

7.5.1 Specification of the protocol

The cache coherence protocol in Jackal is more complex than an interleaved execution of the threads, where each thread executes in program order. The permitted set of execution traces is a superset of the simple interleaved execution of the individual threads. Furthermore, the μCRL specification is an exhaustive nondeterministic description of the cache coherence protocol. This may lead to *state explosion*. To deal with this problem, we made some abstractions of each component. In the following discussion, we present the μCRL specification for each component, together with the abstractions we made. For the sake of presentation, we only give parts of the specification to illuminate the crucial points, and omit the specification of data types. The complete specification can be found at Appendix A (also available at `http://www.cwi.nl/~pangjun/ccp/`).

Our model of the cache coherence protocol is a parallel composition of threads, processors, regions, protocol lock managers and message queues upon a set of communication actions. Fig. 7.2 shows the various components and their interactions in the μCRL specification. P_i are identities of processors, and T_i identities of threads. By means of these communications, data can be transferred between two processes. The complete μCRL specification of this protocol consists of around 1000 lines.

```
proc  Thread(tid:ThreadId,pid:ProcessId,FlushList:RegionIdSet)=
      write(tid).ThreadWrite(tid,pid,FlushList)
      +
      flush(tid).ThreadInvalidate(tid,pid,FlushList)
      ◁not(empty(FlushList))▷δ
```

Table 7.1: Specification of a thread

```
      % Synchronization between actions.
comm  s_refresh | r_refresh = c_refresh
      s_norefresh | r_norefresh = c_norefresh
      s_sendback | r_sendback = c_sendback

proc  ThreadWrite(tid:ThreadId,pid:ProcessId,FlushList:RegionIdSet)=
      % The thread has written the region (rid1) before.
      Thread(tid,pid,FlushList)
      ◁test(rid1,FlushList)▷
      ∑r:Region r_sendback(tid,pid,r).
      % Write the region at home, if pid=gethome(r).
        (s_norefresh(tid,pid).WriteHome(tid,pid,insert(rid1,FlushList))
        ◁eq(gethome(r),pid)▷
      % Otherwise, write the region from remote.
        s_norefresh(tid,pid).WriteRemote(tid,pid,insert(rid1,FlushList)))
```

Table 7.2: Specification of a thread writing

Threads

Each thread runs on a processor, and can perform a number of actions: *read, write* and *invalidate*. It maintains two lists: *ReadList* contains the identities of regions that it is reading or recently read from, and *WriteList* contains the identities of regions that it is writing or recently wrote to.[1] When a thread starts reading from or writing to a region, the corresponding access check determines whether there is a valid local copy of this region at the thread's processor. The *server_lock* is needed if the thread runs on the region's home (i.e., if the thread reads or writes at home); otherwise, the *fault_lock* of the thread's processor is acquired (i.e., if the thread reads or writes from remote). When a fault_lock is granted, the thread retrieves an up-to-date copy of the region from its home node. The thread continues reading from or writing to the region and finally releases the lock by sending an unlock message to the protocol lock manager (see Table 7.2 and Table 7.3).

When a thread invalidates, it empties both its ReadList and WriteList. If

[1] We only model threads with a *FlushList* in μCRL. See later discussion.

```
        % Synchronization between actions.
comm    s_require_faultlock | r_require_faultlock = c_require_faultlock
        s_no_faultwait | r_no_faultwait = c_no_faultwait
        s_signal_faultwait | r_signal_faultwait = c_signal_faultwait
        s_data_require | r_i_data_require = c_i_data_require
        s_signal | r_signal = c_signal
        s_free_faultlock | r_free_faultlock = c_free_faultlock

        % Thread writes from remote, requires a fault lock,
        % and asks for a fresh copy of the region.
proc    WriteRemote(tid:ThreadId,pid:ProcessId,FlushList:RegionIdSet) =
        s_require_faultlock(pid).
        (r_no_faultwait(pid)+r_signal_faultwait(pid)).
        (∑_{r:Region} r_sendback(tid,pid,r).
        % Ask for a fresh copy of the region.
         s_data_require(tid,pid,gethome(r)).s_norefresh(tid,pid).
        % Copy arrives, the thread is notified.
         (∑_{pid':ProcessId} r_signal(tid,pid').
          (∑_{newr:Region} r_sendback(tid,pid,newr).
          s_refresh(tid,pid,setlocalthreads(newr,S(getlocalthreads(newr)))).
          s_free_faultlock(pid).Thread(tid,pid,wl))))
```

Table 7.3: Specification of a thread writing to a region from remote

the thread invalidates a region in its WriteList at home, and it may find out that there is only one processor left from which threads are writing to the region, then the home of the region migrates to this processor. If the thread invalidates a region in its WriteList from remote, it sends a *Flush* message to the home of the region, the Flush message also contains a diff with the difference between the region's object and twin data. The home processor of the region will take charge of automatic home migration. The *flush_lock* of the home of each region is acquired before invalidating, and released after invalidating (see Table 7.4 and Table 7.5).

In the μCRL specification, each thread is modeled as a separate process with a unique identity (see Table 7.1). It contains one parameter *pid* to indicate on which processor the corresponding thread executes. Since the behavior of reading from a region is part of the behavior of writing to a region, and since writing is far more critical for the correctness of the protocol than reading, we abstracted away from the read action of threads. As a result, a thread only maintains a *FlushList* and flushes the regions in this FlushList.

```
        % Synchronization between actions.
comm    s_require_flushlock | r_require_flushlock = c_require_flushlock
        s_no_flushwait | r_no_flushwait = c_no_flushwait
        s_signal_flushwait | r_signal_flushwait = c_signal_flushwait

proc    ThreadInvalidate(tid:ThreadId,pid:ProcessId,FlushList:RegionIdSet) =
        % Thread requires a flush lock.
        s_require_flushlock(pid).
        (r_no_flushwait(pid)+r_signal_flushwait(pid)).
        ∑_{r:Region} r_sendback(tid,pid,r).
        % Invalidate at home, we only model one region: rid1.
          (FlushHome(tid,pid,remove(rid1,FlushList),r)
          ◁eq(gethome(r),pid)▷
        % Otherwise, invalidate from remote.
          FlushRemote(tid,pid,remove(rid1,FlushList),r)))
```

Table 7.4: Specification of a thread invalidating

```
        % Synchronization between actions.
comm    s_flush | r_i_flush = c_i_flush
        s_free_flushlock | r_free_flushlock = c_free_flushlock

proc    FlushRemote(tid:ThreadId,pid:ProcessId,FlushList:RegionIdSet,
          r:Region) =
        % No thread is using this region. Set the last parameter as true.
        s_flush(tid,pid,gethome(r),r,T).
        % Refresh the region's information.
        % We use *new-information-of-the-region* to indicate the updating.
        s_refresh(tid,pid,*new-information-of-the-region*).
        s_free_flushlock(pid).
        % This invalidation is finished, the thread is notified.
        ∑_{pid':ProcessId} r_signal(tid,pid').Thread(tid,pid,FlusList)
        ◁eq(sub1(getlocalthreads(r)),0)▷
        % Otherwise, set the last parameter as false.
        s_flush(tid,pid,gethome(r),r,F).
        % Refresh the region's information.
        s_refresh(tid,pid,*new-information-of-the-region*).
        s_free_flushlock(pid).
        % This invalidation is finished, the thread is notified.
        ∑_{pid':ProcessId} r_signal(tid,pid').Thread(tid,pid,FlusList)
```

Table 7.5: Specification of a thread flushing a region from remote

```
        % pid indicates where the region is;
        % r contains the region's information.
proc    Region(pid:ProcessId, r:Region) =
        % Communication with threads.
        ∑tid:ThreadId s_sendback(tid,pid,r).
          (r_norefresh(tid,pid).Region(pid,r)
          +∑r':Region r_refresh(tid,pid,r').Region(pid,r'))
        % Communication with processors.
        + s_sendback(pid,r).
        (r_norefresh(pid).Region(pid,r)
        +∑r':Regionr_refresh(pid,r').Region(pid,r')))
```

Table 7.6: Specification of a region

Regions

Jackal uses a single shared virtual address space. Each region occupies the same virtual address range on all processors that store a copy of it. When a region is created on one processor, a copy of this region is also created on every other processor. A region contains the following information:

1. Location: A processor's identity, denoting at which node the region (or a copy) is.
2. Home: A processor's identity, denoting the home node for this region.
3. State: A region can evolve into four kinds of states. When no thread uses this region, the state of the region is *Unused*; if a region is only used by threads on its home node, its state is *Homeonly*; if all accesses of a region are read actions, the state of this region is *Readonly*; in all other cases, the state of a region is *Shared*.
4. ReaderList: A list of processors' identities containing threads that are reading or recently read from this region. It is only maintained at the home node.
5. WriterList: A list of processors' identities containing threads that are writing or recently wrote to this region. It is only maintained at the home node.
6. Object data: An array of bytes.
7. Twin data: An array of bytes. It is a copy of the object data for diffing at non-home nodes; initially it is null.
8. Localthreads: A natural number, the number of threads accessing this region at the location of the region.

In μCRL, each region is modeled as a separate component. As a result of our abstraction of the behavior of threads, we made some corresponding abstractions for regions. Each region has only two states; we kept the *Unused* state, while the other three states are mapped to a state *Used*. The region only needs to maintain the *WriterList*. Furthermore, we did not model object and twin data, since they are not relevant to our requirements for the protocol. So in our model a thread cannot write any value to a region. Still, when a thread flushes a region from remote, a message (without a diff) is sent back to the home of this region to unlock its fault_lock.

We use a set of synchronized actions to ensure that during an access to a region, no other processes can change the information of this region (see Table 7.2). For example, a thread gets the information of a region by performing a synchronized action *r_sendback*, and the accesses to this region are blocked until this thread executes another synchronized action *s_norefresh* (if it has changed nothing) or *s_refresh* (if it has changed some information of the region). The synchronized actions on a region are presented in Table 7.6, together with the specification for regions in μCRL. To avoid state explosion, we only analyzed configurations containing one region with identity *rid1*.

Messages to processors

Four kinds of messages can be delivered to a processor.

1. *Data_Request*: This message is sent when a thread starts writing to a region from remote. When a processor gets this message, and it is the home of the region, it adds the thread's processor into the WriterList of the region and sends back an up-to-date copy of the region to the thread's processor by a *Data_Return* message. If it is not the home of the region (meaning that the region migrated its home in the meantime), it forwards the Data_Request message to the region's new home.

2. *Data_Return*: This message is received by a processor when an up-to-date copy of a region has arrived. The processor updates the object and twin data of the region. Moreover, if the message is a home node migration message, then the processor becomes the home of this region, and starts maintaining the WriterList and the state of the region.

3. *Flush*: This message is sent when a thread flushes from remote. When a processor gets this message, and it is the home of the region, it removes the thread's processor from the WriterList of the region; moreover, it may send a home node migration message to a new home of this region (by a *Region_Sponmigrate* message). When it is not the home of the region, it forwards the Flush message to the region's new home.

4. *Region_Sponmigrate*: When a processor gets this message, it becomes the home of the region in question.

```
        % Synchronization between actions.
comm    s_i_data_require | r_data_require = c_o_data_require
        s_free_homequeuelock | r_free_homequeuelock = c_free_homequeuelock
        s_data_return | r_o_data_return = c_i_data_return
        s_i_region_sponmigrate | r_region_sponmigrate = c_o_region_sponmigrate

proc    Processor(pid:ProcessId) =
        % Processor gets a Data_Request message (forwarded) from processor pid'.
        ∑tid:ThreadId ∑pid':ProcessId r_data_require(tid,pid',pid).
        % We only model one region: rid1. Check the current state of the region.
        % If the processor is not the home of the region,
        % then the message is forwarded to the real home.
          r_sendback(pid,rid1).
          (s_data_require(tid,pid',gethome(rid1)).
          s_norefresh(pid).s_free_homequeuelock(pid).Processor(pid)
          ◁not(eq(gethome(rid1), pid))▷
        % Refresh the region's information, and send the region back.
        % If the region is UNUSED, then the Data_Return message
        % is also a home migration message. Set the last parameter as true.
            (s_data_return(tid,pid',pid,*new-infomation-of-the-region*,T).
            s_refresh(pid,*new-infomation-of-the-region*).
            s_free_homequeuelock(pid).Processor(pid)
            ◁eq(getstate(rid1),UNUSED)▷
        % It is not a home migration message. Set the last parameter as false.
            s_data_return(tid,pid',pid,*new-infomation-of-the-region*,F).
            s_refresh(pid,*new-infomation-of-the-region*).
            s_free_homequeuelock(pid).Processor(pid)))
        +
        % Processor gets a Region_Sponmigrate message.
        % It becomes the region's home node by refreshing
        % the region's parameters.
        ∑tid:ThreadId ∑pid':ProcessId ∑r':Region r_region_sponmigrate(tid,pid',pid,r').
          (∑r:Region r_sendback(pid,r).
        % Set the home by itself; maintain the state and writerlist.
          s_refresh(pid,*new-infomation-of-the-region*).
          s_free_homequeuelock(pid).Processor(pid))
        + ...
```

Table 7.7: Specification of a processor dealing with a message

```
        % Synchronization between actions.
comm    s_require_homequeuelock | r_require_homequeuelock
          = c_require_homequeuelock
        s_no_homequeuewait | r_no_homequeuewait
          = c_no_homequeuewait
        s_signal_homequeuewait | r_signal_homequeuewait
          = c_signal_homequeuewait
        s_region_sponmigrate | r_i_region_sponmigrate
          = c_i_region_sponmigrate

proc    HomeQueue(pid:ProcessId)=
        % Home queue gets a Data_Request message.
        % To deal with it, the homequeue_lock is needed.
        ∑tid:ThreadId ∑pid':ProcessId
        % Put a message into the queue.
          r_i_data_require(tid,pid',pid).s_require_homequeuelock(pid).
          (r_no_homequeuewait(pid)+r_signal_homequeuewait(pid)).
        % The processor takes this message.
          s_i_data_require(tid,pid',pid).HomeQueue(pid)
        +
        % Home queue gets a Region_Sponmigrate message.
        % To deal with it, the homequeue_lock is needed.
        ∑tid:ThreadId ∑pid':ProcessId∑r:Region
        % Put a message into the queue.
          r_i_region_sponmigrate(tid,pid',pid,r).s_require_homequeuelock(pid).
          (r_no_homequeuewait(pid)+r_signal_homequeuewait(pid)).
        % The processor takes this message.
          s_i_region_sponmigrate(tid,pid',pid,r).HomeQueue(pid)
        + ...
```

Table 7.8: Part of the specification of a home queue

```
        % Synchronization between actions.
comm    s_require_remotequeueulock | r_require_remotequeuelock
          = c_require_remotequeuelock
        s_no_remotequeuewait | r_no_remotequeuewait
          = c_no_remotequeuewait
        s_signal_remotequeuewait | r_signal_remotequeuewait
          = c_signal_remotequeuewait
        s_o_data_return | r_data_return
          = c_o_data_return

proc    RemoteQueue(pid:ProcessId) =
        % Remote queue gets a Data_Return message.
        % To deal with it, the remotequeue_lock is needed.
        ∑tid:ThreadId ∑pid':ProcessId ∑r:Region ∑b:Bool
        % Put a message into the queue.
          r_o_data_return(tid,pid',pid,r,b).s_require_remotequeuelock(pid).
          (r_no_remotequeuewait(pid)+r_signal_remotequeuewait(pid)).
        % The processor takes this message.
          s_o_data_return(tid,pid',pid,r,b).RemoteQueue(pid)
```

Table 7.9: Specification of a remote queue

In μCRL, each processor is modeled as a separate component (with a unique identity). How a processor deals with Region_Sponmigrate and Data_Request messages is specified in Table 7.7.

Each processor maintains two message queues to store incoming messages. The *HomeQueue* is designed to buffer messages containing a request, while the *RemoteQueue* buffers messages containing a reply. For example, when a thread tries to get an up-to-date data copy from a region's home, first a Data_Request message is put into the home node's HomeQueue. When a Data_Return message arrives, it is put into the RemoteQueue of the thread's processor. The μCRL process for a message queue contains one parameter *pid* to indicate to which processor this message queue belongs to (see Table 7.8 and Table 7.9). To avoid state explosion, we only modeled queues that can contain one message.

Protocol locks

As already explained in the specification of threads, *protocol locks* guarantee exclusivity when threads write to or flush a region. Each processor acts as the protocol lock manager of its regions and region copies. To acquire a protocol lock, a protocol lock request message is sent to the region's home. If the lock is available, the manager replies with a grant message. Otherwise, the requester needs to wait for the lock to be released, and the protocol lock manager adds the requester into the lock's waiting list. To unlock, the current lock owner sends

```
        % We only present those parameters whose values are changed.
proc    Locker(pid:ProcessId,faulters:Bool,flushers:Bool,
          homequeue:Bool,remotequeue:Bool,wait_faulters:Natural,
          wait_flushers:Natural,wait_homequeue:Natural,
          wait_remotequeue:Natural)=
        % Get a request for the fault lock. If this lock can be granted,
        % send a no-wait message.
        r_require_faultlock(pid).
        (s_no_faultwait(pid).Locker(t/faulters)
        ◁and(faulters,flushers)▷
        % Otherwise, increase the number of threads waiting for this lock.
        % Later on, the thread waiting on fault lock will be signaled.
        Locker(S(wait_faulters)/wait_faulters))
        % The fault lock is released, if a thread can be notified,
        % send a signal_wait message, and decrease the waiting number.
        + r_free_faultlock(pid).
        ((s_signal_homequeuewait(pid).
          Locker(f/faulters,t/homequeue,
            sub1(wait_homequeue)/wait_homequeue)
          ◁and(not(eq(wait_homequeue,0)),homequeue)▷...)
        ◁and(not(and(eq(wait_homequeue,0),
          eq(wait_remotequeue,0))),flushers)▷...)
        + ...
```

Table 7.10: Part of the specification of a protocol lock management

an unlock message to the protocol lock manager. When the manager gets an unlock message, it checks whether a thread waiting for this lock can be notified, under some constraints. For instance, a fault_lock can be granted only if this fault_lock and the flush_lock are not held by other threads.

There are five protocol locks for each processor: *homequeue_lock*, *remotequeue_lock*, *server_lock*, *fault_lock* and *flush_lock*. The homequeue_lock and remotequeue_lock are needed to make sure that the handling of a popped message from a HomeQueue or a RemoteQueue by its processor is completed before the next message is popped from the queue. The cache coherence protocol allows writes to a region at home and from remote to happen concurrently. The server_lock, fault_lock and flush_lock ensure exclusivity between threads at a processor. The server_lock and flush_lock must be mutually exclusive for the home of a region, to protect the integrity of region data values and other region's information; likewise, the fault_lock and flush_lock must be mutually exclusive for non-home nodes of a region. When a thread writes at home or from remote, the server_lock or the fault_lock of the thread's processor is needed, respectively. When a thread flushes, the flush_lock of its processor is needed.

Protocol lock management of a processor is modeled in μCRL as a separate

component (see Table 7.10). Each protocol lock is modeled as a boolean variable, since a protocol lock can be held by at most one thread at a time. The waiting list of a lock is modeled as a natural number, representing the number of threads in the waiting list, to enable checking for emptiness; waiting lists do not need to contain thread identities, since waiting and notification are specified by means of a pair of synchronized actions, carrying the identity of the waiting thread as a parameter. When a protocol lock is available, the protocol lock manager randomly selects a waiting thread to notify.

Assertions from the developers

The developers added many assertions into the description and required that the protocol should not violate any of them. The assertions are modeled as a part of the μCRL specification. They can be divided into two classes: *order assertions* and *preconditions*.

- Order assertions: This class of assertions imposes a certain order on the usage of the system's resources. For example, when a thread performs an action on a region, the corresponding protocol lock should already be held by the thread. Order assertions are modeled in μCRL by imposing a certain order on the execution of actions. In the aforementioned example, in the μCRL specification, the behavior of a thread is modeled like this: only after execution of the action *r_no_serverwait* or *r_signal_serverwait*, the thread can access a region at home.
- Preconditions: This class of assertions requires that only when a certain precondition is satisfied, the description after it can be executed. For example, only under certain conditions (see Section 7.4.4) the home of the region automatically migrates. Preconditions are modeled in the μCRL specification as boolean terms in conditional expressions.

7.5.2 Requirements

We formulated three requirements for the cache coherence protocol.

1. Deadlock freeness: The protocol never ends up in a state where it cannot perform any action.
2. Relaxed cache coherence: For each region, at any time there exists one home node.
3. Liveness: Requests for writing to or flushing a region cannot be bounced around the network forever.

7.5.3 Validation of the requirements

The μCRL toolset was used to check the syntax and the static semantics of the specification, and also to transform it into a linear form. The linear form

was used to generate LTSs for various configurations of processors and threads. Next, we validated the three requirements with respect to these configurations.

Requirement 1

We used the μCRL toolset to check for deadlocks. This deadlock checking exercise led to the detection of many mistakes both in the informal description and in the μCRL specification of the protocol. For the first case, when the developers extracted a C-like description of the protocol from its implementation, they abstracted away from certain implementation details; some of these details were actually crucial for the correctness of the μCRL specification. For the second case, at some points the analyzers understood the description differently from what the developers really meant. Whenever a deadlock trace was found, it was simulated to understand the reason for the deadlock. This analysis took us a lot of time, since many of the traces were quite long (typically more than 300 transitions) and difficult to comprehend. Whenever a mistake was found, the μCRL specification was adapted and checked for deadlocks again.

One deadlock found by the analyzers, on a configuration of two processors each containing one thread, was a real problem in the implementation. When a thread wants to write to a region from remote, it acquires the fault_lock of its home node by sending a lock message. If the lock is unavailable, the thread waits for the lock to be released. Whenever it is notified, it continues with its access to the region and holds the fault_lock until it sends an unlock message to the home node. In the deadlock trace, we found that while a thread is waiting for a fault_lock, the home of the region may migrate to the thread's processor. Then in fact the thread writes to the region at home, it needs to acquire the server_lock instead of the fault_lock. This error resulted in a deadlock in the implementation. The chosen solution is that after a thread obtains a fault_lock, it checks whether it still writes from remote. If this is not the case, it sends an unlock message to release the held fault_lock, and then sends a message to acquire the server_lock. After fixing this problem as proposed, no more deadlocks were found.

Requirement 2

Due to automatic home node migration, it needs to be checked that at any time there exists at most one home node for each region. We divided this requirement into two parts.

2.1 Each region has at most one home node.

2.2 If the system is stable, each region has no more than $n - 1$ copies, where n is the number of processors.

To verify these two parts, actions *s_home* and *r_home* were added to the specification of a region, when a region finds that its location equals its home node; *s_copy* and *r_copy* were added, when a region finds that its location does not

```
        % Synchronization between actions.
comm    s_home | r_home = c_home
        s_copy | r_copy = c_copy

proc    Region(pid:ProcessId, r:Region)=
        % This part remains the same as before.
        ... +
        % s_home, r_home indicate pid is the home.
        r_home.Region(pid,r)◁eq(pid,gethome(r))▷δ
        + s_home.Region(pid,r)◁eq(pid,gethome(r))▷δ
        % s_copy, r_copy indicate pid has a copy.
        + δ◁eq(pid,gethome(r))▷r_copy.Region(pid,r)
        + δ◁eq(pid,gethome(r))▷s_copy.Region(pid,r)
```

Table 7.11: Modified specification of a region

equal its home node. We synchronized s_home and r_home into *c_home*, s_copy and r_copy into *c_copy* (see Table 7.11). Furthermore, we encapsulated s_home, r_home, s_copy and r_copy, so that these actions are forced to synchronize.

We verified requirement 2.1 by checking the absence of c_home in the generated LTSs. This is formulated in the regular alternation-free μ-calculus (see Section 2.4) as follows:

2.1 $[\mathsf{T}^*\cdot\mathsf{c\_home}]\ \mathsf{F}$

It says that if an execution sequence contains c_home, then in the resulting state false holds. This formula was checked to be true by Evaluator, a model checker from the CADP toolset.

For requirement 2.2, a stable state of a system means that no protocol lock is held, and that the message queues are empty. We added actions *homequeue_empty* and *remotequeue_empty* to the μCRL specification of queues to indicate that queues are empty, and added an action *lock_empty* to the specification of the protocol lock manager to indicate that no lock is held. Then for a model with two processors, we checked that the generated LTS does not contain a state which can perform c_copy, lock_empty, homequeue_empty and remotequeue_empty. This requirement is presented in the regular alternation-free μ-calculus as follows:

2.2 $\neg\langle\mathsf{T}^*\rangle\ (\langle\mathsf{c\_copy}\rangle\ \mathsf{T} \wedge \langle\mathsf{lock\_empty}\rangle\ \mathsf{T} \wedge \langle\mathsf{homequeue\_empty}\rangle\ \mathsf{T} \wedge \langle\mathsf{remotequeue\_empty}\rangle\ \mathsf{T})$

Note that the above two formulas only work for configurations with two processors, meaning that there are two copies for each region.

A second error in the implementation of the protocol was found while model checking this property on a configuration of two processors, with two threads running on one processor and a third thread on the other processor. The error

can happen when a thread is writing to a region from remote. During its waiting for an up-to-date copy of the region from the region's home, the home node may migrate (by a Region_Sponmigrate message) to the processor where the thread resides. When the Data_Return message with an up-to-date copy of the region arrives, the thread refreshes the region's home by the sender of the answer message. In the resulting state of the protocol, neither of the two processors is the home of the region. So c_copy may happen even in a stable state. The chosen solution is that when a processor gets a Region_Sponmigrate message, it informs those local threads that are writing to the region at the previous home node, so that these threads will behave as writing at home. After fixing this problem as proposed, property 2.2 was successfully model checked.

Requirement 3

The third requirement, that requests of writing to or flushing a region cannot be bounced around the network forever, is a liveness property. Actions *writeover* and *flushover* were added to the μCRL specification of a thread to indicate that a thread completed its pending actions. The following shows the code in the regular alternation-free μ-calculus for this requirement.

3.1 A thread eventually finishes writing to a region:

$[\mathsf{T}^{*}\cdot\mathsf{write}(\star)]\ \mu Y.\langle\mathsf{T}\rangle\ \mathsf{T} \wedge [\neg\mathsf{writeover}(\star)]\ Y$

3.2 A thread eventually finishes its flush of a region:

$[\mathsf{T}^{*}\cdot\mathsf{flush}(\star)]\ \mu Y.\langle\mathsf{T}\rangle\ \mathsf{T} \wedge [\neg\mathsf{flushover}(\star)]\ Y$

We use '$\star$' to indicate any identity of a thread. These two formulas express that after a thread initiates its action (writer($\star$) or flush($\star$)), the end of this action (writeover($\star$) or flushover($\star$)) is inevitable. This requirement was successfully model checked on two configurations.

7.5.4 Verification results

We applied advanced techniques for generating LTSs on a cluster at CWI, consisting of eight nodes. Each node is a dual AMD Athlon MP 1600+ system, with 1.4Ghz processors 2GB RAM and 40GB disk. The nodes are connected by a private ethernet network (100baseT switch) and by a public fast ethernet network (1000baseT switch). Our case study benefited a lot from the μCRL distributed LTS generation tool [22], and also pushed forward its development.

The sizes of the generated LTSs and the verification results are summarized in Table 7.12. Due to the complexity of this protocol, the size of the LTS grows very rapidly with respect to the number of threads and processors. With the current μCRL toolset, we could generate LTSs for the following three configurations: 1) two processors, each with one thread; 2) two processors, one with one thread, the other with two threads; 3) three processors, each with one thread. For the third configuration, we could only check the first requirement, because

Configuration	States	Transitions	Requirements Checked
1	65,234	460,162	1, 2, 3
2	5,424,848	40,476,069	1, 2, 3
3	82,371,105	893,181,444	1

Table 7.12: Verification results

the generated LTS was too large to serve as input to the model checker. The shortest error traces for the two flaws in the original implementation of the protocol that were detected during the model checking phase (see Section 7.5.3) both consisted of more than 100 transitions.

7.6 Conclusions

In this chapter, we used formal specification and model checking techniques to analyze a cache coherence protocol for a Java DSM implementation. We specified the protocol in μCRL and analyzed it. Some general requirements were formulated and verified for several configurations. Our analysis uncovered a lot of inconsistencies between the description and the implementation of this protocol. Two errors were found and fixed in the implementation, which improved the design and implementation of this protocol.

During the specification and analysis phase, we encountered quite a few difficulties. First, it took a relatively long time to obtain a μCRL specification of the protocol. During this period, the developers made important changes to the protocol, so that the μCRL specification had to be updated a number of times. Such gaps between an implementation and its formal model could be avoided if formal methods were used at an earlier design phase. Second, both the developers and analyzers made mistakes in their work. In our analysis, many deadlocks were due to the inconsistencies and misunderstandings. Third, more advanced techniques for distributed/parallel state space generation, reduction, and model checking are highly needed. Our future work will mainly focus on verifying whether the cache coherence protocol implements the JMM in [67, Chapter 17], and checking the requirements on more configurations.

Chapter 8

Simplifying Itai-Rodeh Leader Election for Anonymous Rings

8.1 Introduction

Leader election is the problem of electing a unique leader in a network, in the sense that the leader (process) knows that it has been elected and the other processes know that they have not been elected. Leader election algorithms require that all processes have the same local algorithm and that each computation terminates, with one process elected as leader. This is a fundamental problem in distributed computing and has numerous applications. For example, it is an important tool for breaking symmetry in a distributed system. By choosing a process as the leader it is possible to execute centralized protocols in a decentralized environment. Leader election can also be used to recover from token loss for token-based protocols, by making the leader responsible for generating a new token when the current one is lost.

There exists a broad range of leader election algorithms; see e.g. the summary in the text books [167, 116]. These algorithms have different message complexity in worst and/or average case. Furthermore, they vary in communication mechanism (*asynchronous vs. synchronous*), process names (*unique identities vs. anonymous*), and network topology (e.g. *ring, tree, complete graph*).

A first leader election algorithm for unidirectional rings was given by Le Lann [113]. It requires that each process has a unique identity, with a total ordering on identities; the process with the largest identity becomes the leader. The basic idea of Le Lann's algorithm is that each process sends a message around the ring bearing its identity. Thus it requires a total of n^2 messages, where n is the number of processes in the ring. Chang and Roberts [32] improved Le Lann's algorithm by letting only the message with the largest identity complete the round trip; their algorithm still requires in the order of n^2 messages in the worst case, but only $n \log n$ on average. Franklin [58] developed an leader election algorithm for bidirectional rings with a worst-case message complexity of $\mathcal{O}(n \log n)$. Peterson [138] and Dolev, Klawe, and Rodeh [44] independently adapted Franklin's algorithm so that it also works for unidirectional rings. All

the above algorithms work both for asynchronous and for synchronous communication, and do not require a priori knowledge about the number of processes.

Sometimes the processes in a network cannot be distinguished by means of unique identities. First, as the number of processes in a network increases, it may become difficult to keep the identities of all processes distinct; or a network may accidentally assign the same identity to different processes. Second, identities cannot always be sent around the network, for instance for reasons of efficiency. An example of the latter is FireWire, the IEEE 1394 high performance serial bus (see Section 8.2 for a more detailed description). A leader election algorithm that works in the absence of unique process identities is also desirable from the standpoint of fault tolerance. In an *anonymous network*, processes do not carry an identity. Angluin [5] showed that there does not exist a terminating algorithm for electing a leader in an asynchronous anonymous network. According to this result, a *Las Vegas* algorithm (meaning that the probability that the algorithm terminates is greater than zero, and all terminal configurations are correct) is the best possible option.

Itai and Rodeh [95, 96] proposed a probabilistic leader election algorithm for anonymous unidirectional rings, based on the Chang-Roberts algorithm. Each process selects a random identity from a finite domain, and processes with the largest identity start a new election round if they detect a name clash. It is assumed that the size of the ring is known to all processes, so that each process can recognize its own message (by means of a hop counter that is part of the message). The Itai-Rodeh algorithm is a Las Vegas algorithm that terminates with probability one; it takes $n \log n$ messages on average.

The Itai-Rodeh algorithm makes no assumptions about channel behavior, except fair scheduling. An old message, that has been overtaken by other messages in the ring, could in principle result in a situation where no leader is elected (see Figure 8.1 in Section 8.3.2). In order to avoid this problem, the algorithm proceeds in successive rounds, and each process and message is supplied with a round number. Thus an old message can be recognized and ignored. Due to the use of round numbers, the Itai-Rodeh algorithm has an infinite state space.

In this chapter, we make the assumption that channels are FIFO. We show that in this case round numbers can be omitted from the Itai-Rodeh algorithm. We present two adaptations of the Itai-Rodeh algorithm, that are correct in the presence of FIFO channels. In the first algorithm, a process may only choose a new identity when its message has completed the round trip, as is the case in the Itai-Rodeh algorithm. In the second algorithm, a process selects a new identity as soon as it detects that another process in the ring carries the same identity (even though this identity may not be the largest one in the ring). Since both algorithms do not use round numbers, they are finite-state. This means that we can apply model checking [35] to automatically verify properties of an algorithm, specified in some temporal logic. These properties can be checked against the explicit (finite) state space of the algorithm, for specific ring sizes. We used PRISM [107], a model checker that can be used to model and analyze systems containing probabilistic aspects. We specified both algorithms in the PRISM language, and for rings up to size four we verified the property: "with

probability one, eventually exactly one leader is elected". Furthermore, we present a manual correctness proof for both algorithms, for arbitrary ring size.

PRISM offers the possibility to calculate the probability that our algorithms have terminated after some number of messages. These statistics show that the first algorithm on average requires more messages to terminate than the second algorithm.

Finally, we show that if processes can select identities from a set of only two elements, then our algorithms also work correctly for non-FIFO channels.

Outline of the chapter. Related work is summarized in Section 8.2. Section 8.3 contains the original Itai-Rodeh algorithm. In Sections 8.4 and 8.5, we present two probabilistic leader election algorithms for anonymous rings with FIFO channels. We explain our verification results with PRISM, and give a manual correctness proof for each algorithm. Section 8.6 reveals some experimental results using PRISM on the number of messages needed to terminate. In Section 8.7, we prove that if the domain of identities contains only two elements, the requirement that channels are FIFO can be dropped. We conclude this chapter in Section 8.8.

8.2 Related Work

On the web page of PRISM (`http://www.cs.bham.ac.uk/~dxp/prism/`), the Itai-Rodeh algorithm for asynchronous rings was adapted for synchronous rings. In PRISM, processes synchronize on action labels, so a synchronous ring can simply be modeled by excluding channels from the specification. Processes are synchronized in the same round, thus round numbers are not needed (similar to our Algorithm $\mathcal{A}$). The state space therefore becomes finite, and PRISM could be used to verify the property "with probability one, eventually a unique leader is elected", for rings up to size eight. Also the probability of electing a leader in one round was calculated.

Garavel and Mounier [62] described both the Chang-Roberts algorithm and Le Lann's algorithm using the process algebraic language LOTOS. They studied these two algorithms in the presence of unreliable communication network and/or unreliable processes and suggested some improvements. Their verification was performed using the model checker CADP. Fredlund *et al.* [60] gave a manual correctness proof of the Dolev-Klawe-Rodeh algorithm in the process algebraic language μCRL, for arbitrary ring size. Brunekreef *et al.* [26] designed a number of leader election algorithms for a broadcast network, where processes may participate and crash spontaneously. They used linear-time temporal logic to manually prove that the algorithms satisfy their requirements.

The IEEE 1394 high performance serial bus (called "FireWire") is used to transport video and audio signals within a network of multimedia devices. In the tree identify phase of IEEE 1394, which takes place after a bus reset in the network, a leader is elected. For the sake of performance, identities of nodes cannot be sent around the network, so that it is basically an anonymous

network. The leader election algorithm in the IEEE 1394 standard works for acyclic, connected networks. If a cycle is present, it produces a timeout. The algorithm has been specified and verified with a number of different formal techniques. We give an overview of these case studies.

Shankland and van der Zwaag [157] manually verified the leader election algorithm in μCRL, at three different levels of detail. Shankland and Verdejo [156] used E-LOTOS to manually verify the algorithm. Abrial *et al.* [1] used an event-driven approach with the B Method to develop mathematical models of the algorithm; the internal consistency of each model as well as its correctness with regard to its previous abstraction were proved mechanically. Verdejo *et al.* [175] described the algorithm at different abstract levels, using the language Maude based on rewriting logic; they verified the algorithm by an exhaustive exploration of the state space that always exactly one leader is chosen. Moreover, they gave a manual correctness proof for general acyclic networks. Devillers *et al.* [39] verified the algorithm using an I/O automata model; the main part of their proof has been checked with the theorem prover PVS. Romijn [150] extended their I/O automata model with timing parameters from the IEEE 1394 standard, and manually proved that under certain timing restrictions the algorithm behaves correctly. Calder and Miller [28] verified some properties of the algorithm using the model checker Spin, for networks with up to six nodes. Schuppan and Biere [155] used the model checker SMV to check the correctness of the algorithm for networks with up to ten nodes.

8.3 Itai-Rodeh Leader Election

We consider an *asynchronous*, *anonymous*, *unidirectional* ring consisting of $n \geq 2$ processes $p_0, \ldots, p_{n-1}$. Processes communicate asynchronously by sending and receiving messages over channels, which are assumed to be reliable. Channels are unidirectional: a message sent by p_i is added to the message queue of $p_{(i+1) \bmod n}$. The message queues are guided by a *fair scheduler*, meaning that in each infinite execution sequence, every sent message eventually arrives at its destination. Processes are anonymous, so they do not have unique identities. The challenge is to present a uniform local algorithm for each process, such that one leader is elected among the processes.

8.3.1 The Itai-Rodeh algorithm

Itai and Rodeh [95, 96] studied how to break the symmetry in anonymous networks using probabilistic algorithms. They presented a probabilistic algorithm to elect a leader in the above network model, under the assumption that processes know that the size of the ring is n. It is a Las Vegas algorithm that terminates with probability one. The Itai-Rodeh algorithm is based on the Chang-Roberts algorithm [32], where processes are assumed to have unique identities, and each process sends out a message carrying its identity. Only the message with the largest identity completes the round trip and returns to its originator, which becomes the leader.

In the Itai-Rodeh algorithm, each process selects a *random identity* from a finite set. So different processes may carry the same identity. Again each process sends out a message carrying its identity. Messages are supplied with a *hop counter*, so that a process can recognize its own message (by checking whether the hop counter equals the ring size n). Moreover, a process with the largest identity present in the ring must be able to detect whether there are other processes in the ring with the same identity. Therefore each message is supplied with a bit, which is dirtied when it passes a process that is not its originator but shares the same identity. When a process receives its own message, either it becomes the leader (if the bit is clean), or it selects a new identity and starts the next election round (if the bit is dirty). In this next election round, only processes that shared the largest identity in the ring are *active*. All other processes have been made *passive* by the receipt of a message with an identity larger than their own. The active processes maintain a *round number*, which initially starts at zero and is augmented at each new election round. Thus messages from earlier election rounds can be recognized and ignored.

We proceed to present a detailed description of the Itai-Rodeh algorithm.

The Itai-Rodeh algorithm.

- Initially, all processes are active, and each process p_i randomly selects its identity $id_i \in \{1, \ldots, k\}$ and sends the message $(id_i, 1, 1, \mathit{true})$.
- Upon receipt of a message $(id, round, hop, bit)$, a passive process p_i ($state_i = \mathit{passive}$) passes on the message, increasing the counter hop by one; an active process p_i ($state_i = \mathit{active}$) behaves according to one of the following steps:
  - if $hop = n$ and $bit = \mathit{true}$, then p_i becomes the leader ($state'_i = \mathit{leader}$);
  - if $hop = n$ and $bit = \mathit{false}$, then p_i selects a new random identity $id'_i \in \{1, \ldots, k\}$, moves to the next round ($round'_i = round_i + 1$), and sends the message $(id'_i, round'_i, 1, \mathit{true})$;
  - if $(round, id) = (round_i, id_i)$ and $hop < n$, then p_i passes on the message $(id, round, hop + 1, \mathit{false})$;
  - if $(round, id) > (round_i, id_i)$,[a] then p_i becomes passive ($state'_i = \mathit{passive}$) and passes on the message $(id, round, hop + 1, bit)$;
  - if $(round, id) < (round_i, id_i)$, then p_i purges the message.

[a] We compare $(round, id)$ and $(round_i, id_i)$ lexicographically.

Each process p_i maintains three parameters:

- $id_i \in \{1, \ldots, k\}$, for some $k \geq 2$, is its identity;
- $state_i$ ranges over $\{\mathit{active}, \mathit{passive}, \mathit{leader}\}$;

- $round_i \in \mathbb{N}^+$ represents the number of the current election round.

Only active processes may become the leader; passive processes simply pass on messages. At the start of a new election round, each active process sends a message of the form $(id, round, hop, bit)$, where:

- the values of *id* and *round* are taken from the process that sends the message;
- *hop* is a counter that initially has the value one, and which is increased by one every time it is passed on by a process;
- *bit* is a bit that initially is *true*, and which is set to *false* when it visits a process that has the same identity but that is not its originator.

We say that an execution sequence of the Itai-Rodeh algorithm has *terminated* if each process is either passive or elected as leader, and there are no remaining messages in the channels.

Theorem 8.3.1 [95] The Itai-Rodeh algorithm terminates with probability one, and upon termination a unique leader has been elected.

8.3.2 Round numbers are needed

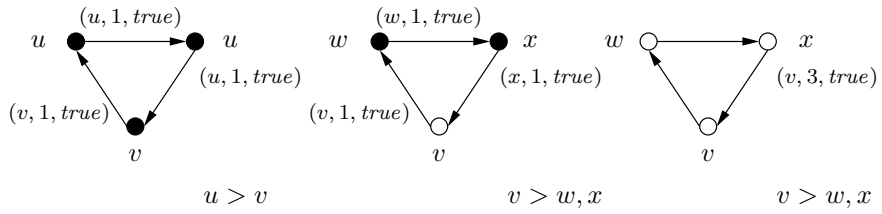


Figure 8.1: Round numbers are essential if channels are not FIFO

Figure 8.1 presents a scenario to show that if round numbers were omitted, the Itai-Rodeh algorithm could produce an execution sequence in which all processes become passive, so that no leader is elected. This example uses the fact that channels are not FIFO. Let $k \geq 3$. Figure 8.1 depicts a ring of size three; black processes are active and white processes are passive. Initially, all processes are active, and the two processes above select the same identity u, while the one below selects an identity $v < u$. (See the left side of Figure 8.1.) The three processes send a message with their identity, and at the receipt of a message with identity u, process v becomes passive. Since channels are not FIFO, the message $(v, 1, true)$ can be overtaken by the other two messages with identity u. The latter two messages return to their originators with a dirty bit. So the processes with identity u detect a name clash, select new identities $w < v$ and $x < v$, and send messages carrying these identities. (See the middle part

of Figure 8.1.) Finally, the message with identity v makes the processes with identities w and x passive. The three messages in the ring are passed on forever by the three passive processes. (See the right side of Figure 8.1.)

8.4 Leader Election without Round Numbers

We observe that if channels are FIFO, round numbers are redundant. Thus we obtain a simplification of the Itai-Rodeh algorithm. Algorithm $\mathcal{A}$ is obtained by considering only those cases in the Itai-Rodeh algorithm where the active process p_i and the incoming message have the same round number. Correctness of Algorithm $\mathcal{A}$ follows from the proposition below.

Algorithm $\mathcal{A}$.

- Initially, all processes are active, and each process p_i randomly selects its identity $id_i \in \{1, \ldots, k\}$ and sends the message $(id_i, 1, \mathit{true})$.
- Upon receipt of a message (id, hop, bit), a passive process p_i ($state_i = \mathit{passive}$) passes on the message, increasing the counter hop by one; an active process p_i ($state_i = \mathit{active}$) behaves according to one of the following steps:
  - if $hop = n$ and $bit = \mathit{true}$, then p_i becomes the leader ($state'_i = \mathit{leader}$);
  - if $hop = n$ and $bit = \mathit{false}$, then p_i selects a new random identity $id'_i \in \{1, \ldots, k\}$ and sends the message $(id'_i, 1, \mathit{true})$;
  - if $id = id_i$ and $hop < n$, then p_i passes on the message $(id, hop + 1, \mathit{false})$;
  - if $id > id_i$, then p_i becomes passive ($state'_i = \mathit{passive}$) and passes on the message $(id, hop + 1, bit)$;
  - if $id < id_i$, then p_i purges the message.

Proposition 8.4.1 Consider the Itai-Rodeh algorithm where all channels are FIFO. When an active process receives a message, then the round number of the process and of the message are always the same.

Proof. Let message $m = (id_j, round_j, hop, bit)$, which originates from process p_j, arrive at active process p_i. Suppose that up to this moment, messages never arrived at active processes with a different round number. We prove that $round_i = round_j$. We derive the desired equality in two steps.

- $round_i \leq round_j$.

  Let $round_i > 1$, for else we are done. Then a message m' with round number $round_i - 1$ originated at p_i and completed the round trip, where all the active processes that it visited had round number $round_i - 1$. FIFO

behavior guarantees that after m' returned to p_i, no other message with round number $\leq round_i - 1$ can have arrived at p_i. So $round_i \leq round_j$.

- $round_i \geq round_j$.

  Let $round_j > 1$, for else we are done. Then a message m'' with round number $round_j - 1$ originated at p_j and completed the round trip, where all the active processes that it visited (so in particular p_i) had round number $round_j - 1$. Since m'' completed the round trip and passed p_i while this process remained active, it follows that both p_i and p_j had the maximal identity in round $round_j - 1$. So the message m''' that originated at p_i with round number $round_j - 1$ also completed the round trip. FIFO behavior guarantees that m''' arrived at p_j before m'', so that m''' passed p_j before m was created at p_j. FIFO behavior guarantees that m''' arrived at p_i before m. So $round_i \geq round_j$.

Hence, $round_i = round_j$. ⊠

Theorem 8.4.2 Let channels be FIFO. Then Algorithm $\mathcal{A}$ terminates with probability one, and upon termination exactly one leader is elected.

Proof. By Theorem 8.3.1 together with Proposition 8.4.1, upon termination exactly one leader is elected. Namely, the execution traces are a subset of the execution traces of the Itai-Rodeh algorithm.

We have to redo the probability analysis, since a probabilistic result for a set of execution traces is not always inherited by subsets of execution traces.

When there are $\ell \geq 2$ active processes in the ring, these processes all remain active if and only if they all the time choose the same identity. Otherwise, at least one active process will become passive. The probability that all active processes select the same identity in one "round" is $(\frac{1}{k})^{\ell-1}$. So the probability for all ℓ active processes to choose the same identity m times in a row is $(\frac{1}{k})^{m(\ell-1)}$. Since $k \geq 2$, the probability that the number of active processes eventually decreases is one.

Clearly, when there is only one active process in the ring, it will be elected as the leader. After the round trip of its final message there are no remaining messages, because channels are FIFO. ⊠

8.4.1 Automated verification with PRISM

Owing to the elimination of round numbers, Algorithm $\mathcal{A}$ is finite-state, contrary to the Itai-Rodeh algorithm. Hence we can apply explicit state space generation and model checking to establish the correctness of Algorithm $\mathcal{A}$ for fixed ring sizes. This analysis of Algorithm $\mathcal{A}$ was actually performed before constructing the manual correctness proof of Algorithm $\mathcal{A}$ from the previous section, as a means to confirm our intuition that Algorithm $\mathcal{A}$ works correctly in case of FIFO channels. Moreover, this model checking exercise has some additional value compared to Theorem 8.4.2. Namely, since the manual proofs of Theorem 8.3.1, Proposition 8.4.1 and Theorem 8.4.2 were not formalized and checked with a theorem prover, there is no absolute guarantee that they are free of flaws.

A short introduction to PRISM

PRISM [107] is a probabilistic model checker. It allows one to model and analyze systems and algorithms containing probabilistic aspects. PRISM supports three kinds of probabilistic models: continuous-time Markov chains (CTMCs), discrete-time Markov chains (DTMCs) and Markov decision processes (MDPs). Analysis is performed through model checking such systems against specifications written in the probabilistic temporal logic PCTL [83, 11] if the model is a DTMC or an MDP, or CSL [10] in the case of a CTMC.

In order to model check probabilistic properties of Algorithm $\mathcal{A}$, we first encoded the algorithm as a DTMC model using the PRISM language, which is a simple, state-based language, based on the Reactive Modules formalism of Alur and Henzinger [4]. A system is composed of a number of modules that contain local variables, and that can interact with each other. The behavior of a DTMC is described by a set of commands of the form:

$$[a]\ g \rightarrow \lambda_1 : u_1 + \ldots + \lambda_\ell : u_\ell$$

a is an action label in the style of process algebras, which introduces synchronization into the model. It can only be performed simultaneously by all modules that have an occurrence of action label a in their specification. If a transition does not have to synchronize with other transitions, then no action label needs to be provided for this transition. The symbol g is a predicate over all the variables in the system. Each u_i describes a transition which the module can make if g is true. A transition updates the value of the variables by giving their new *primed* value with respect to their *unprimed* value. The λ_i are used to assign probabilistic information to the transition. It is required that $\lambda_1 + \cdots + \lambda_\ell = 1$. This probabilistic information can be omitted if $\ell = 1$ (and so $\lambda_1 = 1$). PRISM considers states without outgoing transitions as error states; terminating states can be modeled by adding a self-loop. A more detailed description of PRISM can be found in [107].

Verifying Algorithm $\mathcal{A}$ with PRISM

We used PRISM to verify that Algorithm $\mathcal{A}$ satisfies the probabilistic property "with probability 1, eventually exactly one leader is elected". We modeled each FIFO channel and each process as a separate module in PRISM. The following code in the PRISM language gives the specification for a channel of size two. The channel *channel1* receives a message (*mes1_id*,*mes1_counter*,*mes1_bit*) from process p_1 (synchronized on action label *rec_from_p1*) and sends it to process p_2 (synchronized on action label *send_to_p2*). Each position $i \in \{1, 2\}$ in the channel is represented by a triple of natural numbers: one for the process identity contained in a message (*b_1_2_i1*), one for the hop counter (*b_1_2_i2*), and one for the bit (*b_1_2_i3*). If the natural numbers for a position in a channel are greater than zero, it means this position is occupied by a message. Otherwise, the position is empty.

We present the channel between processes p_1 and p_2. Both the number of processes and the size of the identity set are two ($N = 2$; $K = 2$).

```
module channel1
    b_1_2_11: [0..K]; b_1_2_12:[0..N]; b_1_2_13:[0..1];
    b_1_2_21: [0..K]; b_1_2_22:[0..N]; b_1_2_13:[0..1];
    [rec_from_p1] b_1_2_11=0
            → (b_1_2_11'=mes1_id) & (b_1_2_12'=mes1_counter) &
              (b_1_2_13'=mes1_bit);
    [rec_from_p1] (b_1_2_11>0) & (b_1_2_21=0)
            → (b_1_2_21'=mes1_id) & (b_1_2_22'=mes1_counter) &
              (b_1_2_23'=mes1_bit);
    [send_to_p2] b_1_2_11>0
            → (b_1_2_11'=b_1_2_21) & (b_1_2_12'=b_1_2_22) &
              (b_1_2_13'=b_1_2_23) & (b_1_2_21'=0) &
              (b_1_2_22'=0) & (b_1_2_23'=0);
endmodule
```

mes1_id, *mes1_counter* and *mes1_bit* are *shared* variables. They are used in the module *process1* below for receiving and sending messages. Only in that module values can be assigned to these variables. *mes1_id* carries the identity of a message, *mes1_counter* its hop counter, and *mes1_bit* the clean (1) or dirty (0) bit. If no message is present, all three variables have the value zero. (So *mes1_bit* = 0 can have two meanings: either there is no message, or the bit is dirty.)

Each process p_i is specified by means of a variable *processi_id*:[0..K] for its identity (where 0 means that the process is passive or selecting a new identity), a variable *si*:[0..5] for its local state (this is explained below), and a variable *leaderi*:[0..1] (where in state 0 means that the process is passive, and 1 that it is the leader). The following PRISM code is the specification for process p_1.

```
module process1
    process1_id:[0..K]; s1:[0..5]; leader1:[0..1];
    mes1_id:[0..K]; mes1_counter:[0..N]; mes1_bit:[0..1];
```

When a process is in state 0, it is active and can randomly (modeled by the probability rate $R = 1/K$) select its identity, build a new message with this identity, and set its state to 1.

```
[ ] s1=0
    → R: (s1'=1) & (process1_id'=1) & (mes1_id'=1) &
      (mes1_counter'=1) & (mes1_bit'=1)
     + R: (s1'=1) & (process1_id'=2) & (mes1_id'=2) &
      (mes1_counter'=1) & (mes1_bit'=1);
```

When $s1 = 1$, the process sends the new message into channel 1 (modeled by a synchronization with module *channel1* on action *rec_from_p1*), and moves to state 2.

```
[rec_from_p1] s1=1
    → (s1'=2) & (mes1_id'=0) & (mes1_counter'=0) &
      (mes1_bit'=0);
```

In state 2 the process can receive a message from channel 2 (modeled by a synchronization with module *channel2* on action *send_to_p1*), and go to state 3. Note that *b_2_1_11, b_2_1_12* and *b_2_1_31* are shared variables, representing the first position in the module *channel2*.

```
[send_to_p1] s1=2
    → (s1'=3) & (mes1_id'=b_2_1_11) &
      (mes1_counter'=b_2_1_12) & (mes1_bit'=b_2_1_13);
```

When a process is in state 3, it has received a message and takes a decision. If the process got its own message back ($mes1\_counter = N$) and the bit of the message is clean ($mes1\_bit = 1$), the process is elected as the leader ($leader1' = 1$), and moves to state 4.

```
[ ] (s1=3) & (mes1_counter=N) & (mes1_bit=1)
    → (s1'=4) & (process1_id'=0) & (mes1_id'=0) &
      (mes1_counter'=0) & (mes1_bit'=0) & (leader1'=1);
```

If $mes1\_counter = N$ and $mes1\_bit = 0$, the process changes its state to 0 and will select a new random identity.

```
[ ] (s1=3) & (mes1_counter=N) & (mes1_bit=0)
    → (s1'=0) & (process1_id'=0) & (mes1_id'=0) &
      (mes1_counter'=0) & (mes1_bit'=0);
```

If $mes1\_id = process1\_id$ and $mes1\_counter < N$, the process has received a message with the same identity, but the message does not originate from itself. It increases the hop counter in the message by one, makes the bit dirty, and moves to state 5 to pass on the message.

```
[ ] (s1=3) & (mes1_id=process1_id) & (mes1_counter<N)
    → (s1'=5) & (mes1_counter'=mes1_counter+1) &
      (mes1_bit'=0);
```

If $mes1\_id < process1\_id$, the process purges the message, and moves back to state 2 to receive another message.

```
[ ] (s1=3) & (mes1_id<process1_id)
    → (s1'=2) & (mes1_id'=0) & (mes1_counter'=0) &
      (mes1_bit'=0);
```

If $mes1\_id > process1\_id$, the process increases the hop counter in the message by one, and goes to state 4 where it becomes passive (i.e., the value of *leader1* remains zero).

```
[ ] (s1=3) & (mes1_id>process1_id)
    → (s1'=4) & (process1_id'=0) &
      (mes1_counter'=mes1_counter+1);
```

In state 5, a process passes on a message, and moves to state 2.

```
[rec_from_p1] (s1=5)
    → (s1'=2) & (mes1_id'=0) & (mes1_counter'=0) &
      (mes1_bit'=0);
```

In state 4, a passive process (*leader1* = 0) can only pass on messages with their hop counter increased by one.

```
[send_to_p1] (s1=4) & (leader1=0) & (mes1_id=0)
    → (mes1_id'=b_2_1_11) & (mes1_counter'=b_2_1_12+1) &
      (mes1_bit'=b_2_1_13);
[rec_from_p1] (s1=4) & (leader1=0) & (mes1_id>0)
    → (mes1_id'=0) & (mes1_counter'=0) & (mes1_bit'=0);
```

We added the conjunct *leader1* = 0 to the predicate in order to emphasize that the leader does not have to deal with incoming messages. Namely, when a process is elected as the leader there are no remaining messages, owing to the fact that channels are FIFO.

A self-loop with synchronization on an action label *done* is added to processes in state 4, to avoid deadlock states.

```
    [done] (s1=4) → (s1'=s1);
endmodule
```

Other channels and processes can be constructed by carefully *module renaming* modules *channel1* and *process1*. The initial value of each variable is the minimal value in its range.

Below we specify the property "with probability 1, eventually exactly one leader is elected" for a ring with two processes as a PCTL formula:

```
Property: P>=1 [ true U (s1=4 & s2=4 & leader1+leader2=1 &
                         b_1_2_11+b_2_1_11=0) ]
```

It states that the probability that eventually both p_1 and p_2 get into state 4 ($s1 = 4 \wedge s2 = 4$), with exactly one process elected as the leader ($leader1 + leader2 = 1$), is at least one. In addition, we check that the algorithm terminates with no message in the ring ($b\_1\_2\_11 + b\_2\_1\_11 = 0$).

To model check this property, the algorithmic description (in the module-based language) was parsed and converted into an MTBDD [61]. In PRISM, reachability is performed to identify non-reachable states and the MTBDD is filtered accordingly. Table 8.1 shows statistics for each model we have built. The first part gives the parameters for each model: the ring size n, the size of the identity set, and the size of the channel. It is not hard to see that at any time there are at most n messages in the ring, so channel size n suffices; and having n different possible identities means that in each "round", all active processes can select a different identity. The second part gives the number of states and transitions in the MTBDD representing the model.

Property was successfully checked on all the ring networks in Table 8.1 (we used the model checker PRISM 2.0 with its default options). Note that for

	Processes	Identities	Channel size	FIFO	States	Transitions
Ex.1	2	2	2	yes	127	216
Ex.2	3	3	3	yes	5,467	12,360
Ex.3	4	3	4	yes	99,329	283,872

Table 8.1: Model checking result for Algorithm $\mathcal{A}$ with FIFO channels

$n = 4$, we could only check the property for an identity set of size three. For $n = 4$ and an identity set of size four, and in general for $n \geq 5$, PRISM fails to build a model due to the lack of memory.

8.5 Leader Election without Bits

In this section, we present another leader election algorithm, which is a variation of Algorithm $\mathcal{A}$. Again channels are assumed to be FIFO. We observe that when an active process p_i detects a name clash, meaning that it receives a message with its own identity and hop counter smaller than n, it is not necessary for p_i to wait for its own message to return. Instead p_i can immediately select a new random identity and send a new message. Algorithm $\mathcal{B}$ is obtained by adapting Algorithm $\mathcal{A}$ according to this observation. In particular all occurrences of bits are omitted.

Algorithm $\mathcal{B}$.

- Initially, all processes are active, and each process p_i randomly selects its identity $id_i \in \{1, \ldots, k\}$ and sends the message $(id_i, 1)$.
- Upon receipt of a message (id, hop), a passive process p_i ($state_i = passive$) passes on the message, increasing the counter hop by one; an active process p_i ($state_i = active$) behaves according to one of the following steps:
  - if $hop = n$, then p_i becomes the leader ($state'_i = leader$);
  - if $id = id_i$ and $hop < n$, then p_i selects a new random identity $id'_i \in \{1, \ldots, k\}$ and sends the message $(id'_i, 1)$;
  - if $id > id_i$, then p_i becomes passive ($state'_i = passive$) and passes on the message $(id, hop + 1)$;
  - if $id < id_i$, then p_i purges the message.

We first discuss the automatic verification of Algorithm $\mathcal{B}$ with PRISM in Section 8.5.1. Then we give a manual correctness proof for Algorithm $\mathcal{B}$, for arbitrary ring size, in Section 8.5.2.

8.5.1 Automated verification with PRISM

Channels are modeled in the same way as in Section 8.4. We present each process p_i with a variable *process* i*_id*:[0..K] for its identity, a variable $s\,i$:[0..4] for its local state, and a variable *leader* i:[0..1]. We present only part of the PRISM specification for process p_1. The parts when a process is in state $0, 1, 2$ or 4 are omitted, as this behavior is very similar to Algorithm $\mathcal{A}$ (see Section 8.4.1). State 5 is redundant here, because a process selects a new identity as soon as it detects a name clash.

```
module process1
    process1_id:[0..K]; s1:[0..4]; leader1:[0..1]; mes1_id:[0..K];
    mes1_counter:[0..N];
```

When a process in state 3, it has received a message from the channel and takes a decision. If *mes1_counter* $= N$, the process is elected as the leader (*leader1*$' = 1$), and moves to state 4.

```
    [ ] (s1=3) & (mes1_counter=N)
        → (s1'=4) & (process1_id'=0) & (mes1_id'=0) &
          (mes1_counter'=0) & (leader1'=1);
```

If *mes1_id* $=$ *process1_id* and *mes1_counter* $< N$, the process goes back to state 0 and will select a new identity.

```
    [ ] (s1=3) & (mes1_id=process1_id) & (mes1_counter<N)
        → (s1'=0) & (mes1_id'=0) & (mes1_counter'=0) &
          (process1_id'=0);
```

If *mes1_id* $<$ *process1_id*, the process purges the message, and moves back to state 2 to receive another message.

```
    [ ] (s1=3) & (mes1_id<process1_id)
        → (s1'=2) & (mes1_id'=0) & (mes1_counter'=0);
```

If *mes1_id* $>$ *process1_id*, the process becomes passive, increases the hop counter of the message by one, and goes to state 4.

```
    [ ] (s1=3) & (mes1_id>process1_id)
        → (s1'=4) & (process1_id'=0) &
          (mes1_counter'=mes1_counter+1);
    ...
endmodule
```

Other channels and processes can be constructed by module renaming.

Property was successfully model checked with respect to Algorithm $\mathcal{B}$, in a setting with FIFO channels, for rings up to size five. For any larger ring size, and in case of ring size five and an identity domain containing three elements, PRISM fails to produce an MTBDD. Table 8.2 summarizes the verification results for Algorithm $\mathcal{B}$ with PRISM.

	Processes	Identities	Channel size	FIFO	States	Transitions
Ex.1	2	2	2	yes	97	168
Ex.2	3	3	3	yes	6,019	14,115
Ex.3	4	3	4	yes	176,068	521,452
Ex.4	4	4	4	yes	537,467	1,615,408
Ex.5	5	2	5	yes	752,047	2,626,405

Table 8.2: Model checking result for Algorithm $\mathcal{B}$ with FIFO channels

8.5.2 The correctness proof

In this section we give a correctness proof for Algorithm $\mathcal{B}$, in case of FIFO channels, with respect to ring networks of arbitrary size.

Definition 8.5.1 The processes and messages *between* a process p and a message m are the ones that are encountered when traveling in the ring from p to m.

Lemma 8.5.2 Let active process p have identity id_p and message m have identity id_m. If $id_p \neq id_m$, then there is an active process or message between p and m with an identity $\geq \min\{id_p, id_m\}$.

Proof. We apply induction on execution sequences.
Basis: Prior to the first arrival of a message, every process is active and has generated a message with its own identity; thus the lemma trivially holds.
Induction step: When a message arrives at a passive process, it is simply forwarded. Assume a message $m = (id, hop)$ arrives at an active process p_i with identity id_i. If $hop = n$, then p_i is elected as the leader. Since channels are FIFO, in this case the round trip of the final message of p_i guarantees that there are no remaining messages; thus the lemma trivially holds. Now suppose that $hop < n$. We consider three cases. In each case we only consider each pair of an active process and a message that could violate the condition of the lemma due to the arrival of m at p_i.

- $id_i > id$. Then m is purged by p_i.

  Let p_j be an active process with identity id_j and m' a message with identity id', such that p_i and m are between p_j and m', and $id \geq \min\{id_j, id'\}$. The active process p_i between p_j and m' has identity $id_i > \min\{id_j, id'\}$.

- $id_i < id$. Then p_i becomes passive and sends the message $(id, hop + 1)$.

  Let p_j be an active process with identity id_j and m' a message with identity id', such that p_i and m are between p_j and m', and $id_i \geq \min\{id_j, id'\}$. The message $(id, hop + 1)$ between p_j and m' has identity $id > \min\{id_j, id'\}$.

- $id_i = id$. Then p_i selects a new identity id'_i and sends the message $(id'_i, 1)$.

  We consider three cases, covering each pair of an active process and a message with different identities that is either newly created (the first two cases) or that could violate the condition of the lemma due to the new identity of p_i (the third case).

  CASE 1: For any message m' with identity $id' \neq id'_i$, $(id'_i, 1)$ is a message between p_i and m' with identity $id'_i \geq min\{id'_i, id'\}$.

  CASE 2: For any active process p_j with identity $id_j \neq id'_i$, p_i is an active process between p_j and $(id'_i, 1)$ with identity $id'_i \geq min\{id_j, id'_i\}$.

  CASE 3: Let p_j be an active process with identity id_j and m' a message with identity $id' \neq id_j$, such that p_i and m are between p_j and m', and $id_i \geq \min\{id_j, id'\}$. Since $id' \neq id_j$, either $id_j \neq id_i$ or $id_i \neq id'$. So by induction there is an active process or message either between p_j and m with an identity $\geq \min\{id_j, id_i\}$, or between p_i and m' with an identity $\geq \min\{id_i, id'\}$. Since $id_i \geq \min\{id_j, id'\}$, in either case there is an active process or message between p_j and m' with an identity $\geq \min\{id_j, id'\}$.

⊠

Definition 8.5.3 An active process p is *related to* a message m if they have the same identity id, and all active processes and messages between p and m have an identity smaller than id.

Lemma 8.5.4 Let active process p be related to message m. Let ξ be the maximum of all identities of active processes and messages between p and m ($\xi = 0$ if there are none).

1. Between p and m, there is an equal number of active processes and of messages with identity ξ; and
2. if p is not the originator of m, then there is an active process or message between p and m.

Proof. We apply induction on execution sequences.
Basis: Prior to the first arrival of a message, every process is active and has generated a message with its own identity; thus the lemma trivially holds.
Induction step: When a message arrives at a passive process, it is simply forwarded. Assume a message $m = (id, hop)$ arrives at an active process p_i with identity id_i. If $hop = n$, then p_i is elected as the leader. Since channels are FIFO, in this case the round trip of the final message of p_i guarantees that there are no remaining messages; thus the lemma trivially holds. Now suppose that $hop < n$. We consider three cases. In each of these cases we only consider related pairs that were either created or affected by the arrival of m at p_i.

- $id_i > id$. Then m is purged by p_i.

Let p_i be between an active process p_j and a message m'. Clearly, id is not the maximal identity of active processes and messages between p_j and m'. So if p_j and m' are related after the purging of m, they were also related before this moment. Hence, by induction, the pair p_j and m' satisfies condition 1 of the lemma. Furthermore, p_i is an active process between p_j and m', so the pair also satisfies condition 2.

- $id_i < id$. Then p becomes passive and sends the message $(id, hop + 1)$.

  If an active process p' is related to $(id, hop+1)$, then clearly it was also related to m. So by induction the pair p' and $(id, hop+1)$ satisfies conditions 1 and 2.

  Let p_i and $(id, hop+1)$ be between an active process p_j and a message m'. Clearly, id_i is not the maximal identity of active processes and messages between p_j and m'. So if p_j and m' are related after p_i has become passive, they were also related before this moment. Hence, by induction, the pair p_j and m' satisfies condition 1 of the lemma. Furthermore, $(id, hop + 1)$ is a message between p_j and m', so the pair also satisfies condition 2.

- $id_i = id$. Then p_i selects a new identity id'_i and sends the message $(id'_i, 1)$.

  Note that p_i is the only active process related to $(id'_i, 1)$, and vice versa. Clearly, conditions 1 and 2 of the lemma are satisfied by this pair.

  Let an active process p_j with identity id_j be related to a message m', such that p_i and $(id'_i, 1)$ are between p_j and m'. Since p_i is between p_j and m', condition 2 is satisfied by this pair. We proceed to prove condition 1 for this pair. We consider three cases.

  Case 1: $id_i > id_j$. Then by Lemma 8.5.2 there is an active process or message between p_i and m' with identity $\geq id_j$. This active process or message is also between p_j and m', which contradicts the fact that p_j is related to m'.

  Case 2: $id_i < id_j$. Then p_j and m' were already related before m reached p_i, so by induction this pair satisfied condition 1 before m reached p_i. Let ξ denote the maximum of all identities of active processes (and of messages) between p_j and m' before m reached p_i; and let $\#$ denote the number of active processes (and of messages) between p_j and m' with identity ξ before m reached p_i. Moreover, let ξ'_π and ξ'_μ denote the maximum of all identities of active processes and messages, respectively, between p_j and m' after m reached p_i; and let $\#'_\pi$ and $\#'_\mu$ denote the number of active processes and messages, respectively, between p_j and m' with identity ξ'_π and ξ'_μ, respectively, after m reached p_i. Clearly $id_i \leq \xi$. We consider five cases.

  If $id'_i > \xi$, then $\xi'_\pi = id'_i = \xi'_\mu$ and $\#'_\pi = 1 = \#'_\mu$.

  If $id'_i = \xi$ and $id_i = \xi$, then $\xi'_\pi = \xi = \xi'_\mu$ and $\#'_\pi = \# = \#'_\mu$.

  If $id'_i = \xi$ and $id_i < \xi$, then $\xi'_\pi = \xi = \xi'_\mu$ and $\#'_\pi = \# + 1 = \#'_\mu$.

If $id'_i < \xi$ and $id_i = \xi$, then $\xi'_\pi = \xi = \xi'_\mu$ and $\#'_\pi = \# - 1 = \#'_\mu$. Namely, since $id_i < id_j$, by Lemma 8.5.2 there must be an active process or message between p_i and m' with identity $\geq id_i$. Since $id_i = \xi$, this identity must be equal to id_i.

If $id'_i < \xi$ and $id_i < \xi$, then $\xi'_\pi = \xi = \xi'_\mu$ and $\#'_\pi = \# = \#'_\mu$.

Case 3: $id_i = id_j$. Then before m reached p_i, p_j was related to m and p_i was related to m'. So by induction, before m reached p_i, these pairs satisfied condition 1. Let ξ_1 and ξ_2 denote the maximum of all identities of active processes (and of messages) between p_j and m and between p_i and m', respectively, before m reached p_i; and let $\#_1$ and $\#_2$ denote the number of active processes (and of messages) between p_j and m and between p_i and m', respectively, before m reached p_i. Moreover, let ξ'_π, ξ'_μ, $\#'_\pi$ and $\#'_\mu$ have the same meaning as in the previous case. We consider seven cases.

If $id'_i > max\{\xi_1, \xi_2\}$, then $\xi'_\pi = id'_i = \xi'_\mu$ and $\#'_\pi = 1 = \#'_\mu$.

If $\xi_1 > max\{id'_i, \xi_2\}$, then $\xi'_\pi = \xi_1 = \xi'_\mu$ and $\#'_\pi = \#_1 = \#'_\mu$.

If $\xi_2 > max\{id'_i, \xi_1\}$, then $\xi'_\pi = \xi_2 = \xi'_\mu$ and $\#'_\pi = \#_2 = \#'_\mu$.

If $id'_i = \xi_1 > \xi_2$, then $\xi'_\pi = id'_i = \xi'_\mu$ and $\#'_\pi = \#_1 + 1 = \#'_\mu$.

If $id'_i = \xi_2 > \xi_1$, then $\xi'_\pi = id'_i = \xi'_\mu$ and $\#'_\pi = \#_2 + 1 = \#'_\mu$.

If $\xi_1 = \xi_2 > id'_i$, then $\xi'_\pi = \xi_1 = \xi'_\mu$ and $\#'_\pi = \#_1 + \#_2 = \#'_\mu$.

If $id'_i = \xi_1 = \xi_2$, then $\xi'_\pi = id'_i = \xi'_\mu$ and $\#'_\pi = \#_1 + \#_2 + 1 = \#'_\mu$.

⊠

We say that an active process or message is *maximal* if its identity is maximal among the active processes or messages in the ring, respectively. In the following proposition we write ξ_π and ξ_μ for the identity of maximal active processes and messages, respectively. The number of active processes and messages with the same identity id is denoted by $\#_\pi^{id}$ and $\#_\mu^{id}$, respectively. We write $\#_\pi$ and $\#_\mu$ for the number of maximal active processes and messages, respectively.

Proposition 8.5.5 Until a leader is elected, there exist active processes and messages in the ring, and $\xi_\pi = \xi_\mu$ and $\#_\pi = \#_\mu$.

Proof. We apply induction on execution sequences.
Basis: Prior to the first arrival of a message, every process is active and has generated a message with its own identity; thus the proposition trivially holds.
Induction step: By induction, $\xi_\pi = \xi_\mu$ and $\#_\pi = \#_\mu$; we write ξ for ξ_π and ξ_μ, and $\#$ for $\#_\pi$ and $\#_\mu$. When a message arrives at a passive process, it is simply forwarded. Assume a message $m = (id, hop)$ arrives at an active process p_i with identity id_i. If $hop = n$, then p_i is elected as the leader. Now suppose that $hop < n$. We consider four cases.

- $id_i > id$. Since $\xi_\pi = \xi_\mu$, m is not a maximal message. It is purged by p_i. The values of ξ_π and ξ_μ remain unchanged.
- $id_i < id$. Since $\xi_\pi = \xi_\mu$, p_i is not a maximal process. It becomes passive. The values of ξ_π and ξ_μ remain unchanged.
- $id_i = id < \xi$. Then p_i selects a new identity id'_i, and sends the message $(id'_i, 1)$. If $id'_i > \xi$, then $\xi'_\pi = id'_i = \xi'_\mu$ and $\#'_\pi = 1 = \#'_\mu$. If $id'_i = \xi$, then $\xi'_\pi = \xi = \xi'_\mu$ and $\#'_\pi = (\# + 1) = \#'_\mu$. If $id'_i < \xi$, then $\xi'_\pi = \xi = \xi'_\mu$ and $\#'_\pi = \# = \#'_\mu$.
- $id_i = id = \xi$. Then p_i selects a new identity id'_i, and sends the message $(id'_i, 1)$. We distinguish two cases.

  Case 1: $\# > 1$. If $id'_i > \xi$, then $\xi'_\pi = id'_i = \xi'_\mu$ and $\#'_\pi = 1 = \#'_\mu$. If $id'_i = \xi$, then $\xi'_\pi = \xi = \xi'_\mu$ and $\#'_\pi = \# = \#'_\mu$. If $id'_i < \xi$, then $\xi'_\pi = \xi = \xi'_\mu$ and $\#'_\pi = (\# - 1) = \#'_\mu$.

  Case 2: $\# = 1$. Then clearly p_i is related to m, and all other active processes and messages are between them. Since $hop < n$, p_i is not the originator of m, so by Lemma 8.5.4.2 there is some active process or message between them. Let $\xi_0 > 0$ be the maximum of all identities of active processes $\neq p_i$ and messages $\neq m$. By Lemma 8.5.4.1, $\#_\pi^{\xi_0} = \#_\mu^{\xi_0}$. If $id'_i > \xi_0$, then $\xi'_\pi = id'_i = \xi'_\mu$ and $\#'_\pi = 1 = \#'_\mu$. If $id'_i = \xi_0$, then $\xi'_\pi = \xi_0 = \xi'_\mu$ and $\#'_\pi = (\#_\pi^{\xi_0} + 1) = \#'_\mu$. If $id'_i < \xi_0$, then $\xi'_\pi = \xi_0 = \xi'_\mu$ and $\#'_\pi = \#_\pi^{\xi_0} = \#'_\mu$.

⊠

Theorem 8.5.6 Let channels be FIFO. Then Algorithm $\mathcal{B}$ terminates with probability one, and upon termination exactly one leader is elected.

Proof. By Proposition 8.5.5, some processes remain active until a leader is elected. A process can be elected as the leader only if it receives a message with a hop counter equal to n, which means the message has passed through all other processes and made them passive. Hence, we have uniqueness of the leader.

It remains to show that the algorithm terminates with probability one. When there are $\ell \geq 2$ active processes in the ring, these processes all remain active if and only if they all the time choose the same identity. Otherwise, at least one active process will become passive. The probability that all active processes select the same identity in one "round" is $(\frac{1}{k})^{\ell-1}$. So the probability for all ℓ active processes to choose the same identity m times in a row is $(\frac{1}{k})^{m(\ell-1)}$. Since $k \geq 2$, the probability that the number of active processes eventually decreases is one.

Clearly, when there is only one active process in the ring, it will be elected as the leader. After the round trip of its final message there are no remaining messages, because channels are FIFO. ⊠

8.6 Performance Analysis

A probabilistic analysis in [95] reveals that if $k = n$, the expected number of rounds required for the Itai-Rodeh algorithm to elect a leader in a ring with size n is bounded by $e \cdot \frac{n}{n-1}$. The expected number of messages for each round is $\mathcal{O}(n \log n)$. Hence, the average message complexity of the Itai-Rodeh algorithm is $\mathcal{O}(n \log n)$. Likewise, Algorithms $\mathcal{A}$ and $\mathcal{B}$ have an average message complexity of $\mathcal{O}(n \log n)$.

The probabilistic temporal logic PCTL [83, 11] can be used to express *soft deadlines*, such as "the probability of electing a leader within t discrete time steps is at most 0.5".[1] A PCTL formula to calculate the probability of electing a leader within t discrete time steps for a ring with two processes is

```
P=? [ true U<=t (s1=4 & s2=4 & leader1+leader2=1)]
```

We used PRISM to calculate the probability that Algorithms $\mathcal{A}$ and $\mathcal{B}$ terminate within a given number of transitions, for rings of size two and three. The experimental results presented in Figure 8.2 and Figure 8.3 indicate that Algorithm $\mathcal{B}$ seems to have a better performance than Algorithm $\mathcal{A}$. Note that when t moves to infinity, both algorithms elect a leader with probability one.

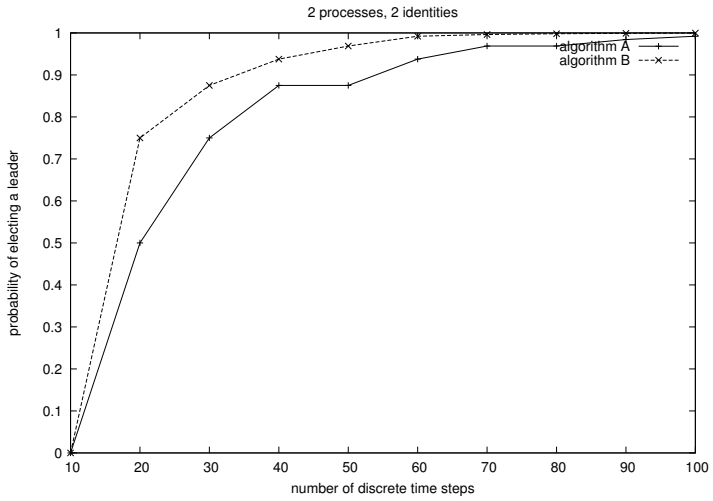


Figure 8.2: The probability of electing a leader with deadlines.

8.7 Leader Election with Two Identities

In this section we show that when $k = 2$, both Algorithm $\mathcal{A}$ and Algorithm $\mathcal{B}$ (with some small adaptations) are correct even if channels are not FIFO. Note

[1] Each discrete time step corresponds to one transition in the algorithm.

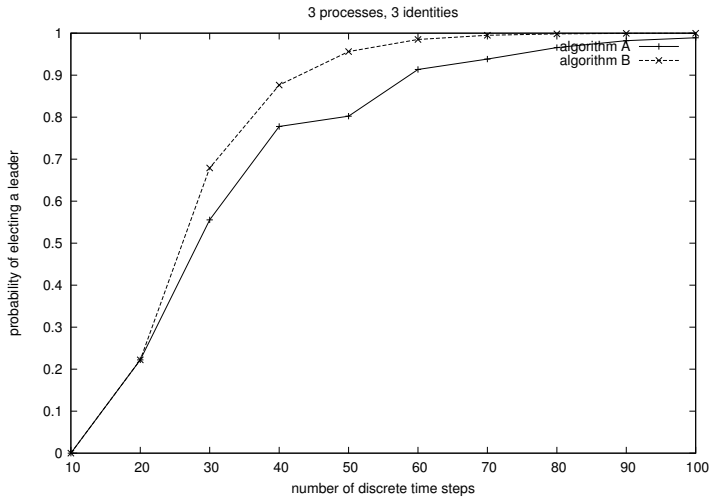


Figure 8.3: The probability of electing a leader with deadlines.

that if $k = 2$, then in Figure 8.1 we cannot find identities u, v, w, x such that $u > v > w, x$.

We first explain the changes that need to be made to Algorithms $\mathcal{A}$ and $\mathcal{B}$. If channels are not FIFO, then when a leader is elected, there may still be messages in the ring. So to guarantee that the algorithms terminate with no message in the ring, the leader must be able to purge incoming messages.

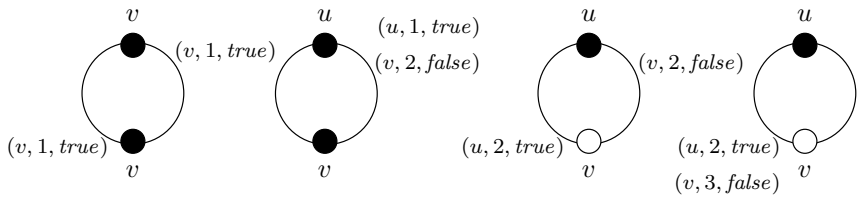


Figure 8.4: Algorithm $\mathcal{A}$: if channels are not FIFO, hop counters can be greater than n.

We need to make one more minor adaptation to the PRISM model of Algorithm $\mathcal{A}$. Namely, the domain of hop counters has to be enlarged from $[0..N]$ to $[0..2N-1]$. Figure 8.4 presents a scenario to show that a message can continue after completing a round trip. It depicts a ring of size two; black processes are active and white processes are passive. Initially, both processes are active, select the smaller of the two identities v, and send a message with their identity. (See the left side of Figure 8.4.) The message from the top node arrives back at its originator, which selects as new identity $u > v$ and sends a message with its

	Processes	Channel size	FIFO	States	Transitions
Ex.1	2	2	no	533	898

Table 8.3: Model checking result for Algorithm $\mathcal{A}$ with $k = 2$

	Processes	Channel size	FIFO	States	Transitions
Ex.1	2	2	no	391	666
Ex.2	3	3	no	63,433	147,660

Table 8.4: Model checking result for Algorithm $\mathcal{B}$ with $k = 2$

identity. (See the second part of Figure 8.4.) Since channels are not FIFO, the message with identity v can be overtaken by the message with identity u, and the latter message makes the bottom node passive. (See the third part of Figure 8.4.) Finally, the message $(v, 2, \mathit{false})$ is passed on by its passive originator to become $(v, 3, \mathit{false})$. (See the right side of Figure 8.4.)

We verified Algorithms $\mathcal{A}$ and $\mathcal{B}$ (with the aforementioned adaptations) using PRISM in the setting that $k = 2$ and channels are not FIFO. Here, we omit the PRISM specification, and only present the verification results in Table 8.3 and Table 8.4. We successfully analyzed Algorithm $\mathcal{A}$ for a ring of size two, and Algorithm $\mathcal{B}$ for rings up to size three. For any larger ring size, PRISM fails to build a model.

Theorem 8.7.1 Let $k = 2$. Algorithm $\mathcal{A}$ terminates with probability one, and upon termination exactly one leader has been elected.

Proof. Since $k = 2$, the identity set contains only two elements. Let u denote the largest element. First, we present a proposition.

Proposition 8.7.2 Until a leader is elected, there exist active processes and messages in the ring.

We apply induction on execution sequences.
Basis: Prior to the first arrival of a message, every process is active and has generated a message with its own identity; thus the proposition trivially holds.
Induction step: When a message arrives at a passive process, it is simply forwarded. Assume that message $m = (id, hop, bit)$ arrives at active process p_i with identity id_i. We distinguish two cases.

- $id_i = id$.

  If $hop = n$ and $bit = \mathit{true}$, then p_i is elected as the leader.

  If $hop = n$ and $bit = \mathit{false}$, then p_i remains active, selects a new identity id'_i and sends the message $(id'_i, 1, \mathit{true})$.

If $hop < n$, then p_i remains active and sends the message $(id, hop + 1, false)$.

- $id_i \neq id$.

  If $id_i = u$, then p_i is the originator of a message with identity u. This message will complete the round trip, since no process has an identity larger than u; so this message is still in the ring. p_i remains active and purges m.

  If $id = u$, then m originates from a process p_j with identity u. p_j remains active until m has completed the round trip, since no message can have an identity larger than u. p_i becomes passive and sends the message $(id, hop + 1, bit)$.

It follows from Proposition 8.7.2 that some processes remain active until a leader is elected. An active process can be elected as the leader only if it receives a message with hop counter n and bit *true*, which means the message has passed through all other processes and made them passive. Hence, we have uniqueness of the leader.

The proof that the algorithm terminates with probability one is similar to the probability analysis in the proof of Theorem 8.4.2. When a leader is elected, it purges the remaining messages in the ring. ⊠

Theorem 8.7.3 Let $k = 2$. Algorithm $\mathcal{B}$ terminates with probability one, and upon termination exactly one leader has been elected.

Proof. Since $k = 2$, the identity set contains only two elements. Let u denote the larger element. First, we present a proposition. We write $\#_\pi$ and $\#_\mu$ for the number of active processes and messages with identity u, respectively.

Proposition 8.7.4 Until a leader is elected, there exist active processes and messages in the ring, and $\#_\pi = \#_\mu$.

We apply induction on execution sequences.
Basis: Prior to the first arrival of a message, every process is active and has generated a message with its own identity; thus the proposition trivially holds.
Induction step: By induction, $\#_\pi = \#_\mu$; we write $\#$ for $\#_\pi$ and $\#_\mu$. When a message arrives at a passive process, it is simply forwarded. Assume that message $m = (id, hop)$ arrives at active process p_i with identity id_i. If $hop = n$, then p_i is elected as the leader. Let $hop < n$. We distinguish two cases.

- $id_i = id$.

  Then p_i remains active, selects a new identity id'_i, and sends the message $(id'_i, 1)$. If $id_i = id'_i$, then $\#'_\pi = \# = \#'_\mu$. If $id_i = u$ and $id'_i \neq u$, then $\#'_\pi = \# - 1 = \#'_\mu$. If $id_i \neq u$ and $id'_i = u$, then $\#'_\pi = \# + 1 = \#'_\mu$.

- $id_i \neq id$.

  Then clearly $\# > 0$.

  If $id = u$, then p_i becomes passive and sends the message $(id, hop + 1)$. $\#'_\pi = \# = \#'_\mu$.

  If $id_i = u$, then p_i remains active and purges m. $\#'_\pi = \# = \#'_\mu$.

By Proposition 8.7.4, some processes remain active until a leader is elected. An active process can be elected as the leader only if it receives a message with a hop counter equal to n, which means the message has passed through all other processes and made them passive. Hence, we have uniqueness of the leader.

The proof that the algorithm terminates with probability one is similar to the probability analysis in the proof of Theorem 8.5.6. When a leader is elected, it purges the remaining messages in the ring. ⊠

8.8 Conclusions

In this chapter, we presented two probabilistic leader election algorithms for anonymous unidirectional rings with FIFO channels. Both algorithms were specified and successfully model checked with PRISM. They satisfy the property "with probability 1, eventually exactly one leader is elected". The complete specifications in PRISM can be found at `http://www.cwi.nl/~pangjun/leader/`. The generation of state spaces and the verifications were performed on a 1.4 GHz AMD Athlon^{TM} Processor with 512 Mb memory. We also gave a manual correctness proof for each algorithm. Future work is to formalize and check these proofs by means of a theorem prover such as PVS.

Itai and Rodeh [95] stated:

> "We could have used any of the improved algorithms [27], [44], [88], [138]."

Following this direction, we developed two more probabilistic leader election algorithms, based on the Dolev-Klawe-Rodeh algorithm [44, 58]. Both of them are finite-state, and we model checked them successfully in μCRL [21] up to ring size six. The adaptations of the Dolev-Klawe-Rodeh algorithm are very similar to our adaptations (Algorithms $\mathcal{A}$ and $\mathcal{B}$) of the Chang-Roberts algorithm; i.e., processes again select random identities, and name clashes are resolved in exactly the same way. Therefore our adaptations of the Dolev-Klawe-Rodeh algorithm are not presented here. The interested reader can find the specifications of all our algorithms at `http://www.cwi.nl/~pangjun/leader/`. These specifications are in the language μCRL, which was used for an initial non-probabilistic model checking exercise.

Chapter 9

Conclusions

Conclusions have been drawn for Chapters 3 to 8 separately. In this chapter, I will give some concluding remarks, from the perspective of the entire project.

Recall that the general goal of the project is:

> "to establish whether it is possible to achieve reliable quality of software for medium size embedded systems, and to better utilize the formal methods in industry."

and that the major question to be answered is:

> "whether the current technology developed in the past by the formal methods research community can indeed become an effective practical tool within a development environment."

The research proposal argued that most of the published case studies of formal verification in the literature were quite remote from the actual product design process and generally only dealt with fractions of a system, as the total system tends to be too complex. The situation at Weidmüller/Add-Controls is quite different. The products they design, embedded controllers, are relatively not very complex. Moreover, direct communication with the development department is possible, which provides an ideal platform for experiments on the trajectory from formal design towards real products.

However, the project progressed in an unexpected way. This project was initially proposed by Jan Friso Groote and Jos van Wamel at CWI. Not long after the project started in August of 2000, both Jan Friso and Jos left CWI, and Wan Fokkink succeeded as the project leader at CWI. In 2001, the division of Weidmüller supporting this project decided to set up a new company – Add-Controls. During the initial phase of Add-Controls, there was no new development of embedded systems. The distributed lift system (see Chapter 6) became its main commercial product.

Nevertheless, we have tried to stick to the spirit of the project. The distributed lift system was first analyzed in 2000 and 2001, and then was redesigned at Add-Controls. The analysis of the redesign took place in 2002 and 2003. In

order to perform another real-life case study during the design phase, in 2001 and 2002, we analyzed the cache coherence protocol for Jackal system, which is a distributed shared memory implementation of the Java language. In 2003 and the beginning of 2004, we used formal verification techniques to design new distributed algorithms and show their correctness. In the meantime, some theoretical research has been carried out for the project. A protocol verification method was developed and supplied with mechanical support (see Chapter 3). The usefulness of this method was illustrated by a challenging case study (see Chapter 4).

To summarize, in this project:

- Different formal verification techniques such as manual proof, model checking and theorem proving have been applied for the analysis of distributed system. Theorem proving was applied in Chapter 5. The combination of manual proof and theorem proving was applied in Chapters 3 and 4. Model checking was applied in Chapters 6 and 7. The combination of manual proof and model checking was applied in Chapter 8.

- We have tried different tools for the verification of different aspects of distributed systems. The theorem prover PVS [131] was used in Chapters 3, 4 and 5. The μCRL tool set [21] and the model checker CADP [49, 63] were used in Chapters 6 and 7. The real-time model checker UPPAAL [111] was used in Chapter 6. The probabilistic model checker PRISM [107] was used in Chapter 8. The tool for conformance testing TorX [14] and the model checker for hybrid systems HyTech [86] were used in two abandoned case studies.

- Formal verification has been applied in different phases of system development. The implementation of original design of the distributed lift system was analyzed in Chapters 6, while the redesign of the system was analyzed before implementing. During its formal verification, the cache coherence protocol in Chapters 7 was still under implementation, and some evolution of its design took place. In Chapters 8, formal verification was used to develop new distributed algorithms for leader election.

- The case studies cover a wide range of distributed systems; namely an embedded controller (Chapters 6), a communication protocol (Chapter 4), a cache coherence protocol (Chapter 7), and distributed algorithms (Chapters 5 and 8).

Within this project, we have achieved certain positive results. Formal verification can find problems in real-life distributed systems, and suggest possible solutions. Formal verification can also be used to prove protocols and algorithms correct. Therefore, the proper use of formal methods does lead to more reliable, dependable systems. Using formal methods in the industrial system development can be effective, at least for embedded controllers.

On the other hand, the situation of using formal methods in the industrial system development described in the thesis of Judi Romijn [149, Chapter 8] has

not improved dramatically in the last five years. Thus, to make formal methods an effective practical tool within an industrial development environment, significant developments in formal methods still have to be made. For example, the cones and foci method developed in this thesis is still far from a practical tool, which can be used directly in industry. How to integrate formal methods into the whole development process of industrial systems partly remains an open question.

I draw some conclusions on what I have learned from the project:

- Model checking is useful for detecting errors in real-life systems and for gaining more confidence about the design of a system. Theorem proving is useful for giving correctness proofs.

- Both researchers of formal methods and their industrial partners need to speak each other's language. Researchers need to understand the system designed and implemented by the industry in order to perform better formal analysis. On the other hand, developers from industry need some knowledge of formal specification languages and verification methods in order to give feedback and appreciate the result of the formal analysis.

- Researchers must take the input from developers seriously. Analyzing a formal model that deviates too much from the actual system or has a very high level of abstraction is not useful in practice (see e.g., Chapter 6). Developers of industrial systems must take the input from the formal analysis of researchers seriously. As shown in Chapter 6, the formal analysis of the original design of the lift system in μCRL would have saved the developers considerable effort in the redesign.

- The developers of the lift system stress that formal methods should be applied in the early design phases to save testing effort and cost.

- It is important that experiments within the formal analysis process can be reproduced easily. When a system is under formal analysis, its design and implementation can still be modified by the developers (see e.g., Chapter 7). After some changes took place, the experiments that had been done before needed to be repeated in order to check whether the changes have effect on the correctness of the system.

- It is necessary for researchers to have the ability of using different formalisms and tools in order to verify different aspects of systems. In my experience, the translation of a formal model of a system into another formalism is in general not very difficult (see e.g., Chapter 6 and Chapter 8).

- Not all system errors can be detected with formal methods, which is a lesson I learned from an abandoned case study.

From my personal viewpoint, I give some remarks on improving the effectiveness of using formal methods in industry.

First, for researchers to improve formal methods, we must: 1) reduce the learning curve of formal methods such that they are easy to learn, and quick to use; 2) increase the expressiveness of formal methods such that they can be used to specify and verify more systems; 3) develop new efficient and effective verification techniques such that they can deal with large and complex systems; 4) integrate different formal verification techniques in a uniform framework such that within a verification task we can benefit from different techniques and switch among different methods smoothly; 5) transfer formal methods to potential users by educating under-graduate students in formal methods and performing more case studies for industry; 6) invest more time and manpower in project, like the one in this thesis.

Second, to apply formal methods in a industrial system development, it is important for industry: 1) to know in which projects using formal methods can be beneficial; 2) to recognize when and where to apply formal methods in such projects; 3) to educate their designers in formal methods; 4) to support more research project, like the one in this thesis.

Appendix A

μCRL Code of the Cache Coherence Protocol

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For data types, equality function defintions are all omitted.
% Sort: Bool
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Bool
func T,F:->Bool
map  if:Bool#Bool#Bool->Bool
     not:Bool->Bool
     and,or,eq:Bool#Bool->Bool
var  b,b':Bool
rew  if(T,b,b')=b if(F,b,b')=b'
     not(T)=F not(F)=T not(not(b))=b
     and(T,b)=b and(F,b)=F and(b,T)=b and(b,F)=F
     or(T,b)=T or(F,b)=b or(b,T)=T or(b,F)=b
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sort: Natural.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Natural
func 0:->Natural
     S:Natural->Natural
map  sub1: Natural->Natural
     eq,gt: Natural#Natural->Bool
var  n,m:Natural
rew  sub1(0)=0 sub1(S(n))=n
     gt(0, n)=F gt(S(n),0)=T gt(S(n),S(m))=gt(n,m)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sort: ThreadId
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort ThreadId
func tid1,tid2,tid3:->ThreadId
map  eq,le:ThreadId#ThreadId->Bool
var  t:ThreadId
```

```
rew  le(t,t)=T le(tid1,t)=T le(tid2,tid1)=F le(tid2,tid3)=T
     le(tid3,tid1)=F le(tid3,tid2)=F
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sort: ProcessorId
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort ProcessorId
func pid1,pid2 :->ProcessorId
map  eq,le:ProcessorId#ProcessorId->Bool
var  p:ProcessorId
rew  le(pid1,p)=T le(pid2,pid1)=F  le(pid2,pid2)=T
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sort: RegionId, only one region with identity rid1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort RegionId
func rid1 :->RegionId
map  eq:RegionId#RegionId->Bool
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This sort is used for a region, which maintains a list of processors
% which have written to the region recently.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort ProcessorIdSet
func ema:->ProcessorIdSet
     in:ProcessorId#ProcessorIdSet->ProcessorIdSet
map  remove:ProcessorId#ProcessorIdSet->ProcessorIdSet
     test:ProcessorId#ProcessorIdSet->Bool
     empty:ProcessorIdSet->Bool
     if:Bool#ProcessorIdSet#ProcessorIdSet->ProcessorIdSet
     eq:ProcessorIdSet#ProcessorIdSet->Bool
     count:ProcessorIdSet->Natural
     % Get the identity when there is only one processor.
     getIden:ProcessorIdSet->ProcessorId
     insert:ProcessorId#ProcessorIdSet->ProcessorIdSet
var  a,a':ProcessorId A,A':ProcessorIdSet
rew  remove(a,ema)=ema
     remove(a,in(a',A))=if(eq(a,a'),remove(a,A),in(a',remove(a,A)))
     test(a,ema)=F test(a,in(a',A))=if(eq(a,a'),T,test(a,A))
     empty(ema)=T empty(in(a,A))=F
     if(T,A,A')=A if(F,A,A')=A'
     count(ema)=0 count(in(a,A))=S(count(remove(a,in(a,A))))
     getIden(in(a,A))=a
     insert(a,ema)=in(a,ema)
     insert(a,in(a',A'))=if(eq(a,a'),in(a',A'),
       if(le(a,a'),in(a,in(a',A')),in(a',insert(a,A'))))
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%This sort is used for a thread, which maintians a list of regions
% where the thread has written recently.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort RegionIdSet
func ridema:->RegionIdSet
     in:RegionId#RegionIdSet->RegionIdSet
```

```
% These functions are defined similarly as in ProcessorIdSet. Omitted.
map  remove:RegionId#RegionIdSet->RegionIdSet
     test:RegionId#RegionIdSet->Bool
     empty:RegionIdSet->Bool
     if:Bool#RegionIdSet#RegionIdSet->RegionIdSet
     eq:RegionIdSet#RegionIdSet->Bool
     count:RegionIdSet->Natural
     getIden:RegionIdSet->RegionId
     insert:RegionId#RegionIdSet->RegionIdSet
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% State of regions, initially, the region is UNUSED.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort State
func UNUSED,USED:->State
map  eq: State#State->Bool
     if:Bool#State#State->State
var  s1,s2:State
rew  if(T,s1,s2)=s1 if(F,s1,s2)=s2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sort: Region
% Id, Home, State, accessorlist, Data, Twin, the number of local threads
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Region
func reg:RegionId#ProcessorId#State#ProcessorIdSet#Natural->Region
map  getid:Region->RegionId
     gethome:Region->ProcessorId
     getstate:Region->State
     getaccessorlist:Region->ProcessorIdSet
     getlocalt:Region->Natural
     sethome:Region#ProcessorId->Region
     setstate:Region#State->Region
     setaccessorlist:Region#ProcessorIdSet->Region
     setlocalt:Region#Natural->Region
     eq:Region#Region->Bool
var  id,id': RegionId h,h':ProcessorId w,w':ProcessorIdSet
     s,s':State lt,lt':Natural region:Region
rew  getid(reg(id,h,s,w,lt))=id
     gethome(reg(id,h,s,w,lt))=h
     getstate(reg(id,h,s,w,lt))=s
     getaccessorlist(reg(id,h,s,w,lt))=w
     getlocalt(reg(id,h,s,w,lt))=lt
     sethome(reg(id,h,s,w,lt),h')=reg(id,h',s,w,lt)
     setstate(reg(id,h,s,w,lt),s')=reg(id,h,s',w,lt)
     setaccessorlist(reg(id,h,s,w,lt),w')=reg(id,h,s,w',lt)
     setlocalt(reg(id,h,s,w,lt),lt')=reg(id,h,s,w,lt')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Actions: We synchronize s_* and r_* into an action c_*.
% The communication functions will be omitted.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act
```

```
s_require_faultlock,r_require_faultlock,c_require_faultlock: ProcessorId
s_require_flushlock,r_require_flushlock,c_require_flushlock: ProcessorId
s_require_serverlock,r_require_serverlock,
     c_require_serverlock: ProcessorId
s_require_homequeuelock,r_require_homequeuelock,
     c_require_homequeuelock: ProcessorId
s_require_remotequeuelock,r_require_remotequeuelock,
     c_require_remotequeuelock: ProcessorId
s_free_faultlock,r_free_faultlock,c_free_faultlock: ProcessorId
s_free_flushlock,r_free_flushlock,c_free_flushlock: ProcessorId
s_free_serverlock,r_free_serverlock,c_free_serverlock: ProcessorId
s_free_homequeuelock,r_free_homequeuelock,
     c_free_homequeuelock: ProcessorId
s_free_remotequeuelock,r_free_remotequeuelock,
     c_free_remotequeuelock: ProcessorId
s_no_faultwait,r_no_faultwait,c_no_faultwait: ProcessorId
s_no_flushwait,r_no_flushwait,c_no_flushwait: ProcessorId
s_no_serverwait,r_no_serverwait,c_no_serverwait: ProcessorId
s_no_homequeuewait,r_no_homequeuewait,
     c_no_homequeuewait: ProcessorId
s_no_remotequeuewait,r_no_remotequeuewait,
     c_no_remotequeuewait: ProcessorId
s_signal_faultwait,r_signal_faultwait,c_signal_faultwait: ProcessorId
s_signal_flushwait,r_signal_flushwait,c_signal_flushwait: ProcessorId
s_signal_serverwait,r_signal_serverwait,c_signal_serverwait: ProcessorId
s_signal_homequeuewait,r_signal_homequeuewait,
     c_signal_homequeuewait: ProcessorId
s_signal_remotequeuewait,r_signal_remotequeuewait,
     c_signal_remotequeuewait: ProcessorId
s_data_require,r_i_data_require,c_i_data_require,
s_i_data_require,r_data_require,c_o_data_require:
     ThreadId#ProcessorId#ProcessorId
s_data_return,r_o_data_return,c_i_data_return,
s_o_data_return,r_data_return,c_o_data_return:
     ThreadId#ProcessorId#ProcessorId#Region#Bool
s_flush,r_i_flush,c_i_flush, s_i_flush,r_flush,c_o_flush:
     ThreadId#ProcessorId#ProcessorId#Region#Bool
s_region_sponmigrate,r_i_region_sponmigrate,c_i_region_sponmigrate,
s_i_region_sponmigrate,r_region_sponmigrate,c_o_region_sponmigrate:
     ThreadId#ProcessorId#ProcessorId#Region
s_sendback,r_sendback,c_sendback:ThreadId#ProcessorId#Region
s_refresh,r_refresh,c_refresh:ThreadId#ProcessorId#Region
s_norefresh,r_norefresh,c_norefresh:ThreadId#ProcessorId
s_sendback,r_sendback,c_sendback: ProcessorId#Region
s_refresh,r_refresh,c_refresh: ProcessorId#Region
s_norefresh,r_norefresh,c_norefresh: ProcessorId
s_signal,r_signal,c_signal: ThreadId#ProcessorId
write,writeover,flush,flushover:ThreadId
r_home s_home c_home r_copy s_copy c_copy
lockempty,homequeueempty,remotequeueempty:ProcessorId
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: Thread
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Thread(tid:ThreadId,pid:ProcessorId,FlushList:RegionIdSet)=
     write(tid).ThreadWrite(tid,pid,FlushList) +
     flush(tid).ThreadInvalidate(tid,pid,FlushList)
     <| not(eq(FlushList, ridema)) |>delta
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: ThreadWrite
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc ThreadWrite(tid:ThreadId,pid:ProcessorId,FlushList:RegionIdSet)=
     Thread(tid,pid,FlushList)
     <| test(rid1, FlushList) |>
     sum(r:Region,r_sendback(tid,pid,r).
        (s_norefresh(tid,pid).
         WriteHome(tid,pid,insert(rid1,FlushList))
         <| eq(gethome(r), pid) |>
         s_norefresh(tid,pid).
         WriteRemote(tid,pid,insert(rid1,FlushList))
     )  )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: WriteHome, thread writes at home.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc WriteHome(tid:ThreadId,pid:ProcessorId,FlushList:RegionIdSet)=
     s_require_serverlock(pid).
     (r_no_serverwait(pid)+r_signal_serverwait(pid)).
     sum(r:Region,r_sendback(tid,pid,r).
        (  (s_refresh(tid,pid,setlocalt(setstate(setaccessorlist(
              r,insert(pid,getaccessorlist(r))),USED),S(getlocalt(r)))).
            s_free_serverlock(pid).
            writeover(tid).Thread(tid,pid,FlushList)
            <| eq(getstate(r), UNUSED) |>
            s_refresh(tid,pid,setlocalt(setaccessorlist(
               r,insert(pid,getaccessorlist(r))),S(getlocalt(r)))).
            s_free_serverlock(pid).
            writeover(tid).Thread(tid,pid,FlushList)
            )
         <| eq(gethome(r), pid) |>
         s_norefresh(tid,pid).
         s_free_serverlock(pid).
         WriteRemote(tid,pid,FlushList)
     )  )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: WriteRemote, thread writes from remote.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc WriteRemote(tid:ThreadId,pid:ProcessorId,FlushList:RegionIdSet)=
     s_require_faultlock(pid).
     (r_no_faultwait(pid)+r_signal_faultwait(pid)).
     sum(r:Region,r_sendback(tid,pid,r).
        (s_data_require(tid,pid,gethome(r)).
```

```
         s_norefresh(tid,pid).
         sum(pid':ProcessorId,r_signal(tid,pid').
            sum(newr:Region,r_sendback(tid,pid,newr).
                s_refresh(tid,pid,setlocalt(newr,S(getlocalt(newr)))).
                s_free_faultlock(pid).
                writeover(tid).Thread(tid,pid,FlushList)
         )  )
         <| not(eq(gethome(r),pid)) |>
         s_norefresh(tid,pid).
         s_free_faultlock(pid).
         WriteHome(tid,pid,FlushList)
     )  )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: ThreadInvalidate
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc ThreadInvalidate(tid:ThreadId,pid:ProcessorId,
                      FlushList:RegionIdSet)=
     Thread(tid,pid,FlushList)
     <| eq(FlushList, ridema) |>
     s_require_flushlock(pid).
     (r_no_flushwait(pid)+r_signal_flushwait(pid)).
     sum(r:Region,r_sendback(tid,pid,r).
        (FlushHome(tid,pid,remove(rid1,FlushList),r)
         <| eq(gethome(r),pid) |>
         FlushRemote(tid,pid,remove(rid1,FlushList),r)
     )  )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: FlushHome, thread invalidates at home.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc FlushHome(tid:ThreadId,pid:ProcessorId,FlushList:RegionIdSet,
               r:Region)=
     (  s_refresh(tid,pid,setlocalt(setstate(setaccessorlist(
          r,remove(pid,getaccessorlist(r))),UNUSED),sub1(getlocalt(r)))).
        s_free_flushlock(pid).
        flushover(tid).Thread(tid,pid,FlushList )
        <| empty(remove(pid,getaccessorlist(r))) |>
        (  (s_region_sponmigrate(tid,pid,
              getIden(remove(pid,getaccessorlist(r))),
              setaccessorlist(r,remove(pid,getaccessorlist(r)))).
            s_refresh(tid,pid,sethome(setlocalt(setstate(
              setaccessorlist(r,ema),UNUSED),sub1(getlocalt(r))),
              getIden(remove(pid,getaccessorlist(r))))).
            s_free_flushlock(pid).
            flushover(tid).Thread(tid,pid,FlushList)
            <| not(eq(getIden(remove(pid,getaccessorlist(r))), pid))|>
            s_refresh(tid,pid,setlocalt(setaccessorlist(
              r,remove(pid,getaccessorlist(r))),sub1(getlocalt(r)))).
            s_free_flushlock(pid).
            flushover(tid).Thread(tid,pid,FlushList)
          )
```

```
          <| eq(count(remove(pid,getaccessorlist(r))),S(0)) |>
             s_refresh(tid,pid,setlocalt(setaccessorlist(
               r,remove(pid,getaccessorlist(r))),sub1(getlocalt(r)))).
             s_free_flushlock(pid).
             flushover(tid).Thread(tid,pid,FlushList)
     )  )
     <| eq(sub1(getlocalt(r)),0) |>
     s_refresh(tid,pid,setlocalt(r,sub1(getlocalt(r)))).
     s_free_flushlock(pid).
     flushover(tid).Thread(tid,pid,FlushList)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: FlushRemote, threads invalidates from remote.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc FlushRemote(tid:ThreadId,pid:ProcessorId,FlushList:RegionIdSet,
                 r:Region)=
     s_flush(tid,pid,gethome(r),r,T).
     s_refresh(tid,pid,setlocalt(setaccessorlist(setstate(
       r,UNUSED),ema),sub1(getlocalt(r)))).
     s_free_flushlock(pid).
     sum(pid':ProcessorId,r_signal(tid,pid').
        flushover(tid).Thread(tid,pid,FlushList)
     )
     <| eq(sub1(getlocalt(r)),0) |>
     s_flush(tid,pid,gethome(r),r,F).
     s_refresh(tid,pid,setlocalt(setaccessorlist(
       r,ema),sub1(getlocalt(r)))).s_free_flushlock(pid).
     sum(pid':ProcessorId,r_signal(tid,pid').
        flushover(tid).Thread(tid,pid,FlushList)
     )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: Region, both thread and processor can access the information.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Region(pid:ProcessorId,r:Region)=
     sum(tid:ThreadId, s_sendback(tid,pid,r).
        (  r_norefresh(tid,pid).Region(pid,r)+
           sum(r':Region, r_refresh(tid,pid,r').Region(pid,r'))
     )  )
     + s_sendback(pid,r).
     (  r_norefresh(pid).Region(pid,r)+
        sum(r':Region,r_refresh(pid,r').Region(pid,r'))
     )
     + r_home.Region(pid,r)<| eq(pid,gethome(r)) |>delta
     + s_home.Region(pid,r)<| eq(pid,gethome(r)) |>delta
     + r_copy.Region(pid,r)<| not(eq(pid,gethome(r))) |>delta
     + s_copy.Region(pid,r)<| not(eq(pid,gethome(r))) |>delta
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: Processor, dealing with four messages.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Processor(pid:ProcessorId)=
     sum(tid:ThreadId,sum(pid':ProcessorId,sum(r':Region,sum(b:Bool,
```

```
    r_data_return(tid,pid,pid',r',b).
    (  sum(r:Region,r_sendback(pid,r).
        (s_signal(tid,pid).
         s_refresh(pid,sethome(setstate(
           r,getstate(r')),gethome(r'))).
         s_free_remotequeuelock(pid).
         Processor(pid)
         <| not(eq(gethome(r),pid)) |>
         s_signal(tid,pid).
         s_refresh(pid,sethome(setstate(
           r,USED),pid)).s_free_remotequeuelock(pid).
         Processor(pid)
       ))
    <| not(b) |>
        sum(r:Region,r_sendback(pid,r).
            s_signal(tid,pid).
            s_refresh(pid,sethome(setstate(setaccessorlist(
              r,getaccessorlist(r')),USED),pid)).
            s_free_remotequeuelock(pid).
            Processor(pid)
    )   )
))))
+ sum(tid:ThreadId,sum(pid':ProcessorId,
    r_data_require(tid,pid',pid).
    sum(r:Region,
        r_sendback(pid,r).
        (  s_data_require(tid,pid',gethome(r)).
           s_norefresh(pid).
           s_free_homequeuelock(pid).
           Processor(pid)
           <| not(eq(gethome(r),pid)) |>
           (  (  s_data_return(tid,pid',pid,sethome(setstate(
                   setaccessorlist(r,insert(pid',
                   getaccessorlist(r))),UNUSED),pid'),T).
                 s_refresh(pid,sethome(setstate(setaccessorlist(
                   r,ema),UNUSED),pid')).
                 s_free_homequeuelock(pid).
                 Processor(pid)
                 <| eq(getstate(r),UNUSED) |>
                 s_data_return(tid,pid',pid,
                   setstate(setaccessorlist(r,
                   insert(pid',getaccessorlist(r))),USED),F).
                 s_refresh(pid,setstate(setaccessorlist(
                   r,insert(pid',getaccessorlist(r))),USED)).
                   s_free_homequeuelock(pid).
                   Processor(pid)
              )
              <| not(eq(pid,pid')) |>
              s_signal(tid,pid).
              s_refresh(pid,setstate(setaccessorlist(
```

```
              r, insert(pid',getaccessorlist(r))),USED)).
            s_free_homequeuelock(pid).
            Processor(pid)
    )   )  )
))
+ sum(tid:ThreadId,sum(pid':ProcessorId,sum(r':Region,sum(b:Bool,
    r_flush(tid,pid',pid,r',b).
    sum(r:Region,
        r_sendback(pid,r).
        (  s_flush(tid,pid',gethome(r),r',b).
           s_norefresh(pid).
           s_free_homequeuelock(pid).
           Processor(pid)
           <| not(eq(gethome(r), pid)) |>
           (  s_signal(tid,pid).
              s_refresh(pid,r).
              s_free_homequeuelock(pid).
              Processor(pid)
              <| not(b) |>
              (s_signal(tid,pid).
               s_refresh(pid,setstate(setaccessorlist(
                 r,remove(pid',getaccessorlist(r))),UNUSED)).
               s_free_homequeuelock(pid).
               Processor(pid)
               <| empty(remove(pid',getaccessorlist(r))) |>
               ((s_region_sponmigrate(tid,pid,
                    getIden(remove(pid',getaccessorlist(r))),
                    setaccessorlist(r,
                    remove(pid',getaccessorlist(r)))).
                 s_signal(tid,pid).
                 s_refresh(pid,sethome(setstate(
                    setaccessorlist(r,ema),UNUSED),
                    getIden(remove(pid',getaccessorlist(r))))).
                 s_free_homequeuelock(pid).
                 Processor(pid)
                 <| not(eq(getIden(remove(pid',
                     getaccessorlist(r))),gethome(r))) |>
                 s_signal(tid,pid).
                 s_refresh(pid,setstate(setaccessorlist(
                    r,remove(pid',getaccessorlist(r))),USED)).
                 s_free_homequeuelock(pid).
                 Processor(pid)
                 )
                 <|eq(count(remove(pid',getaccessorlist(r))),S(0))|>
                 s_signal(tid,pid).
                 s_refresh(pid,setaccessorlist(
                    r,remove(pid',getaccessorlist(r)))).
                 s_free_homequeuelock(pid).
                 Processor(pid)
    )   )  )   ))
```

```
     )))))
     + sum(tid:ThreadId,sum(pid':ProcessorId,sum(r':Region,
         r_region_sponmigrate(tid,pid',pid,r').
         sum(r:Region,
             r_sendback(pid,r).
             s_refresh(pid,sethome(setstate(setaccessorlist(
               r,getaccessorlist(r')),USED),pid)).
             s_free_homequeuelock(pid).
             Processor(pid)
     ))))
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: HomeQueue, size one.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc HomeQueue(pid: ProcessorId)=
     sum(tid:ThreadId,sum(pid':ProcessorId,
         r_i_data_require(tid,pid',pid).
         s_require_homequeuelock(pid).
         (r_no_homequeuewait(pid)+r_signal_homequeuewait(pid)).
          s_i_data_require(tid,pid',pid).HomeQueue(pid)
     ))
     + sum(tid:ThreadId,sum(pid':ProcessorId,sum(r:Region,sum(b:Bool,
         r_i_flush(tid,pid',pid,r,b).
         s_require_homequeuelock(pid).
         (r_no_homequeuewait(pid)+r_signal_homequeuewait(pid)).
          s_i_flush(tid,pid',pid,r,b).HomeQueue(pid)
     ))))
     + sum(tid:ThreadId,sum(pid':ProcessorId,sum(r:Region,
         r_i_region_sponmigrate(tid,pid',pid,r).
         s_require_homequeuelock(pid).
         (r_no_homequeuewait(pid)+r_signal_homequeuewait(pid)).
          s_i_region_sponmigrate(tid,pid',pid,r).HomeQueue(pid)
     )))
     + homequeueempty(pid).HomeQueue(pid)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: RemoteQueue, size one.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc RemoteQueue(pid: ProcessorId)=
     sum(tid:ThreadId,sum(pid':ProcessorId,sum(r:Region,sum(b:Bool,
         r_o_data_return(tid,pid,pid',r,b).
         s_require_remotequeuelock(pid).
         (r_no_remotequeuewait(pid)+r_signal_remotequeuewait(pid)).
         s_o_data_return(tid,pid,pid',r,b).RemoteQueue(pid)
     ))))
     + remotequeueempty(pid).RemoteQueue(pid)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: Locker
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Locker(pid:ProcessorId,faulters:Natural,flushers:Natural,
            homequeue:Natural,remotequeue:Natural,
            wait_faulters:Natural,wait_flushers:Natural,
```

```
       wait_homequeue:Natural,wait_remotequeue:Natural)=
lockempty(pid).
Locker(pid,faulters,flushers,homequeue,remotequeue,
     wait_faulters,wait_flushers,wait_homequeue,wait_remotequeue)
<| and(and(and(and(and(and(and(
    eq(faulters,0),eq(flushers,0)),eq(homequeue,0)),
    eq(remotequeue,0)),eq(wait_faulters,0)),eq(wait_flushers,0)),
    eq(wait_homequeue,0)),eq(wait_remotequeue,0)) |>delta
+
r_require_faultlock(pid).s_no_faultwait(pid).
Locker(pid,S(faulters),flushers,homequeue,remotequeue,
     wait_faulters,wait_flushers,wait_homequeue,wait_remotequeue)
<| and(eq(faulters,0), eq(flushers,0)) |>
r_require_faultlock(pid).
Locker(pid,faulters,flushers,homequeue,remotequeue,
     S(wait_faulters),wait_flushers,wait_homequeue,wait_remotequeue)
+
r_require_flushlock(pid).s_no_flushwait(pid).
Locker(pid,faulters,S(flushers),homequeue,remotequeue,
     wait_faulters,wait_flushers,wait_homequeue,wait_remotequeue)
<| and(and(and(eq(faulters,0),eq(flushers,0)),
    eq(homequeue,0)),eq(remotequeue,0)) |>
r_require_flushlock(pid).
Locker(pid,faulters,flushers,homequeue,remotequeue,
     wait_faulters,S(wait_flushers),wait_homequeue,wait_remotequeue)
+
r_require_serverlock(pid).s_no_serverwait(pid).
Locker(pid,faulters,flushers,S(homequeue),remotequeue,
     wait_faulters,wait_flushers,wait_homequeue,wait_remotequeue)
<| and(eq(homequeue,0),eq(flushers,0)) |>
r_require_serverlock(pid).
Locker(pid,faulters,flushers,homequeue,remotequeue,
     wait_faulters,wait_flushers,S(wait_homequeue),wait_remotequeue)
+
r_require_homequeuelock(pid).s_no_homequeuewait(pid).
Locker(pid,faulters,flushers,S(homequeue),remotequeue,
     wait_faulters,wait_flushers,wait_homequeue,wait_remotequeue)
<| and(eq(homequeue,0),eq(flushers,0)) |>
r_require_homequeuelock(pid).
Locker(pid,faulters,flushers,homequeue,remotequeue,
     wait_faulters,wait_flushers,S(wait_homequeue),wait_remotequeue)
+
r_require_remotequeuelock(pid).s_no_remotequeuewait(pid).
Locker(pid,faulters,flushers,homequeue,S(remotequeue),
     wait_faulters,wait_flushers,wait_homequeue,wait_remotequeue)
<| and(eq(remotequeue,0),eq(flushers,0)) |>
r_require_remotequeuelock(pid).
Locker(pid,faulters,flushers,homequeue,remotequeue,
     wait_faulters,wait_flushers,wait_homequeue,S(wait_remotequeue))
+
```

```
r_free_faultlock(pid).
(  (  (  s_signal_serverwait(pid).
         Locker(pid,sub1(faulters),flushers,S(homequeue),
                remotequeue,wait_faulters,wait_flushers,
                sub1(wait_homequeue),wait_remotequeue)
         +
         s_signal_homequeuewait(pid).
         Locker(pid,sub1(faulters),flushers,S(homequeue),
                remotequeue,wait_faulters,wait_flushers,
                sub1(wait_homequeue),wait_remotequeue)
      )
      <| and(not(eq(wait_homequeue,0)),eq(homequeue,0)) |>
      (  (  s_signal_remotequeuewait(pid).
            Locker(pid,sub1(faulters),flushers,homequeue,
                   S(remotequeue),wait_faulters,wait_flushers,
                   wait_homequeue,sub1(wait_remotequeue))
            <| not(eq(wait_remotequeue,0)) |>
            Locker(pid,sub1(faulters),flushers,homequeue,
                   remotequeue,wait_faulters,wait_flushers,
                   wait_homequeue,wait_remotequeue)
         )
         <| eq(remotequeue,0) |>
         Locker(pid,sub1(faulters),flushers,homequeue,
                remotequeue,wait_faulters,wait_flushers,
                wait_homequeue,wait_remotequeue)
   )  )
   <| and(not(and(eq(wait_homequeue,0),
       eq(wait_remotequeue,0))),eq(flushers,0)) |>
   (  s_signal_flushwait(pid).
      Locker(pid,sub1(faulters),S(flushers),homequeue,
             remotequeue,wait_faulters,sub1(wait_flushers),
             wait_homequeue,wait_remotequeue)
      <| and(and(and(and(not(eq(wait_flushers,0)),eq(flushers,0)),
        eq(homequeue,0)),eq(remotequeue,0)),eq(sub1(faulters),0)) |>
      (  s_signal_faultwait(pid).
         Locker(pid,faulters,flushers,homequeue,
                remotequeue,sub1(wait_faulters),wait_flushers,
                wait_homequeue,wait_remotequeue)
         <| and(and(and(not(eq(wait_faulters,0)),eq(homequeue,0)),
             eq(flushers,0)),eq(sub1(faulters),0)) |>
         Locker(pid,sub1(faulters),flushers,homequeue,
                remotequeue,wait_faulters,wait_flushers,
                wait_homequeue,wait_remotequeue)
)  )  )
+
r_free_flushlock(pid).
(  (  (  s_signal_serverwait(pid).
         Locker(pid,faulters,sub1(flushers),S(homequeue),
                remotequeue,wait_faulters,wait_flushers,
                sub1(wait_homequeue),wait_remotequeue)
```

```
          +
          s_signal_homequeuewait(pid).
          Locker(pid,faulters,sub1(flushers),S(homequeue),
                 remotequeue,wait_faulters,wait_flushers,
                 sub1(wait_homequeue),wait_remotequeue)
       )
       <| and(not(eq(wait_homequeue,0)),eq(homequeue,0)) |>
       (  (  s_signal_remotequeuewait(pid).
             Locker(pid,faulters,sub1(flushers),homequeue,
                    S(remotequeue),wait_faulters,wait_flushers,
                    wait_homequeue,sub1(wait_remotequeue))
             <| not(eq(wait_remotequeue,0)) |>
             Locker(pid,faulters,sub1(flushers),homequeue,
                    remotequeue,wait_faulters,wait_flushers,
                    wait_homequeue,wait_remotequeue)
          )
          <| eq(remotequeue,0) |>
          Locker(pid,faulters,sub1(flushers),homequeue,
                 remotequeue,wait_faulters,wait_flushers,
                 wait_homequeue,wait_remotequeue)
    )  )
    <| and(not(and( eq(wait_homequeue,0),
        eq(wait_remotequeue,0))),eq(sub1(flushers),0)) |>
    (  s_signal_flushwait(pid).
       Locker(pid,faulters,flushers,homequeue,
              remotequeue,wait_faulters,sub1(wait_flushers),
              wait_homequeue,wait_remotequeue)
     <| and(and(and(and(not(eq(wait_flushers,0)),eq(remotequeue,0)),
        eq(homequeue,0)),eq(sub1(flushers),0)),eq(faulters,0)) |>
          (  s_signal_faultwait(pid).
             Locker(pid,S(faulters),sub1(flushers),homequeue,
                    remotequeue,sub1(wait_faulters),wait_flushers,
                    wait_homequeue,wait_remotequeue)
           <| and(and(and(not(eq(wait_faulters,0)),eq(homequeue,0)),
                eq(sub1(flushers),0)),eq(faulters,0)) |>
             Locker(pid,faulters,sub1(flushers),homequeue,
                    remotequeue,wait_faulters,wait_flushers,
                    wait_homequeue,wait_remotequeue)
)   )    )
+
r_free_serverlock(pid).
(  ( (  s_signal_serverwait(pid).
        Locker(pid,faulters,flushers,homequeue,
               remotequeue,wait_faulters,wait_flushers,
               sub1(wait_homequeue),wait_remotequeue)
        +
        s_signal_homequeuewait(pid).
        Locker(pid,faulters,flushers,homequeue,
               remotequeue,wait_faulters,wait_flushers,
               sub1(wait_homequeue),wait_remotequeue)
```

```
          )
          <| and(not(eq(wait_homequeue,0)),eq(sub1(homequeue),0)) |>
          (  (  s_signal_remotequeuewait(pid).
                Locker(pid,faulters,flushers,sub1(homequeue),
                       S(remotequeue),wait_faulters,wait_flushers,
                       wait_homequeue,sub1(wait_remotequeue))
                <| not(eq(wait_remotequeue,0)) |>
                Locker(pid,faulters,flushers,sub1(homequeue),
                       remotequeue,wait_faulters,wait_flushers,
                       wait_homequeue,wait_remotequeue)
             )
             <| eq(remotequeue,0) |>
             Locker(pid,faulters,flushers,sub1(homequeue),
                    remotequeue,wait_faulters,wait_flushers,
                    wait_homequeue,wait_remotequeue)
      ) )
      <| and(not(and(eq(wait_homequeue,0),
           eq(wait_remotequeue,0))),eq(flushers,0)) |>
      (  s_signal_flushwait(pid).
         Locker(pid,faulters,S(flushers),sub1(homequeue),
                remotequeue,wait_faulters,sub1(wait_flushers),
                wait_homequeue,wait_remotequeue)
      <| and(and(and(and(not(eq(wait_flushers,0)),eq(remotequeue,0)),
         eq(sub1(homequeue),0)),eq(flushers,0)),eq(faulters,0)) |>
         (  s_signal_faultwait(pid).
            Locker(pid,S(faulters),flushers,sub1(homequeue),
                   remotequeue,sub1(wait_faulters),wait_flushers,
                   wait_homequeue,wait_remotequeue)
            <| and(and(and(not(eq(wait_faulters,0)),eq(flushers,0)),
                eq(sub1(homequeue),0)),eq(faulters,0)) |>
            Locker(pid,faulters,flushers,sub1(homequeue),
                   remotequeue,wait_faulters,wait_flushers,
                   wait_homequeue,wait_remotequeue)
)   )  )
+
r_free_homequeuelock(pid).
(  (  (  s_signal_serverwait(pid).
         Locker(pid,faulters,flushers,homequeue,
                remotequeue,wait_faulters,wait_flushers,
                sub1(wait_homequeue),wait_remotequeue)
         +
         s_signal_homequeuewait(pid).
         Locker(pid,faulters,flushers,homequeue,
                remotequeue,wait_faulters,wait_flushers,
                sub1(wait_homequeue),wait_remotequeue)
       )
       <| and(eq(sub1(homequeue),0),not(eq(wait_homequeue,0))) |>
       (  (  s_signal_remotequeuewait(pid).
             Locker(pid,faulters,flushers,sub1(homequeue),
                    S(remotequeue),wait_faulters,wait_flushers,
```

```
                    wait_homequeue,sub1(wait_remotequeue))
              <| not(eq(wait_remotequeue,0)) |>
              Locker(pid,faulters,flushers,sub1(homequeue),
                     remotequeue,wait_faulters,wait_flushers,
                     wait_homequeue,wait_remotequeue)
           )
           <| eq(remotequeue,0) |>
           Locker(pid,faulters,flushers,sub1(homequeue),
                  remotequeue,wait_faulters,wait_flushers,
                  wait_homequeue,wait_remotequeue)
   ) )
      <| and(not(and(eq(wait_homequeue,0),
         eq(wait_remotequeue,0))),eq(flushers,0)) |>
      (  s_signal_flushwait(pid).
         Locker(pid,faulters,S(flushers),sub1(homequeue),
                remotequeue,wait_faulters,sub1(wait_flushers),
                wait_homequeue,wait_remotequeue)
    <| and(and(and(and(not(eq(wait_flushers,0)),eq(remotequeue,0)),
        eq(sub1(homequeue),0)),eq(flushers,0)),eq(faulters,0)) |>
        (  s_signal_faultwait(pid).
           Locker(pid,S(faulters),flushers,sub1(homequeue),
                  remotequeue,sub1(wait_faulters),wait_flushers,
                  wait_homequeue,wait_remotequeue)
           <| and(and(and(not(eq(wait_faulters,0)),
           eq(sub1(homequeue),0)),eq(flushers,0)),eq(faulters,0)) |>
           Locker(pid,faulters,flushers,sub1(homequeue),
                  remotequeue,wait_faulters,wait_flushers,
                  wait_homequeue,wait_remotequeue)
) ) )
+
r_free_remotequeuelock(pid).
(  (  (  s_signal_serverwait(pid).
         Locker(pid,faulters,flushers,S(homequeue),
                sub1(remotequeue),wait_faulters,wait_flushers,
                sub1(wait_homequeue),wait_remotequeue)
         +
         s_signal_homequeuewait(pid).
         Locker(pid,faulters,flushers,S(homequeue),
                sub1(remotequeue),wait_faulters,wait_flushers,
                sub1(wait_homequeue),wait_remotequeue)
       )
       <| and( eq(homequeue,0),not(eq(wait_homequeue,0))) |>
       (  (  s_signal_remotequeuewait(pid).
             Locker(pid,faulters,flushers,homequeue,
                    remotequeue,wait_faulters,wait_flushers,
                    wait_homequeue,sub1(wait_remotequeue))
             <| not(eq(wait_remotequeue,0)) |>
             Locker(pid,faulters,flushers,homequeue,
                    sub1(remotequeue),wait_faulters,wait_flushers,
                    wait_homequeue,wait_remotequeue)
```

```
              )
              <| eq(sub1(remotequeue),0) |>
              Locker(pid,faulters,flushers,homequeue,
                     sub1(remotequeue),wait_faulters,wait_flushers,
                     wait_homequeue,wait_remotequeue)
         ) )
         <| and(not( and(eq(wait_homequeue,0),
             eq(wait_remotequeue,0))),eq(flushers,0)) |>
         (  s_signal_flushwait(pid).
            Locker(pid,faulters,S(flushers),homequeue,
                   sub1(remotequeue),wait_faulters,sub1(wait_flushers),
                   wait_homequeue,wait_remotequeue)
            <| and(and(and(and(not(eq(wait_flushers,0)),
               eq(sub1(remotequeue),0)),eq(faulters,0)),
               eq(homequeue,0)),eq(flushers,0)) |>
            (  s_signal_faultwait(pid).
               Locker(pid,S(faulters),flushers,homequeue,
                      sub1(remotequeue),sub1(wait_faulters),
                      wait_flushers,wait_homequeue,wait_remotequeue)
               <| and(and(and(not(eq(wait_faulters,0)),
                   eq(homequeue,0)),eq(flushers,0)),eq(faulters,0)) |>
               Locker(pid,faulters,flushers,homequeue,
                      sub1(remotequeue),wait_faulters,wait_flushers,
                      wait_homequeue,wait_remotequeue)
      ) ) )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The protocol with 2 processors, 3 threads and 1 region.
% Each processor has a copy of the region.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init hide ({...}, % Omitted. Hide all communication actions here.
       encap({...}, % Omitted. Enfore all r_* s_* into c_*.
         Processor(pid1) || Processor(pid2) ||
         Thread(tid1,pid1,ridema) || Thread(tid2,pid2,ridema) ||
         Thread(tid3,pid1,ridema) ||
         Locker(pid1,0,0,0,0,0,0,0,0) || Locker(pid2,0,0,0,0,0,0,0,0) ||
         HomeQueue(pid1) || HomeQueue(pid2) ||
         RemoteQueue(pid1) || RemoteQueue(pid2) ||
         Region(pid1,reg(rid1,pid1,UNUSED,ema,0)) ||
         Region(pid2,reg(rid1,pid1,UNUSED,ema,0))
     ) )
```

Bibliography

[1] J.-R. Abrial, D. Cansell, and D. Méry. A mechanically proved and incremental development of IEEE 1394 FireWire tree identify protocol. *Formal Aspects of Computing*, 14(3):215-227, 2003.

[2] L. Aceto, P. Bouyer, A. Burgueño, and K.G. Larsen. The power of reachability testing for timed automata. *Theoretical Computer Science*, 300(1-3):411-475, 2003.

[3] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7-48, 1999.

[5] D. Angluin. Local and global properties in networks of processors (extended abstract). In *Proc. 12th ACM Symposium on Theory of Computing*, pp. 82-93. ACM, 1980.

[6] T. Arts and I.A. van Langevelde. Correct performance of transaction capabilities. In *Proc. 2nd Conference on Application of Concurrency to System Design*, pp. 35–42. IEEE Computer Society, 2001.

[7] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[8] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51:129–176, 1987.

[9] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1990.

[10] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking continuous-time Markov chains by transient analysis. In *Proc. 12th Conference on Computer Aided Verification*, LNCS 1855, pp. 358–372. Springer, 2000.

[11] C. Baier and M.Z. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125-155, 1998.

[12] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1/2):70-120, 1982.

[13] T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.

[14] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. *Proc. 12th Workshop on Testing of Communicating Systems*, IFIP Conference Proceedings 147, pp. 179-196. Kluwer, 1999.

[15] J. Bengtsson, W.O.D. Griffioen, K.J. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson, and Y. Wang. Automated analysis of an audio control protocol using UPPAAL. *Journal of Logic and Algebraic Programming*, 52-53:163–181, 2002.

[16] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1-3):109–137, 1984.

[17] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[18] J.A. Bergstra and J.W. Klop. Verification of an alternating bit protocol by means of process algebra. In *Proc. Spring School on Mathematical Methods of Specification and Synthesis of Software Systems*, LNCS 215, pp. 9–23. Springer, 1986.

[19] M.A. Bezem and J.F. Groote. A correctness proof of a one bit sliding window protocol in μCRL. *The Computer Journal*, 37(4):289–307, 1994.

[20] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In *Proc. 5th Conference on Concurrency Theory*, LNCS 836, pp. 401–416. Springer, 1994.

[21] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *Proc. 13th Conference on Computer Aided Verification*, LNCS 2102, pp. 250–254. Springer, 2001.

[22] S.C.C. Blom, I.A. van Langevelde, and B. Lisser. Compressed and distributed file formats for labeled transition systems. In *Proc. 2nd Workshop on Parallel and Distributed Model Checking*, ENTCS 89(1). Elsevier, 2003.

[23] S.C.C. Blom and J.C. van de Pol. State space reduction by proving confluence. In *Proc. 14th Conference on Computer Aided Verification*, LNCS 2404, pp. 596–609. Springer, 2002.

[24] M. Broy, S. Merz, and M. Spies, eds. *Formal Systems Specification: The RPC-Memory Specification Case Study*, LNCS 1169. Springer, 1996.

[25] J.J. Brunekreef. Sliding window protocols. In *Algebraic Specification of Protocols*. Cambridge Tracts in Theoretical Computer Science 36, pp. 71–112. Cambridge University Press, 1993.

[26] J.J. Brunekreef, J.-P. Katoen, R.L.C. Koymans, and S. Mauw. Algebraic specification of dynamic leader election protocols in broadcast networks. *Distributed Computing*, 9(4):157-171, 1996.

[27] J.E. Burns. A formal model for message passing systems. Technical Report TR-91, Indiana University, 1980.

[28] M. Calder and A. Miller. Using Spin to analyse the tree identity phase of the IEEE 1394 high-performance serial bus (FireWire) protocol. *Formal Aspects of Computing*, 14(3):247-266, 2003.

[29] R. Cardell-Oliver. Using higher order logic for modelling real-time protocols. In *Proc. 4th Joint Conference on Theory and Practice of Software Development*, LNCS 494, pp. 259–282. Springer, 1991.

[30] V.G. Cerf and R.E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, 1974.

[31] K.M. Chandy and J. Misra. *Parallel Program Design. A Foundation*. Addison Wesley, 1988.

[32] E.J.H. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communication of the ACM*, 22(5):281-283, 1979.

[33] D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In *Proc. 9th Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2619, pp. 113–127. Springer, 2003.

[34] A. Cimatti, F. Giunchiglia, P. Pecchiari, B. Pietra, J. Profeta, D. Romano, P. Traverso, and B. Yu. A provably correct embedded verifier for the certification of safety critical software. In *Proc. 9th Conference on Computer Aided Verification*, LNCS 1254, pp. 202–213. Springer, 1997.

[35] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.

[36] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design*. Addison Wesley, 1994.

[37] B. Courcelle. Recursive applicative program schemes. In *Handbook of Theoretical Computer Science, Volume B, Formal Methods and Semantics*, pp. 459–492. Elsevier, 1990.

[38] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proc. 12th Conference on Computer Aided Verification*, LNCS 1855, pp. 53–68. Springer, 2000.

[39] 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. *Formal Methods in System Design*, 16(3):307–320, 2000.

[40] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[41] E.W. Dijkstra. Self-stabilization in spite of distributed control. In *Selected Writings on Computing: A Personal Perspective*, pp. 41–46, Springer, 1982.

[42] E.W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5-6, Springer, 1986.

[43] D.L. Dill. The Murphi verification system. In *Proc. 8th Conference on Computer Aided Verification*, LNCS 1102, pp. 390-393. Springer, 1996.

[44] D. Dolev, M. Klawe, and M. Rodeh. An $\mathcal{O}(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3:245-260, 1982.

[45] M. Dubois, J.-C. Wang, L. Barroso, K. Lee, and Y.-S. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *Proc. 1991 ACM/IEEE Conference on Supercomputing*, pp. 197–206, 1991.

[46] P.H.J. van Eijk, C.A. Vissers, and M. Diaz, eds. *The formal description technique LOTOS*. Elsevier, 1989.

[47] E.A. Emerson and J.Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time. *Journal of the ACM*, 33(1):151-178, 1986.

[48] E.A. Emerson and C.-L. Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, 8(3):275-306, 1987.

[49] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – A protocol validation and verification toolbox. In *Proc. 8th Conference on Computer-Aided Verification*, LNCS 1102, pp. 437–440. Springer, 1996.

[50] W.J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2000.

[51] W.J. Fokkink, J.F. Groote, J. Pang, B. Badban, and J.C. van de Pol. Verifying a sliding window protocol in μCRL. In *Proc. 10th Conference on Algebraic Methodology and Software Technology*, LNCS 3116. pp. 148-163. Springer, 2004.

[52] W.J. Fokkink, J.-H. Hoepman, and J. Pang. A note on K-state self-stabilization in a ring with $K = N$. CWI Technical Report SEN-R0402, 2004.

[53] W.J. Fokkink, N.Y. Ioustinova, E. Kesseler, J.C. van de Pol, Y.S. Usenko, and Y.A. Yushtein. Refinement and verification applied to an in-flight data acquisition unit. In *Proc. 13th Conference on Concurrency Theory*, LNCS 2421, pp. 1–23. Springer, 2002.

[54] W.J. Fokkink and J. Pang. Cones and foci for protocol verification revisited. In *Proc. 6th Conference on Foundations of Software Science and Computation Structures*, LNCS 2620, pp. 267–281. Springer, 2003.

[55] W.J. Fokkink and J. Pang. Formal verification of timed systems using cones and foci. In *Proc. 6th AMAST Workshop on Real-Time Systems*, ENTCS, Elsevier, 2004. To appear.

[56] W.J. Fokkink and J. Pang. Simplifying Itai-Rodeh leader election for anonymous rings. CWI Technical Report SEN-R0405, 2004.

[57] W.J. Fokkink and J.C. van de Pol. Simulation as a correct transformation of rewrite systems. In *Proc. 22nd Symposium on Mathematical Foundations of Computer Science*, LNCS 1295, pp. 249–258. Springer, 1997.

[58] R. Franklin. On an improved algorithm for decentralized extrema finding in circular configurations of processors. *Communication of the ACM*, 25(5):336-337, 1982.

[59] G.N. Frederickson and N.A. Lynch. Electing a leader in a synchronous ring. *Journal of the ACM*, 34(1):98-115, 1987.

[60] L.A. Fredlund, J.F. Groote, and H.P. Korver. Formal verification of a leader election protocol in process algebra. *Theoretical Computer Science*, 177(2):459-486, 1997.

[61] M. Fujita, P.C. McGeer, and J.C-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149-169, 1997.

[62] H. Garavel and L. Mounier. Specification and verification of various distributed leader election algorithms for unidirectional ring networks. *Science of Computer Programming*, 29(1/2):171-197, 1997.

[63] H. Garavel, F. Lang and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology Newsletter* 4:13-24, 2002.

[64] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[65] P. Godefroid and D.E. Long. Symbolic protocol verification with Queue BDDs. *Formal Methods and System Design*, 14(3):257–271, 1999.

[66] W. Goerigk and F. Simon. Towards rigorous compiler implementation verification. In *Collaboration between Human and Artificial Societies, Coordination and Agent-Based Distributed Computing*, LNCS 1624, pp. 62–73. Springer, 1999.

[67] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[68] R.A. Groenveld. Verification of a sliding window protocol by means of process algebra. Report P8701, University of Amsterdam, 1987.

[69] J.F. Groote. *Process Algebra and Structured Operational Semantics.* PhD thesis, University of Amsterdam, 1991.

[70] J.F. Groote and B. Lisser. Computer assisted manipulation of algebraic process specifications. In *Proc. 3rd Workshop on Verification and Computational Logic*, Technical Report DSSE-TR-2002-5. Department of Electronics and Computer Science, University of Southampton, 2002.

[71] J.F. Groote and H.P. Korver. Correctness proof of the bakery protocol in μCRL. In *Proc. 1st Workshop on the Algebra of Communicating Processes*, Workshops in Computing, pp. 63–86. Springer, 1995.

[72] J.F. Groote, F. Monin, and J.C. van de Pol. Checking verifications of protocols and distributed systems by computer. In *Proc. 9th Conference on Concurrency Theory*, LNCS 1466, pp. 629–655. Springer, 1998.

[73] J.F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. *Journal of Logic and Algebraic Programming*, 55(1/2):21–56, 2003.

[74] J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In *Proc. Workshop on Semantics of Specification Languages*, Workshops in Computing, pp. 232–251. Springer, 1994.

[75] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Proc. 1st Workshop on the Algebra of Communicating Processes*, Workshops in Computing Series, pp. 26–62. Springer, 1995.

[76] J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1/2):39–72, 2001.

[77] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, eds. *Handbook of Process Algebra*, pp. 1151–1208. Elsevier, 2001.

[78] J.F. Groote and M.P.A Sellink. Confluence for process verification. *Theoretical Computer Science*, 170(1/2):47–81, 1996.

[79] J.F. Groote and J. Springintveld. Focus points and convergent process operators. A proof strategy for protocol verification. *Journal of Logic and Algebraic Programming*, 49(1/2):31–60, 2001.

[80] J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proc. 17th Colloquium on Automata, Languages and Programming*, LNCS 443, pp. 626–638. Springer, 1990.

[81] J.F. Groote and J.J. van Wamel. The parallel composition of uniform processes with data. *Theoretical Computer Science*, 266(1/2):65-75, 2001.

[82] B.T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic.* LNCS 129, Springer, 1982.

[83] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing* 6(5): 512-535, 1994.

[84] K. Havelund, K.G. Larsen, and A. Skou. Formal verification of a power controller using the real-time model checker UPPAAL. In *Proc. 5th AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, LNCS 1601, pp. 277-298. Springer, 1999.

[85] M.C.B. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137-161, 1985.

[86] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1/2):110-122, 1997.

[87] T.A. Henzinger, S. Qadeer, and S. Rajamani. Verifying sequential consistency on shared memory multiprocessor systems. In *Proc. 11th Conference on Computer Aided Verification*, LNCS 1633, pp. 301–315. Springer, 1999.

[88] D.S. Hirschberg and J.B. Sinclair. Decentralized extrema-finding in circular configurations of processes. *Communication of the ACM*, 23(11):627-628, 1980.

[89] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[90] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[91] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[92] G.J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279-295, 1997.

[93] J. Hooman and J.C. van de Pol. Formal verification of replication on a distributed data space architecture. In *Proc. ACM 2002 Symposium on Applied Computing, Special Track on Coordination Models, Languages and Applications*, pp. 351-358. ACM, 2002.

[94] G.E. Hughes and M.J. Cresswell. *A Companion to Modal Logic*. Methuen, 1984.

[95] A. Itai and M. Rodeh. Symmetry breaking in distributive networks. In *Proc. 22nd Annual Symposium on Foundations of Computer Science*, pp. 150–158. IEEE Computer Society, 1981.

[96] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60-87, 1990.

[97] B. Jonsson. *Compositional Verification of Distributed Systems*. PhD thesis, Department of Computer Science, Uppsala University, 1987.

[98] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Proc. 6th Conference on Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1785, pp. 220–234. Springer, 2000.

[99] R. Kaivola. Using compositional preorders in the verification of sliding window protocol. In *Proc. 9th Conference on Computer Aided Verification*, LNCS 1254, pp. 48–59. Springer, 1997.

[100] B. Karstens. *Formal verification of the redesign of a distributed lift system using UPPAAL*. Master thesis, Utrecht University, 2003.

[101] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proc. USENIX Winter 1994 Conference*, pp. 115–132, 1994.

[102] R.M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[103] D.E. Knuth. Verification of link-level protocols. *BIT*, 21:21–36, 1981.

[104] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333-354, 1983.

[105] C.P.J. Koymans and J.C. Mulder. A modular approach to protocol verification using process algebra. In *Applications of Process Algebra*, Cambridge Tracts in Theoretical Computer Science 17, pp. 261–306. Cambridge University Press, 1990.

[106] S.S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In *Proc. 4th Workshop on Self-Stabilization*, pp. 33-40. IEEE Computer Society, 1999.

[107] M.Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In *Proc. 12th Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, LNCS 2324, pp. 200-204. Springer, 2002.

[108] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125-143, 1977.

[109] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Transaction on Computers*, 28(9):690–691, 1979.

[110] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley, 2003.

[111] K.G. Larsen, P. Pettersson, and Y. Wang. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1/2):134–152, 1997.

[112] T. Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In *Proc. 22nd Conference on Application and Theory of Petri Nets*, LNCS 2075, pp. 242–262. Springer, 2001.

[113] G. Le Lann. Distributed systems: Towards a formal approach. *Information Processing 77, Proc. of the IFIP Congress*, pp. 155-160, 1977.

[114] M. Lindahl, P. Pettersson, and Y. Wang. Formal design and analysis of a gear controller. *Software Tools for Technology Transfer*, 3(3):353–368, 2001.

[115] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley/Teubner, 1996.

[116] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[117] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symposium on Principles of Distributed Computing*, pp. 137–151. ACM, 1987.

[118] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations. Part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[119] E. Madelaine and D. Vergamini. Specification and verification of a sliding window protocol in Lotos. In *Proc. 4th Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, IFIP Transactions (C-2), pp. 495-510. North-Holland, 1991.

[120] J. Maessen, Arvind, and X. Shen. Improving the Java memory model using CRF. In *Proc. 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 1–12. ACM, 2000.

[121] J. Manson and W. Pugh. Core semantics of multithreaded Java. In *Proc. ACM 2001 Java Grande Conference*, pp. 29–38. ACM, 2001.

[122] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255-281, 2003.

[123] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, 13(2):85–139, 1990.

[124] S. Merz. On the verification of a self-stabilizing algorithm. Technical Report, University of Munich, 1998.

[125] A. Middeldorp. Specification of a sliding window protocol within the framework of process algebra. Report FVI 86-19, University of Amsterdam, 1986.

[126] R. Milner. *A Calculus of Communicating Systems*. LNCS 92, Springer, 1980.

[127] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267-310, 1983.

[128] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[129] G.C. Necula. Translation validation for an optimizing compiler. In *Proc. 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 83–94. ACM, 2000.

[130] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[131] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proc. 8th Conference on Computer-Aided Verification*, LNCS 1102, pp. 411-414. Springer, 1996.

[132] K. Paliwoda and J.W. Sanders. An incremental specification of the sliding-window protocol. *Distributed Computing*, 5:83–94, 1991.

[133] J. Pang. Analysis of a security protocol in μCRL. In *Proc. 4th Conference on Formal Engineering Methods*, LNCS 2495, pp. 396-400. Springer, 2002.

[134] J. Pang, W.J. Fokkink, R.F.H. Hofman, and R. Veldema. Model checking a cache coherence protocol for a Java DSM implementation. In *Proc. 8th Workshop on Formal Methods for Parallel Programming: Theory and Applications*, 238. IEEE Computer Society, 2003.

[135] J. Pang, B. Karstens, and W.J. Fokkink. Analyzing the redesign of a distributed lift system in UPPAAL. In *Proc. 5th Conference on Formal Engineering Methods*, LNCS 2885, pp. 504-522. Springer, 2003.

[136] J. Pang, J.C. van de Pol, and M. Valero Espada. Abstraction of parallel uniform processes with data. In *Proc. 2nd Conference on Software Engineering and Formal Methods*, IEEE Computer Society, 2004, To appear.

[137] D.M.R. Park. Concurrency and automata on infinite sequences. In *Proc. 5th GI-Conference on Theoretical Computer Science*, LNCS 104, pp. 167-183. Springer, 1981.

[138] G.L. Peterson. An $\mathcal{O}(n \log n)$ unidirectional algorithm for the circular extrema problem. *IEEE Transactions on Programming Languages and Systems*, 4(4):758-762, 1982.

[139] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundations of Computer Science*, pp. 46-57. IEEE Computer Society, 1977.

[140] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. 4th Conference on Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1384, pp. 151–166. Springer, 1998.

[141] J.C. van de Pol. A prover for the μCRL toolset with applications – version 0.1. Technical Report SEN-R0106, CWI Amsterdam, 2001.

[142] J.C. van de Pol and M. Valero Espada. Formal specification of JavaSpaces$^{\text{TM}}$ architecture using μCRL. In *Proc. 5th Conference on Coordination Models and Languages*, LNCS 2315, pp. 274–290. Springer, 2002.

[143] F. Pong and M. Dubois. Formal automatic verification of cache coherence in multiprocessors with relaxed memory models. *IEEE Transaction on Parallel and Distributed Systems*, 11(9):989–1006, 2000.

[144] S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In *Proc. IFIP Working Conference on Programming Concept and Methods*, pp. 424-442. Chapman & Hall, 1998.

[145] 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.

[146] J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in Xesar of the sliding window protocol. In *Proc. 7th Conference on Protocol Specification, Testing and Verification*, pp. 235–248. North-Holland, 1987.

[147] Robert Bosch GmbH, Postfach 30 02 40, D-70442 Stuttgart, Germany. *CAN Specification. Version 2.0*, 1991.

[148] C. Röckl and J. Esparza. Proof-checking protocols using bisimulations. In *Proc. 10th Conference on Concurrency Theory*, LNCS 1664, pp. 525–540. Springer, 1999.

[149] J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. PhD thesis, University of Twente, 1999.

[150] J.M.T. Romijn. A timed verification of the IEEE 1394 leader election protocol. *Formal Methods in System Design*, 19(2):165–194, 2001.

[151] A. Roychoudhury and T. Mitra. Specifying multithreaded Java semantics for program verification. In *Proc. ACM SIGSOFT Conference on Software Engineering*, pp. 192–201. ACM, 2002.

[152] V. Rusu. Verifying a sliding-window protocol using PVS. In *Proc. 21st Conference on Formal Techniques for Networked and Distributed Systems*, IFIP Conference Proceedings 197, pp. 251-268. Kluwer, 2001.

[153] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45-67, 1993.

[154] A.A. Schoone. *Assertional Verification in Distributed Computing*. PhD thesis, Utrecht University, 1991.

[155] V. Schuppan and A. Biere. Verifying the IEEE 1394 FireWire tree identity protocol with SMV. *Formal Aspects of Computing*, 14(3):267-280, 2003.

[156] C. Shankland and A. Verdejo. A case study in abstraction using E-LOTOS and the FireWire. *Computer Networks*, 37(3/4):481-502, 2001.

[157] 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.

[158] X. Shen, Arvind, and L. Rodolph. Cachet: an adaptive cache coherence protocol of distributed shared memory systems. In *Proc. 13th ACM Conference on Supercomputing*, pp. 135–144, 1999.

[159] S.K. Shukla, D.J. Rosenkrantz, and S.S. Ravi. Simulation and validation tool for self-stabilizing protocols. In *Proc. 2nd SPIN Workshop*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science (32), 1996.

[160] M.A. Smith and N. Klarlund. Verification of a sliding window protocol using IOA and MONA. In *Proc. 20th Conference on Formal Techniques for Distributed System Development*, IFIP Conference Proceedings 183, pp. 19–34. Kluwer, 2000.

[161] J.L.A. van de Snepscheut. The sliding window protocol revisited. *Formal Aspects of Computing*, 7(1):3–17, 1995.

[162] K. Stahl, K. Baukus, Y. Lakhnech, and M. Steffen. Divide, abstract, and model-check. In *Proc. 6th SPIN Workshop*, LNCS 1680, pp. 57–76. Springer, 1999.

[163] N.V. Stenning. A data transfer protocol. *Computer Networks*, 1(2):99–110, 1976.

[164] J. Stoy, X. Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *Formal Methods for Increasing Software Productivity: Proc. Symposium of Formal Methods Europe*, LNCS 2021, pp. 43–71. Springer, 2001.

[165] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 1981.

[166] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[167] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[168] O.E. Theel. Exploitation of Ljapunov theory for verifying self-stabilizing algorithms. In *Proc. 14th Conference on Distributed Computing*, LNCS 1914, pp. 209-222. Springer, 2000.

[169] T. Tsuchiya, S. Nagano, R.B. Paidi, and T. Kikuno. Symbolic model checking for self-stabilizing algorithms. *IEEE Transaction on Parallel and Distributed Systems*, 12(1):81-95, 2001.

[170] Y.S. Usenko. Linearization of μCRL specifications (extended abstract). In *Proc. 3rd Workshop on Verification and Computational Logic*, Technical Report DSSE-TR-2002-5. Department of Electronics and Computer Science, University of Southampton, 2002.

[171] F.W. Vaandrager. Verification of two communication protocols by means of process algebra. Report CS-R8608, CWI, Amsterdam, 1986.

[172] G. Varghese. *Self-Stabilization by Local Checking and Corrections*. PhD thesis, MIT, 1992.

[173] R. Veldema, R.F.H. Hofman, R. Bhoedjang, and H.E. Bal. Runtime optimizations for a Java DSM implementation. In *Proc. ACM Java Grande Conference*, pp. 153–162. ACM, 2001.

[174] R. Veldema, R.F.H. Hofman, R. Bhoedjang, C. Jacobs, and H.E. Bal. Source-level global optimizations for fine-grain distributed shared memory systems. In *Proc. 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 83–92. ACM, 2001.

[175] A. Verdejo, I. Pita, and N. Marti-Oliet. Specification and verification of the tree identify protocol of IEEE 1394 in rewriting logic. *Formal Aspects of Computing*, 14(3):228-246, 2003.

[176] J.J. van Wamel. A study of a one bit sliding window protocol in ACP. Report P9212, University of Amsterdam, 1992.

[177] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Analyzing the CRF Java memory model. In *Proc. 8th Asia-Pacific Software Engineering Conference*, pp. 21–28. IEEE Computer Society, 2001.

[178] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Specifying Java thread semantics using a uniform memory model. In *Proc. ACM 2002 Java Grande Conference*, pp. 192–201. ACM, 2002.

[179] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release-consistency protocols for shared virtual memory systems. In *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation*, pp. 75–88, 1996.

[180] M.B. van der Zwaag. The cones and foci proof technique for timed transition systems. *Information Processing Letters*, 80(1):33–40, 2001.

Summary

The design and implementation of distributed systems are error-prone and becoming extremely complex. Formal methods can be used to specify systems in a precise, consistent and non-ambiguous way. Moreover, formal verification techniques, such as model checking and theorem proving, can be used to verify whether a system has desired properties. The proper use of formal methods will lead to more reliable, dependable, and secure systems in the future.

Chapter 3 presents a cones and foci proof method, which rephrases the question whether two system specifications are branching bisimilar in terms of proof obligations on relations between data objects. Compared to the original cones and foci method from Groote and Springintveld [79], this method is more generally applicable, and does not require a preprocessing step to eliminate internal loops. The method has been formalized and proved correct using the theorem prover PVS [131]. Thus a framework for mechanical protocol verification has been established.

Chapter 4 presents the verification of one of the most complex sliding window protocols presented in Tanenbaum's *Computer Networks* textbook [165] using the cones and foci method and its mechanical framework in PVS. We proved the correctness of this sliding window protocol with an arbitrary finite window size n and sequence numbers modulo $2n$. We showed that the external behavior of this protocol is equivalent to a FIFO queue of capacity $2n$. This proof is entirely based on the axiomatic theory underlying μCRL and the axioms characterizing the data types, and was checked with the help of PVS.

Chapter 5 presents that, contrary to common belief, Dijkstra's K-state mutual exclusion algorithm on a ring [40, 41] also stabilizes when the number K of states per process is one less than the number $N + 1$ of processes in the ring. The algorithm and the proof has been formalized and checked in PVS, based on Qadeer and Shankar's work [144].

Chapter 6 presents the analysis of a distributed system for lifting trucks. When testing the implementation of the system, the developers found problems. They solved these problems by trial and error, partly because the causes of problems were unclear. The analysis of the original design of the system in μCRL [75, 21] in combination with the CADP toolset [49, 63] revealed the reasons for the problems. Another new problem was found in the model, which was indeed present in the implementation of the system. Solutions were proposed and included in the μCRL specification, and we showed by model checking that

the problems were solved indeed. The developers tried to solve the problems independently. They made a redesign of the lift system based on their own solutions, The redesign was analyzed by using the real-time model checker UPPAAL [111]. We showed that the solutions of the developers do not solve the problems completely, while a refined version of our solutions contained in the μCRL specification does. Currently, the lift system is under revision, and our solutions to the problems are being implemented.

Chapter 7 presents the analysis of a self-invalidation based, multiple-writer cache coherence protocol for Jackal, which is a fine-grained, distributed shared memory implementation of Java. The verification allowed to discover two errors in the design of the cache coherence protocol. Also, a large number of inconsistencies and misunderstandings were found, mostly caused by the evolution of the implementation simultaneously with the formal analysis process. This case study benefited a lot from the μCRL distributed state space generation tool, and also pushed forward its development.

Chapter 8 presents two probabilistic leader election algorithms for anonymous unidirectional rings with FIFO channels, based on an algorithm from Itai and Rodeh [95]. In contrast to the Itai-Rodeh algorithm, our algorithms are finite-state. So they can be analyzed using explicit state space exploration; we used the probabilistic model checker PRISM [107] to verify, for rings up to size four, that eventually a unique leader is elected with probability one.

Nederlandse Samenvatting

Formele Verificatie van Gedistribueerde Systemen

Het ontwerp en implementeren van gedistribueerde systemen is zeer gecompliceerd geworden, en daarmee gevoelig voor fouten. Formele methoden kunnen worden gebruikt voor precieze en consistente specificatie van systemen. Bovendien kunnen formele verificatie gereedschappen, zoals model checkers en automatische stellingbewijzers, worden gebruikt om na te gaan of een systeem de gewenste eigenschappen heeft. Goed gebruik van formele methoden zal in de toekomst leiden tot betrouwbaarder en veiliger gedistribueerde systemen.

Hoofdstuk 3 presenteert een cones en foci bewijsmethode, die de vraag of twee systeem-specificaties equivalent zijn herformuleert in termen van bewijsverplichtingen en relaties tussen data-objecten. Deze methode is algemener toepasbaar dan de originele cones en foci methode van Groote en Springintveld [79], en is geformaliseerd en correct bewezen met behulp van de stellingbewijzer PVS [131]. Aldus wordt een raamwerk voor mechanische protocol-verificatie verkregen.

Hoofdstuk 4 bevat de verificatie van één van de meest ingewikkelde sliding window protocollen uit Tanenbaum's *Computer Networks* tekstboek [165], op basis van het raamwerk uit het vorige hoofdstuk. De correctheid van dit sliding window protocol wordt aangetoond voor een willekeurige omvang van de windows, en voor volgnummers modulo $2n$. Het externe gedrag van het protocol is equivalent met een FIFO queue van capaciteit $2n$. Het bewijs is volledig gebaseerd op de axiomatische theorie die ten grondslag ligt aan μCRL, en de axioma's voor de data-types.

Hoofdstuk 5 laat zien dat (in tegenstelling tot wat soms wordt beweerd) Dijkstra's *K-state mutual exclusion* algoritme voor een ring [40, 41] ook stabiliseert wanneer het aantal K van toestanden per proces één minder is dan het aantal $N+1$ van processen in de ring. Het algoritme en het bewijs zijn geformaliseerd in PVS, op basis van eerder werk van Qadeer and Shankar [144].

Hoofdstuk 6 presenteert de analyse van een gedistribueerd systeem voor het optillen van voertuigen zoals vrachtwagens en treinen. Tijdens het testen van een implementatie liepen de ontwerpers van het systeem tegen problemen aan. Deze problemen werden ad hoc opgelost, zonder dat de oorzaken van de problemen echt duidelijk waren geworden. Door middel van een analyse van

het oorspronkelijke systeem-ontwerp met μCRL [75, 21], in combinatie met de CADP toolset [49, 63], konden we de oorzaken voor de problemen aantonen. Bovendien werd een nieuw probleem gedetecteerd, dat inderdaad aanwezig bleek te zijn in de implementatie. We stelden oplossingen voor en namen die op in de μCRL specificatie. Door middel van model checken met CADP werd aannemelijk gemaakt dat de problemen aldus werkelijk waren opgelost. De ontwerpers echter maakten in de tussentijd onafhankelijk een herontwerp van het liftsysteem, en namen daarin andere oplossingen op voor bovengenoemde problemen. We analyseerden dit herontwerp met behulp van de tijdsgebaseerde model checker UPPAAL [111]. Deze analyse toonde aan dat de oplossingen van de ontwerpers de problemen niet volledig oplossen, terwijl een verfijnde versie van onze oplossingen dat wel doet. Momenteel is het liftsysteem opnieuw onder revisie, en worden onze oplossingen geïmplementeerd.

Hoofdstuk 7 bevat de analyse van een *multiple-writer cache coherence* protocol voor een gedistribueerde *shared memory* implementatie van Java, genaamd Jackal. Tijdens de verificatie, door middel van model checken, werden twee fouten ontdekt in het ontwerp van dit cache coherence protocol. Ook werden een groot aantal tegenstrijdigheden en misverstanden aan het licht gebracht, in de meeste gevallen veroorzaakt door de ontwikkeling van de implementatie in parallel met onze verificatie. Doordat bij deze verificatie zeer grote toestandsruimten gegenereerd werden, was het gebruik van een gedistribueerde generator essentieel. Anderzijds bleek deze case-studie een belangrijke drijfveer tot verdere verbetering van deze gedistribueerde generator.

Hoofdstuk 8 presenteert twee probabilistische *leader election* algoritmes voor anonieme, unidirectionele ringen met FIFO kanalen, gebaseerd op een algoritme van Itai en Rodeh [95]. In tegenstelling tot het Itai-Rodeh algoritme hebben onze algoritmes een eindige toestandsruimte. Aldus kunnen zij worden geanalyseerd door middel van expliciete exploratie van de toestandsruimte; wij hebben de probabilistische model checker PRISM [107] gebruikt om te verifiëren, voor ringen ter grootte Vier, dat met kans één uiteindelijk een unieke leider wordt gekozen.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

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, UL. 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

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java Programs in Higher Order Logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: Syntax and Semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical Verification of Concurrency Control and Recovery Protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and Presentation of Formal Mathematical Documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive Machine Control: A Simulation Approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing State Space Reduction Techniques for Model Checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14