

VRIJE UNIVERSITEIT

On Distributed Verification and Verified Distribution

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 donderdag 25 november 2004 om 10.45 uur
in het auditorium van de universiteit,
De Boelelaan 1105

door

Simona Mihaela Orzan
geboren te Iași, Roemenië

promotor: prof.dr. W.J. Fokkink
copromotor: dr. J.C. van de Pol

On Distributed Verification and Verified Distribution

Simona Orzan

© Simona Orzan, Amsterdam 2004
Printed by Ponsen & Looijen
ISBN 90-6196-528-4
IPA dissertation series 2004-17



The research in this thesis has been carried out at the Centre for Mathematics and Computer Science (CWI), under the auspices of the research school IPA (Institute for Programming research and Algorithmics). It has been financially supported by PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research (NWO), the Dutch Ministry of Economic Affairs and the Technology Foundation STW, grant CES.5009 “Formal Design, Tooling, and Prototype Implementation of a Real-Time Distributed Shared Dataspace”.

Lots of thanks to ...

✍️ Jaco van de Pol, my supervisor, for his careful guidance and for doing everything with such an exceptional combination of enthusiasm and common sense. He always listens patiently, asks good questions, gives precious advice and recognizes the right path.

📧 my promotor Wan Fokkink. He maintained a relaxed but productive environment in SEN2, and in my last few months at CWI he efficiently led the process of writing this thesis to a successful termination.

🌱 Stefan Blom, for our fruitful and pleasant cooperation. He let me join the project of “distributing” the μ CRL toolset and this became a long exciting journey through algorithmic exercises and programming traps. His sharp, sometimes cynical, observations on everything catching his attention, from scientific curiosities to Liga biscuits, were often the salt and pepper of my working days at CWI.

✉️ my reading committee: Farhad Arbab, Luboš Brim, Jan Friso Groote, Wim Hesselink and Jan Rutten, for their constructive comments. Especially Wim’s thorough reading led to many corrections and presentation improvements. I am also grateful to Boudewijn Haverkort and Jozef Hooman, who agreed to act as opponents.

✓ the anonymous referees of the papers contained in this thesis. Their expert feedback contributed to the quality of the material and helped to place it into a wider perspective.

☺ the present and former members of the SEN2 group and all the PAM participants. From them I learned much about various things related to formal methods, but also about how to exchange scientific ideas, conduct technical conversations and give good presentations.

🧑 Erik de Vink and Sjouke Mauw for their patience while I’m taking my time diving into security, and my whole new Eindhoven group for their constantly happy mood.

► Gabriel Ciobanu, Dan Cristea and Dorel Lucanu, my former teachers in Romania, who gave me the first reasons to suspect that research might be fun.

😊 Alexandru, Bernadet, Claudița, Daniel, Dorina, Julian, Jun, Mark, Miguel, Natalia, noi_toti, Oana, Pieter Jan, Rob, Sander, Stefan, Vincent, Yuliya (in alphabetical order), for, also in alphabetical order, encouraging (e-)smiles and (e-)words, feeding my ficus while I was away, long necessary coffee breaks, longer and just as necessary nights out, reading and correcting my introduction, the exciting hours spent on the climbing walls or at the bridge table, and very much more!

♥ my sister Alex and my parents M&M, who tirelessly cheered for me all the way! Bas’s family for always making me feel welcome! and Bas, for the bit of discipline and the lot of love he brings into my life.

Contents

1	Introduction	1
I	Distributed Algorithms for Verification	5
2	Parallel and Distributed Model Checking	7
2.1	Model checking	7
2.2	Distributed algorithms	9
2.3	State spaces	12
3	Strong Bisimulation Reduction	17
3.1	The relational coarsest partition problem	19
3.2	A naive sequential solution	19
3.3	...and its distributed implementation	21
3.4	An optimized sequential solution	28
3.5	...and its distributed implementation	33
3.6	Experiments	37
3.7	Conclusions	49
4	Branching Bisimulation Reduction	51
4.1	Partition refinement based on signatures computation	52
4.2	Correctness of the naive algorithm	53
4.3	Sequential branching bisimulation minimization	56
4.4	Distributed branching bisimulation minimization	59
4.5	Experiments	61
4.6	Conclusions	64
5	Detecting Strongly Connected Components	65
5.1	Preliminaries	66
5.2	Graph transformations	69
5.3	Distributed implementation of the transformation routines	71
5.4	Three example algorithms	79

5.5	Experiments	81
5.6	Conclusions	83
II	Verification of Data Distribution Architectures	85
6	Shared Dataspace Software Architectures	87
6.1	Some distributed shared dataspace coordination models	87
6.2	Views and models of Splice	88
6.3	Introduction to μ CRL	89
7	Expressiveness and Distribution of a Simple Shared Dataspace Architecture	91
7.1	Process algebra with data	94
7.2	Distributed implementation	97
7.3	Expressiveness	99
7.4	Complete distribution of requirements specifications	105
7.5	Full proofs	109
7.6	Conclusions	113
8	Verification and Prototyping of Distributed Dataspace Architectures	117
8.1	The space calculus	119
8.2	The verification and prototyping tools	125
8.3	Examples	130
8.4	Conclusions	131
9	Directions for Future Research	133
	Summary	143
	Samenvatting	145

1

Introduction

The central keywords of this thesis are “verification” and “distribution”. *Verification* refers to the process of finding, by formal means, design errors in complex hardware and software systems, or assessing their absence. The necessity of formal verification is supported by many examples of fatal, dramatic, expensive, or just annoying design errors, see for example [Der]. *Distribution*, the art of making many different, sometimes geographically distant, computers cooperate to achieve a common goal, plays a role in this thesis both as means (Part I) and as object (Part II) of verification.

Verification is usually performed by use of various *formal methods* that allow the precise description of systems and their desired properties, as well as reasoning about them. The systems are modeled in a language with clear mathematically defined syntax and semantics, like logics, automata and process algebra. Using axioms, rules and proof techniques, one can check whether properties are met, show that different models have equivalent behaviors, etc.

This thesis consists of two parts, one about distributed tools to support the verification process and the other about verifying a specific type of distributed software architectures. The following sections detail the background and contributions of each part separately.

Distributed verification

Analyzing formal specifications can be done manually for simple systems, but for large real-life systems automated support is essential. The formal verification tools fall roughly in two categories: *theorem provers* and *model checkers* [CGP00]. In the theorem proving approach, the system and the desired property are expressed as formulas in some logic, then the theorem prover assists in finding a proof of that property for that system. This requires some degree of human expert intervention, which makes it

an insightful approach, but also specialized and time-consuming. In the model checking approach, on which we focus in this thesis, a finite model (state space) of a system is built and the required properties are checked against it. Model checking does not require a high degree of expertise from the users and can be completely automatized, which makes it a usually fast operation. Moreover, when violations of the properties are found, counterexamples are produced (representing subtle errors in design) that can be very helpful for debugging.

But there is one serious problem that model checking has: the combinatorial explosion of the state space when the system specified is made up of many interacting components. For complex industrial systems, the state space can become huge, much larger than the internal memory of one machine. Many techniques have been developed to cope with this problem: symbolic representations, abstractions, compositional verification, partial order reduction, on-the-fly processing, using the disk as extra storage, or using clusters of computers. The latter approach, distributed processing, is the topic of intensive research in recent years, due to the availability at low cost of clusters. Distributed solutions for many verification problems are produced, from state space generation to the actual model checking.

The first part of this thesis contributes to this direction by proposing distributed message passing algorithms for computing state space equivalences. These algorithms allow reducing huge state spaces while preserving interesting properties. It is often the case that a state space needs the memory of a few machines to be stored, while its equivalent minimized version does fit in the memory of one machine and can be model checked by single-threaded tools. The algorithms and implementations that we propose achieve the property that the memory usage per machine decreases almost linearly with the number of machines employed. Since memory usage is the bottleneck, this is an important feature that ensures that linearly larger state spaces can now be verified.

The equivalence reduction algorithms can be used for equivalence checking as well. Equivalence checking is a way to compare a specification with an implementation. If the two models can be identified, then this is a proof that the implementation correctly implements the specification, according to the notion of correctness captured by the chosen equivalence.

Verified distribution

A distributed software system is generally seen as a number of single-threaded applications together with a distributed communication layer that coordinates them. To solve the task of coordination, various models have been proposed, among which the *shared dataspace* is one of the most popular. A shared dataspace architecture is a (possibly distributed) storage of information and/or resources, viewed as an abstract global store, that applications use to coordinate their actions by reading, writing and

removing pieces of data. These architecture models are used in parallel computing applications (Linda, Gamma), network applications (Bonita), WCL), command and control systems (Splice), management of federations of devices (JavaSpaces).

There is a vast literature devoted to modeling, analysis and verification of this type of architectures. Research issues include comparison of the expressive power of different shared dataspace paradigms, formal semantics for them, efficient implementations, etc. The attention is justified by the vital role that the choice and implementation of the architecture plays in the correctness and the performance of a software system.

In the second part of this thesis, we approach some of these issues with process algebraic modeling and proof techniques and we explore the possibility of applying existing model checking tools to the verification of shared dataspace architectures.

Overview

In order to facilitate reading, we have kept the chapters as self-contained as possible. The two parts can be read independently and each has its own introductory chapter.

Part I

Chapter 2 provides an introduction to parallel and distributed model checking, outlines our framework and fixes the most important definitions, notations and conventions occurring in the first part.

In Chapter 3 we present two distributed algorithms for the reduction of state spaces modulo strong bisimulation equivalence. Until now this problem has only been solved by sequential or parallel (shared memory) algorithms. Our algorithms are a straightforward and an optimized distributed version of the “naive” reduction method given by Kanellakis and Smolka in [KS83]. We give a detailed description and analysis of both, including correctness and complexity proofs. We also show by performance data that they scale up well in both memory and time and comment on how the two solutions compare to existing (sequential) tools and to each other.

Chapter 4 continues the development of the basic algorithm from Chapter 3 by adapting it to deal with another behavior equivalence, namely branching bisimulation. This is more complicated due to the fact that the transitive closure of the τ -transitions must be taken into account. We prove the correctness of the adapted basic algorithm, discuss its complexity and show that the implementation scales up. This is the first non-sequential solution given for branching bisimulation.

Chapter 5 treats the well known problem of detecting strongly connected components of a graph. This problem has a linear sequential solution, but which unfortunately is difficult to implement in a parallel/distributed setting. However, for the specific type of graphs that represent state spaces, we give a collection of heuristics that solve

it (distributedly) in reasonable time. Finding the strongly connected components has applications both to the reduction modulo branching bisimulation and to model checking state spaces against LTL/CTL formulas.

Part II

Chapter 6 provides a short literature survey on coordination models, shared dataspace architectures and issues regarding their distribution.

Chapter 7 is a study on the expressiveness and distributed implementation of the simplest imaginable shared dataspace coordination architecture, namely a data set with two primitives: one to write an element and another to read an element. First we give — and prove correct — a distributed implementation of this architecture, where the global storage is replaced by local caches. Next, we formalize a notion of implementation on this architecture and we show that in fact any specification of system requirements admits such an implementation. This proves that this minimal setting has maximal expressiveness.

Chapter 8 introduces a modeling language (space calculus) for distributed dataspace architectures. We give a syntax and an operational semantics to this language and provide tool support for the functional and performance analysis of its expressions. This is also a step towards establishing a formal link between the implementation model and the actual implementation, for the specific case of distributed dataspace systems. We illustrate the approach with two small examples of studying transparent replication in Splice and transparent distribution in JavaSpaces.

Chapter 9 draws some global conclusions and examines possible future plans.

Chapters 3 and 4 are the result of joint work with Stefan Blom and have been published as [BO02, BO03b, BO03a]. An extended version of [BO02] will appear as [BO05]. Chapters 5, 7 and 8 are joint work with Jaco van de Pol. The latter two are based on [OP02, OP03]. Chapter 5 has not yet been published.

Part I

Distributed Algorithms for Verification

2

Parallel and Distributed Model Checking

Parallel and distributed data processing is a very active area of research, driven by the ever growing need for speed and space. Many scientific and engineering applications use fast multiprocessor supercomputers or networks of computers to deliver better performance. Examples include real-time video processing, factoring large numbers, large-scale simulations for difficult problems like weather forecasting, numerical applications, etc. Our main concern and motivation is the verification of large industrial systems, which is a big challenge to the current technological possibilities. From the many techniques and tools being developed for it, we choose to concentrate on exploiting the power of distributed memory machines, or clusters of workstations.

Part I presents four distributed algorithms belonging to the area of enumerative model checking of large state spaces. In this chapter, we introduce the most relevant concepts, namely enumerative model checking (Section 2.1), distributed algorithms (Section 2.2) and state spaces (Section 2.3).

2.1 Model checking

Symbolic versus enumerative. Verification by model checking has been developing in two main directions: *symbolic* model checking, and *enumerative*, or explicit state, model checking. In the symbolic approach, compressed representations of state spaces are used. They seem especially advantageous for hardware verification, where a system is described as a set of processes progressing together synchronously. The enumerative approach, where all the states are generated and all transitions computed, is usually considered more appropriate for software verification. The enumerative generation tools can be sub-divided into *on-the-fly* and *full-generation*. An on-the-fly tool will compute the transitions of a state on demand, while it is checking a property. (Symbolic tools can also work on-the-fly.) A full-generation tool will first compute the

whole state space and only then start checking the property. The main advantage of on-the-fly tools is that if the property can be proved or refuted by exploring only a small part of the state space, the unnecessary generation of the rest of the state space is avoided. However, if proving a property requires visiting the whole state space then full-generation tools have an important advantage: after generating a state space it can be reduced modulo an equivalence that preserves the properties to be checked.

State space explosion. The most serious problem that all verification methods have to face, and especially the full-generation ones, is the *state space explosion*, which is the combinatorial growth of the state space of a process when it is made out of several parallel components. State space explosion is the reason why the size of systems that verification tools can handle is traditionally very small, and why many interesting applications still remain out of reach.

2.1.1 Parallel and distributed approaches to model checking

The easiest solution to solving this memory shortage problem is to use a machine with more memory. However, machines with a really big amount of memory are very expensive. Due to recent advances in the development of networks, clusters have become an appealing low-cost alternative platform for parallel computing. Besides wide availability and scalability, they also have the advantage of open-source software like Linux, file systems (NFS, PVFS) and communication libraries (MPI, PVM). Therefore, a lot of effort is currently invested in building distributed tools, in both the symbolic and the enumerative verification communities.

Below we highlight different problems that occur in the model checking process and we take a look at some parallel and distributed algorithms developed for them.

State space generation. Quite a few state space generation methods have been proposed that use shared and distributed memory to obtain better performance. A few examples are [HBB99, CCM01, Cia01]. In [GMS01], large state spaces are generated on a cluster of workstations. The run time performance is drastically improved with respect to similar sequential tools, but the size of the state spaces that can be generated is not bigger, since in the end the whole state space is collected on one machine. In [BLL03], the state space is generated in a distributed format [BLL03] (see also paragraph 2.3.4), on which the reduction tools described in the next three chapters are applied.

Equivalence reduction. Reduction of a state space modulo some property preserving equivalence can considerably decrease the size of the state space that needs to be verified. This operation is particularly important in cases where the original state space is too big to fit on a single machine, as it happened in some recent case studies

with the μ CRL toolset [PFHV03, PV03], where only the reduced state space could fit on one machine. The most well known solutions for the reduction problem have been given by Kanellakis-Smolka [KS83] and Paige-Tarjan [PT87]. These are sequential algorithms, and based on them parallel shared memory algorithms have been constructed as well [ZS93, RL98]. The algorithms that we propose in this thesis are the first that make use of distributed memory.

Equivalence checking. Since the reduction algorithms actually compute the equivalence classes, it turns out that they can also be used for *checking equivalence* of two state spaces. More precisely, if we apply the reduction algorithm on the union of the two state spaces, their roots get collapsed to the same reduced state if and only if the state spaces are equivalent. This holds for any equivalence. Unlike equivalence reduction, equivalence checking can easily and naturally be performed on-the-fly. Recently, on-the-fly equivalence reduction has been distributed [JM04].

State space exploration and model checking. To actually check desired properties on the generated (and possibly reduced) state space, or on-the-fly, a logic to specify these properties is used. Common logics are LTL (Linear Temporal Logic), CTL (Computation Tree Logic) and μ -calculus. The properties expressed are of two basic types: safety, stating that “something bad never happens”, and liveness, stating that “something good eventually happens”. Parallel and distributed model checking started with the parallelization of the Mur ϕ verifier [SD97] and continued with the distribution of the algorithm used by SPIN to verify safety properties [LS99] and the distribution of the symbolic model checker UPPAAL [BHV00]. The algorithm in [LS99] was later extended [BBS01] in order to cover liveness properties too. Also μ -calculus model checking has been parallelized [BLW01].

2.2 Distributed algorithms

We now introduce a number of useful notions and explain the context and the assumptions under which we later develop the algorithms in Chapters 3, 4 and 5.

2.2.1 Distributed algorithms and parallel algorithms

A clear distinction should be made between the parallelism on shared memory machines (SMM) (for instance [RL98, KR90]) and the parallelism on distributed memory machines (DMM) (for instance [BBS01, FHP00], this thesis). The algorithms in the first category usually assume, besides a huge amount of memory, the presence of many processors as well. The number of processors available is in the order of the problem size, while for distributed memory machines the number of processors is much much

smaller. Also, for shared memory, communication between processors is not an issue, while for distributed memory the latency of communication plays an important role.

So, in our opinion, programs designed for shared memory cannot easily run on a DMM, because most of the time the latencies of a virtual shared memory system are too high. (However, if the algorithm is really tolerant to high latencies then of course this is possible. In this case, there is also the possibility of using a disk as extra storage rather than remote memory.) In the other direction however (from DMM to SMM), program migration does produce acceptable results, as we will show on some examples in Chapter 3.

2.2.2 Distributed algorithms and distributed algorithms

The term “distributed algorithms” as used in the emerging field of distributed verification – and in this thesis – does not refer to classical distributed algorithms like the Dining Philosophers, Leader Election, Stabilization etc. From our viewpoint these latter algorithms are rather distributed *protocols*, since communication plays the essential role and the goal is mostly to achieve some kind of global decision, synchronization or consistency.

Our distributed algorithms act on top of such distributed protocols and could sometimes use them as communication primitives. They focus on the computation side, on solving together a large problem. They are actually parallel algorithms working on distributed memory machines. But we prefer to call them distributed in order to make the distinction with parallel algorithms working on shared memory machines, for which the problem is again different. The main difficulty in designing “our” distributed algorithms is dividing the computation in such a way that communication is triggered rather infrequently, but in the same time avoiding large idle times.

Another relevant difference is that in a classical distributed algorithm the network topology plays an important role, while our distributed algorithms live at a higher level of abstraction and are not aware of topology aspects. We take the simple approach that every two processors can send/receive messages to each other.

2.2.3 Complexity measures for distributed algorithms

To evaluate the performance of our designs, we use the time/message/bit complexity measures for distributed algorithms, as defined in [AW98]. We call the parallel pieces of such an algorithm *workers*.

Definition 2.1 (time complexity) *The worst-case time complexity is the maximal time, among all possible executions for all possible inputs, elapsed between the moment when the first worker starts execution and the moment when last worker stops.*

Sending a message counts as one time unit.

Definition 2.2 (message complexity) *The worst-case message complexity is the maximal number of messages sent by all workers during a run (largest out of all runs).*

Definition 2.3 (bit complexity) *The worst-case bit complexity is the maximal number of bits (messages \times their size) sent by all workers during a run (again, largest out of all possible runs).*

2.2.4 Performance measures for distributed algorithms

In many cases, the worst-case theoretical complexity of a distributed algorithm is not a good indication of its usefulness in practice. To show the qualities of a distributed tool, also performance measurements are made, usually on input types to which the tool is targeted (in our case, state spaces).

An important measure for the practical performance of a (parallel or) distributed algorithm is the *speedup*. If D is a distributed algorithm, S its fastest sequential version, and $T(D, p)$, $T(S)$ the times they need on p processors and one processor, respectively, then the speedup of D on p processors is

$$\text{speedup}_D^p = \frac{T(S)}{T(D, p)}$$

2.2.5 Clusters of workstations

The target architecture of the algorithms that we develop is a cluster whose nodes are connected by a high bandwidth network (a Distributed Memory Machine). We assume that both the nodes and the network are reliable (no node failure, no message loss) and that the order of messages between nodes is preserved (the communication channels are FIFO queues). We also assume that the channels are unbounded. Processes communicate by executing **SEND TO** and **RECEIVE** operations; **SEND TO** is non-blocking, **RECEIVE** is blocking. A few other communication primitives built on top of these two will be used as well. They will be explained when they first occur.

We implemented all communication primitives using the LAM/MPI library (Message Passing Interface) [SL03]. The correctness proofs of our distributed algorithms assume correctness of these primitives. For a thorough discussion regarding implementation of communication primitives in the distributed model checking literature, see [Jou03].

We will sometime mention the latency, bandwidth or throughput of a network. These notions are explained below.

The *message delay* $D(s)$ of a message of size s is its travel time.

The *latency* of a network is the time it takes a small packet to travel from its source to its destination. Formally, it is $D(s_0)$, meaning the message delay of the smallest message admissible by the communication system (s_0).

The (*asymptotic*) *bandwidth* of a network is the amount of data that can be transmitted in a fixed amount of time. Formally, it is the value of the function $\frac{size}{D(size)}$ as $size > 0$ approaches infinity. Bandwidth is expressed in *bps* (bits per second). The most often encountered networks nowadays are Fast Ethernet, with a bandwidth of 100 Mbps, and Gigabit Ethernet (bandwidth 1 Gbps).

The (*transmission*) *throughput* is the transfer rate measured at the sender, that is the data rate at which an infinite stream of messages can be pushed into the network without causing data loss.

2.2.6 P-completeness

In complexity theory, P is the class of problems for which an efficient (i.e. deterministic polynomial time) solution exists. When moving from sequential to parallel computation models, the question arises which problems in P also admit an efficient parallel solution, that is one that runs in polylogarithmic time ($\mathcal{O}((\log N)^{\mathcal{O}(1)})$) using a polynomial number of processors ($\mathcal{O}(N^{\mathcal{O}(1)})$). The class NC contains the problems with an efficient parallel solution. It is known that $NC \subseteq P \subseteq NP$ and it is widely believed that both inclusions are strict, although the proofs of that are difficult open problems. Just like NP-complete problems are collected in order to shed some light on the difference between P and NP, *P-complete* problems are believed to form the difference between NC and P and considered to be not efficiently parallelizable, or *inherently sequential*. Formally, a problem is P-complete if it is in P and reducing any other problem in P to it is in NC.

The relational coarsest partition problem [KS83], which is the basis of equivalence checking problems, has been proved P-complete [ABG91]. Also the standard depth first search, typically used by the algorithms for detection of strongly connected components, is P-complete [Rei85]. Although these results do not directly apply to our setting, which is distributed, rather than parallel ($\mathcal{O}(1)$ processors instead of $\mathcal{O}(N)$), they are an indication that the equivalence checking and the detection of strongly connected components are challenging problems to solve in a distributed manner. Note that for verification purposes good memory performance is more important than time and also that the P-completeness results do not exclude the existence of polynomial time parallel algorithms that could still give a good speedup.

2.3 State spaces

2.3.1 Labeled transition systems

Let **Act** be a fixed set of labels, representing actions. **Act** contains a special action τ that stands for an internal (silent) step. A labeled transition system (LTS) is a triple (S, T, s_0) consisting of a set of states S , a set of transitions $T \subseteq S \times \text{Act} \times S$ and an initial state $s_0 \in S$. The transition relation will also be denoted by \rightarrow and

we will write $p \xrightarrow{a}_T q$ for $(p, a, q) \in T$ or $p \xrightarrow{a} q$ when T is understood. LTSs are a convenient representation format for the behavior of systems, requirements specifications, implementations etc.

2.3.2 Behavioral equivalences

Verification techniques use behavioral equivalences and preorders to decide, prove, check or test that two systems can be identified, or that a system implements a specification. Some equivalences are also congruences with respect to parallel composition and thus allow application of efficient compositional verification techniques (see for example [TLG03]). A comprehensive presentation of many equivalences is given in [Gla01]. Here we introduce four popular LTS equivalences that are used or mentioned at various points in the thesis. They are all based on the concept of bisimulation, which captures the relation between two systems that behave in a similar manner, matching each other's moves.

Strong bisimulation equivalence is the most powerful behavioral equivalence. It lets two systems be equal only if they perfectly simulate each other.

Definition 2.4 (strong bisimulation equivalence [Par81]) *Let $(S^1, \rightarrow_1, s_0^1)$ and $(S^2, \rightarrow_2, s_0^2)$ be two LTSs. A binary relation $R \subseteq S^1 \times S^2$ is a strong bisimulation if for all $a \in \text{Act}$ and all $p \in S^1, q \in S^2$ such that $p R q$:*

- if $p \xrightarrow{a}_1 p'$ then $\exists q' \in S^2 : q \xrightarrow{a}_2 q' \wedge p' R q'$
- if $q \xrightarrow{a}_2 q'$ then $\exists p' \in S^1 : p \xrightarrow{a}_1 p' \wedge p' R q'$

The union of all strong bisimulation relations is itself a strong bisimulation and moreover an equivalence relation; it is therefore named strong bisimulation equivalence. Two states identified by strong bisimulation equivalence are called strongly bisimilar. Two LTSs are bisimilar if their initial states are bisimilar.

Bisimulations that abstract from internal computation are particularly useful, because they equate more states while preserving interesting properties. Weak bisimulation equivalence is the equivalence obtained from strong bisimulation equivalence by relaxing the transition relation such that an arbitrary number of internal steps ($\tau \rightarrow$) is allowed before and after a visible step (\xrightarrow{a}). Branching bisimulation abstracts from the execution of internal steps too, but only of those that do not participate in a choice, so that the branching structure of processes is preserved. τ^*a -equivalence, weaker than branching and stronger than weak equivalence, is useful because it preserves safety properties (“something bad never happens”).

Definition 2.5 (weak bisimulation equivalence [Mil89]) *Let $(S^1, \rightarrow_1, s_0^1)$ and $(S^2, \rightarrow_2, s_0^2)$ be two LTSs. A binary relation $R \subseteq S^1 \times S^2$ is a weak bisimulation if for all $a \in \text{Act}$ and all $p \in S^1, q \in S^2$ such that $p R q$:*

- if $p \xrightarrow{a}_1 p'$ then $(a \equiv \tau \wedge p' R q) \vee (\exists q' \in S^2 : q \xrightarrow{\tau}_2^* \xrightarrow{a}_2 \xrightarrow{\tau}_2^* q' \wedge p' R q')$
- if $q \xrightarrow{a}_2 q'$ then $(a \equiv \tau \wedge p R q') \vee (\exists p' \in S^1 : p \xrightarrow{\tau}_1^* \xrightarrow{a}_1 \xrightarrow{\tau}_1^* p' \wedge p' R q')$

Weak bisimulation equivalence, or observational equivalence, is the union of all weak bisimulations and it is the largest weak bisimulation [Mil89]. Two LTSs are weakly bisimilar if their initial states are weakly bisimulation equivalent.

Definition 2.6 (branching bisimulation equivalence [GW96]) Let $(S^1, \rightarrow_1, s_0^1)$ and $(S^2, \rightarrow_2, s_0^2)$ be two LTSs. A binary relation $R \subseteq S^1 \times S^2$ is a branching bisimulation if for all $a \in \text{Act}$ and all $p \in S^1, q \in S^2$ such that $p R q$:

- if $p \xrightarrow{a}_1 p'$ then $(a \equiv \tau \wedge p' R q) \vee (\exists q', s \in S^2 : q \xrightarrow{\tau}_2^* q' \xrightarrow{a}_2 s \wedge p R q' \wedge p' R s)$
- if $q \xrightarrow{a}_2 q'$ then $(a \equiv \tau \wedge p R q') \vee (\exists p', s \in S^1 : p \xrightarrow{\tau}_1^* p' \xrightarrow{a}_1 s \wedge p' R q \wedge s R q')$

Like in the strong bisimulation case, it turns out that the union of all branching bisimulations is an equivalence relation [Bas96], branching bisimulation equivalence, or branching bisimilarity. Two LTSs are branching bisimilar if their initial states are branching bisimilar.

Definition 2.7 (τ^*a -equivalence [BFG⁺91]) Let $(S^1, \rightarrow_1, s_0^1)$ and $(S^2, \rightarrow_2, s_0^2)$ be two LTSs. A binary relation $R \subseteq S^1 \times S^2$ is a τ^*a -bisimulation if for all $a \in \text{Act}$ and all $p \in S^1, q \in S^2$ such that $p R q$:

- if $p \xrightarrow{a}_1 p'$ then $(a \equiv \tau \wedge p' R q) \vee (\exists q' \in S^2 : q \xrightarrow{\tau}_2^* \xrightarrow{a}_2 q' \wedge p' R q')$
- if $q \xrightarrow{a}_2 q'$ then $(a \equiv \tau \wedge p R q') \vee (\exists p' \in S^1 : p \xrightarrow{\tau}_1^* \xrightarrow{a}_1 p' \wedge p' R q')$

The τ^*a -equivalence is the equivalence relation obtained by considering the union of all τ^*a -bisimulations. Two LTSs are τ^*a -equivalent if their initial states are.

2.3.3 State spaces are special!

All the algorithms that we propose exploit the fact that the LTSs on which they act represent state spaces. Due to the way they are generated, namely as interleavings of several parallel processes, state spaces have some special characteristics:

- there is a special initial state (root);
- there are multiple labels, but in a small constant number compared to the number of states;
- the number of transitions originating in each state is bounded by a constant (bounded fanout);

- the depth (i.e. longest distance between any state and the root) is relatively small compared to the size of the state space (given by the number of transitions);
- the diameter (longest shortest path between any two connected states) is also much smaller than the size;
- strongly connected components (SCCs) are usually not very long cycles, but small dense knots, and many larger SCCs are built out of smaller ones.

An analysis of typical state space characteristics has recently been made in [Pel04]. The tool proposed in [GH03] offers insights in the structure of state spaces by means of visualization techniques. Some of the characteristics mentioned above are visible on the 3D images of various state spaces. A collection of state spaces is created and maintained [CI] under the name VLTS benchmark.

The input of our algorithms is always an LTS (S, T, s_0) with N states and M transitions. Since it represents a state space, this LTS usually conforms to the description above. Our algorithms, presented in the following three chapters, are designed with these characteristics in mind and will perform best on state spaces. This does not imply any technical restriction on the input – our algorithms are correct and work for any LTS.

2.3.4 Input/output formats

Due to the growing number of model checking tools and the many different directions in which they evolve, efforts are now being made on the development of standard state space storage formats and standard interfaces. Some examples are the *FC2* file format [Mad92], Aldebaran’s textual format *AUT* [FGK⁺96], the Binary Coded Graphs format *BCG* [FGK⁺96], the *SVC* format [BLL03].

The main input/output format used by the tools described in this thesis is *SVC2* [BLL03], which was especially designed for distributed settings. In this format, the set of states is divided into n subsets (*segments*) and the set of transitions into n^2 subsets, accordingly (one transition subset for each pair of segments). In order to allow easy access, each transition subset is ordered – i.e., transformed into a list – and split into 3 sublists: source states, labels, destination states. In the current implementation, every sublist is stored as a different file. Our sequential tools (Sections 3.2, 3.4, 4.3) accept inputs in *AUT* and *BCG* format.

3

Strong Bisimulation Reduction

In this chapter two distributed algorithms for strong bisimulation reduction of LTSs are presented. They are useful in the context of enumerative verification, when large state spaces (represented as transition systems) are first fully generated, and only then analyzed. To make analysis easy, such a state space is first reduced modulo an equivalence preserving interesting properties. Strong bisimulation preserves all properties expressible as HML formulas [HM80].

Related work Very good sequential algorithms have been described for bisimilarity reduction and bisimilarity checking: [KS83, PT87] and based on these, [Fer90]. In [BGS92], the bisimilarity checking problem was proved P-complete, which means that it is hard to have it parallelized efficiently (Section 2.2.6 contains detailed explanations).

Parallel versions of [KS83] and [PT87] have been proposed [ZS93, RL98], with time complexity $\mathcal{O}(N^{1+\epsilon})$ using $\frac{M}{N^\epsilon}$ CREW PRAM processors (for any fixed $\epsilon < 1$), and $\mathcal{O}(N \log N)$ with $\mathcal{O}(\frac{M}{N})$ CREW PRAM processors, respectively. These algorithms are designed for shared memory machines and they are difficult to translate efficiently to a distributed memory setting. [JKOK98] proposes a randomized parallel implementation of the Kanellakis-Smolka algorithm for bisimilarity checking, that works in linear time $\mathcal{O}(N)$ on $\mathcal{O}(N^2)$ processors. This solution does not consider the case of multiple labels, and it is not precise (has some small probability of error).

Like the Kanellakis-Smolka [KS83] and Paige-Tarjan [PT87] solutions, our algorithms are based on partition refinement. The computed refinements are precisely the refinements computed by the “naive” reduction algorithm mentioned by Kanellakis and Smolka. That is, in the initial partition all states are in the same block and in every refinement step the next partition distinguishes everything that can be distinguished with respect to the previous partition. This is different from the

Kanellakis-Smolka and Paige-Tarjan algorithms, which in each iteration will select two blocks and a label and then refine the first block with respect to possible transitions to the second block, having the selected label.

In our implementations, a unique ID (an integer) is assigned to each block and partitions are represented as arrays of IDs indexed by states. The signature of a state x with respect to a partition is a set of pairs of labels and IDs, such that a pair (a, id) is in the set if and only if there is a transition with the label a from the state x to another state belonging to the block with the ID id . Two states are distinguishable with respect to a partition if they have different signatures with respect to that partition.

The straightforward implementation, presented in Sections 3.2 and 3.3, computes the signatures of all states in every iteration and randomly assigns IDs to each signature. It terminates if the number of signatures becomes stable.

The optimized implementation, discussed in Sections 3.4 and 3.5, does not recompute the signatures on each iteration. Instead, it modifies the old signatures. While this recomputation goes on, the states with modified signatures are marked. Next, we assign new IDs to the signatures of marked states as follows: if some of the states in a block are unmarked then the signatures of the marked states all get new IDs; if all states in a block are marked then the old ID is reused for the signature which occurs most often and new IDs are assigned to the others. The algorithm terminates if there are no more changes. By assigning the old ID to the most often occurring signature, we minimize the number of signatures which must be recomputed in the next iteration. Note that this is similar to the strategy used in the Paige-Tarjan algorithm, which always splits with respect to the smallest block.

Why Kanellakis-Smolka and not Paige-Tarjan? As starting point for our distributed algorithms, we chose a very simple bisimulation reduction, called “the naive method” by Kanellakis and Smolka [KS83]. From the point of view of theoretical complexity, it cannot compete with the Paige-Tarjan algorithm ($\mathcal{O}(MN + N^2)$ vs. $\mathcal{O}(M \log N)$), but in practice it performs quite well. Moreover, the Paige-Tarjan algorithm cannot easily be extended to branching bisimulation and weak bisimulation. For the naive algorithm this is feasible (see Chapter 3).

Both the Paige-Tarjan algorithm and the naive algorithm use iterations. The Paige-Tarjan algorithm in each iteration carefully selects a number of states to work on. The naive algorithm works on all states independently. The data structures needed to make the selection cost a lot of memory in any case and require modification to allow an efficient distributed implementation. In contrast, the naive algorithm has a natural parallel implementation. For both algorithms the worst case number of iterations is the number of states. However, in practice the Paige-Tarjan algorithm needs many more iterations than the naive algorithm.

Another practically relevant difference is that Paige-Tarjan is efficient for unlabeled

transition systems. Adapting it to work on LTSs is at the cost of accepting extra iterations, while the naive algorithm exploits the different labels, by multi-way splitting, precisely to reduce the number of iterations.

3.1 The relational coarsest partition problem

For the definitions of LTS and strong bisimulation equivalence, we refer to the preliminaries chapter, Section 2.3.1.

The problem that we focus on, *bisimulation minimization*, is to find the equivalence classes of the largest strong bisimulation on the states of a given LTS. Or, in other words, given an LTS, find the LTS that is strongly bisimilar to it and has the minimal number of states.

A related problem is that of *bisimilarity checking*: given an LTS $\mathcal{S} = (S, T, s_0)$ and two states $p, q \in S$, decide whether p and q are strongly bisimilar. This problem reduces to the minimization problem, since it suffices to check whether p and q are in the same equivalence class of the largest bisimulation relation definable on the states of \mathcal{S} . The way of deciding whether two transition systems represent the same behavior is to apply a bisimulation minimization algorithm to the compound LTS $(S^1 \cup S^2, T^1 \cup T^2, s_0^1)$ and see whether s_0^1 and s_0^2 end up in the same class.

For an LTS (S, T, s_0) , a *partition* of the elements of S is a set of disjoint blocks $\{B_i \mid i \in I\}$ s.t. $\cup_{i \in I} B_i = S$. An equivalence relation can be represented as a partition with a block for every equivalence class. A partition π' is a refinement of π if every block of π' is contained in a block of π : $\forall C \in \pi' : \exists B \in \pi : C \subseteq B$.

The bisimilarity minimization problem is equivalent to the *relational coarsest partition problem* which is to find, for a given LTS and a given initial partition π^0 of S , a partition π s.t.:

1. π is a refinement of π^0
2. $\forall p, q \in B \in \pi : \forall a \in \text{Act} : \forall B' \in \pi : \exists p' \in B' : (p, a, p') \in T \text{ iff } \exists q' \in B' (q, a, q') \in T$
3. π has the fewest blocks among all partitions satisfying 1 and 2.

The algorithms discussed in this chapter solve the bisimulation minimization problem by solving the Relational Coarsest Partition Problem with $\pi^0 = \{S\}$.

3.2 A naive sequential solution

In Figure 3.1 we have described an implementation of the naive algorithm, for a given LTS (S, T, s_0) . The idea is that signatures computed with respect to the current partition determine the next partition. More precisely, the blocks of the new partition are sets of states with identical signatures. Keeping track of the current partition is

```

1 for  $x \in S$  do   ID( $x$ ) := 0 enddo
2 oldcount := 0; newcount := 1
3 while oldcount  $\neq$  newcount do
4     /* compute the signatures */
5     for  $x \in S$  do
6         sig( $x$ ) :=  $\{(a, \text{ID}(y)) \mid x \xrightarrow{a} y\}$ 
7     enddo
8     for  $x \in S$  do
9         insert sig( $x$ ) in HashTable
10        and get new value for ID( $x$ )
11    enddo
12    /* count the blocks of the new partition */
13    oldcount := newcount
14    newcount := card( $\{\text{ID}(x) \mid x \in S\}$ )
15 enddo
16 return ID

```

Figure 3.1: (SSN). Single threaded implementation of the naive algorithm

done by assigning every block an unique identifier (natural number). The function $\text{ID} : S \rightarrow \mathbf{N}$ indicates to which block every state belongs. Thus, the current partition is represented by the ID function. We define the *signature* of a state x with respect to it as

$$\text{sig}(x) = \{(a, \text{ID}(y)) \mid x \xrightarrow{a} y\}.$$

Since our signatures of interest are always computed with respect to the current partition, we will not index them; instead, we will make sure it is always clear what the current partition is. In steps 8-11, new values are assigned to ID, such that $\forall x, y \in S$:

the new value of $\text{ID}(x)$ = the new value of $\text{ID}(y)$
iff
 $\text{sig}(x)$ w.r.t. the old ID = $\text{sig}(y)$ w.r.t. the old ID.

A hash table `HashTable`, being a set of $\langle \text{oldsig}, \text{newID} \rangle$ pairs, is used for this purpose. The hash values are signatures. Let us denote by ID^f the ID returned by the algorithm; ID^f determines the final partition.

Note that, unlike the general partition refinement scheme, SSN (Sequential Strong bisimulation Naive algorithm) only computes the new signatures and does not explicitly replace blocks of the old partition with new blocks. The following lemma justifies that the partitions computed in this manner are indeed successive refinements, under the hypothesis that the initial partition is $\{S\}$:

Lemma 3.1 *For $n \geq 0$, denote ID^n , sig^n the ID and sig functions as they are before execution of step 8 of the n th iteration of SSN (first iteration has index 0). Then for every $n > 0$ and for every $x, y \in S$:*

$$\text{sig}^{n-1}(x) \neq \text{sig}^{n-1}(y) \implies \text{sig}^n(x) \neq \text{sig}^n(y).$$

Proof: Let us first examine the relation between the $ID^n (n \geq 0)$ and $\text{sig}^n (n \geq 0)$ series of functions. Due to step 6, for every $x \in S$ and every $n \geq 0$,

$$\text{sig}^n(x) = \{(a, ID^n(y)) \mid x \xrightarrow{a} y\} \quad (3.1)$$

Also, steps 8-11 of the $(n-1)^{th}$ iteration ($\forall n \geq 0$) take care that, for every $x, y \in S$,

$$ID^n(x) = ID^n(y) \text{ iff } \text{sig}^{n-1}(x) = \text{sig}^{n-1}(y). \quad (3.2)$$

We prove the lemma by induction on n . The initial partition is $\{S\}$, which means that $\forall x, y \in S : \text{sig}^0(x) = \text{sig}^0(y)$. Therefore the claim is true for $n = 1$. Let $n > 1$ and suppose there exists a pair of states $x, y \in S$ for which

$$\text{sig}^{n-1}(x) \neq \text{sig}^{n-1}(y) \quad (3.3)$$

$$\text{and } \text{sig}^n(x) = \text{sig}^n(y) \quad (3.4)$$

Then (w.l.o.g.) there must exist a transition $x \xrightarrow{a} z$ ($a \in \text{Act}$, $z \in S$) such that $(a, ID^{n-1}(z)) \notin \text{sig}^{n-1}(y)$, meaning that

$$\nexists t \in S : (y \xrightarrow{a} t \wedge ID^{n-1}(t) = ID^{n-1}(z)) \quad (3.5)$$

The pair $(a, ID^{n-1}(z))$ occurs in $\text{sig}^n(x)$ (3.1). From 3.4 it then follows that there exist a state $v \in S$ such that $y \xrightarrow{a} v$ and $ID^n(v) = ID^n(z)$. From this last equality we derive (3.2) that $\text{sig}^{n-1}(v) = \text{sig}^{n-1}(z)$ and, from the induction hypothesis, $\text{sig}^{n-2}(v) = \text{sig}^{n-2}(z)$. But that also means, applying 3.2 once more, that $ID^{n-1}(v) = ID^{n-1}(z)$, which gives a contradiction with 3.5. \square

3.3 ...and its distributed implementation

The obvious way to distribute a partition refinement algorithm is to distribute the data and keep the control flow centralized. More precisely, the workers perform iterations in which they independently do some refinement and then synchronize the results. This approach is fine in theory, but in practice it turns out that synchronization can take a lot of time. This is another reason to choose the naive algorithm: typically it needs far

```

1 read  $\text{In}_{ji}(\forall j)$ , read  $\text{Out}_{ij}(\forall j)$ 
2  $\text{newcount} := 1$ 
3 for  $\text{ever}$  do
4   /* phase 1: compute signatures */
5   for  $x \in S_i$  do
6      $\text{sig}(x) := \{(a, p) \mid \langle x, a, p \rangle \in \text{Out}_{ij}, 0 \leq j < W\}$ 
7   enddo
8   /* phase 2: compute new IDs */
9    $N_{\text{expected\_answers}} := 0$ 
10   $\text{Send\_Signatures} \parallel \text{Handle\_Messages}$ 
11  /* decide whether this is the last iteration */
12   $\text{oldcount} := \text{newcount}$ 
13   $\text{DBSUM}(\text{newcount}_i, \text{newcount})$ 
14  if  $(\text{oldcount} = \text{newcount})$  break fi
15  /* phase 3: update ID */
16   $\text{Update\_IDs}$ 
17 enddo
18 return ID

```

Figure 3.2: (DSN) Distributed version of SSN (WORKER_i)

```

1 for  $x \in S_i$  do
2    $\text{SEND } \triangleleft \text{hash\_insert} : i, \text{sig}(x) \triangleright \text{TO worker}(\text{hash}(\text{sig}(x)))$ 
3    $N_{\text{expected\_answers}} := N_{\text{expected\_answers}} + 1$ 
4 enddo
5 for  $j : 0 \leq j < W$  do  $\text{SEND } \triangleleft \text{endsig} : \triangleright \text{TO } j$  enddo

```

Figure 3.3: The Send_Signatures routine of WORKER_i

less iterations than Kanellakis-Smolka and Paige-Tarjan, thus fewer synchronizations.

3.3.1 Description

Our distributed reduction algorithm (Figure 3.2) is based on the sequential one (Figure 3.1). The states of the input LTS are evenly divided over the W workers. Worker i is in charge of the set of states S_i . Every iteration, it has to compute the signatures of states in S_i and keep track of the ID of these signatures. It is also responsible for the administration of a part of the hash table used at step 10 (step 8-11 in SSN). We denote i 's part by HashTable_i .

Let T_{ij} be the indexed list of transitions having the source state in S_i and the destination state in S_j . About a transition in T_{ij} , worker i needs to know its source state, its label and the current ID of its destination state, in order to be able to compute the source state's signature. It is worker j 's job to keep i informed about the current ID of the destination state. Therefore, the T_{ij} -concerned knowledge needed

```

1  $N\_active\_workers := W$ 
2 while  $N\_active\_workers > 0 \vee N\_expected\_answers > 0$ 
3   do
4     RECEIVE  $msg$ 
5     case  $msg$ 
6        $\triangleleft hash\_insert : j, s \triangleright$ 
7         insert  $s$  in  $HashTable_i$  and obtain  $ID(s)$ 
8         SEND  $\triangleleft hash\_ID : s, ID(s) \triangleright$  TO  $j$ 
9        $\triangleleft endsig : \triangleright$ 
10         $N\_active\_workers := N\_active\_workers - 1$ 
11        $\triangleleft hash\_ID : s, sid \triangleright$ 
12         $ID(s) := sid$ 
13        $N\_expected\_answers := N\_expected\_answers - 1$ 
14     endcase
15 enddo
16  $newcount_i := card(HashTable_i)$ 

```

Figure 3.4: The *Handle_Messages* routine of $WORKER_i$

```

1 for  $j : 0 \leq j < W$  do
2   SEND  $\triangleleft update : i, [ID(y) \mid y \in In_{ji}] \triangleright$  TO  $j$ 
3 enddo
4  $received := 0$ 
5 while  $received < W$  do
6   RECEIVE  $\triangleleft update : w, IDList \triangleright$ 
7    $received := received + 1$ 
8   update  $Out_{iw}$  with  $IDList$ 
9 enddo

```

Figure 3.5: The *Update_IDs* routine of $WORKER_i$

and maintained by workers i and j is captured in two lists ordered by the same index as T_{ij} :

$$\begin{aligned}
Out_{ij} &= [\langle x, a, ID(y) \rangle \mid x \xrightarrow{a} y \wedge x \in S_i \wedge y \in S_j], \text{ residing in the memory of } i \\
In_{ij} &= [y \mid x \xrightarrow{a} y \wedge x \in S_i \wedge y \in S_j], \text{ in the memory of } j
\end{aligned}$$

Note that elements can occur repeatedly in a list – for instance, a state shows up twice in In_{ij} if it is the destination of two transitions coming from states on i . In_{ij} and the first two fields of the elements from Out_{ij} represent static information about the structure of the LTS. The data that changes throughout the run of the algorithm are the functions sig and ID . Furthermore, workers need to know the number of different signatures of the states in S in the current and in the previous iteration, in order to decide whether the final partition has been reached. As in the sequential case, denote ID^f the final assignment for ID , returned at the end of the algorithm.

The LTS is provided to the workers in the form of the lists In_{ij} and Out_{ij} , $\forall i, j$, the latter using a constant initial ID function: $\text{ID}(x) = 0$ that reflects the initial partition $\{S\}$. Each iteration of DSN (Distributed Strong bisimulation Naive algorithm) consists of three phases:

1. signature computation (steps 5-7),
2. computing globally unique IDs for signatures (step 10) and
3. exchanging ID information. (step 16)

In the first phase (5-7) of every iteration, each worker computes the signatures of its own states.

In the second phase (10, detailed in Figure 3.3 and 3.4), all signatures are inserted in the distributed hash table and are assigned unique IDs. The insertion is based on a hash function $\text{hash} : 2^{\text{Act} \times \mathbf{N}} \rightarrow \mathbf{N}$ and the distribution of the hash table is done by a function $\text{worker} : \mathbf{N} \rightarrow \{0 \dots W - 1\}$. We assume that worker is capable of ensuring a balanced load of signatures on workers. WORKER_i runs two threads. One is busy with sending each signature to the worker responsible for the part of the hash table where it should be inserted (determined using worker). When all signatures are sent, an endsig message is sent to all workers, to mark the end of the stream. The other thread handles the incoming messages. A request for inserting a signature in the local hash table (hash.insert) is handled by looking up the signature and fetching its ID, or, if not found, adding it to the table and assigning it a new ID. The ID is then returned to the owner of the signature. When receiving an answer (hash.ID) to a request sent earlier by Send_Signatures , Handle_Messages fills in the new ID value and decreases the counter of expected answers. Finally, on receiving an end-of-stream message (endsig), it decreases the counter of workers that might still send hash.insert requests. The Handle_Messages thread terminates when all workers announced that they have no more signatures to send to i and all i 's requests have been answered.

After the second phase, we compute how many different signatures there are now (steps 12-13). If the number of signatures did not increase w.r.t. the previous iteration, the stable partition has been reached and the computation must stop (14).

In the third phase (16, shown in detail in Figure 3.5), the lists Out_{ij} are updated. For every transition of the LTS, the new ID of the destination state's signature is sent to the owner of the source state. More precisely, every worker j sends $\text{ID}(\text{In}_{ij})$ to worker i , who will substitute this information on the last fields of its Out_{ij} . This happens correctly due to the fact that the lists In_{ij} and Out_{ij} have the same index.

At the end of the **loop**, the IDs are the states of the reduced LTS and its set of transitions is $\cup_{i,j} \{ \langle \text{ID}(x), a, p \rangle \mid \langle x, a, p \rangle \in \text{Out}_{ij} \}$. They can be dumped independently by the workers, possibly after renumbering the IDs to consecutive numbers.

3.3.2 Correctness and complexity

We now justify that the algorithm described above is correct, i.e. it terminates and it produces the minimal LTS bisimilar to the input LTS. We also give an analysis of its performance in terms of time, memory and number of messages needed during the computation.

Theorem 3.2 (*correctness*) *Let $\mathcal{S} \equiv (S, T, s_0)$ be an LTS. Then DSN applied to any distribution of \mathcal{S} terminates and the resulting ID^f satisfies:*

$$(\{ID^f(x) \mid x \in S\}, \{\langle ID^f(x), a, ID^f(y) \rangle \mid \langle x, a, y \rangle \in T\}, ID^f(s_0))$$

is the minimal LTS bisimilar to \mathcal{S} .

Proof: We first argue that every iteration (steps 5-16) of DSN terminates. For this, we take a closer look at the steps involving communication (10,12-13 and 16). The first thread of step 10 obviously terminates, since it only executes a finite number of **SEND TO** calls (that are always successful).

Handle_Messages's exit condition

$$N_active_workers = 0 \wedge N_expected_answers = 0$$

will eventually be satisfied. $N_active_workers$ becomes 0 when W **hash.ID** messages sent to i will have been received. Note that $N_active_workers$ being 0 is a sign that all **hash.insert** messages directed to i have been received, and also that all **hash.insert** messages, originating from all workers, including i itself, have been sent. In particular, this means that when $N_active_workers$ of i is 0, $N_expected_answers$ of i will not increase anymore. This property rules out the undesired situation that the exit condition is fulfilled while messages for i are still pending. The termination of *Update.IDs* is justified mainly by the fixed number of messages exchanged. Every worker successfully sends exactly W messages (these messages can be very large, but this is not a problem, since we assumed unbounded channels, see Section 2.2.5), then picks up from the network the W messages addressed to it.

It can be easily proved by induction that DSN mimics faithfully the sequential version SSN, depicted in Figure 3.1. That is, formally: for any r , if we consider $ID_{seq} = SSN$'s ID, after step 11 of the r th iteration and $ID_{db}^i = WORKER_i$'s ID, after step 10 of the r th iteration ($\forall i$), then

$$\forall i, j \forall x \in S_i, y \in S_j \quad ID_{db}^i(x) = ID_{db}^j(y) \text{ iff } ID_{seq}(x) = ID_{seq}(y).$$

From this and from the fact that the exit condition from DSN and SSN are identical, it follows that the loop 3-17 of DSN eventually terminates. Moreover, the LTS determined by the ID^f values is exactly the one found by SSN, thus the solution of our

problem. □

To evaluate the performance of DSN, we use the classical time/message complexity measures for distributed algorithms, as defined in [AW98] and recalled in Section 2.2.3.

Theorem 3.3 (*complexity*) *Worst-case time complexity of DSN is $\mathcal{O}(\frac{MN+N^2}{W})$ and message complexity is $\mathcal{O}(N^2)$.*

Proof: For computing the signatures, every state has to be considered and we assumed that the cost per state is linear in the number of outgoing transitions of that state. As workers do this computation independently and we assumed even distribution of states, the time needed is $\mathcal{O}(\frac{M+N}{W})$.

The number of signatures each worker has to insert into the global hash table is at most the number of states it processes: $\lceil \frac{N}{W} \rceil$. Assuming that **worker** is a perfect hash function, each worker has to send $\lceil \frac{N}{W} \rceil$ signatures to every other worker. Every worker therefore receives at most $W \cdot \lceil \frac{N}{W} \rceil$ signatures. (The insertion in the local hash table takes constant time, as well as the computation of a new ID.) The same amount of replies must be sent back. Thus, the cost of computing globally unique identifiers for signatures is $\mathcal{O}(W \cdot \lceil \frac{N}{W} \rceil)$. Under the assumption that $W \ll N$, we can forget about rounding upwards and we evaluate the cost to $\mathcal{O}(\frac{N}{W})$.

To decide termination, we need to compute the total number of different signatures. The cost of this operation is W .

To exchange the new IDs, every worker has to prepare W buffers of total size $\mathcal{O}(\frac{M}{W})$, representing the total number of incoming transitions (see Section 2.2.5). It also has to receive and process W such buffers, from workers that are in charge of successor states.

Summing up, the cost of an iteration is $\mathcal{O}(\frac{M+N}{W})$. Since as many as N iterations might be needed, the worst case time complexity is $\mathcal{O}(\frac{MN+N^2}{W})$.

The message complexity is given by the total number of messages sent by all workers in the whole run of the algorithm. In the worst case, exchanging signatures takes N messages (if every signature has to be sent to another worker), and the update phase W^2 messages. Synchronizing at steps 12-13 takes always W messages. This results in at most $N(N + W + W^2)$ messages over the whole run, that is $\mathcal{O}(N^2 + NW^2)$. Since W is insignificant compared to N , we may conclude a message complexity of $\mathcal{O}(N^2)$. □

The number of iterations is the most important factor in the performance of the algorithm. The worst case is that the number of iterations is the number of states. An example that has this worst case behavior is an LTS whose state are the numbers

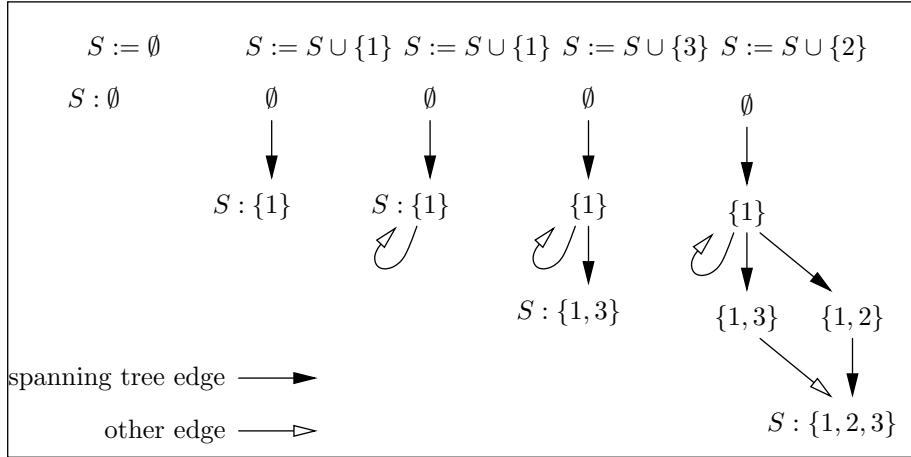


Figure 3.6: Evolution of a set data structure

$0..N$ and the transitions are $i \rightarrow i + 1 (i = 0..(N - 1))$. However, such a long series of events is not typical in state spaces. The typical phenomenon in state spaces is state space explosion: the system would consist of P processes each having N states that run in parallel. The size of the state space would then be N^P , which is a huge number for relatively small N and P . However, if the processes are completely independent then the reduction algorithm needs at most $N + 1$ iterations. Of course neither the long thread nor the complete independence of processes occurs in practice, but they give some intuition about the worst case and why it is unlikely.

The memory needed by one worker can be estimated as follows: $\mathcal{O}(\frac{M+N}{W})$ for the signature information, $\mathcal{O}(\frac{M}{W})$ for the (destination ID, destination state) of incoming transitions, $\mathcal{O}(\frac{M}{W})$ for the (source state, label, destination ID) of outgoing transitions. In total, $\mathcal{O}(\frac{M+N}{W})$, which is the best that can be achieved, W times less than the space used by the single-threaded implementation.

3.3.3 Implementation details

The distributed prototype implements the algorithm in Figure 3.2. In the actual MPI implementation, messages are not sent one by one, but buffered into larger messages. For the update phase (step 16), we issue all the **SEND TO** messages and post **RECEIVE** requests, then wait for all **RECEIVE** requests to be completed. The two threads from step 10 are implemented with explicit interleaving.

For computing signatures we have used a set datatype on which it is easy to add a single element and decide equality. The idea is to maintain a directed graph, whose vertexes are sets and whose labeled edges are an 'obtained by insertion' relation. That is, an edge $S \xrightarrow{e} S'$ is only allowed if $S' = S \cup \{e\}$. In order to efficiently decide if

a certain set is present or not, we maintain the graph in such a way that the edges $S \setminus \{max(S)\} \xrightarrow{max(S)} S$ form a spanning tree. By doing this the set corresponding to an ordered list can be found by starting in the empty set and then following the edges corresponding to the elements in the list. There may be other paths from the empty set to the same set, but if a set exists then this path exists. In Figure 3.6, we have drawn the data structure as it would look when starting with an empty set and adding the elements 1, 1, 3 and 2 in that order. Notice that adding 1 twice creates a cycle in the graph.

On this data structure, we can decide equality of sets in constant time. The complexity of inserting a single element into a set is linear in the size of the set the first time and constant afterwards. (The first time we have to create one or more edges and 0 or more vertexes, afterwards we can find the edge in constant time using a hash table.)

Using this structure it is very easy to write code that computes signatures. However, the order in which the signatures are built matters for the performance of the algorithm. If a set is built in the same order every time then quadratic time is needed for the first build and linear time for every rebuild. The danger comes from the fact that quadratic time and memory may be used for every different order in which the signature is built. This means that to obtain decent performance, we have to sort the transitions ensuring that the amount of different orders is minimal.

3.4 An optimized sequential solution

In the previous two sections, an algorithm was presented that uses the set of all outgoing transitions (signatures) as criteria to distinguish states, as opposed to theoretically more efficient algorithms that check the states of the same block against certain other blocks. The main advantage of the new signatures approach is that it admits a natural distributed implementation.

See the sequential version of this algorithm in Figure 3.1. In that scheme, all signatures are recomputed in every iteration, which can be an unnecessary and costly effort in the case of large input LTSs with a structure that needs a lot of iterations to stabilize and where very few partition blocks can be split per iteration (very few signatures actually change).

The main idea of our second approach, which we will refer to as “optimized”, is to mark, in every iteration, those states that might have suffered a signature change, i.e. the states that have an outgoing transition to a state whose ID changed in the current iteration. (As before, we indicate the current partition by a function $ID : S \rightarrow \mathbf{N}$ that assigns block identifiers to states.) In the next iteration, only the signatures of the marked states need to be recomputed. We will refer to the marked states as *unstable*. Note that, unlike other algorithms, that mark whole blocks as unstable, we insist on reasoning about unstable *states* and not assuming that the states belonging

```

1  E := 1; c(0) := card(S); U := S
2  for x ∈ S do ID(x) := 0; sig(x) := ∅ enddo
3  while U ≠ ∅ do
4      for x ∈ U do sig(x) := {(a, ID(y)) | x  $\xrightarrow{a}$  y} enddo
5      Reusable := {i | 0 ≤ i < E ∧ c(i) = card(U ∩ {x | ID(x) = i})}
6      ST := ∅; νU := ∅
7      for x ∈ U do
8          oid := ID(x)
9          if (sig(x), i) ∈ ST
10             then ID(x) := i
11             else if ID(x) ∉ Reusable
12                 then
13                     c(E) := 0
14                     ID(x) := E
15                     E := E + 1
16                 else Reusable := Reusable − {ID(x)}
17             fi
18             ST := ST ∪ {(sig(x), ID(x))}
19             if oid ≠ ID(x)
20                 then
21                     νU := νU ∪ {y ∈ S | y  $\xrightarrow{a}$  x}
22                     c(oid) := c(oid) − 1
23                     c(ID(x)) := c(ID(x)) + 1
24                 fi
25             fi
26         enddo
27         U := νU
28     enddo
29     for x ∈ S IDf(x) := ID(x)

```

Figure 3.7: (SSO) The optimized algorithm

to the same block are easily retrievable. Extra attention has to be paid to ensure the correctness of the splitting procedure, but it pays off, since the ability to work directly on states provides parallel/distributed workers with a high(er) degree of independence.

The optimized algorithm, **SSO** (Sequential Strong bisimulation Optimized algorithm), is presented in Figure 3.7 and uses the following notations and data structures:

- $U, \nu U$ - the set of unstable states for the current and the next iteration, respectively
- E - the number of blocks in the current partition. Throughout the algorithm, the invariant is kept that the blocks of the current partition are numbered $\{0 \dots E-1\}$.
- $c : \{0 \dots E-1\} \rightarrow \mathbf{N}$ - the number of states in each block
- **Reusable** - the set of block identifiers that can be reused in the next iteration, since all the states belonging to those blocks are marked unstable. These identifiers

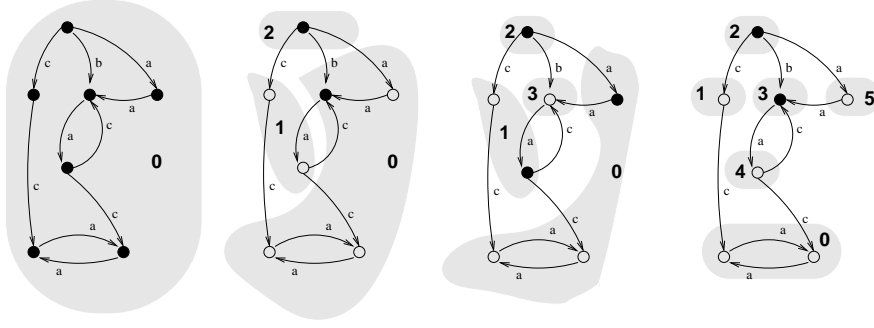


Figure 3.8: A refinement example. The filled circles represent the unstable states at the beginning of each iteration.

should be reused in order to preserve the above mentioned invariant. Moreover, the identifier of every block should be reused for one of its own sub-blocks, to ensure termination of the iterations series.

– ST - a signatures hashtable used to map the signatures of the current iteration to new IDs (the block identifiers of the next iteration).

ID^f is the final partition, the blocks of which represent the states of the minimized LTS. The termination and correctness of SSO follow from a few properties listed below.

Lemma 3.4 *Let \mathcal{U}^n , sig^n , ID^n , E^n denote the set of unstable states, the signatures mapping, the ID mapping, and the number of equivalence classes at the beginning of the n -th iteration of the optimized algorithm (i.e. before the n -th execution of step 3) – the count starts at 0. The following properties hold, for any $n \geq 0$:*

1. $(\forall x \in S) ID^n(x) < E^n$.
2. $(\forall i : 0 \leq i < E^{n-1}) \exists x \in S \text{ s.t. } ID^n(x) = ID^{n-1}(x) = i.$
 $(\forall i : 0 \leq i < E^n) \exists x \in S \text{ s.t. } ID^n(x) = i.$
3. $(\forall x \in S ID^n(x) = ID^{n-1}(x)) \text{ iff } \mathcal{U}^n = \emptyset$
4. $(\forall x, y \in S)$
 $ID^n(x) = ID^n(y) \text{ iff } \text{sig}^n(x) = \text{sig}^n(y) \text{ and}$
 $\text{sig}^n(x) \neq \text{sig}^n(y) =: \text{sig}^{n+1}(x) \neq \text{sig}^{n+1}(y).$
5. $E^{n-1} \leq E^n.$
 $E^n = E^{n-1} \text{ iff } (\forall x \in S ID^n(x) = ID^{n-1}(x)).$

Proof:

The properties 1, 3 and 4 will be proved independently, by induction on n . Property 2 relies on 4 and property 5 relies on 1 and 2.

1. $(\forall x \in S) ID^0(x) = 0 < 1 = E^0$. In every iteration, the only place where fresh values are introduced for ID is step 14 (in step 10 an old value is used). But E is also immediately increased (step 15), therefore the invariant stays true.

2. For $n = 0$ the property is obviously true, since $ID^0(x) = 0$ for all states x . Suppose it is true for E^{n-1} , ID^{n-1} and let us look at how E^n and ID^n are computed, in the $n - 1$ th iteration. First, the set **Reusable** is constructed (step 5), containing the identifiers of the blocks whose states are *all* marked unstable. Let i be any identifier $0 \leq i < E^n$. We distinguish three cases:

- $i \in \text{Reusable}$. Then all states x with $ID^{n-1}(x) = i$ (according to the induction hypothesis, there is at least one) must be in \mathcal{U}^{n-1} . Let y be the first of these states that is handled in the step 7 of the algorithm. $\text{sig}^n(y)$ cannot be already in **ST**, since this would mean that there exist a state z from another block ($ID^{n-1}(z) \neq ID^{n-1}(y)$) with the same signature ($\text{sig}^{n-1}(z) = \text{sig}^{n-1}(y)$), which contradicts point 4 of this lemma. Therefore, y will only be affected by steps 16 and 18, that do not modify the value of ID. Thus, $ID^n(y) = ID^{n-1}(y) = i$.

- $i \notin \text{Reusable} \wedge i < E^{n-1}$. Then there must be a state x for which $ID^{n-1}(x) = i$ and $x \notin \mathcal{U}^{n-1}$. It follows, since the steps sequence 7-26 does not regard x , that $ID^n(x) = ID^{n-1}(x) = i$.

- $E^{n-1} \leq i < E^n$. This means that i is “created” in the steps 13-15 as identifier for a new block. In step 14 the ID^n of the first state of this block is explicitly defined as being i .

3. Let us consider an iteration n that satisfies $\forall x \in S ID^n(x) = ID^{n-1}(x)$. This means that in the iteration $n - 1$, the condition in the step 19, that compares exactly $ID^n(x)$ and $ID^{n-1}(x)$ was never satisfied, thus $\nu\mathcal{U}$ remains \emptyset , that is $\mathcal{U}^n = \emptyset$. The inverse is also true: if $\mathcal{U}^n = \emptyset$ then $\nu\mathcal{U}$ ended up empty in the previous iteration. This could only happen if the condition on line 19 was never met, that is the value of ID was not changed for any state. Formally, $\forall x \in S ID^n(x) = ID^{n-1}(x)$.

4. We prove this by induction on $n \geq 0$. The case $n = 0$ follows from the fact that $(\forall x) \text{sig}^0(x) = 0$ and $(\forall x) ID^0(x) = 0$. To prove the first half of the invariant for an arbitrary n , we consider three cases:

- $x, y \in \mathcal{U}^{n-1}$. In this case, both $\text{sig}^n(x)$ and $\text{sig}^n(y)$ are inserted in the hashtable **ST**, which ensures the same ID value for (and only for) equal signatures.

- $x, y \notin \mathcal{U}^{n-1}$. Then the sigs and IDs do not change, i.e. $\text{sig}^n(x) = \text{sig}^{n-1}(x)$, $ID^n(x) = ID^{n-1}(x)$ and $\text{sig}^n(y) = \text{sig}^{n-1}(y)$, $ID^n(y) = ID^{n-1}(y)$. From the induction hypothesis, it follows that $\text{sig}^n(x) = \text{sig}^n(y)$ iff $ID^n(x) = ID^n(y)$.

- $x \in \mathcal{U}^{n-1}$ and $y \notin \mathcal{U}^{n-1}$. Then there must be a state z that has caused the instability of x , i.e. there is a transition $x \xrightarrow{a} z$ with $ID^{n-1}(z) \neq ID^{n-2}(z)$. Then $ID^{n-1}(z) = i \geq E^{n-2}$, therefore the pair (a, i) cannot be in $\text{sig}^{n-1}(y)$. And since $\text{sig}^n(x)$ is recomputed and $\text{sig}^n(y)$ not, it follows that $\text{sig}^n(x) \neq \text{sig}^n(y)$. It remains to

prove that $ID^n(x) \neq ID^n(y)$ as well. Let us first notice that

$$\text{if } ID^n(x) \neq ID^{n-1}(x) \text{ then } ID^n(x) \geq E^{n-1}. \quad (3.6)$$

If $\text{sig}^{n-1}(x) = \text{sig}^{n-1}(y) = i$, then $i \notin \text{Reusable}$ (since $y \notin \mathcal{U}^{n-1}$), thus $ID^n(x) \neq i$, thus (3.6) $ID^n(x) \geq E^{n-1}$, while $ID^n(y) = ID^{n-1}(y) < E^{n-1}$. If, on the contrary, $\text{sig}^{n-1}(x) \neq \text{sig}^{n-1}(y)$, then $ID^{n-1}(x) \neq ID^{n-1}(y)$ (induction hypothesis). $ID^n(x)$ is computed in the fragment 7-26 and the outcome can be $ID^n(x) = ID^{n-1}(x) \neq ID^{n-1}(y) = ID^n(y)$ or $ID^n(x) \neq ID^{n-1}(x)$. In the latter case, $ID^n(x) \geq E^{n-1}$ (3.6), while $ID^n(y) = ID^{n-1}(y) < E^{n-1}$.

And now we prove the second half of the property. Let x and y be two states for which $\text{sig}^n(x) \neq \text{sig}^n(y)$. Then (w.l.o.g.) there is some pair $(a, ID^{n-1}(z)) \in \text{sig}^n(x)$ and $\notin \text{sig}^n(y)$. If $\text{sig}^n(y)$ does not contain any pair (a, j) then clearly $\text{sig}^{n+1}(x) \neq \text{sig}^{n+1}(y)$. Otherwise, let $y \xrightarrow{a} t$ be any of the a -transitions from y . Then $(a, ID^{n-1}(t)) \in \text{sig}^n(y)$ and $ID^{n-1}(t) \neq ID^{n-1}(z)$, which means (induction hypothesis) that $\text{sig}^{n-1}(t) \neq \text{sig}^{n-1}(z)$ and, further, $\text{sig}^n(t) \neq \text{sig}^n(z)$. Above we have proved that this is equivalent to $ID^n(z) \neq ID^n(t)$. Thus, $\text{sig}^{n+1}(x)$ contains the pair $(a, ID^n(z))$ and $\text{sig}^{n+1}(y)$ does not.

5. From the points 1 and 2 of this lemma it follows that $(\forall n) E^n$ is exactly the number of different values for ID^n . Therefore, if $\forall x \in S ID^n(x) = ID^{n-1}(x)$ then obviously $E^n = E^{n-1}$.

We will now prove the inverse statement. Let n be so that $E^n = E^{n-1}$ and suppose there exist an $x \in S$ with $ID^n(x) = i \neq ID^{n-1}(x)$. The property 2 says that there exists $y \in S$ such that $ID^n(y) = ID^{n-1}(y) = i$. But this would mean $ID^n(x) = ID^n(y)$ and $ID^{n-1}(x) \neq ID^{n-1}(y)$, which comes in contradiction with property 4. \square

Theorem 3.5 (*termination and correctness of SSO*)

For any LTS (S, T, s_0) , SSO terminates and the equivalence relation \approx determined by ID^f ($x \approx y$ iff $ID^f(x) = ID^f(y)$) is the largest strong bisimulation on S .

Proof: It is easy to see that for any iteration $n > 0$, $E^n \geq E^{n-1}$. It is also clear that $E^n > E^{n-1}$ can happen only finitely often, since from the points 1,2 of Lemma 3.4 follows that $\forall n E^n \leq \text{card}(S)$. Hence eventually $E^n = E^{n-1}$ and then the algorithm stops (3,5 of Lemma 3.4 and the exit condition of the loop at step 3). This proves termination.

We will now justify that \approx is a strong bisimulation on S . Let $last$ be the index of the last iteration, that is $ID^f := ID^{last}$. Let x and y be any two equivalent states and let $x \xrightarrow{a} z$ be any transition from x . To prove that \approx is a strong bisimulation, we have to prove that there exists a transition $y \xrightarrow{a} t$ with $t \approx z$. From $ID^{last}(x) = ID^{last}(y)$ and property 4 of Lemma 3.4 it follows that $\text{sig}^{last}(x) = \text{sig}^{last}(y)$. Then, since

$(a, \text{ID}^{last-1}(z))$ must be in sig^{last} , there is a state t with $\text{ID}^{last-1}(t) = \text{ID}^{last-1}(z)$ and $(a, \text{ID}^{last-1}(t)) \in \text{sig}^{last}(y)$. But $last$ is the final iteration, thus $\mathcal{U}^{last} = \emptyset$, that implies (Lemma 3.4, property 3) $\text{ID}^{last}(t) = \text{ID}^{last-1}(t)$ and similarly for z . Thus, $\text{ID}^{last}(t) = \text{ID}^{last}(z)$, or in other words $t \approx z$.

Finally, to prove that \approx is the coarsest strong bisimulation, let \approx' be any other strong bisimulation and show that $\forall x, y \in S \ x \approx' y \Rightarrow x \approx y$. To this end, we prove by induction that $\forall n \geq 0 \ x \approx' y \Rightarrow \text{ID}^n(x) = \text{ID}^n(y)$. The base case $n = 0$ is immediate. Suppose the statement is true for $n - 1$ and let x, y be two states such that $x \approx' y$. Then $\forall x \xrightarrow{a} z \exists y \xrightarrow{a} t$ with $z \approx' t$, and thus also (induction hypothesis) $\text{ID}^{n-1}(z) = \text{ID}^{n-1}(t)$. According to the signature definition, this means that $\text{sig}^n(x) = \text{sig}^n(y)$. From property 4 of Lemma 3.4, $\text{ID}^n(x) = \text{ID}^n(y)$. \square

3.5 ...and its distributed implementation

3.5.1 Description

There are W workers, each consisting of two threads: a *segment manager*, that maintains a part (a segment) of the LTS and computes the signatures of the unstable states, and a *signatures server*, that maintains a part of the signature table ST and computes the new IDs. The data structures occurring in Figure 3.7 are distributed to the workers as follows:

- worker i , actually the segment manager i , is responsible for a subset S_i of S . $S_i \cap S_j = \emptyset, \forall i \neq j$ and $\bigcup_i S_i = S$. The function $SM : S \rightarrow \{0 \dots W - 1\}$ maps every state to its base segment manager.
- transition set T generates for every segment manager i the sets

$$\begin{aligned} \text{In}_i &= \{(x, a, y) \mid y \in S_i \wedge x \xrightarrow{a} y\} \\ \text{Out}_i &= \{(x, a, \text{ID}(y)) \mid x \in S_i \wedge x \xrightarrow{a} y\}, \end{aligned}$$

where ID identifies the current partition.

- the sets of unstable states \mathcal{U} , $\nu\mathcal{U}$ are maintained by managers in the form of $\mathcal{U}_i = \mathcal{U} \cap S_i$ and $\nu\mathcal{U}_i = \nu\mathcal{U} \cap S_i$, respectively.
- the set of block identifiers $\{0 \dots E - 1\}$ is divided into the disjoint sets $\text{IDSET}_0 \dots \text{IDSET}_{W-1}$ and distributed to the W signatures servers by a mapping $SS : \{0 \dots E - 1\} \rightarrow \{0 \dots W - 1\}$. Server j also maintains the part of the counts

```

SEGMENT_MANAGER i
1   $\nu\mathcal{U}_i := \emptyset$ 
2  for  $x \in \mathcal{U}_i$  do
3    compute  $\text{sig}(x)$ 
4    SEND  $\triangleleft \text{hash\_insert} : \text{ID}(x), \text{sig}(x), x \triangleright$  TO  $SS(\text{ID}(x))$ 
5  enddo
6  do loop
7    RECEIVE  $\text{msg}$ 
8    case  $\text{msg}$ 
9       $\triangleleft \text{hash\_ID} : x, i \triangleright$ 
10       for  $y : (y, a, x) \in \text{In}_i$  do
11         SEND  $\triangleleft \text{update} : y, a, \text{ID}(x), i \triangleright$  TO  $SM(y)$ 
12          $\text{ID}(x) := i$ 
13       enddo
14       $\triangleleft \text{update} : x, a, \text{oid}, i \triangleright$ 
15        $\text{Out}_i := \text{Out}_i - (x, a, \text{oid}) + (x, a, i)$ 
16        $\nu\mathcal{U}_i := \nu\mathcal{U}_i \cup \{x\}$ 
17    enddo
18   $\mathcal{U} := \nu\mathcal{U}$ 

SIGNATURES_SERVER i
1   $\text{ST}_i := \emptyset$ 
2  do loop
3    RECEIVE  $\triangleleft \text{hash\_insert} : \text{oid}, s, x \triangleright$ 
4    if  $(\text{oid}, s, Lx) \in \text{ST}_i$ 
5      then  $Lx := Lx + [x]$ 
6      else  $\text{ST}_i := \text{ST}_i \cup \{(\text{oid}, s, [x])\}$ 
7    fi
8     $\text{Reusable}_i := \{\text{oid} \in \text{IDSET}_i \mid c(\text{oid}) = \sum_{(\text{oid}, s, Lx) \in \text{ST}_i} \text{card}(Lx)\}$ 
9  enddo
10  decide on  $\nu\text{IDSET}_i$ 
11  for  $(\text{oid}, s, Lx) \in \text{ST}_i$  do
12    if  $\text{oid} \notin \text{Reusable}_i$ 
13      then take  $id$  from  $\nu\text{IDSET}_i$ 
14    else  $id := \text{oid}$ 
15    fi
16    for  $x \in Lx$  do
17      SEND  $\triangleleft \text{hash\_ID} : x, id \triangleright$  TO  $SM(x)$   $\triangleleft \text{hash\_ID} : x, id \triangleright$ 
18    enddo
19  enddo
20  re-balance  $c$ ,  $\nu\text{IDSET}$ 

```

Figure 3.9: (DSO) A distributed iteration

array c and of the signature table ST corresponding to $IDSET_j$:

$$ST_i = \{(oid, s, Lx) \mid SS(oid) = i \wedge Lx = [x \in S \mid ID(x) = oid \wedge sig(x) = s]\}$$

Here Lx is the list of all unstable states that have s as signature. Lx is necessary because unlike in the sequential implementation, in the distributed one it is not possible to generate the new ID at the moment of signature insertion.

The distributed algorithm executes, like the sequential one, a series of iterations. In between iterations, workers synchronize in order to decide whether the final partition has been reached. The computation inside an iteration is asynchronous and directed only by messages, as sketched in Figure 3.9 (DSO = Distributed Strong bisimulation Optimized algorithm). There are five phases distinguishable within an iteration:

- managers compute the signatures of the unstable states and send them (`hash_insert`) to the appropriate servers
- servers receive the signatures (`hash_insert`) and insert them in their local ST
- servers compute new ID s for the unstable states and send them (`hash_ID`) back to the managers
- managers receive the new ID s for their unstable states (`hash_ID`) and send messages to the parent states of its own states that changed the ID (`update`)
- managers receive and process the update messages (`update`)

In order not to overload the presentation, we leave out the simple mechanism that ensures that the loops $SS2-9$ and $SM6-17$ terminate. Since both these loops receive and treat messages from a fixed number of communication parties, it is enough to let these parties (the managers, in the case of the first loop; the servers in the second) signal when their stream of data has stopped and let the party executing the loop count the stop messages.

Due to the division of tasks between managers and servers, the first and the second phase happen in parallel (steps $SM2-5$, $SS2-9$ in Figure 3.9). Also the last three (steps $SM6-17$, $SS11-19$) are overlapped. The overlapping limits the amount of CPU idle time, by allowing computation and communication to proceed in parallel. For instance, the servers can already proceed with inserting signatures in the table while managers prepare and send more signature messages. In the actual runs of the program, a worker (manager + server) may use one processor. The main advantage of overlapping the phases is memory gain: since the consumers and producers of messages are active at the same time, the messages do not have to be stored. Thus, less memory is used.

3.5.2 Correctness and complexity

Lemma 3.6 *The following properties hold for this distributed algorithm:*

1. in every iteration, the signatures of the states in the same block are sent to the same server
2. every time a block splits, one of the new blocks gets the old id
3. in every iteration, finitely many `hash_insert` and `hash_ID` messages are generated
4. in every iteration, a received `hash_ID` message generates finitely many update messages.
5. $(\forall n > 0)$ if $IDdb^n$ is the state partition at the beginning of iteration n of DSO and $IDseq^n$ is the state partition at the beginning of iteration n of SSO, then

$$(\forall x, y \in S) IDdb^n(x) = IDdb^n(y) \text{ iff } IDseq^n(x) = IDseq^n(y)$$

Proof: 1. Indeed, if $ID(x) = ID(y) = i$, both $\text{sig}(x)$ and $\text{sig}(y)$ are sent (steps 2-5 in the segment manager) to the signature server responsible for i , $SS(i)$.

2. Consider a block with the identifier i . If there are states $x \in S_j - \mathcal{U}_j$ with $ID(x) = i$, then it is clear: all these states are not touched this iteration, i.e. they keep their old ID. If, on the contrary, all the states x with $ID(x) = i$ are in some unstable set $(\forall x \in S ID(x) = i \exists j x \in \mathcal{U}_j)$ then all signatures will be computed and sent to the same server (steps 2-5 in the segment manager). At the signature server side, all these signatures are inserted in ST_i and counted – and i is added to the Reusable_i set. Further, in $SS6-17$ when the first triple (i, s, Lx) is encountered, all the states in Lx get i as new ID.

3. The number of `hash_insert` and `hash_ID` messages is limited by the total size of the sets \mathcal{U}_i , i.e. by $\text{card}(S)$.

4. For each $\langle \text{hash_ID} : x, i \rangle$ message, $\text{card}(\text{In}_i)$ messages (that is $\leq \text{card}(S)$) with the tag `update` are sent.

5. By induction on n . □

Theorem 3.7 (termination and correctness of DSO) *For any LTS (S, T, s_0) , DSO terminates and the $IDdb^f$ function computed is the same as the ID^f computed by SSO.*

Proof: The properties (1), (2) from Lemma 3.6 ensure that the invariants from Lemma 3.4 are also true in the distributed implementation DSO. (3),(4) ensure that the computation within an iteration terminates. The global termination is justified by the one-to-one mapping between iterations in the sequential algorithm SSO and the iterations in the distributed implementation DSO (5). From (5) and the correctness

problem	original			minimized		
	states (in 10^6)	transitions (10^6)	disk space (<i>MB</i>)	states (10^6)	transitions (10^6)	number of iterations
CCP	0.21	0.68	15	0.077	0.24	66
1394-LL	0.37	0.64	15	0.034	0.076	73
lift5	2.2	8.7	101	0.032	0.14	86
CCP-2p3t	7.8	59	678	1.0	6.6	94
token ring	19.9	132	1513	8.4	51.1	6
lift6	33.9	165	1898	0.12	0.65	91
1394-LE	44.8	387	4430	1.1	7.7	51

Figure 3.10: Problem sizes

of SSO (Theorem 3.5) it follows that the partition computed is indeed the correct one. \square

3.6 Experiments

To experiment with the distributed prototype implementations, we used an 8 node dual CPU PC cluster and an SGI Origin 2000. The cluster nodes are dual AMD Athlon MP 1600+ machines with 2G memory each, running Linux and connected by both Fast Ethernet (bandwidth 100Mb/s) and Gigabit Ethernet (1Gb/s). The Origin 2000 is a ccNUMA machine with 32 CPUs and 64G of memory running IRIX, of which we used 1-16 MIPS R10000 processors. On the cluster, we used LAM/MPI 6.5.6 and on the SGI the native MPI implementation.

The case studies The test set consists of a variety of state spaces generated by case studies carried out recently with the μ CRL toolset and of the collection of anonymous LTSs VLTS (Very Large Transitions Systems) [CI]. The μ CRL case studies are mentioned below.

- *1394-LL* [Lut97] is a model of the Link Layer of the FireWire high speed serial data bus, used to connect computers and peripheral devices.
- *1394-LE* models the Leader Election protocol implemented within FireWire. The specification used here is instantiated with 17 nodes and is a variant of the specification in [SZ98].
- *CCP-2p3t* [PFHV03] is a cache coherence protocol model for distributed Java programs. We use the instance with 2 processes and 3 threads. *CCP* is an older (and smaller) version of it.
- *lift5*, *lift6* [GPW03] are models of a distributed system for lifting trucks with 5 and 6 legs.

problem	bcg_min		SSN		SSO	
	time (s)	mem (M)	time (s)	mem (M)	time (s)	mem (M)
CCP	15.0	18	21.3	20	4.5	18
1394-LL	18.5	19	6.2	14	3.3	21
lift5	113	184	64	123	43	214
CCP-2p3t			4363	968	779	1187

Figure 3.11: A comparison of single threaded tools

- *token ring* is the model of a Token Ring leader election protocol for 4 stations ¹. Problem sizes before and after reduction can be found in Figure 3.10.

3.6.1 A comparison of sequential tools

First, in order to validate the use of the naive algorithm, we have compared the memory usage and run times of our single threaded implementations with those of the `bcg_min` reduction tool, which is part of the CADP toolset [FGK⁺96].

The test results can be found in Figure 3.11. These tests were run on a PC running Linux with dual AMD Athlon MP1600+ CPUs and 2G memory. The version of `bcg_min` used was 1.3. It is clear from this table that the performance of our sequential tools is comparable to that of `bcg_min`. Hence, using the naive algorithm is feasible.

It is also clear that the marking strategy (used for the optimized algorithm) can give spectacular gains in time – see the numbers for both cache coherence protocols. The sequential optimized implementation needs more memory than the naive, since it keeps both the straight and the inverse transition systems. On the other hand, the naive one consumes more memory for the hashtable – all signatures have to be inserted, while only some have to be considered by the optimized implementation. Therefore, we expect that the optimized algorithm will be less memory expensive than the naive one when it comes to large examples. The distributed implementation confirms this idea.

3.6.2 Comparing the sequential with the distributed implementation

In Figure 3.12, we show how the performance of the sequential naive implementation compares to that of the distributed naive one with 16 workers on SGI, and how the latter compares to the performance on the cluster.

3.6.3 The distributed implementation: scalability

For the tests in this subsection we used the PC cluster. The inputs were 26 case studies from the VLTS benchmark suite. The selection criterion was no less than 10^5

¹the original LOTOS model [GM97] was translated to μ CRL by Judi Romijn and extended from 3 to 4 stations

problem	SSN SGI		DSN SGI (16)		DSN cluster (16)		DSO cluster (16)	
	time (s)	mem (M)	time (s)	mem (M)	time (s)	mem (M)	time (s)	mem (M)
lift5	64	123			33	460	20	480
CCP-2p3t	10480	1380	1249	5438	550	4430	104	1658
token ring	2505	4367	299	13416	120	10802	231	4508
lift6	15355	2652	1136	15372	702	5958	346	3834
1394-LE	12111	6566	1136	15372	555	15388	428	8737

Figure 3.12: A comparison of single threaded and distributed runs

transitions and small enough to be reduced on a single node. All the values presented are averaged from 5 runs. The speedup was computed from the real time (wall clock time) spent on the reduction only. That is, the time spent on doing I/O operations for reading the LTS and writing the result is not included. The problem size is the number of transitions.

Time In Figure 3.13 we have plotted the reduction times for each of the problems, and their translation to speedups relative to the distributed algorithm running on one node (and one worker). This picture shows clearly that for some problems we obtain good speedups and for others, on the contrary, we get a slow down. Many of the lines in this pictures curve downwards. This means that the efficiency decreases somewhat with the number of processors, due to the increasing influence of communication.

Because we designed the algorithm especially for large transition systems, we were interested in how the problem size influences the speedup. First, we looked at the speedup obtained by moving from a single CPU to a minimal distributed system. In Figure 3.14, we show the speedup relative to the program running on a single CPU for three possibilities: a single dual CPU node, two single CPU nodes and two dual CPU nodes. The considerable amount of extra CPU power in the 2 node, 4 CPU system seems to have had an effect, but apart from that moving from a single CPU to a minimal distributed system does not seem to have much of an advantage. Next, we looked at the speedup we got from moving from a minimal distributed system to larger distributed systems. In Figure 3.15, we show the speedups achieved by using 4 and 8 nodes relative to using 2 nodes for both the single and dual CPU case. Even though the lines are pretty erratic, it is possible to see a tendency of the 4 node lines to converge to 2 and for the 8 node lines to converge to 4. In the dual CPU plot it is also very obvious that using too many CPUs hurts performance.

Finally, the last speedups graphs we show (Figure 3.16) relate the distributed algorithm to the fastest *sequential* algorithm, thus conforming to the standard definition of the speedup for parallel programs.

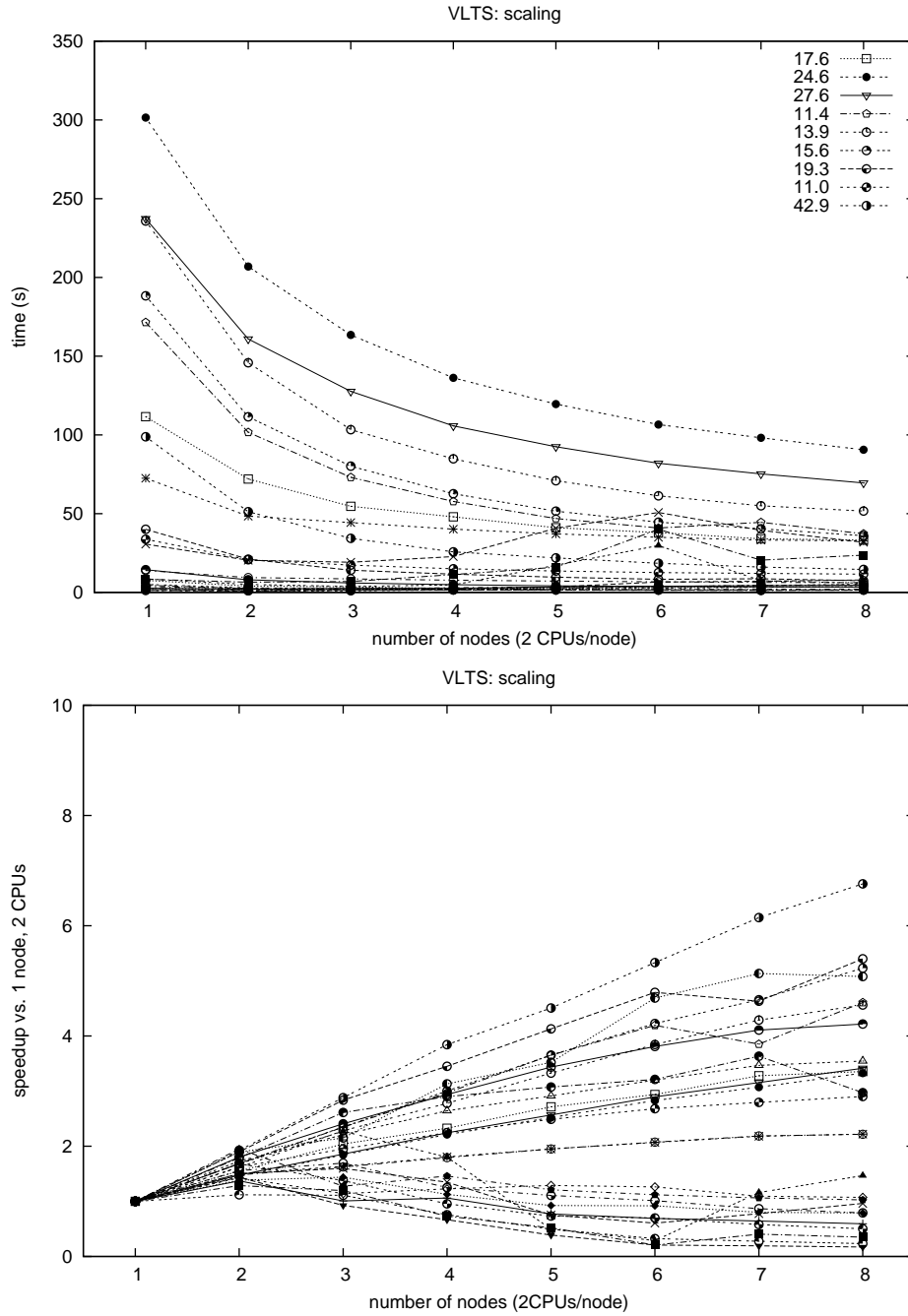


Figure 3.13: (DSN) Distributed reduction times (up) and speedup relative to DSN on one machine (down)

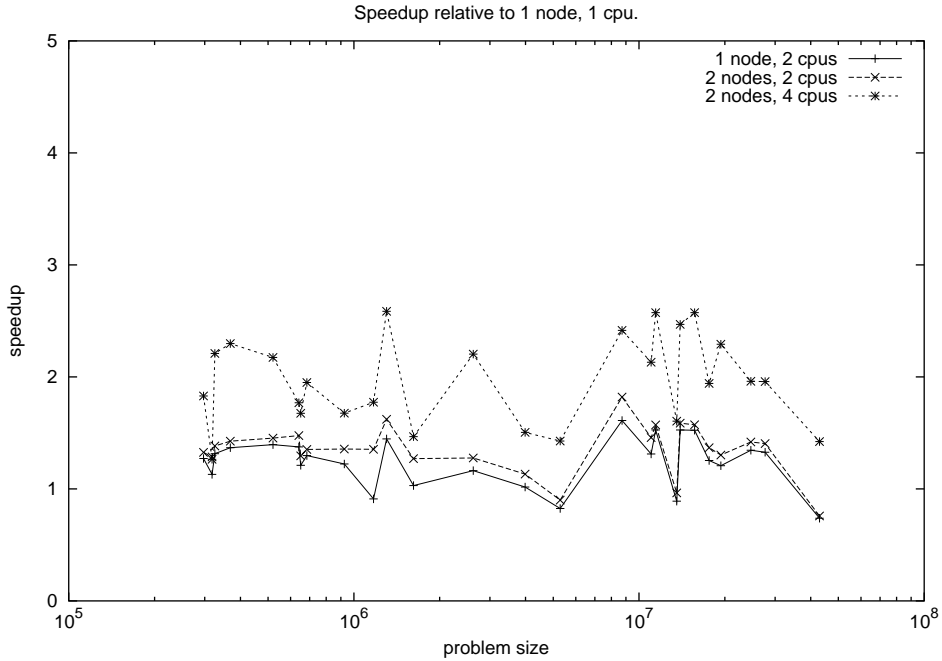


Figure 3.14: (DSN) Initial speedup

Memory We also made memory measurements, in order to observe the efficiency of the distribution and the scalability of the total memory usage. Figure 3.17 shows data collected by measuring the memory usage of each worker for distributed runs with 2 and with 8 workers. To facilitate comparisons and see the scaling up, we divided these values by the memory used by the distributed base run, i.e. the distributed implementation when run with one worker. For each problem, the plots show the minimum/average/maximum value thus obtained. We see in the figure that when 2 workers are used, memory consumption per worker drops to approximately 0.6 of the distributed base run and when 8 are used, to approximately 0.2. Thus, the more workers the less memory needed per worker. Note also that only for three problems there is a substantial difference between minimum and maximum. This means that the function we have used to distribute states across workers performs reasonably well: in most cases the workers need roughly the same amount of memory.

In Figure 3.18 we have plotted the total memory usage against the number of workers for small ($\leq 10^6$ transitions) and large ($> 10^6$ transitions) systems respectively. Here the memory is divided by the memory of the sequential implementation, in order to also provide a distributed/sequential comparison. For small systems the memory usage often increases quite rapidly with the number of workers, but for large problems

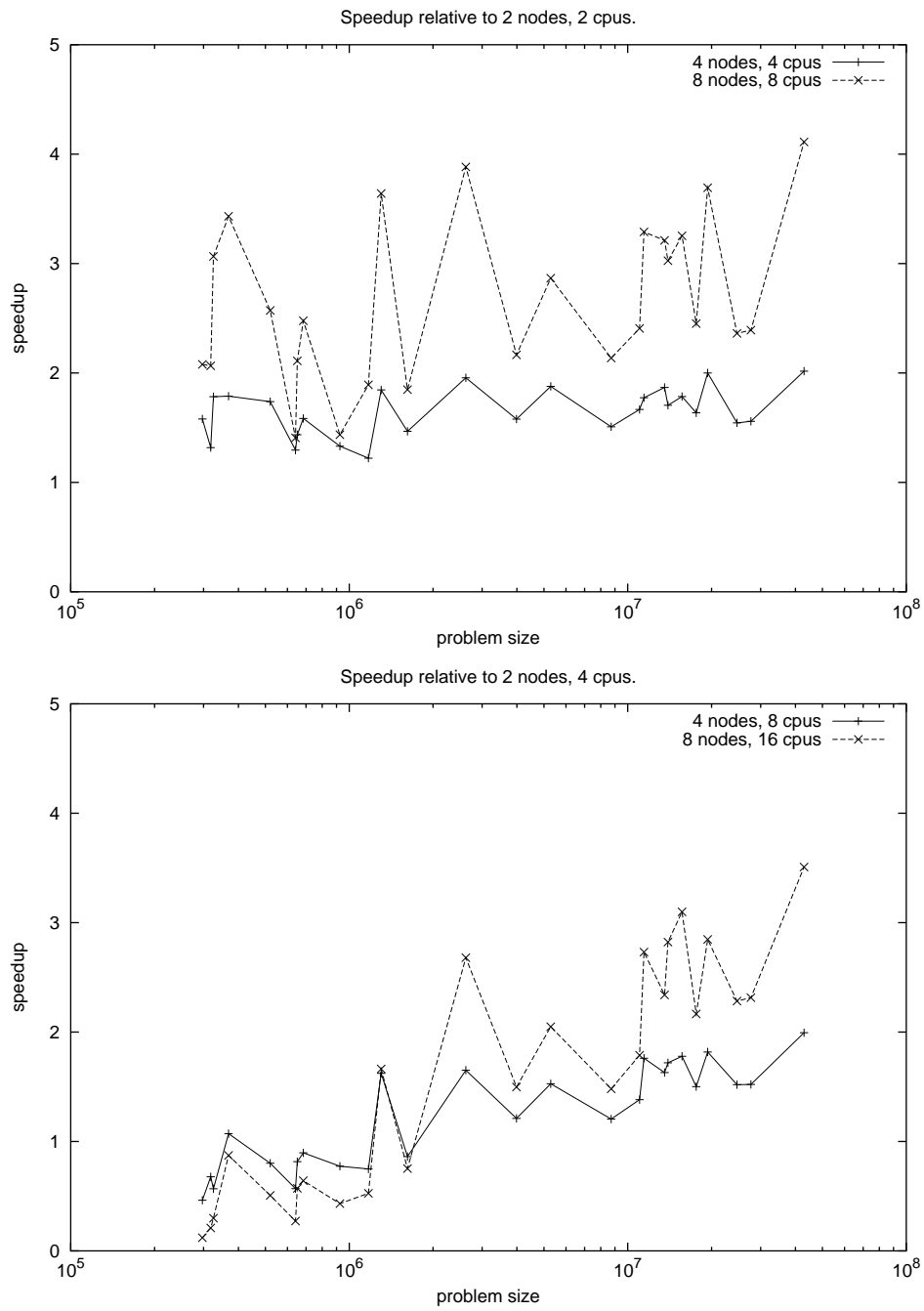


Figure 3.15: (DSN) Additional speedup

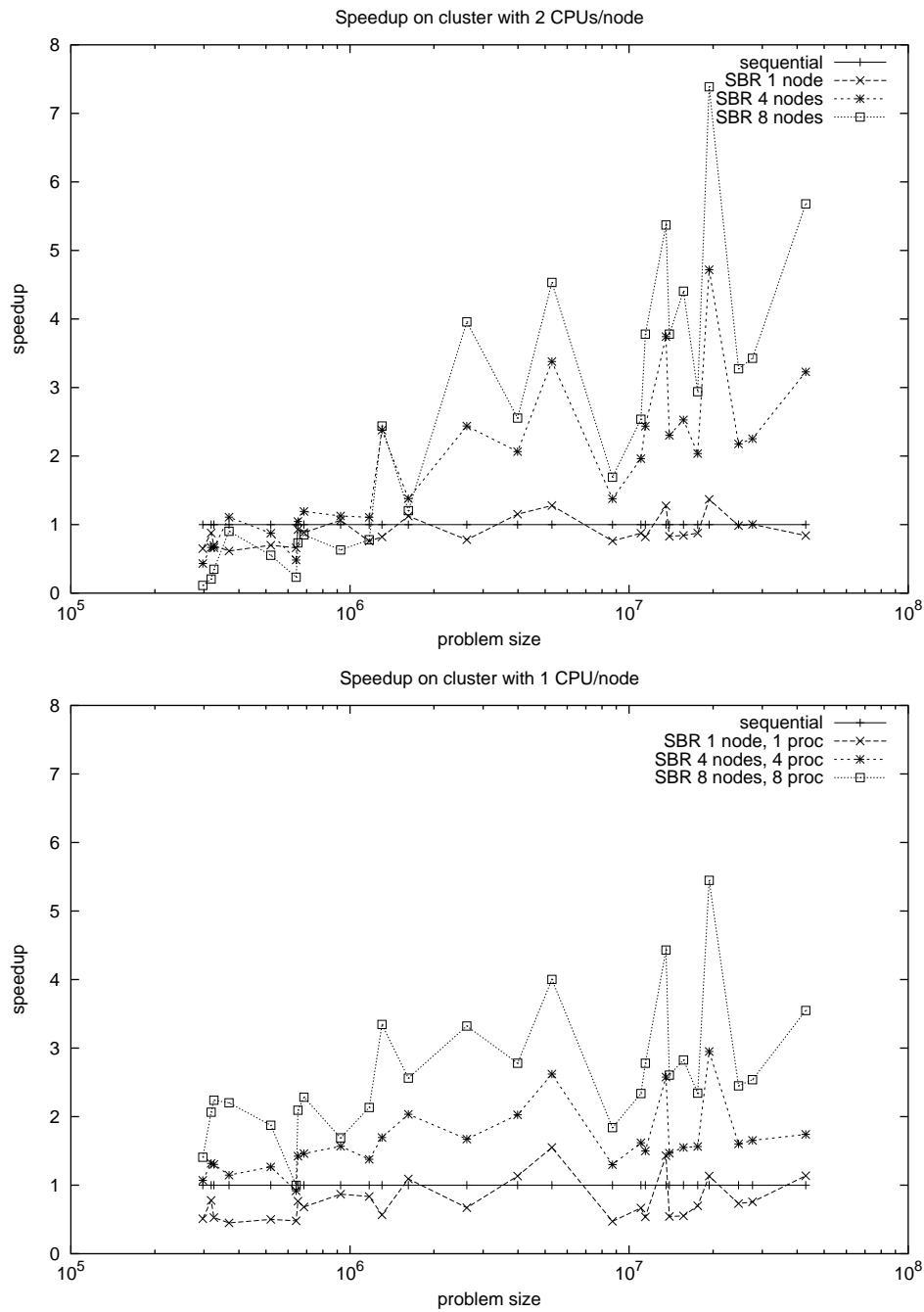


Figure 3.16: (DSN) Speedup with 2 CPUs/node (up) and with 1CPU/node (down)

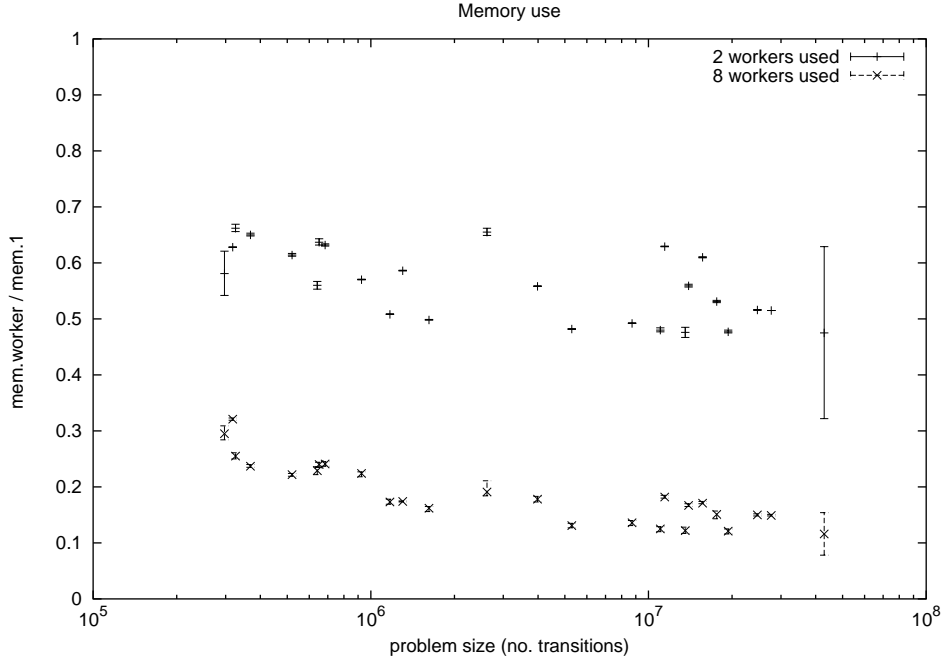


Figure 3.17: (DSN) Memory usage: distribution

the total memory usage does not increase much. From these pictures it is quite clear that for most large problems the memory usage of the distributed tool is between 2 and 3 times as much as the memory usage of the sequential tool. This is not unexpected: at least two copies of all the signatures must be kept (a local copy and a global copy) and ID information is sent using buffers, whose size is linear in the number of transitions.

3.6.4 Distributed naive vs. distributed optimized

Figure 3.12 shows a comparison of the naive and optimized distributed implementations on the cluster, for a number of large LTSs. The numbers listed for the memory usage represent the maximum total memory touched on all 8 workstations during a run.

The runs indicate that the optimized implementation outperforms the naive one most of the time. The optimized is designed to perform better when the partition refinement series needs a large number of iterations to stabilize, yet very few blocks split in every iteration. This is exactly the case for the CCP state space. On the other hand, for state spaces like the Token Ring protocol, where almost all blocks split in every iteration, and the whole process ends in just a few rounds, the naive

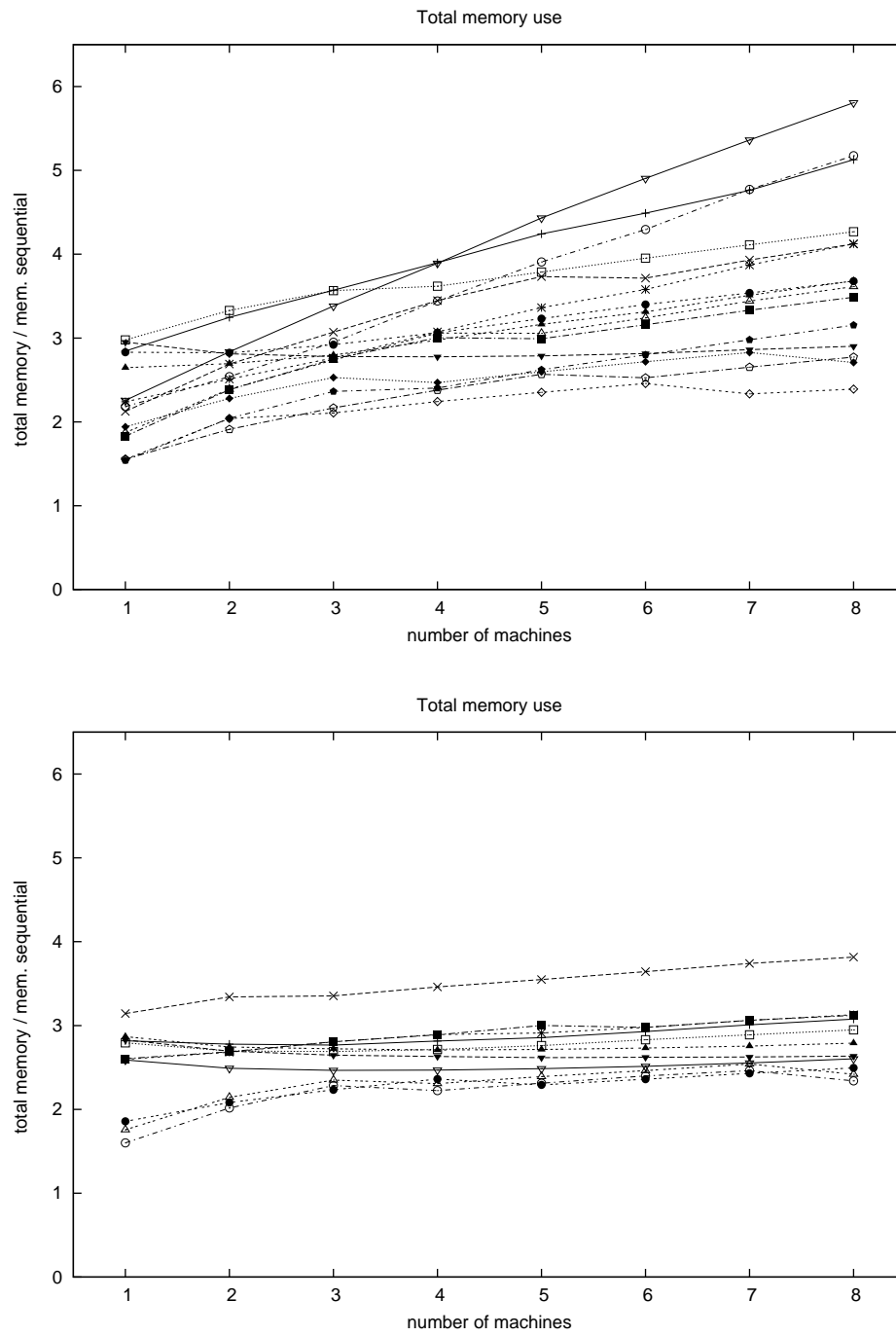


Figure 3.18: (DSN) Memory usage: scaling, small (up) and big LTSs (down)

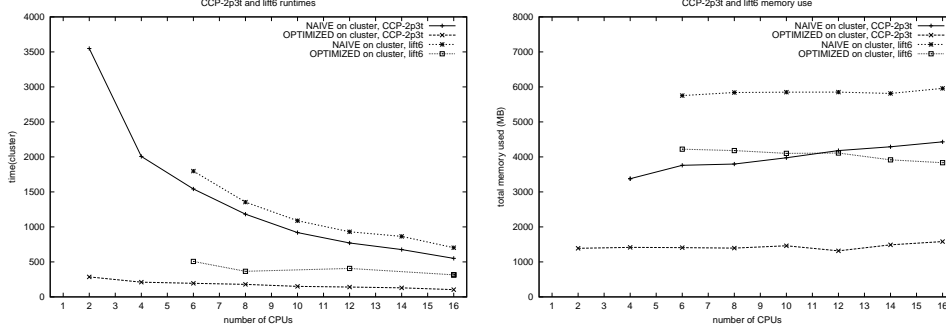


Figure 3.19: Runtimes and memory usage for *CCP-2p3t* and for *lift6*

algorithm works faster, since it does not waste time on administration issues. In all larger examples though, the memory gain is obvious – and for the bisimulation reduction problem, memory is a more critical resource than time.

To test how the optimized distributed algorithm scales, we ran on the cluster series of experiments using 1-8 machines (2-16 processors). Figure 3.19 shows the runtimes (in seconds) needed to reduce *lift6* and *CCP-2p3t*. Since *lift6* is a real industrial case study with serious memory requirements, it could not be run single threaded on a cluster node or distributed on less than 3 nodes. We see that for both distributed implementations and both case studies presented, the memory usage scales well, i.e. the total memory needed on the cluster is almost constant, regardless the number of machines used. Hence, more machines available will mean less resources occupied on each machine.

On runtimes however, the naive implementation scales in a more predictable manner, while the optimized times do not seem to scale up as nicely. This is partly due to the nondeterminism present in the optimized implementation – signatures can arrive at servers in any order, the order influences the new IDs assignment to states, the new IDs determine how many unstable states are there in the next iteration, thus how much time will that iteration cost etc. It is also due to the possibly unbalanced distribution of signatures to servers, which introduces unpredictable idle times. Last, there is some latency due to the MPI implementation. We compared (Figure 3.21) the reduction of *lift5* on the cluster with the reduction on a shared memory machine that uses its native MPI implementation. It appears that the optimized algorithm does scale better on this other MPI.

After analyzing the behavior of the two algorithms on some special case studies, we turn to “anonymous” state spaces from the VLTS benchmark [CI]. Figure 3.20 shows the times and total memory usage of the optimized algorithm relative to those of the naive algorithm. Unlike the other measurements presented, the times considered now are total, that is the I/O operations are included. The 25 state spaces in this selection

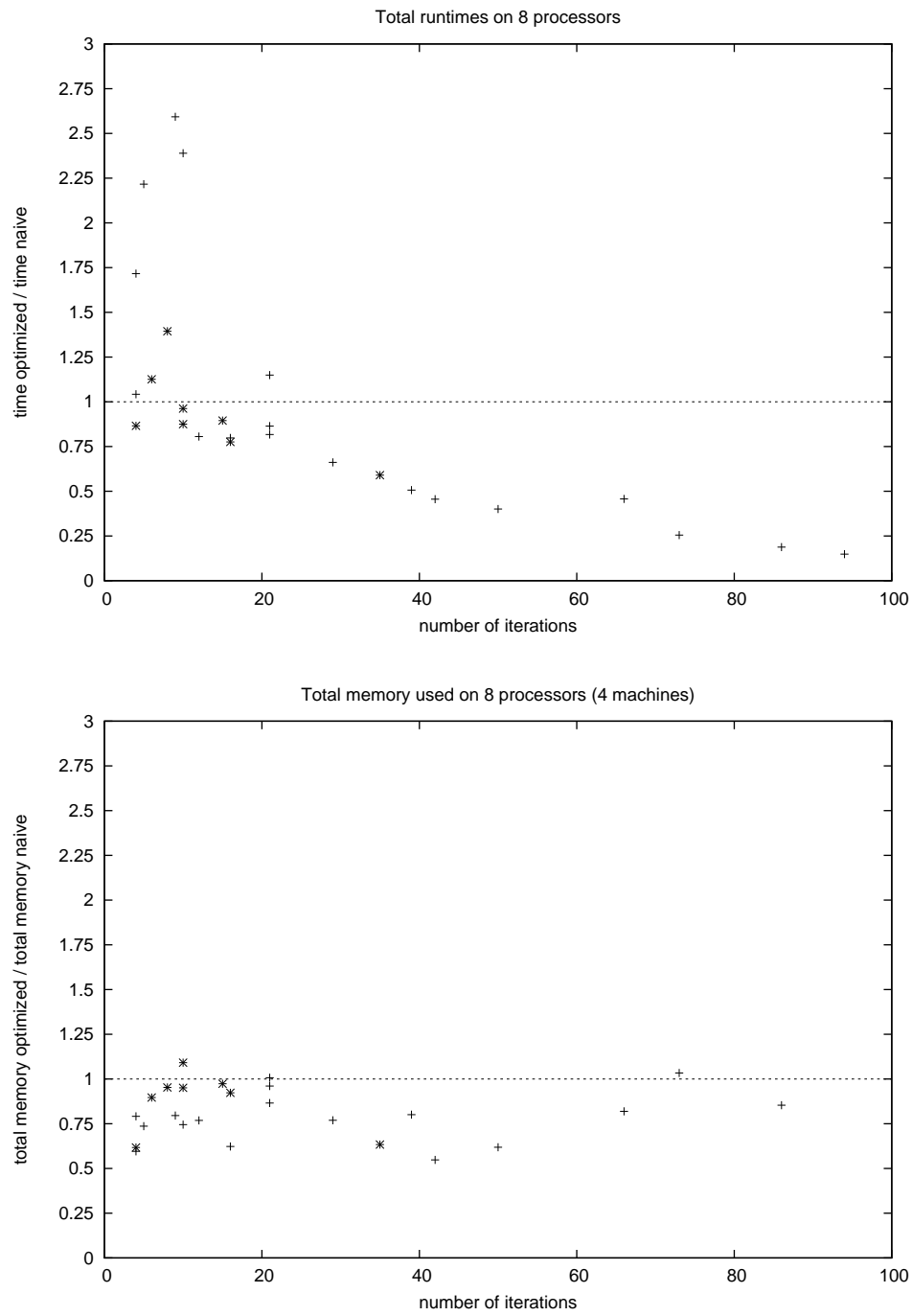


Figure 3.20: (DSO) The VLTS test suite

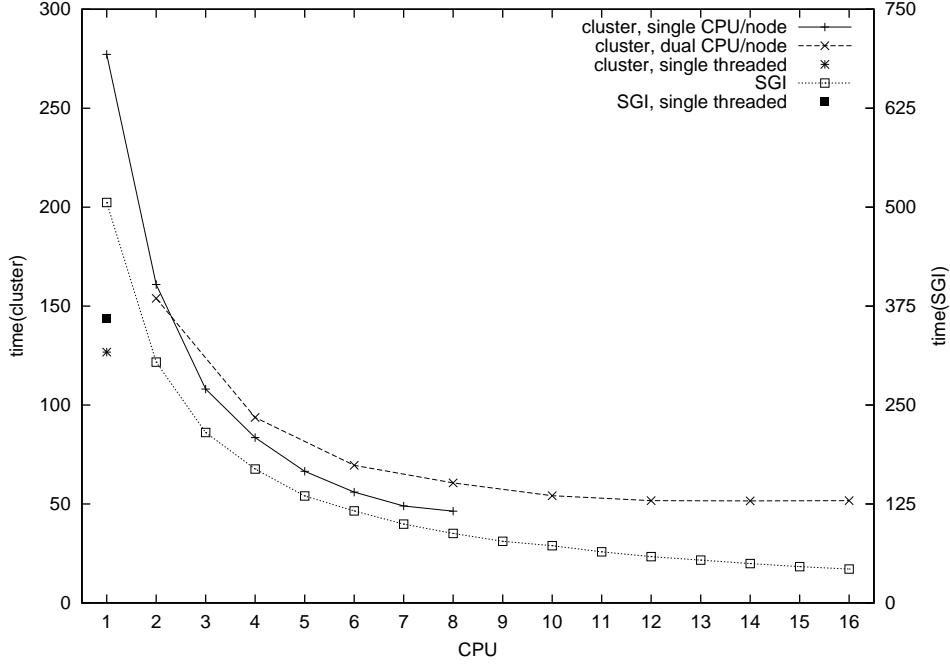


Figure 3.21: (DSO) Runtimes for *lift5* on SGI and on the cluster

are small to medium size (between 0.06 and 12 million states, and between 0.3 million and 60 million transitions) and they get reduced modulo strong bisimulation in less than 100 iterations. The stars mark the very small state spaces, i.e. those that are reduced in less than 5 seconds by both algorithms.

We present the state spaces ordered by the number of iterations in which the reduction procedure stabilizes. This is a relevant order only for the time performance, not for the memory usage.

As apparent from the figure, the relative time performance of the optimized algorithm is indeed influenced by the number of iterations and the size of the state space. This is roughly because, compared to the naive one, it spends (much) more time on the initial setup - and this time pays back only if the reduction process has some length. Note that for very short reductions, it can be almost 3 times slower than the naive, but for lengthy ones it is usually much faster (up to 6 times faster).

Regarding the memory usage, we may notice that the optimized algorithm is indeed almost always an improvement. Exceptions are the small state spaces, where the fixed size buffers used by the optimized are significantly larger than needed. This could be fixed by using dynamic buffers.

problem	DSN	DSN	DSN
	cluster fast	cluster gbit	SGI
1394-LE	1520	475	540
CCP-2p3t	1685	530	670

Figure 3.22: Network analysis

3.6.5 Hardware matters

The Fast Ethernet network connections works at 100 Mbps. The Gigabit Ethernet works at 1000 Mbps. The performance difference between these two networks can be seen in Figure 3.22, where the reduction times on different network connections are shown – as expected, the Gigabit Ethernet outperforms the Fast one.

During our experiments with the 6 leg lift problem, we found that reducing the LTS is not the only problem. The ext3 file system as implemented in the Linux 2.4 kernels is not suitable for reading/writing multiple large files in parallel. As a result, reading the LTS from disk actually took more time than reducing it. For later experiments we have used PVFS (Parallel Virtual File System [CLRT00]) instead of NFS. This is a distributed file system, which uses the disks of multiple machines to present a large file system to the user. Per node the performance of PVFS was roughly equal to that of NFS, but the performance of PVFS scales linearly with the number of nodes so effectively it was 8 times better.

3.7 Conclusions

We took a simple algorithm for strong bisimulation reduction and designed and implemented two distributed versions of it: a straightforward one (3.2,3.3) and a more elaborated and “optimized” one (3.4,3.5). The latter employs a marking technique for incremental computation of partitions and a setting where communication and computation can proceed in parallel. Therefore the performance is improved in memory and in some cases also in time.

We argued that, despite a poor worst-case theoretical complexity, in practice both distributed implementations have a decent speedup for large examples and, more importantly, the total memory consumed does not grow too much with the number of machines used.

The concept of signature refinement also works for other equivalences, like branching bisimulation (treated in the next chapter), weak bisimulation and τ^*a equivalence.

4

Branching Bisimulation Reduction

In the previous chapter, a few sequential and distributed algorithms were presented, for reduction of large LTSs modulo strong bisimulation equivalence. The starting point was the “naive method” of Kanellakis and Smolka [KS83]. In this chapter we adapt that straightforward solution to another very useful equivalence, namely branching bisimulation.

Related work The most commonly used algorithm for computing branching bisimulation is the one of Groote and Vaandrager [GV90]. It is a very good algorithm, but there are two reasons why one does not really want to use it for developing a distributed tool. First, the natural parallelism in the algorithm is very fine grained, which is a bad idea on a cluster, where the often large message latency leads to unacceptable performance. The second reason is that the Groote-Vaandrager algorithm works on LTSs that do not have cycles of silent steps. Cycle elimination requires detection of strongly connected components, which is a difficult problem to solve distributedly, although sequentially the well known Tarjan algorithm [Tar72] solves it in linear time. That sequential algorithm is based on depth first search traversal (DFS) of the graph, an idea very difficult to parallelize – it has been proved [Rei85] that DFS is P-complete (see also Section 2.2.6). The distributed algorithm that we now propose does not rely on the absence of τ cycles, but we learn from sequential studies that it would perform better on a cycle-free LTS. Therefore, it is interesting future work to integrate an initial distributed cycle elimination phase (see also Chapter 5) and optimize the actual reduction algorithm for LTSs without τ cycles.

Kripke structures are directed graphs with labeled states and unlabeled transitions. Branching bisimulation on LTSs resembles the stuttering equivalence on Kripke structures [BCG88] up to divergence sensitivity. Namely, stuttering equivalence distinguishes states where an infinite sequence of invisible steps is possible from states

where such a sequence is impossible, while branching bisimulation does not. Nevertheless, the stuttering equivalence algorithm presented by Browne, Clarke and Grumberg [BCG88] resembles ours in that it employs a similar partition refinement strategy. Only the way in which the refinements are computed is different: the Browne-Clarke-Grumberg algorithm calls for explicit computation of the transitive reflexive closure of silent steps whereas our algorithm avoids doing so. The naive algorithm that we have chosen as starting point is a brute force approach to state space minimization that employs the extensive resources of a cluster. Other approaches to saving space and time in the minimization process include exploiting the natural modular structure of systems [BG01].

Outline This chapter is organized as follows. Section 4.1 revisits some basic notions and gives the signature refinement algorithm for computing branching bisimulation. Section 4.2 proves it correct. The single threaded and distributed implementations are commented in Sections 4.3, 4.4 and their performance is discussed in Section 4.5. We draw conclusions in Section 4.6.

4.1 Partition refinement based on signatures computation

The definitions of LTSs and branching bisimulation equivalence have been given in Section 2.3.1. Note that the silent action τ is a member of the set of labels Act . In this section we fix some notations, recall the theory behind the partition refinement algorithm based on *signature* computation (presented and used in the previous chapter) and argue that it is applicable to branching bisimulation as well. We work with a fixed LTS (S, \rightarrow, s_0) and we use the following notations:

$$\begin{array}{ll}
 s \xrightarrow{a} t & \text{short for } (s, a, t) \in \rightarrow \\
 \xrightarrow{a} & \text{the transitive reflexive closure of } \xrightarrow{a} \\
 \xrightarrow[R]{a} & R \cap \xrightarrow{a} \text{ for any equivalence relation } R \\
 \xrightarrow[R]{a} & \text{the transitive reflexive closure of } \xrightarrow[R]{a}
 \end{array}$$

$\pi = \{B_i \mid i \in I\}$ is a *partition* of S if

$$(\forall B \in \pi : B \neq \emptyset) \text{ and } \bigcup_{i \in I} B_i = S \text{ and } (\forall B', B'' \in \pi : B' = B'' \vee B' \cap B'' = \emptyset) .$$

A partition π' is a *refinement* of a partition π if

$$(\forall B' \in \pi') (\exists B \in \pi) B' \subseteq B .$$

The elements of a partition are referred to as *blocks*. By $\pi(x)$ we denote the unique block B of π such that $x \in B$. We also view a partition π as a relation and abbreviate

$\pi(x) = \pi(y)$ as $x \pi y$. $\xrightarrow[\pi]{\tau}$ is then a particular case of $\xrightarrow[R]{a}$ and represents a sequence of 0 or more τ -steps within a block of π .

In Chapter 3 we defined the notion of *signature* of a state w.r.t. a partition. We based our sequential and distributed algorithms for strong bisimulation reduction on this notion. Now we redefine it, taking into account the silent steps and the new goal, which is characterizing and computing the branching bisimulation equivalence classes. The *signature* of S w.r.t. a partition π of S is a function $\text{sig}_\pi : S \rightarrow 2^{\text{Act} \times 2^S}$:

$$\text{sig}_\pi(s) = \{(a, \pi(t)) \mid \exists s' : s \xrightarrow[\pi]{\tau} s' \xrightarrow{a} t \wedge (a \neq \tau \vee \pi(s) \neq \pi(t))\}$$

The *signature refinement* of π is a new partition, denoted sigref_π , where the states of S having the same signature w.r.t. π are in the same block:

$$\text{sigref}_\pi = \{\{s' \in S \mid \text{sig}_\pi(s) = \text{sig}_\pi(s')\} \mid s \in S\}$$

A partition π is *stable* if $\text{sigref}_\pi = \pi$. The *signature refinement algorithm* iteratively computes π^{n+1} until the stable partition is reached.

$$\begin{aligned} \pi^0 &= \{S\} \\ \pi^{n+1} &= \text{sigref}_{\pi^n} \end{aligned} \tag{4.1}$$

For our LTS $\mathcal{S} \equiv (S, \rightarrow, s_0)$, let \xleftrightarrow{B} be the largest branching bisimulation relation on S , i.e. the one relating most states and the union of all branching bisimulations. Then the partition determined by \xleftrightarrow{B}

$$\pi^B = \{\{s' \in S \mid s \xleftrightarrow{B} s'\} \mid s \in S\}$$

is the coarsest (has a minimal number of blocks) among all partitions determined by branching bisimulations. Thus, the LTS with the minimal number of states that is branching bisimilar to \mathcal{S} is

$$\mathcal{S}^B \equiv (\pi^B, \{(\pi^B(s), a, \pi^B(t)) \mid s \xrightarrow{a} t \wedge (s \xleftrightarrow{B} t =: a \neq \tau)\}, \pi^B(s_0))$$

Our goal is to compute \xleftrightarrow{B} and we claim that the simple signature refinement algorithm above (4.1) does exactly that. However, because of the complex influence of the silent steps on the branching bisimulation relation, this is not trivial to prove.

4.2 Correctness of the naive algorithm

We first prove that the iterations of the algorithm (4.1) produce successive partition refinements (Lemma 4.2). Then we show that throughout the algorithm branching bisimilar states are kept together (Lemma 4.3) and that when the algorithm stops, a branching bisimulation relation has been reached (Lemma 4.4). Finally, in The-

orem 4.5 we put these facts together and justify that (4.1) correctly computes the minimal branching bisimulation.

Lemma 4.1 *If $s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} s_2$ and $s_0 \xleftrightarrow{B} s_2$ then $s_0 \xleftrightarrow{B} s_1$.*

Proof: Follows from the stuttering lemma in [GW96].

Lemma 4.2 *$(\forall n \geq 0)$ π^{n+1} is a refinement of π^n .*

Proof: We prove this claim by induction on n . The induction basis is immediate: any partition is a refinement of π^0 , so in particular π^1 is. As induction hypothesis suppose that for all $i < n$, π^{i+1} is a refinement of π^i . This guarantees that for any states x and y and any i, j such that $i < j \leq n$:

$$\text{if } \pi^i(x) \neq \pi^i(y) \text{ then } \pi^j(x) \neq \pi^j(y) \quad (4.2)$$

$$\text{if } \pi^j(x) = \pi^j(y) \text{ then } \pi^i(x) = \pi^i(y) \quad (4.3)$$

$$\text{if } x \xrightarrow{\tau} y \text{ then } x \xrightarrow{\pi^i} y \quad (4.4)$$

To show that π^{n+1} is a refinement of π^n , we proceed by supposing that this is not the case and deriving a contradiction. If π^{n+1} is not a refinement of π^n then there exist two states s, t for which $\pi^n(s) \neq \pi^n(t)$ and $\pi^{n+1}(s) = \pi^{n+1}(t)$. Then

$$\text{sig}_{\pi^n}(s) = \text{sig}_{\pi^n}(t) \quad (4.5)$$

From the induction hypothesis and the fact that $\pi^0(s) = \pi^0(t)$ it follows that there is a partition π^k ($k < n$) such that

$$(\forall j : 0 \leq j \leq k) \pi^j(s) = \pi^j(t) \text{ and } (\forall j : k < j \leq n) \pi^j(s) \neq \pi^j(t)$$

So, $\text{sig}_{\pi^k}(s) \neq \text{sig}_{\pi^k}(t)$. Without loss of generality, there exists a pair (a, B) with $a \neq \tau$ or $B \neq \pi^k(s)$ such that

$$(a, B) \in \text{sig}_{\pi^k}(s) \quad (4.6)$$

$$(a, B) \notin \text{sig}_{\pi^k}(t) \quad (4.7)$$

(4.6) translates to $(\exists s_1 \cdots s_q \in S, \exists x \in B \in \pi^k) s \xrightarrow{\tau} s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_q \xrightarrow{a} x$. When we turn to partition π^n , there are two situations possible:

- $s_1 \cdots s_q$ are all in $\pi^n(s)$. Let us then further distinguish two cases:
 - $a \neq \tau$. Then it is clear that $(a, \pi^n(x)) \in \text{sig}_{\pi^n}(s)$ and, according to (4.5), $(a, \pi^n(x)) \in \text{sig}_{\pi^n}(t)$. This means that there is a state y with $\pi^n(y) = \pi^n(x)$ and a state $t' \in \pi^n(t)$ such that $t \xrightarrow{\tau} t' \xrightarrow{a} y$. With (4.4) and (4.3), it follows that $t \xrightarrow{\tau} t' \xrightarrow{a} y$ and $\pi^k(y) = \pi^k(x) = B$, thus $(a, B) \in \text{sig}_{\pi^k}(t)$, which contradicts (4.7).

- $a = \tau$ and $B = \pi^k(x) \neq \pi^k(s)$. Then, according to (4.2), $\pi^n(x) \neq \pi^n(s)$. Therefore, $(\tau, \pi^n(x)) \in \text{sig}_{\pi^n}(s)$ and, because of (4.5), $(\tau, \pi^n(x)) \in \text{sig}_{\pi^n}(t)$. Consequently, $\pi^n(t) \neq \pi^n(x)$ and there is a state y with $\pi^n(y) = \pi^n(x)$ and a state $t' \in \pi^n(t)$ such that $t \xrightarrow{\tau/\pi^n} t' \xrightarrow{\tau} y$. Note that $\pi^k(y) = \pi^k(x) \neq \pi^k(s) = \pi^k(t)$. It immediately follows that $(\tau, B) \in \text{sig}_{\pi^k}(t)$, contradicting (4.7).

- Some of the states s_1, \dots, s_q are not in $\pi^n(s)$. Let x be the first of these. Then

$$s \xrightarrow{\tau/\pi^n} s_1 \xrightarrow{\tau/\pi^n} \dots \xrightarrow{\tau/\pi^n} s_r \xrightarrow{\tau} x \notin \pi^n(s) \quad (s_r \equiv s \text{ or } 1 \leq r < q),$$

therefore $(\tau, \pi^n(x)) \in \text{sig}_{\pi^n}(s)$ and thus (4.5) $(\tau, \pi^n(x)) \in \text{sig}_{\pi^n}(t)$ as well. This means that $\pi^n(x) \neq \pi^n(t)$ and there is a state $t' \in \pi^n(t)$ and a state y with $\pi^n(y) = \pi^n(x)$ for which $t \xrightarrow{\tau/\pi^k} t' \xrightarrow{\tau} y$. This path exists also in π^k (4.4):

$$t \xrightarrow{\tau/\pi^n} t' \xrightarrow{\tau} y \quad (4.8)$$

and moreover $\pi^k(y) = \pi^k(x) = \pi^k(s) = \pi^k(t)$ (4.3). Because $k < n$, $\pi^{k+1}(x) = \pi^{k+1}(y)$ and thus $\text{sig}_{\pi^k}(y) = \text{sig}_{\pi^k}(x)$. Since obviously $(a, B) \in \text{sig}_{\pi^k}(x)$, it follows that $(a, B) \in \text{sig}_{\pi^k}(y)$ and with (4.8), we obtain $(a, B) \in \text{sig}_{\pi^k}(t)$, contradicting (4.7). \square

The following lemma states that refining a partition where branching bisimilar states are in the same block results in a partition where branching bisimilar states are still in the same block. (Note that saying that π^B is a refinement of π is equivalent to saying that branching bisimilar states are in the same block of π .)

Lemma 4.3 *For any partition π , if π^B is a refinement of π then π^B is a refinement of sigref_π .*

Proof: We must show that for any s_0, t_0 such that $s_0 \pi^B t_0$, $s_0 \text{sigref}_\pi t_0$ holds. This means that we have to show that $\text{sig}_\pi(s_0) = \text{sig}_\pi(t_0)$, given that $s_0 \pi t_0$. Due to symmetry it suffices to show that $\text{sig}_\pi(s_0) \subseteq \text{sig}_\pi(t_0)$.

For $(a, B) \in \text{sig}_\pi(s_0)$, we can find a path $s_0 \xrightarrow{\tau/\pi} s_1 \xrightarrow{\tau/\pi} \dots s_n \xrightarrow{a} s$, such that $\pi(s) = B$ and $(a \neq \tau \vee \pi(s_0) \neq \pi(s))$. We construct a corresponding path starting from t_0 : given t_i such that $t_i \pi^B s_i$ and $i < n$, we define t_{i+1} by distinguishing two cases:

- If $s_{i+1} \pi^B s_i$ then let $t_{i+1} = t_i$. Then $s_{i+1} \pi^B t_{i+1}$ and $t_i \xrightarrow{\tau/\pi} t_{i+1}$.
- Otherwise, due to bisimulation and the stuttering lemma we can find t_{i+1} such that $t_i \xrightarrow{\tau/\pi^B} t' \xrightarrow{\tau} t_{i+1}$ and $s_{i+1} \pi^B t_{i+1}$. So for some t'_i , we have $t_i \xrightarrow{\tau/\pi^B} t'_i \xrightarrow{\tau} t_{i+1}$. Therefore, $t'_i \pi^B t_i \pi^B s_i \pi s_{i+1} \pi^B t_{i+1}$. Because π^B is a refinement of π , we

can conclude that $t'_i \pi t_{i+1}$. Thus, we have that $t'_i \xrightarrow{\tau/\pi} t_{i+1}$. Again because π^B is a refinement of π , we have that $t_i \xrightarrow{\tau/\pi} t'_i$, so we have $t_i \xrightarrow{\tau/\pi} t_{i+1}$.

If $\pi(s_0) \neq \pi(s)$ then $\pi^B(s_n) \neq \pi^B(s)$, because $\pi(s_0) = \pi(s_n)$ and π^B is a refinement of π . So we have $a \neq \tau \vee \pi^B(s_0) \neq \pi^B(s)$. We also know that $s_n \pi^B t_n$, so by definition of branching bisimulation and the stuttering lemma there exists t such that $t_n \xrightarrow{\tau/\pi^B} t$ and $s \pi^B t$. As π^B is a refinement of π , it follows that $t_n \xrightarrow{\tau/\pi} t$ and $s \pi t$. In turn this implies $t_0 \xrightarrow{\tau/\pi} t$, which means that $(a, B) \in \text{sig}_\pi(t_0)$. \square

Finally, we need to establish that a stable partition is a branching bisimulation.

Lemma 4.4 *If π is a stable partition then π is a branching bisimulation.*

Proof: Given $s \pi t$ and $s \xrightarrow{a} s'$, if $a = \tau$ and $s \pi s'$ then $s' \pi t'$. Otherwise $(a, \pi(s')) \in \text{sig}_\pi(s)$. Because the partition is stable, $\text{sig}_\pi(s) = \text{sig}_\pi(t)$. So for some t' we have $t \xrightarrow{\tau/\pi} t' \xrightarrow{a} t''$ with $s' \pi t''$ and $s \pi t'$. \square

From these three lemmas, the correctness of the partition refinement algorithm for finite LTSs follows easily:

Theorem 4.5 *Given a finite LTS the following program computes π^B in π :*

```

 $\pi := \{S\}$ 
repeat
   $\pi' := \pi$ 
   $\pi := \text{sigref}_\pi$ 
until  $\pi = \pi'$ 

```

Proof: After the n^{th} iteration of the loop, the variable π contains π^n . If the loop exits after n iterations then the partition π is stable. Due to Lemma 4.4 the resulting π is a branching bisimulation. Due to Lemma 4.3 it must be π^B . From Lemma 4.2 we get that sigref_π is a refinement of π . This means that if sigref_π is not the same as π then sigref_π contains more blocks than π . As the number of blocks is limited by the number of states, termination of the loop is guaranteed. \square

4.3 Sequential branching bisimulation minimization

We now describe a single threaded implementation (depicted in Figure 4.1) of the algorithm outlined in (4.1) and proved correct in Theorem 4.5. To represent partitions we assign a unique (integer) identifier to each block and then represent the partition as

```

1  for  $s \in S$  do  $ID[s] := 0$  enddo
2  do
3    /* compute signatures */
4    for  $s \in S$  do  $sig[s] := \emptyset$  enddo
5    for all transitions  $(s, a, t)$  do
6      if  $a \neq \tau \vee ID[s] \neq ID[t]$  then insert  $(s, a, ID[t])$  fi
7    enddo
8    /* reassign ID according to sig */
9    HashTable :=  $\emptyset$ 
10   count := 0
11   for  $s \in S$  do
12     if  $sig[s] \notin Keys(HashTable)$ 
13       then hash_insert (HashTable,  $sig[s]$ , count)
14         count := count + 1
15     fi
16   enddo
17   for  $s \in S$  do  $ID[s] := \text{lookup} (HashTable, sig[s])$  enddo
18 enddo until ID is stable

insert  $(t, a, id)$ :
1  if  $(a, id) \notin sig[t]$ 
2  then
3     $sig[t] := sig[t] \cup \{(a, id)\}$ 
4    for all  $s$  such that  $s \xrightarrow{a} t \wedge ID[s] = ID[t]$  do
5      insert  $(s, a, id)$ 
6    enddo
7  fi

```

Figure 4.1: (SBN) Single threaded naive branching bisimulation minimization

an array of block identifiers, which is indexed by states. Thus, the initial partition can be represented as an array of zeros. The definition of signature considers transitions of all states reachable by τ -steps within blocks. Explicitly computing sets of reachable states should be avoided because this would require too much time and memory. So instead of starting at a state and searching the reachable states for information, we start with the information and propagate it back along the τ -steps within blocks using a backward depth first traversal. Once all signatures have been computed, unique identifiers are assigned to signatures and from these identifiers the next partition is built. Based on the number of identifiers, we can decide if the partition is stable and iterate if necessary.

4.3.1 Comments on complexity

For an LTS with N states and M transitions, the worst case complexity of our algorithm is $\mathcal{O}(N^2M)$ time and $\mathcal{O}(NM)$ space. This is much worse than the $\mathcal{O}(N(N+M))$

```

reduce()
1 for  $s \in S$  parallel do  $ID[s] := 0$  enddo
2 do
3   for  $s \in S$  parallel do
4      $sig[s] := \{(a, ID[t]) \mid s \xrightarrow{a} t \wedge (a \neq \tau \vee ID[s] \neq ID[t])\}$ 
5      $pred[s] := \{t \mid t \xrightarrow{\tau} s \wedge ID[s] = ID[t]\}$ 
6   enddo
7    $new := sig$ 
8   do
9     for  $s \in S$  parallel do  $nextnew[s] := \emptyset$  enddo
10    for  $s \in S$  parallel do
11      for  $t \in pred[s]$  do
12         $nextnew[t] := nextnew[t] \cup (new[s] \setminus sig[t])$ 
13         $sig[t] := sig[t] \cup new[s]$ 
14      enddo
15    enddo
16     $new := nextnew$ 
17  enddo until  $\forall s : new[s] = \emptyset$ 
18  reassign  $ID$  according to  $sig$ 
19 enddo until  $ID$  is stable

```

Figure 4.2: (DBN)Distributed branching bisimulation minimization

time and $\mathcal{O}(N + M)$ space complexity of the Groote-Vaandrager algorithm. However, we expect that for typical state spaces (Section 2.3.3) both algorithms perform practically in $\mathcal{O}(\log(N)(N + M))$ time and $\mathcal{O}(N + M)$ space. Next we will analyze the complexity of an example near to the worst case.

Example 4.6 *Given a natural number N , consider the LTS with states $1, 1', 2, 2' \dots, N, N'$, transitions $i \xrightarrow{a} i'$, $i + 1 \xrightarrow{\tau} i$, $1 \xrightarrow{\tau} N$, $(i + 1)' \xrightarrow{b} i'$ and initial state N . The signatures for this LTS are*

$$\begin{aligned}
sig_{\pi^k}(i) &= \{(a, \pi^k(1')), \dots, (a, \pi^k(N'))\} \\
sig_{\pi^k}(1') &= \emptyset \\
sig_{\pi^k}((i + 1)') &= \{(b, \pi^k(i'))\}
\end{aligned}$$

and the partitions are

$$\begin{aligned}
\pi^0 &= \{\{1, 1', 2, 2' \dots, N, N'\}\} \\
\pi^k &= \{\{1'\}, \dots, \{k'\}, \{(k + 1)', \dots, N'\}, \{1, \dots, N\}\}
\end{aligned}$$

This means that $N + 1$ iterations are needed to get to a stable refinement of π^0 . The cost of computing the signature of i' is constant in each iteration because the signature

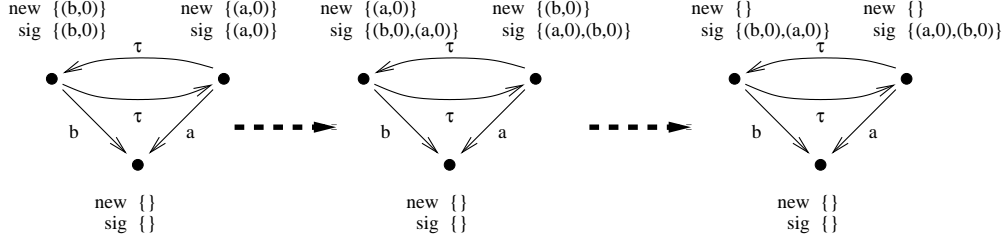


Figure 4.3: Computation of signatures

size is constant. However, the cost of computing the signature of i in the k^{th} linearly grows with k because the size of the signature is k . As we have N signatures of each kind, we get time complexity $\mathcal{O}(N^3)$ and space complexity $\mathcal{O}(N^2)$.

4.4 Distributed branching bisimulation minimization

For details about data distribution and computation of partitions from signatures, we refer to the first part of Chapter 3. We recall that the states are divided among a set of workers and we may apply the function `worker` to a state to get the worker which owns the state.

Let us now see how a distributed version of the algorithm can be implemented. The single threaded algorithm uses sequential depth first traversal for propagating signature information. As the order of signature propagation is irrelevant, we chose breadth first propagation for the distributed algorithm in Figure 4.2. In order to present the global picture in a clear way, we write it as a shared memory algorithm and abstract away the actual location of data. Each worker stores the parts of the arrays corresponding to its owned states. This means that the underlined references to arrays are potentially remote references. There are remote references to three arrays: `ID`, `newsig` and `sig`. In our distributed implementation remote references to `ID` are made local by copying the relevant parts of `ID`. That is, every worker keeps not only the `ID` data for its owned states, but also for the successor states of its owned states. Remote access to `newsig` and `sig` is solved in a different way. Instead of letting the owner of the `new` array perform the assignments, we let the owner of the `new` array send a message containing `s`, `t`, and `new[s]` to the owner of the `newsig` and `sig` arrays. Upon receiving such a message the owner of the `newsig` and `sig` arrays will perform the assignments. This is correct because the order of the assignments to `nextnew` and `sig` does not matter as long as they are atomic (no other assignments carried out in between).

Message passing replacements for lines 3-6 and 10-15 of the code in Figure 4.2 can be found in Figures 4.4 and 4.5, respectively. The replacements consist of multiple threads which are separated by `||`. In Figure 4.4, the initialization of `sig` is done before

```

1 for  $t \in S$  parallel do  $sig[t] := \emptyset$  enddo
2 for  $t \in S$  parallel do
3   for  $s, a$  such that  $s \xrightarrow{a} t$  do
4     SEND  $\triangleleft pi : s, a, t, ID[t] \triangleright$  TO worker( $s$ )
5   enddo
6 enddo
7 ||
8 while RECEIVE  $\triangleleft pi : s, a, t, id \triangleright$  do
9    $ID[t] := id$ 
10  if  $a = \tau$  and  $ID[s] = ID[t]$ 
11    then
12      SEND  $\triangleleft pred : t, s \triangleright$  TO worker( $t$ )
13    else
14       $sig[s] := sig[s] \cup \{(a, id)\}$ 
15    fi
16 enddo
17 ||
18 while RECEIVE  $\triangleleft pred : t, s \triangleright$  do
19    $pred[t] := pred[t] \cup \{s\}$ 
20 enddo

```

Figure 4.4: Message passing replacement for lines 3-6 of DBN

starting the parallel threads. The receive statement blocks until there is a message returning true or until there are no further messages in the system and no further sends can be initiated in which case they return false. For performance reasons the actual implementation buffers a few KB worth of small messages before sending.

In Figure 4.3 we have illustrated the process of signature computation. When the computation starts every state is in partition 0. Initially, the signature sets contain the (transition,id) pairs which are possible in every state and the new sets are set to the same value. In every iteration, the new sets are forwarded along the inverse of the invisible τ -steps, added to the signature sets and the new elements are added to the new sets. So for example in the first iteration (b,0) is sent along the top edge, inserted in the signature of the right state and because it is new it is also put into new. In the second iteration it is sent along the bottom τ -edge and inserted, but because it was already present it is not added to new. This forwarding continues until the new sets are empty.

We have omitted the code for reassigning ID according to sig, because the distributed assignment works the same as the single threaded, with the exception that hashtable lookups are performed by means of message passing rather than by means of memory access.

```

1  for  $t \in S$  parallel do
2    for  $s \in pred[t]$  do
3      for all  $(a, id) \in new[t]$  do
4        SEND  $\triangleleft new : s, a, id \triangleright$  TO worker( $s$ )
5      enddo
6    enddo
7  enddo
8  ||
9  while RECEIVE  $\triangleleft new : s, a, id \triangleright$  do
10    if  $(a, id) \notin sig[s]$ 
11      then
12         $sig[s] := sig[s] \cup \{(a, id)\}$ 
13         $nextnew[s] := nextnew[s] \cup \{(a, id)\}$ 
14      fi
15    enddo

```

Figure 4.5: Message passing code for lines 10-15 of DBN

4.5 Experiments

We have built prototype implementations of both sequential and distributed branching bisimulation minimization algorithms. The distributed implementation uses MPI for communication. The tests were made on a cluster of 8 dual AMD Athlon MP1600+ machines with 2G memory each, running Linux and connected by Gigabit Ethernet.

The examples used are the state space of the FireWire Link Layer protocol [Lut97] (1394-LL), the FireWire Leader Election protocol [SZ98] with 14 nodes (1394-LE), a cache coherence protocol [PFHV03] (CCP-2p3t), and a distributed lift system with 5 and 6 legs [GPW03] (lift5, lift6). See Section 3.6 for a short description of these case studies.

4.5.1 Single-threaded implementations

In order to investigate possibilities, we have implemented four variants of the branching bisimulation reduction scheme based on signatures. They are showed in Figure 4.6, all under the same name SBN. The variant called *cycle* eliminates the τ cycles before starting the iterations series, while *dfs* and *iter* do not. Further, *iter* computes the signatures by performing propagation sub-iterations, as done in the distributed implementation. Finally, *mark* employs a marking procedure that proved helpful in the strong bisimulation reduction case (see Section 3.4). Its basic idea is to restrict the signature recomputation effort of an iteration to those signatures that changed for sure.

Figure 4.6 displays the total run times (read, reduction and write) of these im-

problem	size	bcg_min 1.4	SBN cycle	SBN dfs	SBN iter	SBN mark	number of iterations
	states transitions	time mem	time mem	time mem	time mem	time mem	
1394-LL	0.37 10^6 0.68 10^6	2.27s 2.2M	0.98s 2.8M	0.97s 3.5M	2.5s 3.5M	1.16s 4M	6
lift5	2.2 10^6 8.7 10^6	2m42s 174M	1m18s 108M	1m20s 152M	9m03s 116M	2m30s 410M	16
1394-LE	2.5 10^6 17.6 10^6	1m18s 316M	1m11s 220M	1m08s 411M	1m25s 220M	1m14s 340M	2
CCP-2p3t	7.8 10^6 59 10^6	19m26s 1051M	22m50s 736M	62m52s 968M	- -	- -	46

Figure 4.6: A comparison of sequential implementations.

plementations and the maximum amount of memory occupied. To show that our signature refinement scheme is comparable to the block based refinement scheme, we include *bcg_min* (the reduction tool belonging to the CADP toolset; it implements the Groote-Vaandrager algorithm [GV90]) in this brief comparison. For the CCP-2p3t example, the *iter* implementation takes too much time and *mark* runs out of memory. The reason for the *iter* implementation taking too much time was diagnosed as an inefficient implementation of one sub-routine. Thus, we could avoid making the same mistake in the distributed implementation. We stopped the single threaded tool after more than 24 hours, with only half the job completed. The distributed tool completes the task in roughly 12 minutes on 16 processors.

The first conclusion of this sequential study is that the signature based reduction algorithm works for branching bisimulation. The cycle elimination seems to be an advantage (cycle vs. dfs), therefore it might be interesting to use it also in the distributed version. From the performance data of *iter* it is clear that there is no serious efficiency loss by using mechanisms specific to a distributed implementation. The marking procedure does not deliver spectacular improvements, in fact no improvements at all. The explanation is that this procedure is efficient in the iterations when few changes happen – typically towards the end of the reduction process. But the branching bisimulation algorithm usually stabilizes in a rather small number of iterations, therefore the administrative penalties paid in the first iterations are not regained later. (1394-LL, for instance, stabilizes in 73 iterations for strong and in 6 for branching; lift5 in 86 for strong, 16 for branching; 1394-LE in 51 for strong and only 2 for branching.)

4.5.2 Distributed implementation

Figure 4.7 shows the times and memory usage of the distributed prototype DBN when run on 4,5,6,7 and 8 workstations (with the input *lift6*, that cannot be reduced on less than 4 machines). For comparison, we also show the speedup of the similar distributed

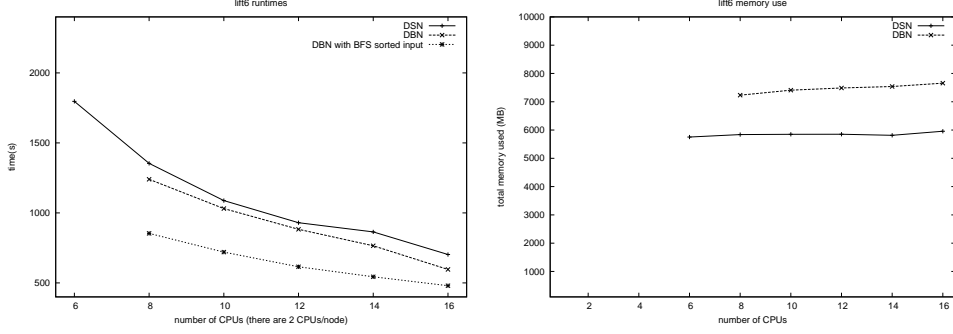


Figure 4.7: Time and memory usage for the reduction of lift6 (34 million states, 165 million transitions)

tool developed for strong bisimulation reduction, DSN (Chapter 3). DBN's memory needs grow slowly from 7232M for 8 processors to 7656M for 16 processors, while DSN used 5752M (8 processors) up to 5958M (16). This shows that the memory usage per worker decreases almost linearly with the number of workers.

As mentioned in the previous subsection, the stable partition with respect to branching bisimulation is most of the time reached in (a lot) less iterations than the stable partition with respect to strong bisimulation. This explains why, although a DBN iteration takes longer than a DSN one, DBN needs on the whole less time. As regard to memory usage, DBN is in all cases more expensive than DSN. This is due to two factors. Firstly, the signatures for the branching bisimulation case are in general larger, since the signatures of a state x must include the signatures of all states reachable by silent steps. And secondly, our current implementation is a first prototype, not yet optimized for memory usage. We expect that a more careful implementation will visibly reduce this difference.

A more interesting comparison is between the run times of DBN for random and for sorted input. (Random meaning a copy without caring about the order and sorted means sorted into the same BFS order written by our distributed state space generation tool.) The data indicates a much better performance in the case when the distribution of the states to the workers is done on BFS order. This means that we should investigate whether other orders exist, which can easily be computed and show even better performance.

Finally, in Figure 4.8 we show a speedup graph obtained by dividing the runtimes of *SBN-cycle* (the best sequential algorithm) to the runtimes of DBN on 1, 4 and 8 nodes (i.e., 2, 8 and 16 CPUs). The wild shape of this graph is due first of all to the fact that the sequential and the distributed algorithms compared are fundamentally different: the sequential eliminates the cycles of internal steps, while DBN incorporates them.

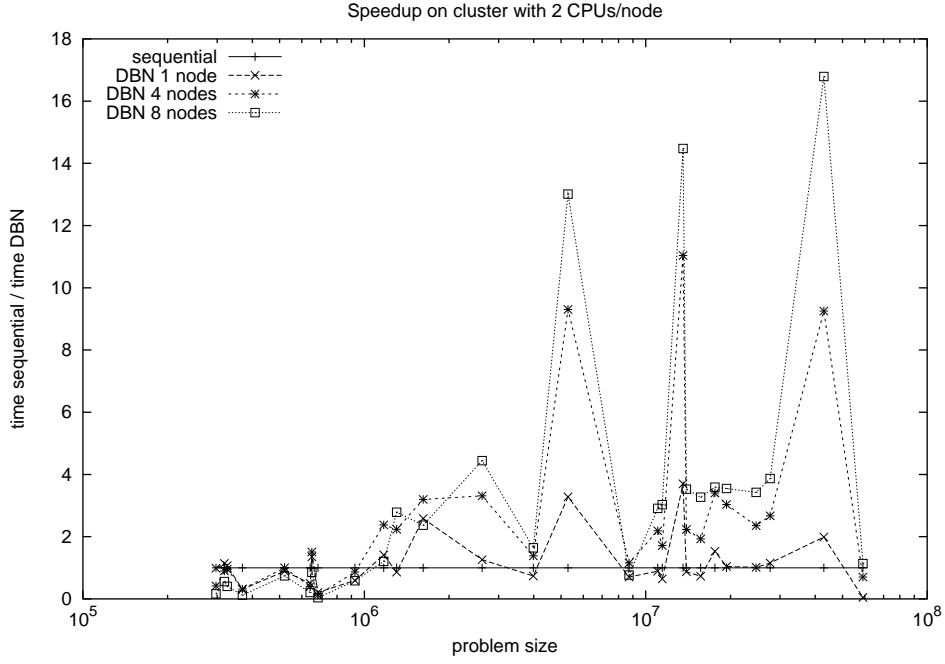


Figure 4.8: Speedup of DBN

For most case studies eliminating the cycles is a good idea, as sustained also by the quite small speedup or even slowdown exhibit by DBN on a number of case studies. But there are also cases when DBN's approach works better.

4.6 Conclusions

The work presented here continues the series of distributed minimization algorithms from Chapter 3. In this chapter we considered *branching bisimulation* as reduction relation and we developed a signature based partition refinement algorithm for it, that works on LTSs with cycles of invisible steps. We proved its correctness, briefly described its implementation and showed by some experimental results that it scales up reasonably both in time and memory usage.

5

Detecting Strongly Connected Components

A strongly connected component (SCC) of a directed graph is a maximal subgraph in which every vertex is reachable from every other vertex. The problem of decomposing a graph into SCCs is a well known and studied one and has an elegant solution, linear in the input size, based on depth first search [Tar72]. The SCC detection problem has applications in many different areas, from data mining to scientific computing to computer-aided design and model checking. Our motivation to study it comes, as expected, from verification by enumerative model checking and our graphs of interest are state spaces. The SCC detection occurs in several stages of this verification process. For instance, the algorithms for branching bisimulation reduction usually employ a preprocessing step in which the cycles of invisible steps (τ s) are eliminated. In other words, the SCCs of the τ -subgraph are detected and collapsed. Another use of SCC detection is in LTL model checking: finding counterexamples means finding cycles reachable from the root state that contain some special accepting states.

In this chapter we investigate distributed message passing solutions for the detection of SCCs. We describe a collection of heuristics that explore the characteristic features of state spaces, especially the small depth and diameter. (See also the discussion in section 2.3.) We state the SCC problem and present the solution in the context of unlabeled graphs (transition systems). The algorithms can immediately be applied to deal with labeled graphs in which only SCCs with a certain label must be found – for instance, the τ -cycles in the case of state spaces.

Related work The sequential very efficient algorithm of Tarjan [Tar72] essentially uses depth first search and is not likely to have an efficient parallel implementation [Rei85]. Therefore, to solve the problem in a parallel/distributed setting, other methods have been explored. For instance, an NC algorithm for finding SCCs that uses matrix multiplication is proposed in [GM88] and improved in [CV89]. A more in-

interesting approach for our application domain is taken in [FHP00] and [MIHPR01], where a divide-and-conquer algorithm is described, analyzed and implemented. However, that algorithm is aimed at another type of graph: typically much smaller than our state spaces and with high outgoing degrees. The essential observation is that the SCC of any state x is exactly the intersection of its successors and predecessors sets. Our coloring transformation (Section 5.3.6) also uses this idea, but instead of picking a pivot state and splitting the graph in three independent (no crossing SCCs) parts, we use a set of prioritized colors and split the graph in many parts at once. This rather brute force approach exhibits more parallelism and it works quite well in practice. The trimming step used in [MIHPR01] is similar to our detection of atomic components.

In the verification world, the problem of detecting SCCs in a distributed graph has so far received attention only in the context of (on-the-fly) LTL/CTL model checking [BČKP01, BBC03, ČP03]. Like in our approach, in [BBC03] the DFS traversal is abandoned in favor of BFS. The algorithm proposed in [ČP03] is inspired by a symbolic algorithm and it also contains a phase where atomic SCCs are eliminated. We encountered the SCC problem when building a distributed tool for branching bisimulation reduction (Chapter 4) and therefore we focus on this application.

A related problem is that of detecting connected components in undirected graphs, to which also parallel [HMB01] and distributed [BT01] solutions exist. Since a SCC is also a CC in the underlying undirected graph, these algorithms could be useful as a first step in detecting SCCs. But not for our application domain, since state spaces are always connected graphs.

Outline Section 5.1 introduces the main definitions, motivation and the SCC detection problem from the verification point of view. Our graph transformation routines are described briefly in Section 5.2 and in more detail in Section 5.3. In Section 5.4 three SCC reduction algorithms using the transformations are described. Then, in Section 5.5, some experiments with the three algorithms are presented. Conclusions are summarized in Section 5.6.

5.1 Preliminaries

Until now, we considered LTSs as representation of state spaces. But for the applications that we are targeting in this chapter, the SCC labels actually do not matter, therefore we will work with unlabeled state spaces (S, T) , where S is, like in an LTS, a set of states and T is a binary relation on S . When T is understood, we will use the notation $p \rightarrow q$ for $(p, q) \in T$ and $p;q$ for the reflexive transitive closure of T . We also introduce the following notations:

- T^* the reflexive transitive closure of a binary relation T
- T^{-1} the binary relation inverse to T

The *incoming degree* of a state is the number of transitions that end in that state. Conversely, the *outgoing degree* of a state is the number of transitions originating in that state.

5.1.1 SCC detection as verification problem

Our intended use of the SCC detection algorithm is as preprocessing step for two verification algorithms: branching bisimulation reduction/equivalence and LTL model checking.

The preprocessing phase in the *branching bisimulation* algorithm consists of merging the states that can reach each other on invisible paths, since they have the same (branching bisimilar) behavior. In other words, it consists in collapsing the SCCs in the graph obtained from the original state space by ignoring the visible transitions. Although observations on sequential implementations show that the preprocessing phase is a big advantage, the distributed algorithm in Chapter 4 avoided using it because a distributed cycle elimination algorithm that can handle very large instances seemed to be a difficult and challenging problem in itself. This is our main motivation to study the SCC problem.

For the *LTL model checking* algorithm the interesting information is whether a given state belongs to a cycle, no matter what labels that cycle might contain. In this case a useful preprocessing step is to detect the SCCs of the graph obtained from the original state space by ignoring the labels and to mark all the states situated on a cycle. This procedure consists of computing *scc*, then performing an extra test for self-loops.

Both applications are instances of:

The SCC detection problem. Given an unlabeled state space (S, T) , find a representative function $\text{scc} : S \rightarrow S$ such that $\forall x, y : \text{scc}(x) = \text{scc}(y)$ iff $(x, y) \in T^* \cap T^{*-1}$.

5.1.2 Sequential SCC detection

The classical approach to the detection of strongly connected components is the Tarjan algorithm [Tar72], that is based on depth-first traversal and solves the problem in linear time. We present a version of this algorithm, explained and justified in [AHU83]. The input is a state space $\mathcal{S} = (S, T)$.

SCCTarjan(\mathcal{S}):

- Perform a depth first search traversal of \mathcal{S} and compute the *finishing times* of all states. This is done by successively calling the *DFS* routine below for a not yet visited state until all states have been visited. The finishing time of x is the moment when the x and all its successors have been visited. The clock I is

initialized at 0 and increased with every new state visited.

```

DFS( $x$ ) :
  mark  $x$  as visited
  for each transition  $(x, y)$  do
    if  $y$  unmarked then DFS( $y$ )
  finish_time[ $x$ ] :=  $I$ 
   $I$  :=  $I + 1$ 

```

- Construct the reverse state space $\mathcal{S}^{-1} = (S, T^{-1})$.
- Perform a depth first search traversal of \mathcal{S}^{-1} starting from the state with the highest finishing time. While there still are unvisited states, call a new *DFS* procedure starting in the remaining state with the highest finishing time.
- Each DFS tree resulted in the second traversal is a strongly connected component of \mathcal{S} .

The correctness of this algorithm, proved in [AHU83], relies on the key observation that all the states of a strongly connected component are contained in the same DFS tree. But DFS is a typical difficult-to-parallelize-efficiently algorithm [Rei85]. In the remainder of this chapter we propose a solution consisting of some local and global transformations on the distributed graph, that are less time-expensive than a direct distributed implementation of the Tarjan algorithm.

5.1.3 Distribution of the state space

In the rest of the exposition we will talk about a fixed state space \mathcal{S} and we assume a distribution of \mathcal{S} on W machines:

$$S = S_0 \cup \dots \cup S_{W-1} \text{ and } \forall i \neq j : S_i \cap S_j = \emptyset.$$

$$T = \bigcup_{0 \leq i, j < W} T_{ij}, \text{ where } T_{ij} = \{(x, y) \in T \mid x \in S_i \text{ and } y \in S_j\}$$

The machine (or: processor, worker) i *owns* the states S_i and the transitions $(\forall j)T_{ij}$. The state spaces are produced in this format by the distributed generation tool [BLL03] from the μCRL toolset. We also assume a globally known hash function $\text{worker} : S \rightarrow \{0 \dots W - 1\}$ that indicates to which worker every state belongs. In our implementations the states are identified by pairs (worker, offset), thus the **worker** function is just a projection.

We call transitions that cross worker boundaries (i.e. in T_{ij} with $i \neq j$) *cross transitions*. The performance of most algorithms on distributed state spaces is influenced by the number of cross transitions. Ideally, we would like to have the state space

distributed in such a way that the number of cross transitions is (much) smaller than that of inside transitions (in T_{ii}), while the number of states owned by different workers is almost the same (for all i, j , small $|S_i - S_j|$). Finding such a distribution is a difficult problem, therefore in reality we work with a random balanced distribution, that ensures about the same number of states to every worker but does not try to optimize the number of cross transitions.

5.2 Graph transformations

5.2.1 Identify atomic components

Usually, a state space will contain a lot of states that do not connect via cycles with any other state. That is, they are SCCs on their own. We call these states *atomic SCCs* and we describe now a simple procedure to discover some of them. We start by the states with incoming degree 0. They are for sure atomic components, since otherwise the state would be reachable from other states. This also means that their outgoing transitions are not internal to a component, and therefore their presence does not change the SCC structure of the graph. Thus, we may remove the states without incoming transitions, together with all their outgoing transitions. This step can be repeated until all states have at least one incoming transition. Since at every step we only have to look at the states with no incoming transitions and their successors, this procedure is quite cheap (one BF pass) and allows for much parallelism. Section 5.3.3 presents it in more detail.

5.2.2 Partial SCC detection

The very efficient (linear) Tarjan DFS algorithm (Section 5.1.2) can be exploited in a distributed environment as well, in several ways. One possibility is to let it perform on the local subgraph (S_i, T_{ii}) of each processor, in order to find and collapse the local components. For each component, one of the states, say x , is chosen as representative and all the others (y) are identified with it by means of the `scc` function: `scc(y) := x`. Then all transitions in the global graph have to be renamed from (x, y) to $(\text{scc}(x), \text{scc}(y))$.

The other extreme application of `SCCTarjan` on a distributed state space is to send the whole graph to one manager worker that will then compute its SCCs using the sequential algorithm and send back the correct values of `scc`. This is of course only possible when the global graph is – or has become, by means of other transformations – sufficiently small.

A good idea for when the global graph is not small enough and the elimination of local components does not shrink it substantially, is an intermediate approach: apply the collapse-global-components transformation on disjoint subsets of workers, in parallel. This way, the managers get a smaller global graph (sometimes, this makes

the difference between not-feasible and feasible). Moreover, the chance of finding components is higher than when collapsing locally. By repeatedly collapsing SCCs on random small sets of workers, we hopefully arrive at a global graph that is small enough to be further reduced on one worker. This procedure is discussed in more detail in Section 5.3.4.

5.2.3 Coloring

By a certain coloring of the states of the graph, a partition of the set of states can be achieved, such that if x and y are in the same SCC then x and y have the same color. This splits the SCC problem in smaller disjoint instances.

The coloring procedure starts with a color function $c : S \rightarrow \mathbf{N}$ satisfying the property

$$\forall x, y \in S \text{ if } c(x) = c(y) \text{ then } (x, y) \in T^* \cap T^{*-1} \quad (\text{safety})$$

If there is no a priori information available that allows the fast construction of such a c , we can choose the identity function, which trivially satisfies the condition. We assume an order on the colors, $<$. The coloring procedure consists in successively modifying c until no modification is possible anymore. At each modification step, every state x passes its color to every successor y for which $c(x) < c(y)$. When the coloring is done, the transitions having their source colored differently than their destination are definitely between components (see Section 5.3.6 for the justification) and, consequently, they get removed. The result is a set of disconnected and smaller state spaces, each of them uniformly colored. Note also that every small state space has one or more special states that kept their initial color – let us call them *roots*.

We can now focus on solving separately the subproblems determined by colors. The final scc mapping is simply the union of the scc sub-mappings thus resulted. Optionally, we can first exploit the colors somewhat more, by finding and extracting some SCCs. Pick a root x . Since the initial coloring was safe, the states wearing x 's color are precisely those reachable from x (Section 5.3.6 contains more details). Thus, the states belonging to the SCC of x are those colored the same as x and that can reach x .

The distributed implementations of the coloring procedure and of the procedure for extracting the roots' components are described and discussed in Sections 5.3.6 and 5.3.7, respectively.

5.2.4 Eliminate reflexive and multiple transitions

As a result of other transformations, transitions of the form (x, x) and multiple occurrences of the same transition can appear, that have no influence on the SCC. Eliminating these reduces the size of the graph. Since for every state all the outgoing

transitions are kept on the same worker, this is a simple local operation and requires no network communication. Therefore, from now on we will ignore it.

5.3 Distributed implementation of the transformation routines

In this section, the distributed implementations of some of the transformations introduced above are presented: the elimination of atomic SCCs (Figure 5.1), the partial SCC detection (Figure 5.2), coloring (Figures 5.4, 5.5) and an auxiliary routine that includes elimination of reflexive and multiple transitions (Figure 5.3).

5.3.1 Distributed data structures

The basic data structures are *sets* and *lists*. On sets, the usual set union, intersection and difference (\cup , \cap , $-$) are defined. Lists are sequences $X = /x_1.x_2.\dots.x_n/$. We use pairs of lists of equal size to implement relations. For example, the relation $\{(1,1), (2,1), (2,5)\}$ is represented as $(/1.2.2/, /1.1.5/)$. This is convenient when sending/receiving buffered messages (see Section 5.3.2). We consider the following notations, operations and predicates for lists:

$/\ /, /x/$	the empty list and the singleton list, respectively
$X[i]$	the element at position i in the list X
$X.Y$	list concatenation
(X, Y)	pair of lists with the same length
$X - x$	remove all occurrences of x in X
$X + x$	$X./x/$
$(X, Y) + (x, y)$	$(X + x, Y + y)$
$x \in X$	there is at least an occurrence of x in X
$(x, y) \in (X, Y)$	there is at least a position i such that $X[i] = x$ and $Y[i] = y$.

We will also use natural extensions of the function `scc` to sets and lists as follows. Let S be any set and X any list. Then

$$\begin{aligned}
\text{scc}(S) &\stackrel{\text{def}}{=} \{\text{scc}(x) \mid x \in S\} \\
\text{scc}(/ /) &\stackrel{\text{def}}{=} / / \\
\text{scc}(/x/.X) &\stackrel{\text{def}}{=} /\text{scc}(x)/.\text{scc}(X)
\end{aligned}$$

We consider the current state space (S, T) , the final set of transitions (`finalT`)

and the current `scc` values as global variables. Every transformation expects them to have a special form, expressed by a *precondition*, and modifies them such that a *postcondition* is ensured.

The worker i maintains the following data:

- S_i = the set of owned states.
- $\text{scc}(S_i)$ = the current scc mapping of the owned states. scc is initialized as identity.
- $T_{ij} = (\text{Source}_{ij}, \text{Dest}_{ij})$, for every worker j , including itself. The set of transitions with the state source owned by worker i and destination owned by worker j is implemented as a pair of lists, one containing the source states (Source_{ij}) and the other the destinations (Dest_{ij}). The order is the same for both. So, $(x, y) \in T_{ij}$ if and only if there is an index p s.t. x is the p th element in Source_{ij} and y the p th element in Dest_{ij} .
- finalT_{ij} = the transitions that are definitely in the final set of transitions, but possibly with another numbering. More precisely, if $(x, y) \in \text{finalT}_{ij}$ then $(\text{scc}(x), \text{scc}(y))$, with scc the final mapping, will be a transition in the SCC-reduced state space.

We make the convention that all sets or lists occurring in the pseudo-code descriptions that are not listed as input, are considered initialized as \emptyset and $/$ /, respectively.

5.3.2 Communication primitives

As already said in Section 2.2.5, our target architecture is a cluster whose nodes are connected by a high bandwidth network (Distributed Memory Machine). Our cluster is not massively distributed, meaning that the number of nodes available is always much smaller than the problem size ($W \ll N$). Processes communicate by executing nonblocking send/receive operations:

- **SEND** m **TO** i - message m to worker i .
- **RECEIVE** m **FROM** i - message m from worker i .

We also use the following two communication patterns, that are easily implementable with the above primitives:

- **DBSUM**($x_i, xall$) - for an arbitrary set of local values $x_0 \cdots x_{W-1}$, collectively compute their sum into a global value $xall$ and store a copy of the result on each worker. This is useful when all the workers are executing a loop and the exit condition depends on a global value.
- **REQ-REP** $j : B := f(B)$ - the worker executing this pattern (i) sends a buffer (request) B to the worker j (not necessarily $\neq i$) and expects j to do the same, i.e. send to i a buffer B' . Upon receiving B' , i creates a list C' of the same length as B' , where if $B'[t] = x$ then $C'[t] = f(x)$. Then it sends C' to j and waits for a similar reply from j , i.e. a list C . In the end, i replaces B with

```

elim-atomic-fwd
Postcondition:  $\forall x \in S \exists y \in S \text{ s.t. } (y, x) \in T$ .
(i.e. all trivial components  $\{x\}$  reachable from a start node (node with incoming degree 0)
have been identified and removed)
1  /* compute all the incoming degrees */
2  for  $x \in S_i$  do  $\text{indegree}[x] := 0$  enddo
3  for all workers  $j$  do
4      SEND  $\text{Dest}_{ij}$  TO  $j$  || RECEIVE  $\text{Dest}_{ji}$  FROM  $j$ 
5      for  $x \in \text{Dest}_{ji}$  do  $\text{indegree}[x] := \text{indegree}[x] + 1$  enddo
6  enddo
7  /* loop: eliminate all the initial states */
8  /* and their outgoing transitions */
9  while there still are states with indegree 0 do
10     for all workers  $j$  :  $B_{ij} := \emptyset$ 
11     for  $x \in S_i$  :  $\text{indegree}[x] = 0$  do
12          $S_i := S_i - \{x\}$ 
13         for  $(x, y) \in T_{ij}$  do
14              $T_{ij} := T_{ij} - (x, y)$ 
15              $\text{finalT}_{ij} := \text{finalT}_{ij} + (x, y)$ 
16              $B_{ij} := B_{ij} + y$ 
17         enddo
18     enddo
19     for all workers  $j$  do
20         SEND  $B_{ij}$  TO  $j$  || RECEIVE  $B_{ji}$  FROM  $j$ 
21         for  $x \in B_{ji}$  do  $\text{indegree}[x] := \text{indegree}[x] - 1$  enddo
22     enddo
23 enddo

```

Figure 5.1: Forward BFS pass in order to identify atomic components and final transitions (worker i)

the newly received list C and j replaces B' with C' . Thus, the effect of the request-reply action is $B := f(B)$ and $B' := f(B')$. The implementation with send/receive:

$$\begin{array}{lll}
 \text{SEND } B \text{ TO } j & \text{and} & \text{RECEIVE } B' \text{ FROM } j \\
 \text{SEND } f(B') \text{ TO } j & \text{and} & \text{RECEIVE } f(B) \text{ FROM } j
 \end{array}$$

Occasionally, in the actual implementation we also used MPI primitives that are more powerful than simple sends and receives. An example is `MPI_AlltoAll`, that transfers data, in parallel, from every worker to every worker in a careful order so as to avoid deadlocks. To keep the presentation simple, we abstract away from these details, and base the exposition of the distributed procedures only on the primitives above.

5.3.3 Distributed identification of atomic components

Figure 5.1 shows a distributed routine that finds and removes some of the atomic SCC, namely those reachable only from states without predecessors. Workers begin by computing together (steps 2-6) the incoming degrees of all states, *indegree*. As explained in Section 5.3.1, transitions T_{ij} are stored by worker i as a pair of lists ($\text{Source}_{ij}, \text{Dest}_{ij}$). The incoming degree of an arbitrary state $x \in S_j$ is the number of transitions that have x as destination state and it is easily computed by counting the occurrences of x in all lists Dest_{ij} . To this end, every list Dest_{ij} is sent to worker j (step 4), where the number of occurrences is updated (step 5). Thus, there will be one message for every pair of workers, and the destination state of every transition will be transferred once. Therefore, the message complexity of computing the incoming degrees is $\mathcal{O}(W^2)$ and the bit complexity $\mathcal{O}(M)$. The time complexity, under the balanced distribution assumption, is $\mathcal{O}((M + N)/W)$.

In the second part (steps 9- 23), all states without incoming transitions are marked as atomic SCCs – in non-atomic SCCs, every state is reachable from any other, therefore it must have at least one incoming transition. Further, any transition with an atomic SCC as source will not be on a path inside an SCC, therefore removing it doesn't influence the scc partition (steps 14, 15). Then the destination state of such a transition has to have its incoming degree updated. This happens in steps 19-22. The total size of the buffers being exchanged in the **while** loop is at most M . As for the total number of messages: in the worst case, every buffer gets always only one transition, which leads to a message complexity of $\mathcal{O}(M)$.

In order to detect as many such atomic components as possible, this procedure should be executed with regard to both forward and backward transitions. We have only discussed the forward pass. The backward pass can be implemented by reversing the graph (see steps 2 -5 in the heads-off routine, Figure 5.5) and calling the forward pass (with the subtle difference that the transitions marked as final should be reversed again).

Note that this procedure is sound, but not complete. The states placed “in between” two cycles will never get the degree 0 as long as the cycles are still in place.

5.3.4 Partial SCC detection

For most distributed graphs, and definitely for distributed state spaces, it is usual that many of the SCCs do not span over all workers, but over a small subset. Figure 5.2 shows three variants of a transformation that employs the very efficient sequential algorithm based on DFS (see Section 5.1) in order to detect and collapse this kind of small SCCs. Let **SOME** be the subset of workers under consideration and let $\mathcal{S}_{\text{SOME}} = (S_{\text{SOME}}, T_{\text{SOME}})$ be the subgraph induced by the states owned by workers in **SOME**. The idea is to simply send $\mathcal{S}_{\text{SOME}}$ to a special worker **M** (step 3), that will locally compute the scc function (step 7) and send it back (step 8) to the workers in **SOME**. Since

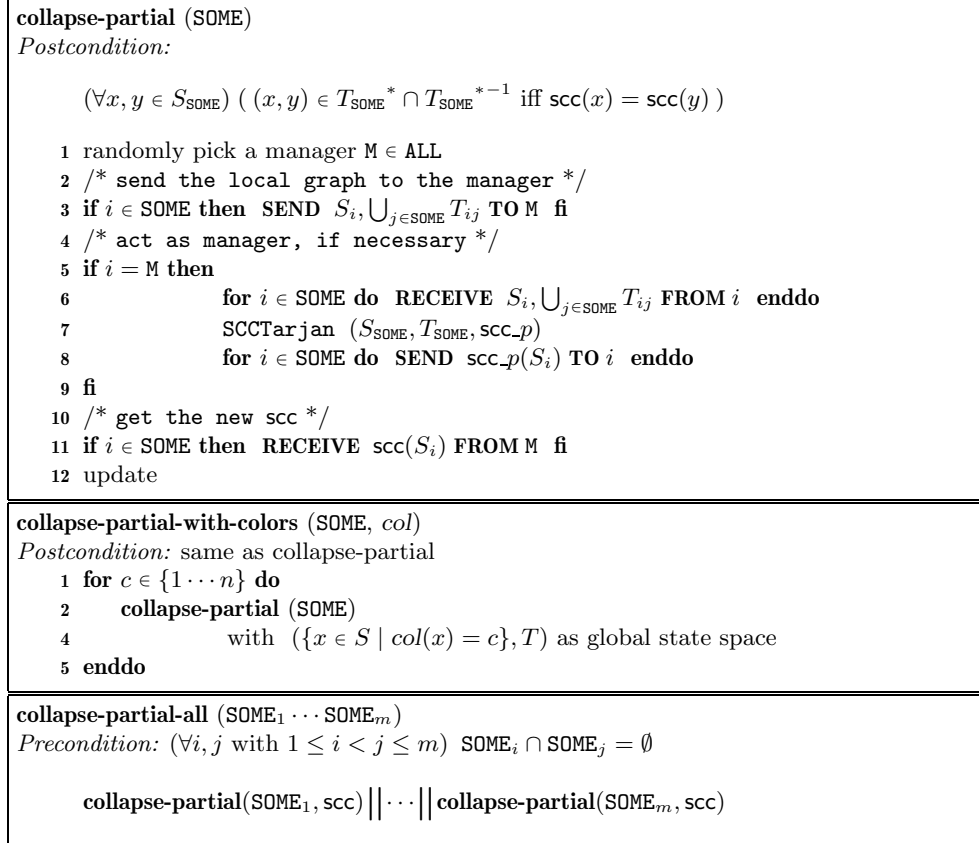


Figure 5.2: Partial SCC detection

disjoint subsets of workers generate disjoint subgraphs, the partial SCC reduction can be executed in parallel on more subsets of workers (**collapse-partial-all**).

The number of messages used in **collapse-partial** is $\mathcal{O}(\text{card}(\text{SOME}))$, with a total size of $\mathcal{O}(\text{card}(S_{\text{SOME}}) + \text{card}(T_{\text{SOME}}))$. The time complexity is also $\mathcal{O}(\text{card}(S_{\text{SOME}}) + \text{card}(T_{\text{SOME}}))$.

Another variant of this routine is **collapse-partial-with-colors** that uses the color information (see Section 5.3.6) to partition the graph $\mathcal{S}_{\text{SOME}}$ into smaller graphs $\mathcal{S}_{\text{SOME}}^1, \mathcal{S}_{\text{SOME}}^2 \dots \mathcal{S}_{\text{SOME}}^n$ and process them independently (sequentially). The independence stems from the fact that the states wearing different colors are definitely not in the same component. This can be exploited to save memory, by letting the manager solve them sequentially and thus keeping the manager's load low (in Figure 5.2). Alternatively, one could gain time by collapsing the small graphs in parallel and thus distributing the manager role among the workers.

```

update

1 /* update and migrate the transitions */
2 for  $j \in \text{ALL}$  do REQ-REP  $j : \text{Dest}_{ij} := \text{scc}(\text{Dest}_{ij})$  enddo
3 for  $j \in \text{ALL}$  do  $\text{Source}_{ij} := \text{scc}(\text{Source}_{ij})$  enddo
4 for  $j \in \text{ALL}, (s, d) \in (\text{Source}_{ij}, \text{Dest}_{ij})$  do
5    $k := \text{worker}(s)$ 
6    $l := \text{worker}(d)$ 
7    $(\text{Source}_{ij}, \text{Dest}_{ij}) := (\text{Source}_{ij}, \text{Dest}_{ij}) - (s, d)$ 
8    $(BS_{ikl}, BD_{ikl}) := (BS_{ikl}, BD_{ikl}) + (s, d)$ 
9 enddo
10 for  $k, l \in \text{ALL}$  do
11   SEND  $BS_{ikl}, BD_{ikl}$  TO  $k$ 
12   RECEIVE  $BS_{kil}, BD_{kil}$  FROM  $k$ 
13    $\text{Source}_{il} := \text{Source}_{il}.BS_{kil}$ 
14    $\text{Dest}_{il} := \text{Dest}_{il}.BD_{kil}$ 
15 enddo
16 for  $s \in S_i$  do
17   if  $\text{scc}(s) \neq s$  then  $S_i := S_i - \{s\}$ 
18   enddo

```

Figure 5.3: Update

5.3.5 Update

The transformations described until now only assign values to the `scc` function, without actually replacing x with `scc(x)`, that is without renaming the transitions from (x, y) to $(\text{scc}(x), \text{scc}(y))$. This is the role of the **update** routine (Figure 5.3). For any transition (x, y) , first the destination state y is replaced by `scc(y)` (step 2), then the source state (step 3) and finally the updated transitions are moved to their new owners (steps 4-15).

5.3.6 Reducing the problem by coloring

Figure 5.4 shows a distributed procedure that takes a color function col on the states of a graph and modifies it repeatedly until it stabilizes, that is until $col[x] \geq col[y]$ for every transition (x, y) . The modifying step identifies the transitions (x, y) that do not conform to this condition and copies the color of the parent to the child.

Let $(S^{start}, T^{start}), col^{start}$ be a state space and an initial color function and let $(S, T), col$ be the state space and color function after the colorify action. Let us also define a set $\text{Roots} = \{x \in S \mid col^{start}(x) = col(x)\}$. The following facts are true:

- every SCC in T^{start} (and T) is painted uniformly by col
 Proof: at the end of the painting procedure, $col[x] \geq col[y]$ for every transition (x, y) . This means that $col[x] \geq col[y]$ for any path $x:y$. If two states x and

<pre> colorify (col^{start}) <i>Precondition:</i> (safe) if $col^{start}(x) = col^{start}(y)$ then $x:y:x$ <i>Postcondition:</i> $\forall x, y \in S \forall (x, y) \in T \ col(x) = col(y)$ 1 $col := col^{start}$ 2 /* loop: the color of the parent propagates */ 3 /* to the child, if the color of the child is weaker */ 4 DBSUM($card(S_i), totalC$) 5 $Changed := S_i$ 6 while $totalC > 0$ do 7 $newC := \emptyset$ 8 for all workers j do 9 $B_{ij} := \{(y, col[x]) \mid (x, y) \in T_{ij} \text{ and } x \in Changed\}$ 10 SEND B_{ij} TO j RECEIVE B_{ji} FROM j 11 for $(y, c) \in B_{ji}$ do 12 if $(c < col[y])$ 13 then 14 $col[y] := c$ 15 $newC := newC \cup \{y\}$ 16 fi 17 enddo 18 enddo 19 $Changed := newC$ 20 DBSUM($card(Changed), totalC$) 21 enddo 22 /* mark as final all the transitions between */ 23 /* states of different colors */ 24 for all workers j do 25 REQ-REP $j : Dest_{ij} := col(Dest_{ij})$ 26 for $(x, y) \in T_{ij}$ s.t. $col[x] \neq col[y]$ do 27 $T_{ij} := T_{ij} - \{(x, y)\}$ 28 $finalT := finalT \cup \{(x, y)\}$ 29 enddo 30 enddo 31 return col </pre>	returns col
---	---------------

Figure 5.4: Graph coloring (worker i)

```

heads-off (col, Roots)
Precondition:  $\forall (x, y) \in T \text{ } col(x) = col(y)$ 
                $\wedge \forall x \in S \exists \text{ a unique } r \in \text{Roots } col(x) = col(r)$ 
1  /* reverse the transitions */
2  for all workers j do
3    SEND Sourceij, Destij TO j
4    RECEIVE Destij, Sourceij FROM j
5  enddo
6  /* paint the roots with their old color */
7  for x  $\in S_i$  do c[x] := N + 1 enddo
8  for r  $\in \text{Roots}$  do c(r) := col(r) enddo
9  c := colorify(c)
10 for x  $\in S_i$  do
11   if c(x) = col(x)
12   then
13     scc(x) := the unique r  $\in \text{Roots}$  s.t. col(r) = col(x)
14   fi
15 enddo
16 /* reverse the transitions again */
17 for all workers j do
18   SEND Sourceij, Destij TO j
19   RECEIVE Destij, Sourceij FROM j
20 enddo
21 update

```

Figure 5.5: Elimination of root components (worker *i*)

- y* are in the same strongly connected component then there are paths *x*:*y* and *y*:*x*. Thus $col[x] \geq col[y]$ and $col[y] \geq col[x]$, hence equal.
- if *x*:*Ty* then $col(y) = col(x)$
 Proof: At the end of the coloring procedure (Figure 5.4, steps 24-30), all the transitions (*x*, *y*) with $col(x) \neq col(y)$ are eliminated.
 - if col^{start} is safe then: if *x* $\in \text{Roots}$ then $col(y) = col(x)$ iff *x*:*Ty*
 Proof: Since col^{start} is safe, all the states *z* with $col^{start}(z) = col^{start}(x)$ must be on a cycle with *x*. If $col(y) = col(x)$ then there is a path (possibly empty) from one of these states to *y*, because the colors propagate only on paths. It follows that there is also a path from *x* to *y*. The other way was proved at the previous point.

These observations justify the claim that the final coloring partitions *S* into subsets $S^0 \dots S^{n-1}$ such that any strongly connected component from the initial graph is completely contained in one of the subgraphs induced $(S^0, T^0) \dots (S^{n-1}, T^{n-1})$. Solving the problem of detecting the strongly connected components in the initial graph reduces to solving it for the *n* subgraphs. Moreover, the subgraphs are actually

the forward reachability sets of a few selected states (roots). The colorify routine finishes in about $M * \text{diameter}$ steps – this is quite OK for state spaces, that usually have a very small diameter.

5.3.7 Heads off

This routine gets as input a coloring of the state space together with a set of *roots* (one root per color) with the property that every root can reach all (and only) the states painted in its color. This means that the states that are reachable from their root also on backward paths, form the root's strongly connected component. An easy way to compute the backward reachable states is reversing the state space (steps 2- 5) and coloring it again, with an initial color function that leaves all the non-root states unpainted. The nodes that get painted in this new coloring round are in their root's SCC and can be marked as such – and removed from S . In the end, the state space gets back to the original orientation (17- 20).

5.4 Three example algorithms

The intended use of the graph transformations described until now is as building blocks for algorithms that compute `scc`. Extra information on the structure of the graph can help in choosing an optimal combination of transformations. Note that every transformation eliminates some of the states and transitions, either by collapsing SCCs or by proving that certain states are atomic or certain transitions are definitely between different components. When discovered, the atomic states are thrown away and the transitions crossing components boundaries are stored in the set `finalT`. After a number of transformations, the set of transitions left in the state space will drop to \emptyset . At that moment, `scc` and `finalT` define the reduced graph, which is the initial one modulo the strong connectivity equivalence relation. But it is possible that `scc` of some states does not hook them directly to their head of component, but via some intermediate states. To get the final `scc` definition, a *flattening* phase must be performed, at the end of which $\forall x \in S : \text{scc}(\text{scc}(x)) = \text{scc}(x)$. The distributed implementation of this phase uses just one BFS pass of the graph. This is possible because throughout all the transformations, the following invariant is preserved:

$$\forall x \in S \exists \text{ a unique } y \in S \text{ s.t. } \text{scc}(y) = y \wedge \text{scc}(\text{scc}(\dots \text{scc}(x))) = y.$$

After flattening, the state space without cycles is:

$$\begin{aligned} S^{\text{scc}} &= \{\text{scc}(x) \mid x \in S\} \\ T^{\text{scc}} &= \{(\text{scc}(x), \text{scc}(y)) \mid (x, y) \in \text{finalT}\} \end{aligned}$$

We describe below three SCC reduction algorithms based on the transformations

Extreme 1:
<pre> elim-atomic groupsize = 1 while groupsize < W do partition ALL in groups of size (at most) groupsize : ALL := $\bigcup_{0 \leq i < W/\text{groupsize}} \text{SOME}_i$ if $\exists i : \sum_{j \in \text{SOME}_i} \text{card}(T^j) > \text{MAX}$ then ERROR : group too large else collapse-partial-all (SOME₀ ... SOME_m) fi groupsize := 2 * groupsize enddo </pre>
Extreme 2:
<pre> while (T ≠ ∅) do elim-atomic c := colorify (Self) Roots := {x ∈ S c(x) = x} heads-off (c, Roots) enddo </pre>
Combination:
<pre> elim-atomic while (card(T) > MAX) do c := colorify(Self) Roots := {x ∈ S c(x) = x} if card(Tⁱ) ≤ MAX then collapse-partial(ALL, 0) else if (∃ Tⁱ : card(Tⁱ) > MAX) then heads-off(c, Roots) else collapse-partial-with-colors(ALL) fi fi enddo collapse-partial(ALL, 0) </pre>

Figure 5.6: Three example algorithms

proposed (Figure 5.6). A constant **MAX** is needed to specify the maximum load (in number of transitions) that a worker can handle.

Extreme 1 Our first algorithm is aimed at speed. It uses a series of collapse-partial-all calls to reduce quickly the size of the distributed graph. The series begins with finding, in parallel, the SCCs on the subgraphs local to every worker (level 0, collapse-partial-all with groupsize 1). Then the groups of workers double in size every step, until only one group including all the workers is considered. If at any step the maximum load is reached, the algorithm stops with an error. This approach will work well for relatively small state spaces and for dense ones, with many small cycles inside of larger ones.

Extreme 2 Our second example algorithm uses only the color-based transformations. **Self** denotes the identity function, $\text{Self}(x) = x$. The algorithm repeatedly colors the state space starting with **Self** as initial color function and extracts the head components. Note that, with **Self** as initial color, every color gets a unique root. This algorithm may in general be slower, but it always terminates successfully, because its memory usage stays more-or-less constant and, moreover, the buffers can be restricted to a convenient size (while in the case of Extreme 1, the manager has to be able to simultaneously store the local graphs of several workers in its memory).

Combination A possible hybrid algorithm uses the technique of partitioning by colors only until the graph pieces are small enough to be collapsed by the groups technique.

5.5 Experiments

We discuss some aspects of the performance of our cycle elimination algorithms on a series of large distributed state spaces generated for the verification of a system for lifting trucks [GPW03] (*lift5*, *lift6*), a cache coherence protocol [PFHV03] (*cache*), some instances of the Sokoban game [sok] (*screen.706*, *screen.801*, *screen.1*) and a sliding window protocol [FGP⁺04] (*swp-piggy*). We also included two state spaces without invisible cycles from the VLTS [CI] benchmark (*vasy-8082*, *vasy-4338*). The problem sizes and some other relevant structural characteristics are summarized in Figure 5.7. The reduction times presented in Figure 5.8 are recorded on a cluster of 4 dual AMD Athlon MP 1600+ nodes with 2G memory each, running Linux and connected by Gigabit Ethernet.

The CE1 and CE2 columns in Figure 5.8 show the runtimes of the first two algorithms, not including the input/output operations. In order to justify that the cycle elimination algorithms can be useful as preprocessing step for branching bisimulation reduction, we also show the runtimes of a distributed branching bisimulation reduction tool [BO03a] on the original state spaces and on the SCC-reduced state spaces.

state space	size of the τ -graph			size reduced		trivial SCCs (N%)	largest SCC
	N (in 10^6)	M (in 10^6)	M%	N%	M%		
cache	7.8	22.8	38.6	99.6	99.7	99.5	248
lift5	2.1	3.8	43.9	99.9	99.9	98.9	165
lift6	33.9	74.1	44.8	99.9	99.9	99.8	486
screen.706	1.2	2.37	87.8	11.4	0	6.6	38
screen.801	20.7	44.9	90.3	9.2	0	4.3	50
screen.1	29.9	65.9	91.2	7.4	3.9	1.2	50
swp-piggy	9.6	30.9	57.9	24.9	46.2	10.5	45
vasy_4338	4.3	3.1	19.9	100.0	100.0	100.0	1
vasy_8082	8.0	2.5	5.9	100.0	100.0	100.0	1

Figure 5.7: Some case studies: size, structure

state	CE1	CE2	BB after CE	BB original
cache	45	47	1331	1394
lift5	8.6	14.7	79	86
lift6	160	305	930	1039
screen.706	7.7	12.4	9.6	106.8
screen.801	172	112.5	43.6	2819.2
screen.1	210	121	36.7	180 000
swp-piggy	125	237	122	341
vasy_4338	6	6	82	82
vasy_8082	11	11	27	27

Figure 5.8: Reduction times (in seconds)

The important observation here is that the cycle reduction times are usually much smaller than the branching bisimulation reduction times. This means that although the cycle elimination step is not always advantageous, it is also not harmful and could be always done, just in case it might provide a spectacular gain (like for the Sokoban screens). Moreover, the current branching bisimulation algorithm is not optimized for the case when the input state space is guaranteed not to contain cycles. There is much space for improving in this direction and then the small penalty paid for eliminating the cycles would completely pay off.

The very low runtime of CE on the two *vasy* case studies is due to elim-atomic, which discovers in one pass that all the states are atomic SCCs. The number of needed color iterations in E2 depends on the diameter of the graph, which is usually rather small for state spaces.

5.6 Conclusions

The cycle detection problem plays an important role in verification algorithms, both explicit and symbolic. In this chapter we concentrated on distributed algorithms for explicit verification. We investigated some practical solutions to the problem of finding strongly connected components in very large distributed state spaces. Given that the best single-threaded solution is not efficiently parallelizable, we proposed two (orthogonal) heuristics that approach the problem by reducing it to smaller instances, solvable in parallel.

The first idea is to use the Tarjan sequential SCC detection algorithm on groups of workers. The local graphs of several workers get sent to a manager, where the combined subgraph is solved sequentially. Depending on the size of the workers' group, this procedure ranges from solving all local subgraphs in parallel to solving the whole global graph. Due to memory limitations, this approach is obviously not always successful. Therefore we also proposed an alternative solution, based essentially on computing forwards and backwards reachability sets. A third heuristics that proved very helpful uses the observation that if a state is on a non-trivial cycle then it must be reachable from at least one other state placed on a cycle. Thus, the states without parents cannot be on cycles. They are their own (atomic) SCC, and so are also all the states reachable only from atomic SCC. Removing these states is quite cheap and the state space usually becomes significantly smaller. The same operation applied on the reversed state space reduces it even more. The most spectacular effect of eliminating atomic components can be seen on state spaces without cycles, when the SCC detection finishes in one pass, thus linear time.

The reasonable performance of our algorithms demonstrates that cycle elimination on very large distributed state spaces is feasible and useful.

Part II

Verification of Data Distribution Architectures

6

Shared Dataspace Software Architectures

The complexity of designing a distributed system is generally managed by introducing a *software architecture*, defining what the components of the system are and how they are coordinated. The component layer and the coordination layer are usually designed, analyzed and implemented separately, in order to efficiently address issues like reusability, compositionality, maintainability and verification.

Coordination models are either *data-driven*, if the components communicate through a common pool of data (e.g., Linda [CG89]), or *control-driven*, if the components communicate by adhering to some communication patterns (e.g., Manifold [BAdB⁺00]). A comprehensive survey of coordination models can be found in [PA98]. In this thesis, we are only interested in data-driven coordination models. However, the formal verification techniques that we advocate are also appropriate for analyzing control-driven models. In fact, this is already being done [MSA04].

In this chapter, we briefly review a few coordination models based on the shared dataspace model. In particular, we introduce and discuss Splice, the software architecture that provided the main inspiration for the research questions in Part II. We also give a quick introduction to the algebraic specification language μCRL [GR01], as it plays an important role in the next two chapters.

6.1 Some distributed shared dataspace coordination models

Linda [CG89] was the first coordination language based on shared data. It is based on generative communication: components communicate by storing and retrieving tuples from a global shared tuple space. The most convenient characteristics of this simple mechanism are the time decoupling and the anonymity of components. Tuples can be both passive (data) and active (executable code). The retrieval works through associative pattern matching.

Bonita [RW97] is a successor of Linda and improves both its functionality and performance. Bonita extends the Linda primitives to work with multiple tuple spaces. For parallel performance reasons, the retrieving operation in Bonita is split into a *request* and an *obtain* part.

KLAIM [DNFP98] (a Kernel Language for Agents Interaction and Mobility) is another successor of Linda designed for distributed environments. Its distinctive feature is the use of explicit *localities* in the primitives. This enables the programmer to distribute the space among different sites.

WCL [Row98] enriches the basic set of Linda primitives, by providing both asynchronous and synchronous tuple space access, bulk primitives and streaming primitives. The novelty of this coordination language is that it specifically targets the situation of geographically distributed agents. To this aim, it supports location transparency and dynamic analysis of the tuple space (in order to achieve efficiency by migrating data).

JavaSpaces [FHA99] This rather new coordination language, developed by Sun, uses one central global dataspace and offers a whole range of useful new primitives: insertion, destructive and non-destructive lookup of shared objects, distributed event notifications, transactions, leasing. In order to fix a clear semantics for the new primitives, a process calculus was proposed in [BGZ00a] and some consequences of choosing different semantics were analyzed. The semantics of JavaSpaces has been formalized in μ CRL as well [PV02, PV03], with the goal of allowing automatic verification. Several interesting applications were model-checked on this specification.

6.2 Views and models of Splice

Splice [Boa93] is a data-oriented software architecture for complex control systems, developed and used at the company Hollandse Signaalapparaten BV (currently Thales Nederland). It is based on the publish-subscribe paradigm and it uses some interesting mechanisms (keys, lifecycles, timestamps, joins) to control the distribution of data from publishers to subscribers. The architecture of a Splice system consists of a (variable) number of *agents* connected by an Ethernet *network*. Each application has its own agent, that acts as the application's buffer to the network and manages its local *database* of data items. The applications communicate with the agents using Splice primitives like *read*, *write*, *subscribe*, *publish*. The Splice system takes care of asynchronously transferring data from the databases of publishers to those of subscribers. The communication is decoupled and anonymous, which allows components to join or leave a Splice system at run-time.

Splice has been regarded and discussed from many perspectives. Depending on the aspects that were put on the foreground, it has been referred to as publish-subscribe software architecture, data distribution middleware, data-driven coordination model, (real-time) (distributed) shared dataspace, or a combination of these. Roughly three categories of questions concerning Splice have been addressed in the literature:

Models and formal semantics A dedicated process algebra for Splice has been defined in [DGJU99]. In [DL00], a detailed μ CRL specification of Splice is reported, where the Ethernet layer is modeled, but the data aspects are not of prime importance. A later μ CRL model of Splice [HP02b] is much simpler and focuses on more relevant functional aspects. Both models contribute to defining a much needed formal semantics to the Splice primitives. Also a denotational semantics has been developed for use in compositional verification and refinements and it was proved equivalent to the operational semantics [HP02a].

Expressiveness of Splice-like models Expressiveness of coordination models that use a shared dataspace has been extensively investigated. [BHJ00, BKZ99a, BKZ99b] compare a number of such models that differ in choosing a set or a multiset as data repository, using destructive or nondestructive primitives, having one global dataspace or few local caches etc. Chapter 7 is an expressiveness study on a very simple shared dataspace model, that reduces the set of coordination primitives to the extreme, namely it only uses *reads* and *writes*.

Verification The verification effort on Splice started with [DL00], where properties like deadlock freedom, soundness and weak completeness were checked in μ CRL + CADP. Theorem proving techniques (PVS) were used in [HH01], where a formal approach to the top-down design of components on Splice was presented. There the real-time characteristics play an important part. Both approaches, model checking with μ CRL and theorem-proving with PVS, were used to study transparent replication of applications on top of Splice [HP02b]. Replication possibilities on Splice were first explored in [DJ00].

6.3 Introduction to μ CRL

The process algebra approach to verification, taken by formalisms like Communicating Sequential Processes (CSP) [BHR84], Calculus of Communicating Systems (CCS) [Mil80], Algebra of Communicating Processes (ACP) [BK85], provide powerful means to represent and study behaviors, nondeterminism, parallelism, system equivalences, abstractions.

The specification language μ CRL [GR01] is the result of extending the process algebra ACP with abstract data types, in order to achieve sufficient expressiveness to

describe real-life applications.

In μCRL , processes are built from atomic actions by certain operators. μCRL inherited the typical process algebra connectives from ACP. For any processes p and q , $p + q$ denotes *non-deterministic choice* between p and q , $p.q$ denotes their *sequential composition*, and $p \parallel q$ denotes the *parallel composition* (defined in terms of interleaving and synchronous communication). The synchronization is allowed only between pairs of communicating actions, which are determined by a communication function γ . There is also an *encapsulation* operator ∂_H , that forces processes to communicate, by making the actions in H act exclusively in communication. The *hiding* operator (τ_I) abstracts away the actions in I . There are two special processes: δ (deadlock, the unit of $+$) and τ (internal action).

In order to use abstract data types in a specification, a signature of multiple sorts and functions can be declared, and axiomatized by equations.

The following connectives connect processes with abstract data types. First, atomic actions can be parameterized with data elements, as in $get(n)$. Then, $\sum_{n:D} P(n)$ denotes alternative (possibly infinite) choice over data domain D , i.e. for any value $d_0 \in D$, the process can behave as $P(d_0)$. Alternative choice is used to model input. Finally, if b is a term of data domain $Bool$ and p and q are processes, then the conditional $(p \triangleleft b \triangleright q)$ is the process “ p if b , else q ”. Conditionals are often used to put a restriction on the surrounding summation over data. In particular, $\sum_{n:Nat} get(n).P(n) \triangleleft n < 20 \triangleright \delta$ denotes the process that gets an arbitrary $n_0 < 20$, and continues as the process $P(n_0)$.

7

Expressiveness and Distribution of a Simple Shared Dataspace Architecture

Inspired by the industrial architecture Splice (introduced in Section 6.2), we study the consequences of choosing an extremely weak and simple coordination model: communication via a global set. The coordination primitives between components are restricted to writing and reading. We imagine that the application components reside at certain physical locations, abstractly represented by natural numbers. The coordination primitive $write(i, v)$ represents that the component at location i adds value v to the global set; if v is present already this action has no effect. The other primitive, $read(i, v)$ denotes a non-destructive, blocking read of a particular value (or template) v by a component at location i . That is, it waits until it actually finds v in the global set, and then proceeds. Note that test for absence and deletion of items is not possible.

Two separate tasks can be distinguished. First, the architecture must be implemented on a distributed network. Second, components must be designed that together implement the requirements of the system under design, using the coordination primitives provided by the architecture.

The two tasks raise two natural questions. The *first question* is whether the architecture itself has an efficient distributed implementation. This is addressed in Section 7.2. We define a distributed implementation of the architecture, in which every component has its own local set. Data items are exchanged between these local sets asynchronously. We prove that the implementation based on local sets is *behaviorally equivalent* to the specification based on a conceptual global set (that is, the bottom layer of Figure 7.1(b) is equivalent to the bottom layer of Figure 7.1(c)). The fact that the difference cannot be noticed is mainly due to the careful selection of the weak coordination primitives. This result is essentially the same as

in [BHJ00, BKZ99a, BKZ99b], albeit in a slightly more general setting. However, the proof is much simpler due to the application of powerful process algebraic proof principles.

The *second question* is whether the architecture is sufficiently expressive to allow the distributed implementation of any system specification. This is investigated in Section 7.3 from a functional point of view – i.e. without taking into account issues like performance or fault tolerance. We show that every specification of functional behavior has a distributed implementation, i.e. one where different types of actions are performed at different physical locations. In particular, the components only use the weak coordination primitives *read* and *write* on a global set. Figure 7.1(a),(b) gives a quick view on this implementation. As far as we know, this main result is not comparable to existing results on expressiveness of coordination models.

Example 7.1 *Consider a very simple logging system, with its behavioral specification input.log , indicating that some input action precedes some log action. This system probably uses two physical devices (e.g. a monitor and an actuator) with their own controllers, so input and log happen at different locations. A distributed implementation with our primitives could be: $\text{input.write}(l_1, d) \parallel \text{read}(l_2, d).\text{log}$. Here l_1 and l_2 are the locations of the components, and d is some data value. With \parallel we denote parallel composition. Assuming that the system starts with the empty data space, the second process is initially blocked, so the only execution of this little program should be $\text{input.write}(l_1, d).\text{read}(l_2, d).\text{log}$. If we hide the communication actions *read* and *write*, we indeed get the desired system behavior input.log . We remark that the system $\text{button}_1 + \text{button}_2$, in which non-deterministically either button_1 or button_2 is pressed, also has a distributed implementation, but this is much harder. In particular, our solution will use an unbounded number of internal communications.*

Finally, in Section 7.4, we connect the two results. First, the architecture with local sets (Figure 7.1(c)) is split into parallel agents (Figure 7.1(d)). Using these, and combining the results from Sections 7.2 and 7.3, we obtain for any requirements specification, a truly distributed implementation consisting of components communicating by synchronous message passing (Figure 7.1(e)). Each component consists of an application process and a local data space agent.

Relationship with Splice The choice of architecture in this chapter is influenced by Splice (Section 6.2). The advantage of the Splice architecture is that the components are loosely coupled, thus increasing the amount of fault tolerance. The data is present at several locations, making replication of components relatively easy. Recent research papers propose to view Splice conceptually as a shared data space, i.e. a set of data common to all components [BHJ00, DJ00]. Viewing the data as a global data space has the advantage that all programs perceive the same data at any moment. In

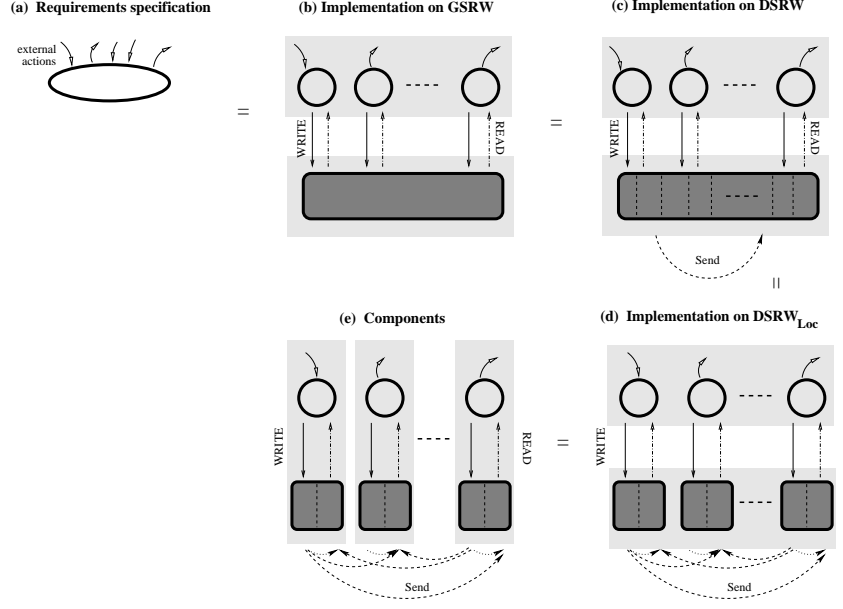


Figure 7.1: From requirements to a distributed implementation

addition, viewing it as a set (instead of a multi-set) opens the way to *transparent* replication of components [DJ00]. See Section 7.6.1 for further related work.

A Process-algebraic Approach A common theme has been to embed the coordination primitives in a host language and give semantics to the resulting coordination language. As an alternative, we adopt a process algebraic point of view. In this view everything is a process, or more precisely: the behavior of every system can be modeled as a process. A system can be modeled as a process at various abstraction levels. Typically, two descriptions are distinguished: *Spec* and *Impl*. The process *Spec* specifies the global behavior of the system, whereas the process *Impl* describes its implementation, typically as the parallel composition of certain communicating processes. The typical process algebraic correctness statement is then: $\text{Spec} = \tau_I(\text{Impl})$, i.e. the specification is behaviorally equivalent to the implementation, after abstraction of internal communications in I .

In our case, the components of the application are processes. Also the architecture itself will be a process; we will define our architecture as the process **GSRW** (Global Set with Read and Write) in Section 7.1.1. Our first problem is to find a distributed implementation of **GSRW**, called **DSRW** (Distributed Set with Read and Write) together with a proof that $\text{GSRW} = \tau_I(\text{DSRW})$. The second problem requires for any specification of a system's global behavior B , a number of components P_i (satisfying

certain syntactic criteria on locations) such that $B = \tau_I(P_1 \parallel \dots \parallel P_n \parallel \text{GSRW})$.

We have chosen the process algebraic approach for a number of reasons. First, it clarifies the concepts. By choosing a formalism, rather vague claims on realizability and expressiveness are turned into clear theorems. Process algebra provides the means to focus on the essential interfaces, by distinguishing external and internal actions, and by encapsulation of data in processes. The next advantage is that our approach yields rigorous formal proofs, apt for mechanic verification. The full proofs are available in Section 7.5. The third advantage is that we can use powerful proof principles developed for process algebra. Finally, by using a standard process algebra, existing tools [BFG⁺01] can be used for simulation and model checking. This has been demonstrated in [HP02b, PV02]. Section 7.4 gives a nice example of process algebraic manipulation. By just applying some distributivity and associativity laws, we transform a system description with two layers (applications and data space) to a system description with components (each consisting of an application part and a communication part), that communicate by synchronous message passing.

7.1 Process algebra with data

For good introductions to process algebra see [BW90, Fok00]. We will present and prove our ideas using the formalism μCRL (see Section 6.3).

7.1.1 GSRW in the syntax of μCRL

We will tacitly assume the following μCRL standard sorts with the usual operations: *Bool* (booleans), *Nat* (natural numbers, to represent locations), *D* (to represent data values, intentionally left unspecified) and *Set* (finite sets over *D*). For $A : \text{Set}$ and $v : D$, $A + v$ denotes $A \cup \{v\}$. It is routine to specify these types algebraically.

To the end of formally defining GSRW in μCRL , we introduce the parameterized atomic actions $\text{Read}(i : \text{Nat}, v : D)$ and $\text{Write}(i : \text{Nat}, v : D)$, where i denotes the location (or: service access point) and v the datum. Given these basic actions, the architecture GSRW is now defined by the following recursive specification, parameterized with the current set A of values of sort D :

$$\begin{aligned} \text{GSRW}(A : \text{Set}) &= \sum_{i:\text{Nat}} \sum_{v:D} \text{Write}(i, v). \text{GSRW}(A + v) \\ &+ \sum_{i:\text{Nat}} \sum_{v:D} \text{Read}(i, v). \text{GSRW}(A) \triangleleft v \in A \triangleright \delta \end{aligned}$$

Thus, GSRW maintains the global set A . At any moment this process allows that either an element is written, or a value can be read, *provided* it is actually present in A . In this way, the blocking character of *read* is captured.

Application processes can read and write by synchronizing with the *Read* and

Write of GSRW. To this end we introduce the actions $read(i : Nat, v : D)$ and $write(i : Nat, v : D)$. These actions should synchronize (cf. function calls or method invocations), so we define the communication function as follows: $Read \mid read = R$ and $Write \mid write = W$. As usual in μCRL , the unsynchronized actions are *encapsulated* by the $\partial_{\{Read, read, Write, write\}}$ construct (in order to enforce communication), and the internal communications are *hidden* using the $\tau_{\{R, W\}}$ construct (in order to abstract from internal detail).

The semantics of Example 7.1 is now captured formally by the following μCRL expression:

$$\tau_{\{R, W\}}(\partial_{\{Read, Write, read, write\}}(input.write(l_1, d) \parallel read(l_2, d).log \parallel \text{GSRW}(\emptyset)))$$

And indeed, it is a trivial exercise to prove that this is behaviorally equivalent to the specification $input.log.\delta$.

7.1.2 Proof methods from process algebra

We noted already that the typical process algebraic notion of refinement is given by the equation $\tau_I(Impl) = Spec$. As equivalence relation between processes we use branching bisimulation. Our results also apply to weak bisimulation, which is a slightly coarser equivalence relation. (see Section 2.3 for the definitions). In [GR01] branching bisimulation on μCRL processes is axiomatized algebraically. Recent papers developed more practical proof methods that will be used here. These methods are related to a particular process format, called *linear process equation*.

Linear Process Equations and Invariants In [Use02] it is demonstrated that the whole class of μCRL specifications can be transformed to linear process equations (LPE). Process terms have an implicit notion of state. The point of the LPE format is that the state is encoded explicitly in a data vector. An LPE is essentially a list of condition-action-effect triples. Given an index i from a finite index set J , action a_i with data parameter $f_i(d, e_i)$ is enabled in state d , if $b_i(d, e_i)$ holds. This action leads to the next state $g_i(d, e_i)$. Here e_i is a local variable, used to encode arbitrary input from a data set E_i . Formally, an LPE is a recursive specification of the following form:

$$Impl(d : D) = \sum_{i \in J} \sum_{e_i : E_i} a_i(f_i(d, e_i)).Impl(g_i(d, e_i)) \triangleleft b_i(d, e_i) \triangleright \delta$$

The advantage of this format is that properties and proof methods can be uniformly expressed, in terms of the state d and the constituents f_j , g_j and b_j .

We assume a special action τ , denoting hidden steps. An LPE is *convergent*, if it doesn't admit infinite sequences of τ -steps. The principle CL-RSP (Recursive Speci-

fication Principle for Convergent LPEs) [BG94, GR01] states that a convergent LPE has a unique solution.

Reachable states are characterized by means of *invariants*. A predicate $I(d)$ is an invariant if and only if it is preserved by all transitions, formally iff the following conjunction holds:

$$(\forall i \in J \forall d : D \forall e_i : E_i) (I(d) \wedge b_i(d, e_i) \rightarrow I(g(d, e_i)))$$

In [GS01, GR01] the *cones and foci* method is described for proving equality between implementation and specifications, which we recall in the next paragraph. This method is only applicable in case of convergent LPEs. If τ -loops exist, we need a fairness assumption on executions in order to ensure that eventually an exit from the τ -loop is chosen. To this end, a fairness rule will be introduced below.

State mappings, cones and focus points The summands of *Impl* above can be split into τ -steps and external steps, $J = Int \uplus Ext$, where $Int = \{i \in J \mid a_i = \tau\}$. Besides the implementation, we assume a given specification:

$$Spec(d' : D') = \sum_{i \in Ext} \sum_{e_i : E_i} a_i(f'_i(d', e_i)). Spec(g'_i(d', e_i)) \triangleleft b'_i(d', e_i) \triangleright \delta$$

Note that the specification must not contain τ -steps. We also assume that the implementation is convergent. Then every state has internal steps to a focus point, i.e. one in which no further τ -steps are possible. The focus points can be easily characterized by the focus condition:

$$FC(d) = (\forall i \in Int) (\nexists e_i : E_i) b_i(d, e_i).$$

An implementation and a specification in the format above can be proved behaviorally equivalent by providing a state mapping $h : D \rightarrow D'$, and proving that the matching criteria $MC_h(d)$ hold, where $MC_h(d)$ is defined as the conjunction of the following:

1. for each $i \in Int$, $\forall(e_i : E_i). b_i(d, e_i) \rightarrow h(d) = h(g_i(d, e_i))$
i.e internal steps don't change the related state.
2. for each $i \in Ext$, $\forall(e_i : E_i). b_i(d, e_i) \rightarrow b'_i(h(d), e_i)$
i.e the specification can mimic all external steps of the implementation (soundness).
3. for each $i \in Ext$, $\forall(e_i : E_i). b'_i(h(d), e_i) \wedge FC(d) \rightarrow b_i(d, e_i)$
i.e each external step of the specification can be mimicked in the related focus points of the implementation (completeness).

4. for each $i \in Ext$, $\forall(e_i : E_i). b_i(d, e_i) \rightarrow f_i(d, e_i) = f'_i(h(d), e_i)$
i.e the data labels on the external transitions coincide.
5. for each $i \in Ext$, $\forall(e_i : E_i). b_i(d, e_i) \rightarrow h(g_i(d, e_i)) = g'_i(h(d), e_i)$
i.e the next states after a visible transition are related.

Theorem 7.2 (from [GS01]) *For specification and convergent implementation in the format above, and given a state mapping h and an invariant I such that $I(d)$ holds and $\forall(d : D). I(d) \rightarrow MC_h(d)$, we have*

$$Impl(d) \triangleleft FC(d) \triangleright \tau.Impl(d) = Spec(h(d)) \triangleleft FC(d) \triangleright \tau.Spec(h(d))$$

The essence of this proof method is that given a state mapping h , and invariant I , the correctness proof boils down to a check of a number of simple criteria.

Fair abstraction The cones-and-foci method only works for convergent LPEs. But we will encounter τ -loops of arbitrary length. In order to eliminate these loops, we need a fairness principle, which states that eventually an exit of the loop is chosen. For this we will use Koomen's Fair abstraction rule (KFAR _{n} for $n > 1$) [BBK87]. Assume we have a v -loop with exits, of the following form:

$$\begin{aligned} X_1 &= v.X_2 + s_1 \\ X_2 &= v.X_3 + s_2 \\ &\dots \\ X_n &= v.X_1 + s_n \end{aligned}$$

Then after abstraction from v we would get a non-convergent LPE. However, according to KFAR _{n} we are sure that after some time one of the exits s_i is taken, so we get:

$$\tau.\tau_{\{v\}}(X_1) = \tau.\tau_{\{v\}}(s_1 + \dots + s_n)$$

7.2 Distributed implementation

In this section a distributed implementation of GSRW is defined and a correctness proof is given. We first introduce the data type *List*, representing a list of local data spaces. It has constructors ϵ (empty list) and $::$ (cons). The elements of the lists are sets of values. Note that GSRW puts no bound on the number of access points. Therefore, we can not assume a fixed length on the list of local data spaces. In order to avoid infinite lists, they are specified in such a way that they "grow on demand". We write L_i for the i -th element of L (counting from 0). If i exceeds the length of L , then L_i is taken to be the empty set. With $L[i : +v]$ we denote the list

$L_0, \dots, L_{i-1}, L_i + v, L_{i+1}, \dots, L_n$. When necessary, $L[i : +v]$ extends L with empty sets to have length at least i , and adds v to L_i .

$$\begin{array}{ll}
\epsilon_i &= \emptyset \\
(A :: L)_0 &= A \\
(A :: L)_{i+1} &= L_i
\end{array}
\qquad
\begin{array}{ll}
\epsilon[0 : +v] &= [\{v\}] \\
\epsilon[(i+1) : +v] &= \emptyset :: \epsilon[i : +v] \\
(A :: L)[0 : +v] &= (A + v) :: L \\
(A :: L)[(i+1) : +v] &= A :: (L[i : +v])
\end{array}$$

In the distributed version DSRW, each component i will write to its private set K_i and reads from its private set L_i . Elements of K_i are sent to all the L_j separately, by executing for each v an action $Send(i : Nat, v : D, j : Nat)$. Therefore, the distributed implementation of GSRW is a process DSRW, having as parameters the lists K and L and defined as follows:

$$\begin{aligned}
\text{DSRW}(K, L : List) = & \quad (7.1) \\
& \sum_{i:Nat} \sum_{v:D} \text{Write}(i, v). \text{DSRW}(K[i : +v], L) \\
& + \sum_{i:Nat} \sum_{v:D} \text{Read}(i, v). \text{DSRW}(K, L) \triangleleft v \in L_i \triangleright \delta \\
& + \sum_{v:D} \sum_{i,j:Nat} \text{Send}(i, v, j). \text{DSRW}(K, L[j : +v]) \triangleleft v \in K_i \setminus L_j \triangleright \delta
\end{aligned}$$

According to DSRW, written elements are not immediately available. Data items might even arrive in a different order in different processes. Nevertheless, we have the following correctness theorem:

Theorem 7.3 $\text{GSRW}(\emptyset) = \tau_{\{Send\}}(\text{DSRW}(\epsilon, \epsilon))$.

Proof: We view GSRW as a specification and $\tau_{\{Send\}}(\text{DSRW})$ as its implementation; the latter equals DSRW with $Send(i, v, j)$ replaced by τ . By the cones-and-foci method, it suffices to give a state mapping and an invariant, and check the matching criteria. As state mapping we define $h(K, L) = (\bigcup K \cup \bigcup L)$. We need the invariant $Inv(K, L) = \forall i. L_i \subseteq \bigcup K$, which can be checked easily. The focus condition $FC(K, L)$ is $\nexists(i, j, v). v \in K_i \setminus L_j$. Assuming the invariant, this can be simplified to $\forall j. L_j = \bigcup K$ (i.e. all written values have arrived and are ready to be read). Convergence of the implementation follows easily: in $\tau_{\{Send\}}(\text{DSRW})$ the number $\sum_i \sum_j \#(K_i \setminus L_j)$ decreases with each τ -step. Now the matching criteria are (skipping the trivial ones):

- (1) $v \in K_i \setminus L_j \rightarrow \bigcup K \cup \bigcup L = \bigcup K \cup \bigcup L[i : +v]$
- (2) $v \in L_i \rightarrow v \in \bigcup K \cup \bigcup L$
- (3) $(v \in \bigcup K \cup \bigcup L) \wedge (\forall j. L_j = \bigcup K) \rightarrow (v \in L_i)$

$$(4) (\bigcup K \cup \bigcup L) + v = \bigcup K[i : +v] \cup \bigcup L$$

These can be proved by simple set-theoretic calculations. Initially, we have $Inv(\epsilon, \epsilon)$ and $FC(\epsilon, \epsilon)$, whence the result follows by Theorem 7.2. \square

In fact this means that GSRW and $\tau_{\{Send\}}(DSRW)$ are indistinguishable. This is a generalization of [BKZ99a, BHJ00], because the application processes may use non-deterministic choice, recursion, or even synchronous communication. Our proof is a standard application of the cones-and-foci method (see Section 7.1.2).

7.3 Expressiveness

In this section we investigate the expressiveness of GSRW, from a system engineering point of view: given the requirements specification of a system under design, can a distributed implementation on GSRW be constructed? We assume that the requirements specification is given by a description of the global behavior, and a localization function. The *behavioral specification* is a process $Spec$. The alphabet of a process is the set of action labels that occur in it. Let A be the alphabet of $Spec$. We also assume some set L of locations, describing for instance physical devices. A localization function is a function $\lambda : A \rightarrow L$.

A component X is *consistent with the localization function* if there exists a fixed location ℓ , such that the alphabet of X contains only the actions *read*, *write* and external actions a with $\lambda(a) = \ell$. For instance, if $\lambda(scan) \neq \lambda(log)$, the implementation can have a component with the alphabet $\{read, write, scan\}$ and another with $\{read, write, log\}$. This notion can be seen as a syntactic criterion to enforce correct distribution and to enforce that processes can only communicate via the coordination primitives.

A *distributed implementation of $Spec, \lambda$ on GSRW* consists of an initial database A_0 , together with a number of components X_1, \dots, X_n that are consistent with λ , and behave like $Spec$, i.e.

$$Spec = \tau_{\{R, W\}} \partial_{\{read, Read, write, Write\}} (X_1 \parallel \dots \parallel X_n \parallel GSRW(A_0)).$$

The matter of distributing functionalities of a requirements specification over more communicating components was also studied in [Lan92] for LOTOS expressions; the synchronization is solved there with message passing, while GSRW coordinates the components using persistent data.

Example 7.4 *We describe a possible implementation on GSRW of a very simple buffer specification. For this, we consider the datasort *Queue*, representing a queue of arbitrary data items (data must be sent out in the same order in which it was scanned).*

It has the constant em , representing the empty queue, and the operations:

$\text{enqueue} : \text{Data} \times \text{Queue} \longrightarrow \text{Queue}$	adds an element to a queue;
$\text{dequeue} : \text{Queue} \longrightarrow \text{Queue}$	extracts the top element of the (non-empty) parameter queue;
$\text{head} : \text{Queue} \longrightarrow \text{Data}$	returns the top element of the queue;
$\text{notempty} : \text{Queue} \longrightarrow \text{Bool}$	true when there is at least one element in the queue.

The buffer interacts with the world through the actions IN , which inputs a data element to the buffer and OUT , which outputs a data element from the buffer. Then the μCRL specification of the buffer is:

$$\begin{aligned} \text{BufSpec}(Q : \text{Queue}) &= \sum_{d:\text{Data}} (IN(d). \text{BufSpec}(\text{enqueue}(d, Q)) \\ &+ OUT(\text{head}(d)). \\ &\quad \text{BufSpec}(\text{dequeue}(Q)) \triangleleft \text{notempty}(Q) \triangleright \delta) \end{aligned} \quad (7.2)$$

In order to build an implementation of BufSpec on GSRW , we use a global set that memorizes values of sort $\text{Nat} \times \text{Data}$, representing pairs (sequence number, data item). We instantiate the architecture to $\text{GSRW}(A : \text{Set}(\text{Nat} \times \text{Data}))$ and we choose a localization function $\lambda : \lambda(IN) = l_1, \lambda(OUT) = l_2$. A possible distributed implementation on GSRW of the buffer is :

$$\text{BufImpl} = B_{in}(0) \parallel B_{out}(0) \parallel \text{GSRW}(\emptyset) \quad (7.3)$$

where

$$\begin{aligned} B_{in}(n : \text{Nat}) &= \sum_{d:\text{Nat}} IN(d).write(l_1, (n, d)).B_{in}(n+1) \\ B_{out}(n : \text{Nat}) &= \sum_{d:\text{Nat}} read(l_2, (n, d)).OUT(d).B_{out}(n+1) \end{aligned}$$

(7.3) is indeed an implementation of (7.2), since it can be proved that $\text{BufSpec}(\text{em})$ is branching bisimilar to $\tau_{\{R, W\}} \partial_{\{\text{Read}, \text{Write}, \text{read}, \text{write}\}} \text{BufImpl}$.

7.3.1 The translation scheme

In the sequel we show how to construct X_i and A_0 for any requirements specification. That is, we describe a translation scheme from an LPE $\text{Spec}(d)$ together with a localization function L to a set of processes X_1, \dots, X_n and some initial database A_0 satisfying the above criteria. The localization criterion will be solved by mapping each action label to a different component. This results in the maximally distributed, most fine-grained implementation of the given specification, from which an implementation

with less parallel components can always be obtained by bundling several components X_i . Then we will prove that this translation scheme is correct.

We assume that a requirements specification is given in LPE format (see Section 7.1):

$$Spec(d : D) = \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i(d, e_i)).Spec(g_i(d, e_i)) \triangleleft b_i(d, e_i) \triangleright \delta \quad (7.4)$$

Each summand of $\sum_{i \in I}$ defines a set of transitions from state d to state $g_i(d, e_i)$ and it is enabled for all e_i for which the guard $b_i(d, e_i)$ is true. Moreover, we assume a localization function $\lambda : \{a_i \mid i \in I\} \rightarrow L$ for a set of locations L .

Let $n = \text{card}(I)$. The distributed implementation will have n components, each responsible for one action a_i . They communicate via GSRW, using a global set of pairs (*timestamp*, *data*) of the sort $Nat \times D$. The timestamp represents the moment when the pair was added to the database or, in other words, the number of visible + invisible steps executed until the time of insertion.

Components are triggered in turns, by the timestamp, in a circular infinite pass: component i will be activated at all moments $t = k.n + i$ ($\forall k$). When activated, it will choose to execute its action or not to execute it. In both cases, it will increase the “global time” and pass the turn to its next sister. This cycle is needed to ensure that the nondeterminism that may exist in the global specification $Spec(d)$ is preserved in the distributed implementation. At any time, all possible actions must have a chance to execute.

In a formal definition, the component X_i , responsible of action a_i is:

$$\begin{aligned} X_i(m) = \sum_{d:D} & read(\lambda(a_i), (m, d)). \\ & (\sum_{e_i:E_i} (a_i(f_i(d, e_i)).write(\lambda(a_i), (m+1, g_i(d, e_i)))) \\ & \triangleleft b_i(d, e_i) \triangleright \delta) \\ & + write(\lambda(a_i), (m+1, d)) \\ & . X_i(m+n) \end{aligned} \quad (7.5)$$

and the initial state of the implementation is

$$\prod_i X_i(i) \parallel \text{GSRW}(\{(0, d)\}) \quad (7.6)$$

The parameter m of X_i is the moment when X_i expects to be activated next. As mentioned before, m is always of the form $k.n + i$. At moment m , $read(\ell, (m, d))$ from X_i synchronizes with $Read(\ell, (m, d))$ from $\text{GSRW}(A)$, for some d . This activates X_i . After “acting”, X_i will set its parameter to the next active moment $(k+1).n + i$, i.e. $m + n$. In its life, X_i passes only through the following local states: 0 –ready to read, 1 –activated; make a choice (execute action or pass the turn), 2 –action performed; pass the turn.

We will prove that this distributed implementation on GSRW of an LPE is almost equivalent to the specification. That is: if we abstract from the actions dealing with the global set (R, W) , then we get the specification $Spec(d)$ with an extra initialization step.

Theorem 7.5 *For every requirements specification expressible as an LPE $Spec(d)$, the components X_i resulted by applying the translation scheme satisfy:*

$$\tau.Spec(d) = \tau.\tau_{\{R,W\}}\partial_{\{Read, Write, read, write\}}\left(\prod_i X_i(i) \parallel GSRW(\{(0, d)\})\right).$$

7.3.2 Correctness proof

This subsection is devoted to proving that the translation defined above is correct. That is, to prove Theorem 7.5. We will do this in three steps:

1. First of all, to be able to compare the two processes appearing in the theorem, we need to *bring the implementation (7.6) to a linearized form* (the specification $Spec(d)$ already is, by assumption).

2. Further, having both specification and implementation in linearized form, we can use the cones-and-foci method to prove their equivalence. But not immediately, since this method requires that the implementation should be convergent (without infinite τ -loops) and this is not the case for ours - infinite τ -loops occur when abstracting from R and W . Therefore, in the second step, *pre-abstraction*, we will consider an intermediate specification Y , in which we abstract only from R 's and the second W (the one generated by the communication between $write(m+1, d)$ and $Write$ from GSRW, see (7.5)), while keeping the other $write(m+1, g_i(d, e_i))$ as a visible action - but renamed to an action without arguments v . In Y we also eliminate the database A .

3. Finally we can prove, using the cones-and-foci method, that the linearized implementation is branching bisimilar to Y . Afterwards we abstract from the remaining visible action v and prove by *fair abstraction* that $\tau.\tau_{\{v\}}Y = \tau.Spec$.

The three steps are detailed below.

1. Linearization of implementation In the linearized version of a process, we view everything globally. The state of the system will be described by the parameters A , $\vec{m} \in N^n$, $\vec{l} \in \{0, 1, 2\}^n$ and $\vec{d} \in D^n$. A is the set of pairs, the database appearing as parameter of process GSRW. \vec{m} is the vector of “moments”, an element m_i (the parameter of process X_i) is the moment when X_i will be activated next. \vec{l} is the vector of local states (l_i is the current local state of component i). Finally, \vec{d} is the vector of data items; d_i is the data that component i knows of, currently. Although in principle there is only one global view on data, components may have temporary different views. That's why we need \vec{d} as parameter, instead of just d .

In the initial state, $A = \{(0, d)\}$ (we are at moment 0 and the current data is the global specification's parameter d); $\vec{l} = 0$ (all the components are in the “start” local state 0); $\vec{m} = (0, 1, \dots, n-1)$ (component i waits to be activated at moment i and first component to be activated is 0, triggered by $(0, d)$, the only pair from the database A); $\vec{d} = \vec{0}$ (in the initial state the values in this vector don't matter, since they will be used only after being initialized by a reading action).

Due to the fact that all components X_i from (7.6) are independent, the linearized version is just the sum of their separate interactions with $\text{GSRW}(A)$. After renaming one of the write actions to v and hiding the read action and the other write, we get the following linearized implementation:

$$\begin{aligned}
\text{Impl } (A, \vec{l}, \vec{m}, \vec{d}) = & \sum_{i=0}^{n-1} (\\
& \sum_y \tau. \text{Impl}(A, \vec{l}[l_i := 1], \vec{m}, \vec{d}[d_i := y]) \\
& \quad \triangleleft l_i = 0 \wedge (m_i, y) \in A \triangleright \delta \\
& + v. \text{Impl}(A \cup \{(m_i + 1, d)\}, \vec{l}[l_i := 0], \vec{m}[m_i := m_i + n], \vec{d}) \\
& \quad \triangleleft l_i = 1 \triangleright \delta \\
& + \sum_{e_i: E_i} (a_i(f_i(d, e_i)). \text{Impl}(A, \vec{l}[l_i = 2], \vec{m}, \vec{d}[d_i := g_i(d, e_i)]) \\
& \quad \triangleleft l_i = 1 \wedge b_i(d, e_i) \triangleright \delta) \\
& + \tau. \text{Impl}(A \cup \{(m_i + 1, d_i)\}, \vec{l}[l_i := 0], \vec{m}[m_i := m_i + n], \vec{d}) \\
& \quad \triangleleft l_i = 2 \triangleright \delta)
\end{aligned} \tag{7.7}$$

The formula

$$\begin{aligned}
\tau_{\{R, W\}} \partial_{\{Read, Write, read, write\}} \left(\prod_i X_i(i) \parallel \text{GSRW}(\{(0, d)\}) \right) \\
= \tau_{\{v\}} \text{Impl}(\{(0, d)\}, \vec{0}, (0, 1, \dots, n-1), \vec{0}) \tag{7.8}
\end{aligned}$$

summarizes what has happened in the linearization step.

2. Pre-abstraction We define the intermediate specification Y as follows:

$$\begin{aligned}
Y(\mathbf{c}, \mathbf{d}) = & \sum_{i=0}^{n-1} (\\
& v. Y((i+1) \bmod n, \mathbf{d}) \quad \triangleleft i = \mathbf{c} \triangleright \delta \\
& + \sum_{e_i: E_i} (a_i(f_i(d, e_i)). Y((i+1) \bmod n, g_i(d, e_i)) \\
& \quad \triangleleft i = \mathbf{c} \wedge b_i(\mathbf{d}, e_i) \triangleright \delta))
\end{aligned} \tag{7.9}$$

The parameter \mathbf{c} is a natural number from the set $\{0, \dots, n-1\}$ and points to the active component $X_{\mathbf{c}}(m_{\mathbf{c}})$. \mathbf{c} 's values in the successive calls of Y reflect the order in which components become active:

$$Y(0, _), Y(1, _), Y(2, _), \dots, Y(n-1, _), Y(0, _), Y(1, _), \dots$$

The other parameter, \mathbf{d} , is the global state of the system.

We aim to show, by using an appropriate state mapping, that this intermediate specification is equivalent to the linearized implementation, i.e. that

$$\tau.\text{Impl}(\{(0, d)\}, \vec{0}, (0, 1, \dots, n-1), \vec{0}) = \tau.Y(0, d). \quad (7.10)$$

The state mapping must relate equivalent states of *Impl* and *Y*. To ensure this, the cones-and-foci method (see Section 7.1.2) requires that certain *matching criteria* should be satisfied, which are easy (but tedious) to prove, using invariants on *Impl*'s states. The complete proof of (7.10), including a list of the invariants, is presented in Section 7.5. Here we will only show some of the invariants and briefly discuss the state mapping.

One of the invariants is that for any “moment” t there is at most one data item d such that $(t, d) \in A$. When this item exists, we will denote it by $\mathbf{data}(A, t)$. Another important invariant is that for any state $\langle A, \vec{l}, \vec{m}, \vec{d} \rangle$ there is exactly one $x \in \{0 \dots n-1\}$ for which $(m_x, _) \in A$ (where $_$ denotes any data instance). The state mapping $h : \text{States}(\text{Impl}) \rightarrow \text{States}(Y)$ can be now defined as follows:

$$h(\langle A, \vec{l}, \vec{m}, \vec{d} \rangle) = \begin{cases} \langle x, \mathbf{data}(A, m_x) \rangle & \text{if } l_x \in \{0, 1\} \text{ and } (m_x, _) \in A \\ \langle (x+1) \bmod n, d_x \rangle & \text{if } l_x = 2 \text{ and } (m_x, _) \in A \end{cases}$$

The idea of this mapping is that it extracts from a global state $\langle A, \vec{l}, \vec{m}, \vec{d} \rangle$ the essential information that characterizes it, namely the index of the active component and the data that this component gets as input.

If we hide v in both *Impl* and *Y*, (7.10) becomes

$$\tau.\tau_{\{v\}}\text{Impl}(\{(0, d)\}, \vec{0}, (0, 1, \dots, n-1), \vec{0}) = \tau.\tau_{\{v\}}Y(0, d). \quad (7.11)$$

3. Abstraction By instantiating the definition (7.9) for $\mathbf{c} \in \{0 \dots n-1\}$ and using the observation that there are no summands for which $i \neq \mathbf{c}$, we obtain:

$$\begin{aligned} Y(0, d) &= v.Y(1, d) + \sum_{e_0: E_0} a_0(f_0(d, e_0)).Y(1, g_0(d, e_0)) \triangleleft b_0(d, e_0) \triangleright \delta \\ Y(1, d) &= v.Y(2, d) + \sum_{e_1: E_1} a_1(f_1(d, e_1)).Y(2, g_1(d, e_1)) \triangleleft b_1(d, e_1) \triangleright \delta \\ &\vdots \\ Y(n-1, d) &= v.Y(0, d) + \sum_{e_{n-1}: E_{n-1}} a_{n-1}(f_{n-1}(d, e_{n-1})).Y(0, g_{n-1}(d, e_{n-1})) \\ &\quad \triangleleft b_{n-1}(d, e_{n-1}) \triangleright \delta \end{aligned}$$

It is easy to see that $Y(0, d) \cdots Y(n-1, d)$ form a $\{v\}$ -cluster, with exits

$$\{a_i(f_i(d, e_i)). Y((i+1) \bmod n, g_i(d, e_i)) \triangleleft b_i(d, e_i) \triangleright \delta \mid 0 \leq i < n, e_i \in E_i\}$$

KFAR_n (Koomen's Fair Abstraction Rule) states that in a fair execution, one of the exits will eventually be taken. In our case, this means that we can write, for all $k \in \{0 \cdots n-1\}$:

$$\begin{aligned} \tau.\tau_{\{v\}}Y(k, d) = \\ \tau.\sum_{i=0}^{n-1} \sum_{e_i \in E_i} a_i(f_i(d, e_i)).\tau_{\{v\}}Y((i+1) \bmod n, g_i(d, e_i)) \\ \triangleleft b_i(d, e_i) \triangleright \delta \end{aligned} \quad (7.12)$$

The right-hand side of this formula does not depend on k , which allows us to say that $\tau.\tau_{\{v\}}Y(0, d) = \tau.\tau_{\{v\}}Y(1, d) = \cdots = \tau.\tau_{\{v\}}Y(n-1, d)$. Consequently, we can replace in (7.12) k with 0 and $a_i(f_i(d, e_i)).\tau_{\{v\}}Y((i+1) \bmod n, g_i(d, e_i))$ with $a_i(f_i(d, e_i)).\tau_{\{v\}}Y(0, g_i(d, e_i))$ and obtain:

$$\tau.\tau_{\{v\}}Y(0, d) = \tau.\sum_{i=0}^{n-1} \sum_{e_i \in E_i} a_i(f_i(d, e_i)).\tau_{\{v\}}Y(0, g_i(d, e_i)) \triangleleft b_i(d, e_i) \triangleright \delta$$

Comparing with (7.4), we see that $\tau.\tau_{\{v\}}Y(0, d)$ and $\tau.Spec(d)$ are solutions of the same equation, thus, by CL-RSP (see Section 7.1.2), they are equal. This equality, together with the linearization (7.8) and the equivalence to the intermediate specification (7.11), proves Theorem 7.5.

7.4 Complete distribution of requirements specifications

In this section, we show that using the results from the previous sections, and applying simple algebraic properties, any requirements specification can be transformed to a distributed implementation, built of components that communicate by synchronous message passing (an overview of the transformation steps is given in Figure 7.1). Each component consists of two parts: an application process, and an agent which implements the local data space and takes care of the communication aspects. We indicate how this result leads to a high degree of compositionality.

First, we show how the distributed architecture DSRW can be transformed to a parallel composition of separate agents. In order to avoid dynamic agent replication, we only deal with the case that the set of locations is finite and known in advance. Therefore, we restrict the definition (7.1) of DSRW to the case where the index i of

locations ranges over a finite set $Loc \subset Nat$ – and obtain $DSRW_{Loc}$:

$$\begin{aligned}
DSRW_{Loc}(K, L : List) = & \\
& \sum_{i:Loc} \sum_{v:D} Write(i, v) \quad . \quad DSRW_{Loc}(K[i : +v], L) \\
& + \sum_{i:Loc} \sum_{v:D} Read(i, v) \quad . \quad DSRW_{Loc}(K, L) \triangleleft v \in L_i \triangleright \delta \\
& + \sum_{v:D} \sum_{i:Loc} \sum_{j:Loc} Send(i, v, j) \quad . \quad DSRW_{Loc}(K, L[j : +v]) \triangleleft v \in K_i \setminus L_j \triangleright \delta
\end{aligned}$$

This process is equivalent to (even the linearized version of) a parallel composition of processes representing applications and local databases (agents). We assume the communication function $send \mid SEND = Send$.

$$\begin{aligned}
DSRW_{Loc} \quad (K, L) = \partial_{\{send, SEND\}} \Big(\prod_{i:Loc} Agent(i, K_i, L_i) \Big) \quad (7.13) \\
Agent \quad (i : Nat, A : Set, B : Set) = & \\
& \sum_{v:D} Write(i, v).Agent(i, A + v, B) \\
& + \sum_{v:D} Read(i, v).Agent(i, A, B) \triangleleft v \in B \triangleright \delta \\
& + \sum_{v:D} \sum_{j:Nat} SEND(i, v, j).Agent(i, A, B) \triangleleft v \in A \triangleright \delta \\
& + \sum_{v:D} \sum_{j:Nat} send(j, v, i).Agent(i, A, B + v) \triangleleft v \notin B \triangleright \delta \\
& + \sum_{v:D} Send(i, v, i).Agent(i, A, B + v) \triangleleft v \in A \setminus B \triangleright \delta
\end{aligned}$$

Note that a $Send(i, v, j)$ action of $DSRW_{Loc}$ is expressed as the communication of a $SEND(i, v, j)$ -action (agent i tries to send item v to agent j) with a $send(i, v, j)$ -action (agent j tries to receive a new item v from agent i). As processes cannot communicate with themselves, a separate line is needed for the $Send(i, v, i)$ -actions.

For readability, we adopt the following notations:

$$\begin{aligned}
P &= \{R, W\} \\
H &= \{Read, Write, read, write\}
\end{aligned}$$

Let us consider a requirements specification expressible as an LPE $Spec(d)$. We can apply the translation scheme (Section 7.3) and get components $X_0 \cdots X_n$, which satisfy (Theorem 7.5)

$$\tau.Spec(d) = \tau.\tau_P \partial_H \Big(\prod_{i:\{0..n\}} X_i(i) \parallel GSRW(\{(0, d)\}) \Big) \quad (7.14)$$

From the proof of Theorem 7.3, it follows that for any lists K and L that satisfy $\forall i. L_i \subseteq \bigcup K$:

$$\text{GSRW}(\bigcup K) = \tau_{\{\text{Send}\}}(\text{DSRW}(K, L))$$

In particular, by instantiating K_i with $\{(0, d)\}$, for every $i \in \{0 \dots n\}$ and with \emptyset for every $i > n$:

$$\text{GSRW}(\{(0, d)\}) = \tau_{\{\text{Send}\}}(\text{DSRW}([\{(0, d)\}, \dots, \{(0, d)\}], \epsilon))$$

With this, (7.14) becomes

$$\tau.\text{Spec}(d) = \tau.\tau_P \partial_H \left(\prod_{i:\{0 \dots n\}} X_i(i) \parallel \tau_{\{\text{Send}\}}(\text{DSRW}([\{(0, d)\}, \dots, \{(0, d)\}], \epsilon)) \right)$$

and because the number of locations used is finite:

$$\begin{aligned} \tau.\text{Spec}(d) = \\ \tau.\tau_P \partial_H \left(\prod_{i:\{0 \dots n\}} X_i(i) \parallel \tau_{\{\text{Send}\}}(\text{DSRW}_{\{0 \dots n\}}([\{(0, d)\}, \dots, \{(0, d)\}], \epsilon)) \right). \end{aligned}$$

Further, let us expand $\text{DSRW}_{\{0 \dots n\}}$, according to (7.13) :

$$\begin{aligned} \tau.\text{Spec}(d) = \\ \tau.\tau_P \partial_H \left(\prod_{i:\{0 \dots n\}} X_i(i) \parallel \tau_{\{\text{Send}\}} \partial_{\{\text{send}, \text{SEND}\}} \left(\prod_{i:\{0 \dots n\}} \text{Agent}(i, \{(0, d)\}, \emptyset) \right) \right) \end{aligned} \quad (7.15)$$

In expression (7.15), we still see two layers, on the left the applications, and on the right the agents implementing the data space architecture. Below we will reorganize this expression, bringing together the processes and agents at the same location i . This reshuffle is based on the associativity of the parallel composition, and the distributivity of hiding (τ_I) and encapsulation (∂_H) over parallel composition (\parallel). The latter is only possible under certain restriction on action names. In [KS98], so-called *alphabet axioms* are presented, that allow manipulation of expressions based on the alphabets of the process involved. Here $\alpha(p)$ denotes the alphabet of process p (the set of action names that occur in it) and $\alpha(p) \mid \alpha(q)$ denotes the set of actions that may be the result of a communication between an action in $\alpha(p)$ and an action in $\alpha(q)$. In

particular, the axioms

$$\begin{array}{ll}
\text{CA1} & (\alpha(x) \mid (\alpha(y) \cap H)) \subseteq H \cup \{\delta\} \quad \rightarrow \quad \partial_H(x \parallel y) = \partial_H(x \parallel \partial_H(y)) \\
\text{CA2} & (\alpha(x) \mid (\alpha(y) \cap I)) \subseteq \{\delta\} \quad \rightarrow \quad \tau_I(x \parallel y) = \tau_I(x \parallel \tau_I(y)) \\
\text{CA3} & \alpha(x) \cap H = \emptyset \quad \rightarrow \quad \partial_H(x) = x \\
\text{CA4} & \alpha(x) \cap I = \emptyset \quad \rightarrow \quad \tau_I(x) = x \\
\text{CA7} & H \cap I = \emptyset \quad \rightarrow \quad \tau_I \partial_H(x) = \partial_H \tau_I(x)
\end{array}$$

allow moving process expressions inside and outside the encapsulation and the hiding operators, subject to an independence condition. We can make the convention that the external actions of the components X_i don't contain any *Send*, *send*, or *SEND* actions (otherwise, they should be renamed). Under this hypothesis, the X_i processes can be pushed inside the ∂ and the τ . Therefore, (7.15) becomes

$$\begin{aligned}
\tau.Spec(d) = & \\
& \tau.P \partial_H \tau_{\{Send\}} \partial_{\{send, SEND\}} \left(\parallel_{i:\{0 \dots n\}} (X_i(i) \parallel \text{Agent}(i, \{(0, d)\}, \emptyset)) \right)
\end{aligned}$$

or (commuting the τ s and the ∂ s)

$$\begin{aligned}
\tau.Spec(d) = & \\
& \tau.P \tau_{\{Send\}} \partial_{\{send, SEND\}} \tau_P \partial_H \left(\parallel_{i:\{0 \dots n\}} (X_i(i) \parallel \text{Agent}(i, \{(0, d)\}, \emptyset)) \right) \quad (7.16)
\end{aligned}$$

In the definition of X_i , all *read* and *write* primitives occurring have the first parameter constant, namely i and the *Read* and *Write* from $\text{Agent}(i, K_i, L_i)$ also have the first parameter constant: i . Thus, we could replace the *read* action by $n + 1$ actions $read_0 \dots read_n$ and view $read(i, a)$ as $read_i(a)$. Similarly, $write(i, a)$ can be seen as $write_i(a)$, $Read(i, a)$ as $Read_i(a)$ and $Write(i, a)$ as $Write_i(a)$. In the next step, we will apply the alphabet axioms, with $read_i$, $write_i$, etc. viewed as the action names occurring in the alphabet of processes. In this way we can push $\tau_P \partial_H$ inside, in order to obtain:

$$\begin{aligned}
\tau.Spec(d) = & \\
& \tau.P \tau_{\{Send\}} \partial_{\{send, SEND\}} \left(\parallel_{i:\{0 \dots n\}} \tau_P \partial_H (X_i(i) \parallel \text{Agent}(i, \{(0, d)\}, \emptyset)) \right) \quad (7.17)
\end{aligned}$$

This is a truly distributed implementation of $Spec(d)$, built up by components consisting of two clearly separated parts: the computation X_i and the coordination **Agent**. The components communicate with each other by synchronous message passing (namely, by executing *Send* actions).

Until now, the locations (i) were needed in order to avoid confusion between applications and the local data spaces. However, in equation (7.17), the binding of an application to an agent is already uniquely specified by means of encapsulation. Therefore, the next step could be to remove all references to locations. This boils down to renaming $read(i, v)$ to $read(v)$, $send(i, v, j)$ to $send(v)$, etc. The \sum_i -operators become superfluous as well.

Note that the renamed agents are exactly identical. At this point, we have arrived at a truly compositional description of the system, in which (a number of) components can be exchanged by (any number of) branching bisimilar components. This compositionality forms the basis of transparent agent replication, or even application replication. However, a detailed description is out of the scope of this chapter (but see Section 7.6.1 on related work).

7.5 Full proofs

7.5.1 Full proofs of invariants

In the following, \star denotes any instance of data from D . We will need the function $\mathbf{active} : States(X) \longrightarrow \{0 \cdots n - 1\}$,

$$\mathbf{active}(\langle A, \vec{l}, \vec{m}, \vec{d} \rangle) = \text{the } x \text{ for which } (m_x, \star) \in A$$

Lemma 7.6 *The following invariants hold for the implementation X :*

1. $(\forall t)$ there is at most one pair $(t, \star) \in A$
2. $\exists i (0 \leq i < n)$ s.t. $(m_i, \star) \in A$ (i.e. \mathbf{active} is well defined), and $m_i = \max_{(t, d) \in A} t$.
3. if $x = \mathbf{active}(\langle A, \vec{l}, \vec{m}, \vec{d} \rangle)$ then $\forall i \in \{0 \cdots n - 1\}$ s.t. $i \neq (x - 1) \bmod n$, $m_{(i+1) \bmod n} = m_i + 1$ and $m_x = m_{(x-1) \bmod n} + 1 - n$.
(in other words: in each state we have an arithmetic progression $m_x < m_{x+1} < \cdots < m_n < m_1 < \cdots < m_{x-1}$, with step 1)
4. $\vec{l} = \vec{0}$
or $\exists i (0 \leq i < n) : l_i = 1$ and $\forall j \neq i l_j = 0$
or $\exists i (0 \leq i < n) : l_i = 2$ and $\forall j \neq i l_j = 0$.
5. if $l_i > 0$ then $\mathbf{active}(\langle A, \vec{l}, \vec{m}, \vec{d} \rangle) = i$.

Proof: We first prove that these invariants hold for X 's initial state,

$$\langle A_0, \vec{l}_0, \vec{m}_0, \vec{d}_0 \rangle = \langle \{(0, d)\}, \vec{0}, (0, 1, \dots, n-1), \vec{0} \rangle :$$

1. immediate.

2. $i=0$.

3. $\mathbf{active}(A_0, \vec{l}_0, \vec{m}_0, \vec{d}_0) = 0$.

$$(\forall i : 0 \leq i \leq n-2) m_{0i+1} = m_{0i} + 1 = (m_{0i} + 1) \bmod n.$$

$$\text{And } m_{00} = 0 = (n-1) + 1 - n = m_{0n-1} + 1 - n = m_{0(-1) \bmod n} + 1 - n.$$

4. $l_0 = \vec{0}$.

5. $l_0 = \vec{0}$.

Now, supposing that they hold for an arbitrary state $\langle A, \vec{l}, \vec{m}, \vec{d} \rangle$, we prove that they are still true for any of X 's next states, let it be denoted by $\langle A', \vec{l}', \vec{m}', \vec{d}' \rangle$. We have to analyze the following possibilities (extracted from the description of process X):

– for some k , $l_k = 0 \wedge (m_k, \star) \in A$ and a τ -step happens.

Then $\langle A', \vec{l}', \vec{m}', \vec{d}' \rangle = \langle A, \vec{l}[l_k := 1], \vec{m}, \vec{d}[d_k := y] \rangle$.

In this case, (1),(2),(3) remain true because state components that occur in these properties (database A and moments' array \vec{m}) haven't changed.

$(m_k, \star) \in A \stackrel{\text{inv.}(2)}{\rightsquigarrow} \mathbf{active}(A, \vec{l}, \vec{m}, \vec{d}) = k \rightsquigarrow j \neq k \stackrel{\text{inv.}(5)}{\rightsquigarrow} l_j = 0 \forall j \neq k$. But $l_k = 0$ too, so $\vec{l} = \vec{0}$, which means that $\vec{l}' = (0 \cdots 1 \cdots 0)$, i.e. (4) is true. The only index i for which $l'_i = 0$ is k ($l'_k = 1$). $k = \mathbf{active}(A', \vec{l}', \vec{m}', \vec{d}')$ (because $A' = A$ and $\vec{m}' = \vec{m}$), so (5) holds too.

– for some k , $l_k = 1$ and a v action happens.

Then $\langle A', \vec{l}', \vec{m}', \vec{d}' \rangle = \langle A \cup \{(m_k + 1, d)\}, \vec{l}[l_k := 0], \vec{m}[m_k := m_k + n], \vec{d} \rangle$

1. We have to prove that the newly added pair $(m_k + 1, d)$ hasn't disturbed this property, i.e. there is no $(m_k + 1, \star)$ already in A . This is true, since $m_k + 1 >$

$$m_k \stackrel{\text{inv.}(2)}{=} \max_{(t,d) \in A} t.$$

2. If $n > 1$, the unique i is $(k+1) \bmod n$, because $m'_{(k+1) \bmod n} = m_k + 1$ (inv. (3)) and $(m_k + 1, d) \in A'$. Uniqueness is given by inv. (1).

If $n = 1$, the active process 0 remains active ($(m_0 + 1, d_0) \in A'$ and $m'_0 = m_0 + 1$).

3. $x' = \mathbf{active}(A', \vec{l}', \vec{m}', \vec{d}') = (k+1) \bmod n$. Notice that $(x' - 1) \bmod n = k$.

For $i \in \{0 \cdots n-1\}, i \neq k, i \neq (k-1) \bmod n$, the property remains true, as $m' = m$ for the values involved.

It remains to be shown that $m'_k = m'_{(k-1) \bmod n} + 1$ and that $m'_{x'} = m'_k + 1 - n$.

$$m'_k = m_k + n \stackrel{\text{inv.}(3)}{=} (m_{(k-1) \bmod n} + 1 - n) + n = m_{(k-1) \bmod n} + 1 = m'_{(k-1) \bmod n} + 1$$

$$m'_{x'} = m_{x'} \stackrel{\text{inv.}(3)}{=} m_k + 1 = (m'_k - n) + 1.$$

4. $l'_i = l_i = 0, \forall i \neq k$ and $l'_k = 0$. So, $\vec{l}' = \vec{0}$.

5. $\vec{l}' = 0$, by invariant (4).

– for some k and some $e_k \in E_k$, $l_k = 1 \wedge b_k(d_k, e_k)$ and an a_k action happens.
Then $\langle A', \vec{l}', \vec{m}', \vec{d}' \rangle = \langle A, \vec{l}[l_k = 2], \vec{m}, \vec{d}[d_k := g_k(d_k, e_k)] \rangle$.

(1),(2),(3) hold because the state components involved are not changed by this step.
 $l_k = 1 \xrightarrow{inv.(4)} l_i = 0 \ \forall i \neq k$. So, since $l'_i = l_i = 0 \ \forall i \neq k$ and $l'_k = 2$, (4) holds in the current state too.

And, finally, (5) holds too, as the active index did not change (it's still k).

– for some k , $l_k = 2$ and a τ -action happens.

Then $\langle A', \vec{l}', \vec{m}', \vec{d}' \rangle = \langle A \cup \{(m_k + 1, d_k)\}, \vec{l}[l_k := 0], \vec{m}[m_k := m_k + n], \vec{d} \rangle$.

All the invariants are shown to hold by a reasoning similar to the second case (“for some k , $l_k = 1$ and a v action happens”). \square

We proved (invariant 1) that for any “moment” t there is at most one data item d such that $(t, d) \in A$. When this item exists, we will denote it by $\mathbf{data}(A, t)$. Note that $\mathbf{data}(A, m_{\mathbf{active}(\langle A, \vec{l}, \vec{m}, \vec{d} \rangle)})$ is defined for all reachable states in $States(X)$.

Lemma 7.7 *If $l_i = 1$ then $\mathbf{data}(A, m_i) = d_i$.*

Proof: If $l_i = 1$ then the most recent step in X was a τ -step (a read from the database). This could happen only if the guard was true: $l_i = 0 \wedge (m_i, y) \in A$, for some y ; by definition, $y = \mathbf{data}(A, m_i)$. The changes that occur in the state $\langle A, \vec{l}, \vec{m}, \vec{d} \rangle$ as a result of this step are $l_i := 1$ and $d_i := y$. That is, $d_i := \mathbf{data}(A, m_i)$. \square

7.5.2 Full proofs of the matching criteria

Lemma 7.8 *For X , Y and h defined in the second step of the correctness proof (Section 7.3.2), the following matching criteria hold:*

1. (a) for all i , $(\forall y \in D) \ l_i = 0 \wedge (m_i, y) \in A \longrightarrow h(\langle A, \vec{l}, \vec{m}, \vec{d} \rangle) = h(\langle A, \vec{l}[l_i := 1], \vec{m}, \vec{d}[d_i := y] \rangle)$
 (b) for all i , $l_i = 2 \longrightarrow h(\langle A, \vec{l}, \vec{m}, \vec{d} \rangle) = h(\langle A \cup \{(m_i + 1, d_i)\}, \vec{l}[l_i := 0], \vec{m}[m_i := m_i + n], \vec{d} \rangle)$
 (internal steps in X don't change the mapped state in Y)
2. (a) for all i , $l_i = 1 \longrightarrow \mathbf{c} = i$ (v enabled in X : v enabled in Y)
 (b) for all i , $(\forall e_i \in E_i) \ l_i = 1 \wedge b_i(d_i, e_i) \longrightarrow \mathbf{c} = i \wedge b_i(\mathbf{d}, e_i) \ (X \text{ can do } a_i(f_i(\star, e_i)): Y \text{ can do } a_i(f_i(\star, e_i)))$
 (soundness: in each state, for each external action, if X can do it then Y can do it, too)

The external actions that X can do are v (that can happen when $l_i = 1$) and $\{a_i(f_i(d, e_i))\}$ (that are enabled when $l_i = 1 \wedge b_i(d_i, e_i)$ is true).

3. $FC(\langle A, \vec{l}, \vec{m}, \vec{d} \rangle) : (\exists i : 1 \leq i \leq n) \text{ s.t. } l_i = 1 \text{ and } l_j = 0 \forall j \neq i,$
that is s.t. $\vec{l} = (0, \dots, 0, 1, 0, \dots, 0)$.

(a) for all i , $FC(\langle A, \vec{l}, \vec{m}, \vec{d} \rangle) \wedge i = \mathbf{c} \longrightarrow l_i = 1$

(b) for all i , $(\forall e_i : E_i) FC(\langle A, \vec{l}, \vec{m}, \vec{d} \rangle) \wedge i = \mathbf{c} \wedge b_i(\mathbf{d}, e_i) \longrightarrow l_i = 1 \wedge b_i(d_i, e_i)$

(completeness: X can do a step : X can do that step, too - eventually after a number of internal steps)

4. for all i , $(\forall e_i : E_i) b_i(d_i, e_i) \longrightarrow f_i(d_i, e_i) = f_i(\mathbf{d}, e_i)$ (the data labels on the visible actions coincide).

5. (a) $l_i = 1 \longrightarrow h(\langle A \cup \{m_i + 1, d_i\}, \vec{l}[l_i := 0], \vec{m}[m_i := m_i + n], \vec{d} \rangle) = \langle (i + 1) \bmod n, \mathbf{d} \rangle$

(b) $l_i = 1 \wedge b_i(d_i, e_i) \longrightarrow h(\langle A, \vec{l}[l_i := 2], \vec{m}, \vec{d}[d_i := g_i(d_i, e_i)] \rangle) = \langle (i + 1) \bmod n, g_i(\mathbf{d}, e_i) \rangle$

(every visible action takes related states to related states, i.e. if the initial states in X and Y are h -mapped, then the states after executing the action are also h -mapped)

Proof:

1. Let $\langle A, \vec{l}, \vec{m}, \vec{d} \rangle$ be X 's state before the τ -step, (\mathbf{c}, \mathbf{d}) the state in Y mapped from it, $\langle A', \vec{l}', \vec{m}', \vec{d}' \rangle$ X 's state after the τ -step and $(\mathbf{c}', \mathbf{d}')$ its h -mapped Y state. We have to prove that (\mathbf{c}, \mathbf{d}) and $(\mathbf{c}', \mathbf{d}')$ are equal.

- (a) After the τ -step, x and m_x are not changed (because A and \vec{m} didn't change). The only different value is of l_x , but this doesn't affect h 's definition since $l_x := l_x + 1 = 1$ is still in $\{0, 1\}$.

So, $\langle \mathbf{c}', \mathbf{d}' \rangle = \langle x, \mathbf{data}(A', m'_x) \rangle = \langle x, \mathbf{data}(A, m_x) \rangle = \langle \mathbf{c}, \mathbf{d} \rangle$.

- (b) $l_i = 2 \xrightarrow{\text{inv.5, def. } h} \langle \mathbf{c}, \mathbf{d} \rangle = \langle (i + 1) \bmod n, d_i \rangle$.

If $n > 1$ then $i = x \neq (x - 1) \bmod n$. Then from (inv.3) $m_{(i+1) \bmod n} = m_i + 1$, thus $\mathbf{active}(\langle A', \vec{l}', \vec{m}', \vec{d}' \rangle) = (i + 1) \bmod n$. In the new state, $\vec{l} = 0$, which means that $l_{(i+1) \bmod n} = 0$.

$$\begin{aligned} \langle \mathbf{c}', \mathbf{d}' \rangle &\stackrel{\text{def. } h}{=} \langle (i + 1) \bmod n, \mathbf{data}(A, m_{(i+1) \bmod n}) \rangle \\ &= \langle (i + 1) \bmod n, d_i \rangle = \langle \mathbf{c}, \mathbf{d} \rangle. \end{aligned}$$

If $n = 1$ then $i = x = (x + 1) \bmod n$, so $\langle \mathbf{c}, \mathbf{d} \rangle = \langle i, d_i \rangle$. The active process

$(i = x = 1)$ remains active in the new state (inv.2). $\langle \mathbf{c}', \mathbf{d}' \rangle \stackrel{l'_i=0, \text{def. } h}{=} \langle i, \mathbf{data}(A', m'_i) \rangle$. $m'_i = m_i + n$, i.e. $m_i + 1$ and in A' there is $(m_i + 1, d_i) \rightsquigarrow \mathbf{data}(A', m'_i) = d_i \rightsquigarrow \langle \mathbf{c}', \mathbf{d}' \rangle = \langle i, d_i \rangle = \langle \mathbf{c}, \mathbf{d} \rangle$.

2. Let $\langle A, \vec{l}, \vec{m}, \vec{d} \rangle$ denote always the current state.

(a) $l_i = 1 \stackrel{\text{inv.5}}{\rightsquigarrow} i = \mathbf{active}(\langle A, \vec{l}, \vec{m}, \vec{d} \rangle) \stackrel{\text{def. } h}{\rightsquigarrow} \mathbf{c} = i$.

(b) True, because $\mathbf{d} = d_i$ from Lemma 7.7 and $l_i = 1 \longrightarrow \mathbf{c} = i$ was shown at (2(a)).

3. (a) Let i_0 be the i from FC ($l_{i_0} = 1$). Then, from Lemma 7.6(invariant 5) and definition of h , $\mathbf{c} = i_0$. But $\mathbf{c} = i$, also, so $i = i_0$. This means that $l_i = l_{i_0} = 1$.

(b) $l_i = 1$ is shown with the same reasoning as in 3(a). $b_i(d_i, e_i)$ is true because $b_i(\mathbf{d}, e_i)$ is true and $\mathbf{d} = d_i$ (Lemma 7.7).

4. The conditions $b_i(d, e_i)$ are evaluated when $l_i = 1$. By Lemma 7.7 and definition of h , we get $\mathbf{d} = d_i$, so $f_i(d_i, e_i) = f_i(\mathbf{d}, e_i)$.

5. Again, we will denote $\langle A, \vec{l}, \vec{m}, \vec{d} \rangle$ and $\langle A', \vec{l}', \vec{m}', \vec{d}' \rangle$ the states of X before and after executing the discussed action. Similarly, $\langle \mathbf{c}, \mathbf{d} \rangle$ and $\langle \mathbf{c}', \mathbf{d}' \rangle$ are the corresponding states in Y .

(a) $l_i = 1 \stackrel{\text{Lemma 7.6(5), Lemma 7.7}}{\rightsquigarrow} \langle \mathbf{c}, \mathbf{d} \rangle = \langle i, d_i \rangle$.
 $\mathbf{active}(\langle A', \vec{l}', \vec{m}', \vec{d}' \rangle) = (i + 1) \bmod n$, by a reasoning similar to 1(b).
 Because $l'_{(i+1) \bmod n} = 0$, we have $\mathbf{c}' = (i + 1) \bmod n$ and $\mathbf{d}' = d_i$. But according to Lemma 7.7, $\mathbf{d} = d_i$. So, $\langle \mathbf{c}', \mathbf{d}' \rangle$ is indeed $\langle (i + 1) \bmod n, \mathbf{d} \rangle$.

(b) $l'_i = 2 \stackrel{\text{Lemma 7.6(5)}}{\rightsquigarrow} \mathbf{active}(\langle A', \vec{l}', \vec{m}', \vec{d}' \rangle) = i \stackrel{\text{def. } h, l'_i=2}{\rightsquigarrow} \langle \mathbf{c}', \mathbf{d}' \rangle = \langle (i + 1) \bmod n, d'_i \rangle = \langle (i + 1) \bmod n, g_i(d_i, e_i) \rangle$. \square

7.6 Conclusions

We have studied the architecture GSRW, based on write and blocking, non-destructive read primitives on a global set. By viewing the architecture as a separate component defined by process algebra, we obtained a nice separation between the tasks of application programming on the architecture, and the distributed implementation of the architecture itself.

GSRW provides a conceptual global view to application programmers, making the development and analysis of applications easier. Our first result shows that maintaining the global view doesn't lead to any overhead in the distributed implementation, like locking protocols. For this, the limited set of coordination primitives is essential.

Due to these restrictions, application processes just cannot observe that their local set is not (yet) up-to-date. Our second result supports this architecture, by indicating that despite these restrictions, the architecture is sufficiently expressive from a functional point of view.

Finally, non-functional requirements, like performance and fault tolerance might lead to stronger coordination primitives, such as destructive or non-blocking read, as in Linda. However, these don't come for free. Either we have to give up the global view, as shown in [BKZ99a, BKZ99b], or complicated protocols are needed in order to guarantee global consistency, as the two-phase-commit protocol in JavaSpacestm. The former compromises ease of application program construction and analysis, the latter compromises run-time performance.

7.6.1 Related work

In [DL00] a more detailed description of Splice is given, at the level of agents communicating on an Ethernet network. However, an abstract specification of this fragment is not given. Instead the model is validated by verifying that a number of scenarios satisfy certain desired temporal logic properties. The description of Splice in [HP02b] is also more detailed, as it includes a description of time stamps and details the structure of the local databases. That study is devoted to transparent component replication. Two approaches were investigated: a model checking approach applied to a μ CRL description in the same style as here, and a theorem proving approach on a denotational semantics specified in PVS. The same specification style was also applied in [PV02] on the different shared data space architecture JavaSpaces.

The distributed implementation that we give is at the same level of abstraction as in [BHJ00, BKZ99a, BKZ99b]. This is sufficient to show that for read/write primitives a global set is equivalent to a number of local sets. In [BKZ99a, BKZ99b] operational semantics corresponding to these views are given, and it is proved that for each program these views yield behaviorally equivalent semantics. Several other variants were considered, based on e.g. multi-sets and stronger coordination primitives. A semantics of JavaSpaces along the same lines is defined in [BGZ00b]. In [BHJ00] denotational semantics are given for distributed and local versions, and it is proved and formally checked by a proof checker, that both semantics yield the same *write-traces* and end up in the same data space.

Although our realizability result resembles this work, the setting is quite different. As we have the architecture as a separate component, we can prove that the global architecture and its distributed implementation are behaviorally equivalent. Therefore our result is language independent and immediately applies to the case where components may use recursion and internal choice. This combination has not been considered in [BHJ00, BKZ99a, BKZ99b]. The proof we give is simpler in our view, as it mainly consists of checking some simple matching criteria, which are generated

by a standard application of the cones-and-foci method.

In [BHJ00] an imperative language is used with as primitive $read(x, q); P$, which is blocked until some value v satisfying query q exists which is then bound in P to x . We obtain the same effect by the process $\sum_x (read(x).P \triangleleft q(x) \triangleright \delta)$. Instead of the action of writing or reading, those authors regard the arrival in the database observable, which we have hidden by a $\tau_{\{Send\}}$ in DSRW. It is interesting future research to see how their semantics can be formally connected with ours.

Our expressiveness result should be contrasted with the result of [BGZ97], where it is shown that additional primitives, like the test-for-absence, are needed to get Turing completeness. There, components are restricted to finite state machines, and the computation power entirely comes from the coordination primitives. We take a system's engineering view, by focusing on the question whether the read and write primitives are sufficiently expressive for solving the coordination between (probably infinite state) application programs. We also focus on the real task of the components: implement the system's external global behavior.

Our construction has similarities with transformations in [Lan92], where a requirements specification is split in parallel parts communicating via message passing, and [NP96], where an encoding of choice in the a-synchronous π -calculus is provided. Both papers introduce internal loops to resolve external choices, similar to our translation. However, those papers are based on event-based coordination, whereas our approach uses a persistent data approach. For this reason, we had to use increasing sequence numbers, and couldn't find a finite state solution.

8

Verification and Prototyping of Distributed Dataspace Architectures

In this chapter, we focus on the problem of designing, verifying and prototyping distributed shared dataspace systems. Building correct distributed systems is a difficult task. Typical required properties include transparent data distribution and fault-tolerance (by application replication and data replication), which are usually ensured at the price of giving up some performance. Many questions occur when deciding on the exact shape of the distributed dataspace. For instance: what data should be replicated (in order to increase efficiency or to prevent single points of failure)? should the local storages be kept synchronized or should they be allowed to have different views on the global space? should the migration of data between local storages be on a subscription basis or rather on demand?

The space calculus, introduced in this chapter, is an experimental framework in which verification and simulation techniques can be applied to the design of distributed systems that use a shared dataspace to coordinate their components.

We provide a tool that translates a space calculus specification into a μ CRL specification. From this code a state space graph can be generated and analyzed by means of the model checker CADP. A second tool generates distributed C code to simulate the system. System designers may use the automatic verification and simulation possibilities provided by the μ CRL toolset to verify properties of their architecture. Complementary, the distributed C prototype can be used for testing purposes, and to get an indication about the performance (e.g. number of messages, used bandwidth, bottlenecks). Several design choices can be rapidly investigated in this way. Ultimately, the prototype C implementation could even be used as a starting point for building a production version.

The operational semantics of our space calculus provides the formal ground for

algebraic reasoning on architectures. Despite its apparent simplicity, our calculus is highly expressive, capable of modeling various destructive/non destructive, global/local primitives. By restricting the allowed space connectives and the allowed coordination primitives, we obtain well known instances, such as the kernels of Splice and JavaSpaces. Some specific features, like the transactions in JavaSpaces or dynamic publish/subscribe in Splice are out of our scope. Our goal is a uniform framework where core characteristics of various dataspace architectures should be present, in order to allow for studies and comparisons. The verification tool will help getting fast insights in the replication and distribution behavior of certain architectures, for instance. The simulation tool can help to identify the classes of applications appropriate to each architecture.

Related work An overview of shared dataspace coordination models is given in [TRG02]. Some work that studies different semantics has been done in [BKZ99a, BGZ00b, BZ01, BMMZ02], on which we based the style of our operational semantics. [HP02a] proposes a compositional denotational semantics for Splice and proves it equivalent to an operational semantics. [BZ01] compares the publish/subscribe with the shared dataspace architectural style by giving a formal operational semantics to each of them. We also aim at being able to compare the two paradigms, but we take a more unifying perspective: we consider both as being particular instances of the more general distributed dataspace model and express them in the same framework. [CR90] was the first attempt to use a Unity-like logic to reason on a shared dataspace coordination model (Swarm). [LAC00] has goals similar to ours. It provides a framework for describing software architectures in the theorem prover PVS. However, it seems that the verification of functional behavior is out of the scope of that chapter. In [CMP98], a language for specification and reasoning (with TLA) about software architectures based on hierarchical multiple spaces is presented. The focus there is on the design of the coordination infrastructure, rather than on the behavior of systems using it. In [HCS01], a translator from the design language VPL to distributed C++ code is presented. VPL specifications can be verified using the CWB-NC toolset. Compared to that approach, our work is more specific. We concentrate on shared dataspace architectures and define a “library” of carefully chosen primitives that are both handy and expressive. In [DL00], scenario-based verification is introduced as a useful technique in between verification and testing. Our language also supports that.

In Section 8.1, we present the syntax and semantics of the space calculus and we comment its main characteristics. Then (Section 8.2) we introduce the supporting tools. Section 8.3 contains two examples. We end with some concluding remarks (Section 8.4).

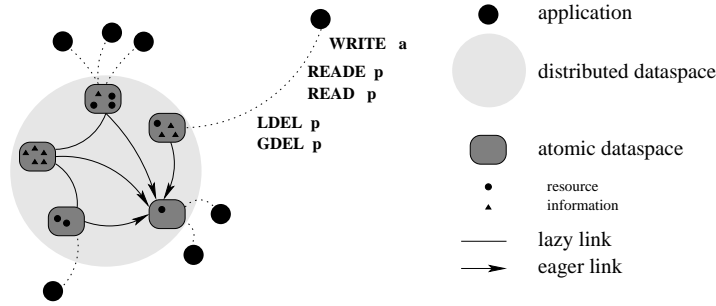


Figure 8.1: A distributed dataspace architecture

8.1 The space calculus

8.1.1 Informal view

We model the shared space as a graph with atomic spaces as nodes (see Figure 8.1). We consider two types of links between spaces: *eager* and *lazy*. When elements are written in a local space, they are asynchronously transferred over all eager links that start in that local space. Eager links are represented in the graph by arcs going from spaces where data is produced (written) to spaces where data should be transferred to and can be used to model subscription and notification mechanisms. On the other hand, the lazy links, represented in the figure by undirected edges, are demand driven. Only when a data item is requested in some atomic space, it is transferred via a lazy link from one of the neighboring spaces. Besides modeling the shared space, the space calculus provides a set of coordination primitives for *applications*: write, blocking and non-blocking read, local and global delete operations. Applications are loosely coupled in the sense that they cannot directly address other applications.

With so many existing shared dataspace models, it is difficult to decide what features are the most representative. Some choices that we are faced with are: atomic spaces can be sets or multisets; when transferring data items between different spaces, they could be replicated or moved; the primitives can be location-aware or location-independent; the retrieve operation can be destructive or non-destructive, etc. The answers depend of course on the specific application or on the purpose of the architecture. In order to allow the modeling of as many situations as possible, we let the user make the distinction between data items that should be treated as *information* (e.g. data from a sensor), for which multiplicity is not relevant, and data items that should be treated as *resource* (e.g. numbers to be added, jobs to be executed), for which multiplicity is essential. When handling elements, the space takes into account their type. The transfer between spaces means copying for information items and moving for resources. Similarly, the lookups requested by applications are destructive for resources and non-destructive for information items.

The atomic spaces are multisets in which the elements tagged as information are allowed to randomly increase their multiplicity. As for the question whether to give to applications the possibility to directly address atomic spaces by using handles, like for instance in [RW97], we have chosen not to, in order to keep the application layer and the coordination layer as separated as possible. The advantage of a clear separation is that the exact distribution of the space is transparent to the applications.

8.1.2 Syntax and semantics

As mentioned before, in our view a system description consists of a number of program applications and a number of connected atomic spaces. We refer to the topology of the distributed space by giving atomic spaces (abstract) locations, denoted by i, j, \dots . The data items come from a set \mathcal{D} of values, ranged over by a, b, \dots . Furthermore, we assume a set of patterns $\mathcal{Pat}(\mathcal{D})$, which are properties that describe subsets of \mathcal{D} . We assume that the patterns describing singletons are the elements of \mathcal{D} themselves, for instance a describes the subset $\{a\}$. Thus, $\mathcal{D} \subseteq \mathcal{Pat}(\mathcal{D})$. p, q, \dots denote patterns. We also postulate two predicates on patterns: $\text{match} : \mathcal{Pat}(\mathcal{D}) \times \mathcal{D} \rightarrow \{\top, \perp\}$ to test if a given pattern matches a given value, and $\text{inf} : \mathcal{Pat}(\mathcal{D}) \rightarrow \{\top, \perp\}$ to specify whether a given pattern should be treated as information or as resource. The predicate $\text{res} : \mathcal{Pat}(\mathcal{D})$ will be used as the complementary of inf .

A process $([P]^i)$ is a program expression P residing at a location i . A program expression is a sequence of coordination primitives: write, read, read if exists, local delete, global delete. These primitives are explained later in this section. Formal parameters in programs are denoted by x, y, \dots , the empty program is denoted by ε , and \perp denotes a special error value.

The lazy and eager behaviors of the connections are specified as special marks: \downarrow_p^i (meaning that atomic space i publishes data matching p), \uparrow_p^i (i subscribes to data matching p), \uparrow_i^j (i and j can reach each other's elements). If \downarrow_p^i and \uparrow_p^j are present, then all data matching $p \wedge q$ written in the space i by an application will be asynchronously forwarded to the space j . We say then that there is an eager link from i to j . The presence of \uparrow_i^j indicates that there is a (symmetric) lazy link from space i to j . That is, all data items of i are visible for retrieve operations issued on to j by an application.

For administrative reasons, the set of data items (a) is extended with buffered items that have to be sent ($!a^j$, a has to be sent to space j), pending request patterns ($?p$, data matching pattern p were requested) and subscription policies (\bigcirc_p^k and $\bigcirc_p^{k,t}$). Subscription policies are inspired by Splice and their function is to filter the data coming into a space as consequence of a subscription. Based on *keys* and *timestamps*, some of the data in the space will be replaced (overwritten) by the newly arrived element. The parameters k, t are functions on data $k : \mathcal{D} \rightarrow \text{Keys}$, $t : \mathcal{D} \rightarrow \mathbb{N}$ that describe how the keys and the timestamps are extracted from data items. If the newly

arrived element a , matching p , meets the filter \bigcirc_p^k , then a will overwrite all data with the key equal to that of a . If it meets the filter $\bigcirc_p^{k,t}$, it will overwrite the data with the key equal to that of a and timestamp not fresher than that of a . With this second filter, it is also possible that the arrival of a is ignored if its timestamp is too old. A configuration (\mathcal{C}) then consists of a number of atomic dataspace and applications, each bound to a location, and a number of links. The parallel composition operator \parallel is associative and commutative.

$$\begin{aligned}
\text{Conf} &::= \text{Data} \mid \text{Proc} \mid \text{Link} \mid \text{Conf} \parallel \text{Conf} \\
\text{Data} &::= \langle D \rangle^i, \text{ where } D \text{ is a finite set over data} \\
\text{Proc} &::= [P]^i \\
\text{Link} &::= \uparrow_i^j \mid \uparrow_p^i \mid \downarrow_p^i \\
\text{data} &::= a \mid !a^j \mid ?p \mid \bigcirc_p^k \mid \bigcirc_p^{k,t} \\
P &::= \varepsilon \mid \text{prim}.P \\
\text{prim} &::= \text{write}(a) \mid \text{read}(p, x) \mid \text{read}\exists(p, x) \mid \text{ldel}(p) \mid \text{gdel}(p)
\end{aligned}$$

The operational semantics of the space calculus is defined by means of a transition relation on configurations, which is defined inductively in Figure 8.2. Note that the operational semantics rules don't explicitly reflect the dual information/resource structure of the systems. This unitary appearance is possible due to a few operators on data, the definition of which (Figure 8.3) encapsulates this distinction. D, B are multisets, $-$ and $+$ denote the usual difference and union of multisets and d is a *data* element (a or $!a^j$ or $?p$ or \bigcirc_p^k or $\bigcirc_p^{k,t}$). We will use the notation $\text{inf}(d)$ to express the value of the predicate inf for the pattern occurring in d . That is, $\text{inf}(!a^j) = \text{inf}(a)$ and $\text{inf}(?p) = \text{inf}(\bigcirc_p^k) = \text{inf}(\bigcirc_p^{k,t}) = \text{inf}(p)$. The same holds for res .

We now explain the intuitive semantics of the coordination primitives:

write(a): write data item a into the local dataspace, to be automatically forwarded to all subscribed spaces. a is added to the local dataspace (W1) and an auxiliary $w(i, a)$ step is introduced. When pushing $w(i, a)$ to the top level, if a matches a pattern published by i , then $!a^j$ items are introduced for all subscriptions \uparrow_p^j matching a (rules W2, W3). At top level, the auxiliary $w(i, a)$ -step gets promoted to a *write*(a)-step (W4). Finally, the a items are sent to the subscribed spaces asynchronously (W5). The operator \uplus in the right-hand side of rule (W5) states that the freshly received data item should be added to the local database taking into account the local subscription policies.

read(p, x): blocking test for presence, in the local space and its lazy linked neighboring spaces, of some item a matching p ; x will be bound to a . This results in generating a $?p$ request, keeping the application blocked (R τ). If a matching a has been found, it is returned and the application is unblocked (R). Meanwhile, the lazy linked neighbors of the local space asynchronously respond to the request $?p$, if they

$$\begin{array}{l}
\text{(W1)} \quad \frac{\langle D \rangle^i \parallel [\text{write}(a).P]^i \xrightarrow{w(i,a,[a])} \langle D \rangle^i \parallel [P]^i}{\mathcal{C} \parallel \langle D \rangle^i \xrightarrow{w(i,a,B)} \mathcal{C}' \parallel \langle D \rangle^i} \\
\text{(W2)} \quad \frac{\mathcal{C} \parallel \langle D \rangle^i \parallel \downarrow_p^i \parallel \uparrow_q^j \xrightarrow{w(i,a,B \boxplus !a^j)} \mathcal{C}' \parallel \langle D \rangle^i \parallel \downarrow_p^i \parallel \uparrow_q^j}{\text{match}(p,a) \wedge \text{match}(q,a)} \\
\text{(W3)} \quad \frac{\mathcal{C} \xrightarrow{w(i,a,B)} \mathcal{C}'}{\mathcal{C} \parallel X \xrightarrow{w(i,a,B)} \mathcal{C}' \parallel X} \\
\quad X \in \{!a^j, [P]^j, \langle D \rangle^j, \downarrow_q^j, \uparrow_p^l\}, \quad p : \neg \text{match}(p,a) \vee \downarrow_p^i \notin \mathcal{C} \\
\text{(W4)} \quad \frac{\mathcal{C} \xrightarrow{w(i,a,B)} \mathcal{C}'}{\mathcal{C} \xrightarrow{w(i,a,B \boxplus a)} \mathcal{C}'} \quad \text{(W5)} \quad \frac{\langle D \rangle^i \parallel \mathcal{C} \xrightarrow{w(i,a,B)} \langle D \rangle^i \parallel \mathcal{C}'}{\langle D \rangle^i \parallel \mathcal{C} \xrightarrow{\text{write}(a)} \langle D \oplus B \rangle^i \parallel \mathcal{C}'} \\
\text{(W6)} \quad \langle D + [!a^j] \rangle^i \parallel \langle D' \rangle^j \xrightarrow{\tau} \langle D \rangle^i \parallel \langle D' \boxplus a \rangle^j \\
\\
\text{(R}\tau\text{)} \quad \langle D \rangle^i \parallel [\text{read}(p,x).P]^i \xrightarrow{\tau} \langle D + [?p] \rangle^i \parallel [\text{read}(p,x).P]^i \quad ?p \notin D \\
\text{(R)} \quad \frac{\langle D + [?p] \rangle^i \parallel [\text{read}(p,x).P]^i \xrightarrow{\text{read}(p,a)} \langle D - [?p] \ominus a \rangle^i \parallel [P[x := a]]^i}{a \in D \wedge \text{match}(p,a)} \\
\text{(R}\exists 1\text{)} \quad \frac{\langle D \rangle^i \parallel [\text{read}\exists(p,x).P]^i \xrightarrow{\text{read}\exists(p,a)} \langle D \ominus a \rangle^i \parallel [P[x := a]]^i}{a \in D \wedge \text{match}(p,a)} \\
\text{(R}\exists 2\text{)} \quad \frac{\langle D \rangle^i \parallel [\text{read}\exists(p,x).P]^i \xrightarrow{\text{read}\exists(p,\perp)} \langle D \rangle^i \parallel [P[x := \perp]]^i}{\nexists a \in D \text{ match}(p,a)} \\
\\
\text{(LD)} \quad \langle D \rangle^i \parallel [l\text{del}(p).P]^i \xrightarrow{l\text{del}(p)} \langle D - [a \in D \mid \text{match}(p,a)] \rangle^i \parallel [P]^i \\
\text{(GD1)} \quad [g\text{del}(p).P]^i \parallel \parallel_j \langle D_j \rangle^j \xrightarrow{g\text{del}(p)} [P]^i \parallel \parallel_j \langle D_j - [a \in D_j \mid \text{match}(p,a)] \rangle^j \\
\text{(GD2)} \quad \frac{\mathcal{C} \xrightarrow{g\text{del}(p)} \mathcal{C}'}{\mathcal{C} \parallel X \xrightarrow{g\text{del}(p)} \mathcal{C}' \parallel X} \quad X \neq \langle D \rangle^i \\
\\
\text{(TAU)} \quad \langle D + [?p] \rangle^i \parallel \parallel_i^j \langle D' \rangle^j \xrightarrow{\tau} \langle D - [?p] \oplus a \rangle^i \parallel \parallel_i^j \langle D' \ominus a \rangle^j \\
\quad a \in D' \wedge \text{match}(p,a) \\
\text{(act)} \quad \frac{\mathcal{C} \xrightarrow{\text{act}} \mathcal{C}'}{\mathcal{C} \parallel \mathcal{C}'' \xrightarrow{\text{act}} \mathcal{C}' \parallel \mathcal{C}''} \quad \text{act} \notin \{g\text{del}(p), \text{write}(a), w(i,a)\}
\end{array}$$

Figure 8.2: Operational semantics of the space calculus

have an item matching p (TAU).

$\text{read}\exists(p,x)$: non-blocking test for presence in the local space. If some item a matching p exists in the local space, it is bound to x ; otherwise a special error value \perp is returned. Delivers a matching a from the local space, if it exists (R $\exists 1$). Otherwise an error value is returned (R $\exists 2$).

$l\text{del}(p)$: atomically removes all elements matching p from the local space (LD).

$g\text{del}(p)$: this is the global remove primitive. It atomically deletes all items matching p , in all atomic spaces. Note that due to its global synchronous nature, $g\text{del}$ can not

$$\begin{array}{lcl}
D \uplus_p a & = & D + [a] \\
D \uplus_{p,k} a & = & D - [b \in D \mid k(b) = k(a)] + [a] \\
D \uplus_{p,k,t} a & = & D - [b \in D \mid k(b) = k(a)] + [a] \\
& & \quad \text{if } \nexists b \in D \, k(b) = k(a) \wedge t(b) > t(a) \\
& & \quad \text{otherwise} \\
D \uplus_d d & = & D - [b \in D \mid b = d] + [d] \\
\\
D \oplus d & = & \begin{cases} D \uplus_d d & \text{if } \text{inf}(d) \\ D + [d] & \text{if } \text{res}(d) \end{cases} \quad D \ominus a = \begin{cases} D & \text{if } \text{inf}(a) \\ D - [a] & \text{if } \text{res}(a) \end{cases} \\
\\
D \oplus [d_1 \cdots d_n] & = & D \oplus d_1 \oplus \cdots \oplus d_n \quad B \boxplus d = \begin{cases} B \uplus_d d & \text{if } \text{inf}(d) \\ [d] & \text{if } \text{res}(d) \end{cases} \\
\\
D \uplus a & = & \begin{cases} D \uplus_{p,k} a & \text{if } \bigcirc_p^k \in D \wedge \text{match}(p, a) \\ D \uplus_{p,k,t} a & \text{if } \bigcirc_p^{k,t} \in D \wedge \text{match}(p, a) \\ D \oplus a & \text{if } \nexists \bigcirc_p^k, \bigcirc_p^{k,t} \in D \text{ s.t. } \text{match}(p, a) \end{cases}
\end{array}$$

Figure 8.3: Auxiliary operators on multisets.

be lifted over atomic spaces (GD2). Finally, the general parallel rule (act) defines parallelism by interleaving, except for *write* and *gdel* which have their own parallel rules to ensure synchronization.

8.1.3 Modeling some dataspace paradigms

The kernels of some well known dataspace paradigms can be obtained by restricting the allowed configurations and primitives.

Splice [Boa93] implements a publish-subscribe paradigm. It has a loose semantics, reflecting the unstable nature of a distributed network. Applications announce themselves as publishers or subscribers of data sorts. Publishers may write data items to their local agents, which are automatically forwarded to the interested subscribers. Typically, the Splice primitives are optimized for real-time performance, and don't guarantee global consistency. The space calculus fragment without lazy links and restricted to the coordination primitives *write*, *read*, *ldel* corresponds to the reliable kernel of Splice. Network searches (modeled by the lazy links) and global deletion (*gdel*) are typically absent. In Splice, data sorts have keys, and data elements with the same key may overwrite each other – namely at the subscriber's location, the fresh data overwrites the old one. The order is given by implicit timestamps that elements get in the moment when they are published. The overwriting is expressible in our calculus, by using the eager links with subscribe policies. Splice's timestamps mechanism is not present, but some timestamping behavior can be mimicked by explicitly

```

Atomic (id:Nat, D:TupleSet, Req: TupleSet,
      ToSend: NatTupleSet, todel:Tuple,
      LL: NatSet, PL: TupleSet, SL: SubscriptionList) =

% W
sum(v:Tuple,
  W(v). sum(NewToSend: NatTupleSet,
            sum(NewD: TupleSet,
                getToSend(v, ToSend, NewToSend, D, NewD).
                Atomic(id, NewD, Req, NewToSend,
                      todel, LL, PL, SL)))
  <| and(isData(v), match(v, PL)) |> delta)
+ sum(v:Tuple,
  W(v).
  Atomic(id, a(v,D), Req, ToSend, todel, LL, PL, SL)
  <| and(isData(v), not(match(v,PL)))|> delta )

% async send
+ sum(x:Nat, sum(y:Tuple,
  el_send(x,y).
  Atomic(id, D, Req, r(x,y,ToSend), todel, LL, PL, SL)
  <| in(x,y,ToSend) |> delta ))

% async receive
+ sum(x:Tuple,
  el_recv(id,x).
  Atomic(id, add(x,D,SL), Req, ToSend, todel, LL, PL, SL))
...

```

Figure 8.4: Fragment from a μ CRL specification of an atomic space

writing and modifying an extra field in the tuples that models the data.

JavaSpaces [FHA99] on the contrary can be viewed as a *global* dataspace. It typically has a centralized implementation, and provides a strongly consistent view to the applications, that can write, read, and take elements from the shared dataspace. The space calculus fragment restricted to a single atomic space to which all coordination primitives are attached, and with the primitives *write*, *read*, *read* \exists forms a fragment of JavaSpaces. Transactions and leasing are not dealt with in our model. Note that with the mechanism of marking the data as being information or resource, we get the behavior of both destructive and non-destructive JavaSpaces lookup primitives: our *read*, *read* \exists works, when used for information, like *read* and *readIfExists* from JavaSpaces, and like *take* and *takeIfExists* when called for resources.

So, interesting parts of different shared dataspace models are expressible in this

framework.

8.2 The verification and prototyping tools

We defined a mapping from every configuration in the operational semantics to a process in the μCRL specification language. An incomplete description of this mapping is given later in this section. The generation of the μCRL specification following this mapping is automated. Therefore, the μCRL toolset can be immediately used to simulate the behavior of a configuration. This toolset is connected to the CADP model checker, so that temporal properties on systems in the space calculus can be automatically verified by model checking. Typical verified properties are deadlock freeness, soundness, weak completeness, equivalence of different specifications.

The state of a μCRL system is the parallel composition of a number of processes. A process is, as explained in Section 6.3, built from atomic actions by sequential composition (\cdot), choice ($+$, \sum), conditionals ($\langle \cdot \triangleright \cdot \rangle$) and recursive definitions. For our purpose, we introduce processes for each atomic space and for each application. An additional process, called the **TokenManager**, has to ensure that operations requiring global synchronization (*gdel*) don't block each other, thus don't introduce deadlocks. Before initiating a global delete operation, a space has to first request and get the token from the manager. When it has finished, it has to return the token to the manager, therefore allowing other spaces to execute their *gdels*. A second additional process, **SubscriptionsManager**, manages the list (multiset) of current subscriptions. When an item a is written to an atomic space, that space synchronizes with the **SubscriptionsManager** in order to get the list of the other atomic spaces where the new item should be replicated or moved.

For simplicity, we model the data items as tuples of natural numbers – fields are modeled by the μCRL datasort **Nat**, tuples by **Tuple**.

An atomic space has two interfaces: one to the application processes, and one to the other atomic spaces. In μCRL calls between processes are modeled as synchronization between atomic actions. The primitives of the space calculus correspond to the following atomic actions of **Atomic**:

$$\{\mathbf{W}, \mathbf{R}, \mathbf{RE}, \mathbf{Ldel}, \mathbf{Togdel}, \mathbf{Gdel}\}.$$

The interface to the other atomic processes is used to send/receive data items and patterns for read requests. In Figure 8.4, the μCRL specification of a space's *write* behavior is shown.

The application programs are also mapped to μCRL processes. Execution of co-ordination primitives is modeled by atomic actions, that synchronize with the corresponding local space's pair actions. This synchronization with the space is described by a communication function.

EXTCOMMAND means $[E][X][T][a - zA - Z]^+$	
INTID means $[i][a - zA - Z0 - 9]^*$	
ID means $[a - zA - Z][a - zA - Z0 - 9]^*$ (that is not INTID)	
INT means $[0 - 9]^+$	
configuration	: settings declarations
settings	: setting settings
setting	: nfields = INT ubound = INT res pattern
declarations	: space declarations link declarations application declarations
space	: space ID (ID) space ID
link	: LL (ID , ID) ID - > pattern ID < - pattern ID < - pattern intlist ID < - pattern intlist INT
pattern	: < tuple >
tuple	: datum tuple , datum
datum	: * INT INTID
intlist	: INT intlist , INT
intexpression	: INT INTID projection intexpression + intexpression
projection	: pattern / INTID ID / INTID

Figure 8.5: The YACC style syntax definition

Another tool translates space calculus specifications to a distributed implementation in C that uses MPI for process communication. Different machines can be specified for different locations, thus getting a real distribution of spaces and applications. By instrumenting this code, relevant performance measures for a particular system under design can be computed. The result of the translation is more than a software simulation. It is actually a prototype, that can be tested in real-time conditions, in a distributed environment.

application	:	app ID @ ID { program }
program	:	
		command ; program
command	:	write pattern
		write ID
		read pattern ID
		readE pattern ID
		ID := pattern
		INTID := intexpression
		ldel pattern
		gdel pattern
		publish pattern
		subscribe pattern subscribe pattern intlist
		subscribe pattern intlist INT
		if condition { program }
		while condition { program }
		EXTCOMMAND
condition	:	ID not(ID) true false

Figure 8.6: The YACC style syntax definition - continued

8.2.1 The space calculus tool language

In order to make the space calculus usable as specification language, the tools supporting it work with a concrete syntax. The data universe considered is *tuples of naturals* and the patterns are incomplete tuples (e.g. $\langle 1, *, 2 \rangle, \langle * \rangle$). Apart from the syntactical constructions already defined, we allow external actions (e.g. **EXTping**), *assignment* of data variables, assignment of tuple variables and *if* and *while* with standard semantics. Now we give a brief description of this language, including a precise syntax written in a slightly simplified YACC format.

Since we allow exactly one space per location, it is nice to give *names* to spaces and to say, instead of saying that a program stays at location i , that the program runs at the space $\langle \text{name} \rangle$. A specification of a configuration consists of:

- (optional) fixing the tuple size (**nfields**) and the first natural value strictly greater than any field of any tuple (**upbound**). The default values are $\text{nfields}=1$, $\text{upbound}=2$.
- (optional) define the **inf/res** predicates, by mentioning the patterns for which **res** should be \top . Any pattern p not included by the **res** declaration has $\text{inf}(p) = \top$.
- describing the spaces, by giving each space a name and, optionally, the machine where it's supposed to live. The default machine is "localhost".
- describing the applications, by specifying for each application its name, the name of the space with which it shares the location (the physical location as well) and its program.

<pre> nfields = 1 upbound = 2 res <*> space JS (mik.sen.cwi.nl) app Ping@JS { write <1>; EXTping; read <0> x; write <1>; EXTping; read <0> x; } app Pong@JS { read <1> x; write <0>; EXTpong; read <1> x; write <0>; EXTpong; } </pre>	<pre> nfields = 1 upbound = 2 res <*> space JS (mik.sen.cwi.nl) space JSbis (boeg.sen.cwi.nl) JS -> <*> JS <- <*> JSbis -> <*> JSbis <- <*> app Ping@JS { write <1>; EXTping; read <0> x; write <1>; EXTping; read <0> x; } app Pong@JSbis { read <1> x; write <0>; EXTpong; read <1> x; write <0>; EXTpong; } </pre>
--	---

Figure 8.7: A Ping-Pong application on one JavaSpace (left) and on two (right)

Apart from the primitives **read**, **readE**, **write**, **ldel**, **gdel**, the actual language includes some extra constructions to provide easy data manipulation and control possibilities: natural variable names and expressions, projection of a tuple on a field, assignments, *if*, *while*, external actions that can be specified as strings.

The condition of *if* and *while* is very simple: a standard boolean value or a variable name that gets tested for correctness. Namely, “*if x*” means “if *x* is not *error*”. Extending the conditions is further work.

The key and timestamp functions needed in the subscription policies are considered to be projections on the fields of the tuples – one field for the timestamp, possibly more for the key. Therefore, key functions are represented as lists of field indexes and timestamps functions as one field index.

```

nfields = 3
upbound = 3
space A1
space A2
space A3
A1 -> <1,*,*>
A2 -> <2,*,*>
A2 <- <1,*,*> 1 3
A3 <- <2,*,*> 1 3

app Producer@A1 {
  itsp := 0; EXTin;
  write <1,0,itsp>;
  itsp := itsp + 1;
  write <1,1,itsp>;
}
app Transformer@A2 {
  while (true) {
    read <1,*,*> x;
    ivx := x/2+1;
    itx := x/3;
    write <2,ivx,itx>;
  };
}
app Consumer@A3 {
  while (true) {
    read <2,*,*> x;
    EXTout;
  };
}

nfields = 3
upbound = 3
space A1
space A2
space A3
space A4
A1 -> <1,*,*>
A2 -> <2,*,*>
A2 <- <1,*,*> 1 3
A3 <- <2,*,*> 1 3
A4 -> <2,*,*>
A4 <- <1,*,*> 1 3

app Producer@A1 {
  itsp := 0; EXTin;
  write <1,0,itsp>;
  itsp := itsp + 1;
  write <1,1,itsp>;
}
app Transformer@A2 {
  while (true) {
    read <1,*,*> x;
    ivx := x/2+1;
    itx := x/3;
    write <2,ivx,itx>;
  };
}
app Transformer@A4 {
  while (true) {
    read <1,*,*> x;
    ivx := x/2+1;
    itx := x/3;
    write <2,ivx,itx>;
  };
}
app Consumer@A3 {
  while (true) {
    read <2,*,*> x;
    EXTout;
  };
}

```

Figure 8.8: The Producer/Consumer/Transformer application with one (left) and two (right) transformers

8.3 Examples

We use the new language to specify two small existing applications, studied in [PV02] and [HP02b], respectively. The goal of these examples is to show that our language is very simple to use and to illustrate the typical kind of problems that space calculus is meant for: transparent distribution of data and transparent replication of applications.

8.3.1 Towards distributed JavaSpaces

One of the initial motivations of our work was to model a distributed implementation of JavaSpaces, still providing the same strongly consistent view to the applications. When restricting the primitives as discussed in Section 8.1.3, the expression $\langle \emptyset \rangle^0$ represents the kernel of JavaSpaces and the expression $\langle \emptyset \rangle^0 \parallel \langle \emptyset \rangle^1 \parallel \downarrow_\star^0 \parallel \uparrow_\star^0 \parallel \downarrow_\star^1 \parallel \uparrow_\star^1$ models a distributed implementation of it, consisting of two spaces eagerly linked by subscriptions matching any item.

Two rounds of the Ping-Pong game [FHA99, PV02] can be written in the space calculus as follows:

$$\begin{aligned} Ping &= write(1).read(0, x).write(1).read(0, x) \\ Pong &= read(1, x).write(0).read(1, x).write(0) \end{aligned}$$

(with $\mathcal{D} = \{0, 1\}$ and $\inf(x) = \perp, \forall x$). We wish that the distribution of the space should be completely transparent to the applications, i.e. that they run on one space exactly the same that they run on two:

$$[Ping]^0 \parallel [Pong]^0 \parallel \langle \emptyset \rangle^0 = [Ping]^0 \parallel [Pong]^1 \parallel \langle \emptyset \rangle^0 \parallel \langle \emptyset \rangle^1 \parallel \downarrow_\star^0 \parallel \uparrow_\star^0 \parallel \downarrow_\star^1 \parallel \uparrow_\star^1$$

We have checked this equivalence by writing the two specifications of the Ping-Pong game (with a single, respectively replicated space) in the tool syntax (Figure 8.8(a)), generating the two state spaces and using the model checker provided by the CADP toolset to verify that they satisfy the *safety equivalence* relation (defined in Section 2.3).

8.3.2 Transparent replication of some Splice applications

Some of the most interesting problems in a system with components are associated with *replication*: which components can be replicated and at what costs? We claim that the space calculus is a good framework for studying this type of questions. In the sequel we give an example of how our space calculus can be used to rapidly check transparent replication of some applications on Splice.

Consider a simple Splice system, composed of three applications: a *Producer* that writes data to the Splice network, based on observations that it makes on the envi-

ronment; a *Transformer* that reads the data, applies some transformations on it and writes it back; and a *Consumer* that reads the transformed data items and uses it further, for instance by displaying it on a screen. The producer and the consumer are the components that interact with the environment, while the transformer works “under water”. Therefore it is reasonable to ask whether it is possible to replicate the transformer without affecting the (external) behavior of the system.

This producer-transformer-consumer example illustrates a specific pattern in Splice systems. The transparent replication of the middle component was extensively studied in [HP02b], using both μCRL and PVS. We show how to model the problem in space calculus (Figure 8.8(b)), for the specific instance when two data items are produced, with values 0 and 1. The `itsp` variable models the local clock. The two specifications have been proved safety equivalent by the CADP model checker.

8.4 Conclusions

This chapter presents our ideas on a unifying framework for the design and analysis of various distributed dataspace systems. We introduced the space calculus, in which basic concepts of some dataspace paradigms can be modeled. A formal syntax and operational semantics provides a rigorous basis to this calculus.

We aim at two goals: comparing the various paradigms with respect to their meta-properties and facilitating the analysis of individual systems based on heterogeneous shared dataspace architectures.

For the first goal, we view a particular dataspace paradigm as a fragment of the space calculus and we address questions like: does a fragment admit transparent replication of data/processes, what are the costs of a distributed implementation, what are the typical applications for a certain fragment. An answer to the last question would facilitate early architectural design decisions. Some of these questions have been answered for Splice already [DJ00, HP02b, OP02].

The second goal is supported by automatic translations to μCRL and to C. The μCRL specifications can be used as an input to a model checker, thus formally establishing the functional correctness of a system. The approach follows a previous successful attempt for JavaSpaces [PV02]. The distributed C simulator can be used to find performance bottlenecks in the high-level architecture. These could be solved by transforming the space calculus expression to a functionally equivalent one with a better performance.

9

Directions for Future Research

In these last pages we briefly comment and speculate on possible interesting directions of future work.

Distributed verification

Parallel algorithms for computing strong bisimulation equivalence exist in the literature, based both on the Kanellakis-Smolka and Paige-Tarjan approach [JKOK98, RL98]. They are designed for shared memory machines (PRAM, Parallel Random Access Machines), but it seems they are easily mapped to distributed memory machines, with at most a logarithmic factor of slow down [Lei92]. Therefore, it would be interesting to see how they work on virtual shared memory; we expect that the latency of the shared memory simulation would seriously affect their performance.

There are many ways to improve our collection of distributed reduction tools. First and most important is the development of (distributed) signature refinement algorithms for minimization modulo other equivalences, like weak bisimulation and safety equivalence. Prototype implementations and preliminary correctness proofs already exist.

We have argued, and verified in practice, that the two proposed algorithms for strong bisimulation reduction (Chapter 3) complement each other: the naive approach functions better on work-intensive iterations, while the optimized version is better on iterations where few changes are performed. Therefore, a hybrid algorithm, that could switch between the algorithms as dictated by the current situation, is also an interesting topic of further research.

Recent developments suggest that our optimized algorithm can reach a theoretical time complexity of $\mathcal{O}(N \log N)$. To obtain this in practice, a complete redesign of the implementation, based on carefully chosen data structures, will be necessary.

Another potential improvement regards the branching bisimulation tool. By integrating the distributed τ -cycle elimination algorithm (Chapter 5) as preprocessing phase, the branching bisimulation reduction tool in Chapter 4 could visibly benefit.

Finally, more clever ways to distribute the state space, possibly dynamically, remain to be found. One of the ideas is exploiting the information about the structure of the state space that can be obtained through abstract interpretation [OPV04].

Building a distributed model checker in the style of XTL [MG98] also comprises a nice continuation of the work presented in this thesis. This would be the last piece to a complete distributed model checking solution for the μ CRL toolset.

Verified distribution

In Chapter 7 we studied the simplest possible shared dataspace coordination model. We proved that *any* requirements specification has a “maximal” distributed implementation on this model, in the sense that every single step is executed on another location. A challenging future research direction is to investigate distribution implementations based on other criteria. We could look for “efficient” implementations, i.e. schemes that would minimize the number of communication steps (i.e. interactions with the database), or schemes that place all actions of a certain type on the same location, etc. To this end, it might be necessary to add new primitives to the current GSRW model or to consider weaker equivalences between specification and implementation. Another interesting possibility is to settle our conjecture that the compositionality obtained in section 7.4 is the basis of transparent agent and application replication.

The space calculus tool language (Chapter 8) is, at present, not sufficiently expressive. In order to be practically useful, it should be embedded in a well developed modeling language, that would allow the specification of realistic applications on top of the distributed space. More examples and case studies will provide further support and feedback on the proposed framework. An extension that would bring the space calculus closer to modeling real-life examples is allowing dynamic creation of spaces and applications and dynamic change of the link structure. The investigation of meta-properties for (fragments) of the space calculus and of behaviour-preserving transformation rules is an enterprise in its own.

Bibliography

- [ABG91] C. Alvarez, J. L. Balcazar, and J. Gabarro. Parallel complexity in the design and analysis of concurrent systems. In *Proceedings PARLE'91*, volume 506 of *LNCS*, pages 288–303, 1991.
- [AHU83] A. Aho, J. Hopcroft, and J. Ullman. *Data structures and algorithms*. Addison-Wesley, 1983.
- [AW98] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 1998.
- [BAdB⁺00] M.M. Bonsangue, F. Arbab, J.W. de Bakker, J.J.M.M. Rutten, A. Scutella, and G. Zavattaro. A transition system semantics for the control-driven coordination language manifold. *TCS*, 240(1):3–47, 2000.
- [Bas96] T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.
- [BBC03] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model checking. In *Proceedings ASE'03*, pages 106–115. IEEE Computer Society, 2003.
- [BBK87] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51(1-2):129–176, 1987.
- [BBS01] J. Barnat, L. Brim, and J. Štříbrná. Distributed LTL model-checking in SPIN. In *Proceedings SPIN'01*, volume 2057 of *LNCS*, pages 200–216, 2001.
- [BCG88] M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.
- [BČKP01] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In *Proceedings FSTTCS'01*, volume 2245 of *LNCS*, pages 96–107, 2001.
- [BFG⁺91] A. Bouajjani, J.C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. In *Proceedings ICALP'91*, volume 510 of *LNCS*, pages 76–92, 1991.
- [BFG⁺01] S.C.C. Blom, W.J. Fokink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μ CRL: A toolset for analysing algebraic specifications. In *Proceedings CAV'01*, volume 2102 of *LNCS*, pages 250–254, 2001.
- [BG94] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In *Proceedings CONCUR'94*, volume 836 of *LNCS*, pages 401–416, 1994.

- [BG01] D. Bustan and O. Grumberg. Modular minimization of deterministic finite-state machines. In *Proceedings FMICS'01*, volume 6, pages 163–178, 2001.
- [BGS92] J.L. Balcazar, J. Gabarro, and M. Santha. Deciding bisimilarity is \mathcal{P} -complete. *Formal Aspects of Computing*, 4(6A):638–648, 1992.
- [BGZ97] N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing equivalence of Linda coordination primitives. In *Proceedings Express'97*, volume 7 of *ENTCS*, 1997.
- [BGZ00a] N. Busi, R. Gorrieri, and G. Zavattaro. On the semantics of JavaSpaces. In *Proceedings FMOODS'00*, pages 3–19, 2000.
- [BGZ00b] N. Busi, R. Gorrieri, and G. Zavattaro. Process calculi for coordination: From Linda to JavaSpaces. In *Proceedings AMAST'00*, volume 1816 of *LNCS*, 2000.
- [BHJ00] R. Bloo, J.M. Hooman, and E. de Jong. Semantical aspects of an architecture for distributed embedded systems. In *Proceedings SAC'00*, pages 149–155, 2000.
- [BHR84] D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.
- [BHV00] G. Behrmann, T. Hune, and F.W. Vaandrager. Distributed timed model checking - How the search order matters. In *Proceedings CAV'00*, volume 1855 of *LNCS*, pages 216–231, 2000.
- [BK85] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.
- [BKZ99a] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *Proceedings SAC'99*, pages 146 – 155, 1999.
- [BKZ99b] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing software architectures for coordination languages. In *Proceedings COORDINATION'99*, volume 1594 of *LNCS*, pages 150–164, 1999.
- [BLL03] S.C.C. Blom, I. van Langevelde, and B. Lissner. Compressed and distributed file formats for labeled transition systems. In *Proceedings PDMC'03*, volume 89 of *ENTCS*, 2003.
- [BLW01] B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free μ -calculus. In *Proceedings TACAS'01*, volume 2031 of *LNCS*, pages 543–558, 2001.
- [BMMZ02] N. Busi, C. Manfredini, A. Montresor, and G. Zavattaro. Towards a data-driven coordination infrastructure for peer-to-peer systems. In *Proceedings P2P'02*, volume 2376 of *LNCS*, 2002.
- [BO02] S.C.C. Blom and S.M. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. In *Proceedings PDMC'02*, volume 68 of *ENTCS*, 2002.
- [BO03a] S.C.C. Blom and S.M. Orzan. Distributed branching bisimulation reduction of state spaces. In *Proceedings PDMC'03*, volume 89 of *ENTCS*, 2003.
- [BO03b] S.C.C. Blom and S.M. Orzan. Distributed state space minimization. In *Proceedings FMICS'03*, volume 80 of *ENTCS*, 2003.

- [BO05] S.C.C. Blom and S.M. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *STTT*, 2005. To appear.
- [Boa93] M. Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, 1993.
- [BPS01] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, 2001.
- [BT01] L. Buš and P. Tvrdík. A parallel algorithm for connected components on distributed memory machines. In *Proceedings of Euro PVM/MPI*, volume 2131 of *LNCS*, 2001.
- [BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.
- [BZ01] N. Busi and G. Zavattaro. Publish/subscribe vs. shared dataspace coordination infrastructures. In *Proceedings WETICE'01*, 2001.
- [CCM01] S. Caselli, G. Conte, and P. Marenzoni. A distributed algorithm for GSPN reachability graph generation. *Journal of Parallel and Distributed Computing*, 61(1):79–95, 2001.
- [CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
- [CGP00] E.M. Clarke, O. Grumberg, and A. Peled. *Model checking*. MIT press, 2000.
- [CI] CWI/SEN2 and INRIA/VASY. The VLTS benchmark. http://www.inrialpes.fr/vasy/cadp/resources/benchmark{_}bcg.html.
- [Cia01] G. Ciardo. Distributed and structured analysis approaches to study large and complex systems. In *Lectures on Formal Methods and Performance Analysis : First EEf/Euro Summer School on Trends in Computer Science*, volume 2090 of *LNCS*, pages 244–274. 2001.
- [CLRT00] P.H. Carns, W.B. III Ligon, R.B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.
- [CMP98] P. Ciancarini, M. Mazza, and L. Pazzaglia. A logic for a coordination model with multiple spaces. *Science of Computer Programming*, 31(2–3):231–261, 1998.
- [ČP03] I. Černá and R. Pelánek. Distributed explicit fair cycle detection (set based approach). In *Proceedings SPIN'03*, volume 2648 of *LNCS*, pages 49–73, 2003.
- [CR90] H. Cunningham and G.-C. Roman. A Unity-style programming logic for shared dataspace programs. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):365–376, 1990.
- [CV89] R. Cole and U. Vishkin. Faster optimal prefix sums and list ranking. *Information and Computation*, 81:334–352, 1989.
- [Der] Nachum Dershowitz. Software horror stories. <http://www.cs.tau.ac.il/~nachumd/horror.html>.

- [DGJU99] P.F.G. Dechering, R. Groenboom, E. de Jong, and J.T. Udding. Formalization of a software architecture for embedded systems: a process algebra for SPLICE. In *Proceedings HICSS'99*, 1999.
- [DJ00] P.F.G. Dechering and E. de Jong. Transparent object replication: A formal model. In *Proceedings WORDS'99*, 2000.
- [DL00] P.F.G. Dechering and I.A. van Langevelde. The verification of coordination. In *Proceedings COORDINATION'00*, volume 1906 of *LNCS*, 2000.
- [DNFP98] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agent interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
- [Fer90] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3):219–236, 1990.
- [FGK⁺96] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In *Proceedings CAV'96*, volume 1102 of *LNCS*, pages 437–440, 1996.
- [FGP⁺04] W. Fokkink, J.F. Groote, J. Pang, B. Badban, and J.C. van de Pol. Verifying a sliding window protocol in μ CRL. In *Proceedings AMAST'04*, *LNCS*, 2004.
- [FHA99] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, 1999.
- [FHP00] L.K. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *Proceedings Irregular'00*, volume 1800 of *LNCS*, pages 505–512, 2000.
- [Fok00] W. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science (EATCS). Springer-Verlag, 2000.
- [GH03] J.F. Groote and F. van Ham. Large state space visualization. In *Proceedings TACAS'03*, volume 2619 of *LNCS*, 2003.
- [Gla01] R.J. van Glabbeek. The linear time - branching time spectrum i. In Bergstra et al. [BPS01], pages 3–99.
- [GM88] H. Gazit and L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Information Processing Letters*, 28:61–65, 1988.
- [GM97] H. Garavel and L. Mounier. Specification and verification of various distributed leader election algorithms for unidirection ring networks. *Science of Computer Programming*, 29(1–2):171–197, 1997.
- [GMS01] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proceedings SPIN'01*, volume 2057 of *LNCS*, pages 217–234, 2001.
- [GPW03] J.F. Groote, J. Pang, and A.G. Wouters. Analyzing a distributed system for lifting trucks. *Journal of Logic and Algebraic Programming*, 55(1–2):21–56, 2003.
- [GR01] J.F. Groote and M.A. Reniers. Algebraic process verification. In Bergstra et al. [BPS01], pages 1151–1208.

- [GS01] J.F. Groote and J.S. 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.
- [GV90] J.F. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proceedings ICALP'90*, volume 443 of *LNCS*, pages 626–638, 1990.
- [GW96] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.
- [HBB99] B.R. Haverkort, A. Bell, and H.C. Bohnenkamp. On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. In *Proceedings PNPM'99*, pages 12–21, 1999.
- [HCS01] D. Hansel, R. Cleaveland, and S. Smolka. Distributed prototyping from validated specifications. In *Proceedings RSP'01*, pages 97–102, 2001.
- [HH01] U. Hannemann and J. Hooman. Formal design of real-time components on a shared data space architecture. In *Proceedings COMPSAC'01*, IEEE, 2001.
- [HM80] M. Hennessey and R. Milner. On observing nondeterminism and concurrency. In *Proceedings ICALP'80*, *LNCS*, pages 295–309, 1980.
- [HMB01] W.M. Hesselink, A. Meijster, and C. Bron. Concurrent determination of connected components. *Science of Computer Programming*, 41:173–194, 2001.
- [HP02a] J.M. Hooman and J.C. van de Pol. Equivalent semantic models for a distributed dataspace architecture. In *Proceedings FMCO'02*, volume 2852 of *LNCS*, 2002.
- [HP02b] J.M. Hooman and J.C. van de Pol. Formal verification of replication on a distributed data space architecture. In *Proceedings SAC'02*, pages 351–358, 2002.
- [JKOK98] C. Jeong, Y. Kim, Y. Oh, and H. Kim. A faster parallel implementation of Kanellakis-Smolka algorithm for bisimilarity checking. In *Proceedings of the International Computer Symposium*, 1998.
- [JM04] C. Joubert and R. Mateescu. Distributed on-the-fly equivalence checking. In *Proceedings PDMC'04*, *ENTCS*, 2004.
- [Jou03] C. Joubert. Distributed model checking: From abstract algorithms to concrete implementations. In *Proceedings PDMC'03*, volume 89 of *ENTCS*, 2003.
- [KR90] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter Algorithms and Complexity, pages 869–932. MIT Press, 1990.
- [KS83] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes and three problems of equivalence. In *Proceedings of 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 228–240, 1983.
- [KS98] H. Korver and A. Sellink. A formal axiomatization for alphabet reasoning with parametrized processes. *Formal Aspects of Computing*, 10(1):30–42, 1998.

- [LAC00] K. Lichtner, P. Alencar, and D. Cowan. A framework for software architecture verification. In *Proceedings ASWEC'00*, pages 149–158, 2000.
- [Lan92] R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, Department of Computer Science, University of Twente, 1992.
- [Lei92] T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, 1992.
- [LS99] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings SPIN'00*, volume 1680 of *LNCS*, 1999.
- [Lut97] S.P. Luttik. Description and formal specification of the Link Layer of P1394. In *Proceedings of the 2nd International Workshop on Applied Formal Methods in System Design*, 1997.
- [Mad92] E. Madelaine. Verification tools from the concur project. *EATCS Bulletin*, 47, 1992.
- [MG98] R. Mateescu and H. Garavel. XTL: A meta-language and tool for temporal logic model-checking. In *Proceedings STTT'98*, number NS-98-4 in BRICS Notes Series, 1998.
- [MIHPR01] W. C. McLendon III, B.A. Hendrickson, S.J. Plimpton, and L. Rauchwerger. Identifying strongly connected components in parallel. In *Proceedings SIAM PP01*, 2001.
- [Mil80] R. Milner. A calculus of communicating systems. volume 92 of *LNCS*, 1980.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [MSA04] M. Mousavi, M. Sirjani, and F. Arbab. Specification, simulation, and verification of component connectors in Reo. Technical Report 04-15, Department of Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 2004.
- [NP96] U. Nestmann and B.C. Pierce. Decoding choice encodings. In *Proceedings CONCUR'96*, volume 1119 of *LNCS*, pages 179–194, 1996.
- [OP02] S.M. Orzan and J.C. van de Pol. Distribution of a simple shared dataspace architecture. In *Proceedings FOCLASA'02*, volume 68 of *ENTCS*, 2002.
- [OP03] S.M. Orzan and J.C. van de Pol. Verification of distributed dataspace architectures. In *Proceedings PSI'03*, volume 2890 of *LNCS*, 2003.
- [OPV04] S.M. Orzan, J.C. van de Pol, and M. Valero Espada. A state space distribution policy based on abstract interpretation. In *Proceedings PDMC'04*, ENTCS, 2004.
- [PA98] G.A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46:330–396, 1998.
- [Par81] D.M.R. Park. Concurrency and automata on infinite sequences. In *Proceedings GI'81*, volume 104 of *LNCS*, pages 167–183, 1981.
- [Pel04] R. Pelánek. Typical structural properties of state spaces. In *Proceedings SPIN'04*, volume 2989 of *LNCS*, 2004.

- [PFHV03] J. Pang, W.J. Fokkink, R. Hofman, and R. Veldema. Model checking a cache coherence protocol for a Java DSM implementation. In *Proceedings FMPPTA'03*, 2003.
- [PT87] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.
- [PV02] J.C. van de Pol and M. Valero Espada. Formal specification of JavaSpacesTM architecture using μ CRL. In *Proceedings COORDINATION'02*, volume 2315 of *LNCS*, pages 274–290, 2002.
- [PV03] J.C. van de Pol and M. Valero Espada. Verification of JavaSpaces parallel programs. In *Proceedings ACSD'03*, pages 196–205, 2003.
- [Rei85] J.H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.
- [RL98] S. Rajasekaran and I. Lee. Parallel algorithms for relational coarsest partition problems. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):687–699, 1998.
- [Row98] A.I.T. Rowstron. WCL: A co-ordination language for geographically distributed agents. *World Wide Web*, 1(3):167–179, 1998.
- [RW97] A.I.T. Rowstron and A.M. Wood. Bonita: a set of tuple space primitives for distributed coordination. In *Proceedings HICSS'97*, pages 379–388, 1997.
- [SD97] U. Stern and D. Dill. Parallelizing the Mur ϕ verifier. In *Proceedings CAV'97*, volume 1254 of *LNCS*, pages 256–278, 1997.
- [SL03] J.M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings of the 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *LNCS*, 2003.
- [sok] Sokoban. <http://www.cs.ualberta.ca/~games/Sokoban/>.
- [SZ98] C. Shankland and M. van der Zwaag. The tree identify protocol of IEEE 1394 in μ CRL. *Formal Aspects of Computing*, 10:509–531, 1998.
- [Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.
- [TLG03] F. Tronel, F. Lang, and H. Garavel. Compositional verification using CADP, of the ScalAgent deployment protocol for software components. In *Proceedings FMOODS'03*, volume 2884 of *LNCS*, pages 244 – 260, 2003.
- [TRG02] R. Tolksdorf and G. Rojec-Goldmann. The SPACETUB models and framework. In *Proceedings COORDINATION'02*, pages 348–363, 2002.
- [Use02] Y.S. Usenko. *Linearization in μ CRL*. PhD thesis, Eindhoven University of Technology, 2002.
- [ZS93] S. Zhang and S.A. Smolka. Towards efficient parallelization of equivalence checking algorithms. In *Proceedings FORTE'92*, volume C-10 of *IFIP Transactions*, pages 133–146, 1993.

Summary

This thesis consists of two parts that have in common the themes of *distribution* and *verification*.

Part I considers the question of whether automatic verification can be distributed in order to make possible the use of the processing power and memory of a network of computers instead of just one computer.

We investigate this in the specific case of verification by enumerative model checking. To verify a system, its state space, a concrete representation of the system's behavior, is automatically generated. Then, the state space is reduced modulo an equivalence that preserves all relevant properties, and inspected in order to validate desired properties or to find counterexamples. The only but serious drawback of this approach is the familiar *state space explosion problem*: the size of the state space grows much faster than the complexity of the system represented. A recent approach to this problem, that we also pursue, is building parallel and distributed tools. They allow larger state spaces to be handled and thus increase the applicability of formal verification techniques to real life industrial systems.

We are particularly interested in the problem of reducing state spaces modulo behavioral equivalences. Very good sequential solutions for this already exist in the literature, as well as some adaptations to the parallel shared memory setting. We propose in Chapters 3 and 4 new distributed memory algorithms for state space reduction modulo strong and branching bisimulation equivalence. They use communication by message passing, where, unlike in the case of shared memory, latency plays an essential role. The three algorithms presented draw their main inspiration from the sequential ones of Kanellakis and Smolka [KS83]. We prove their correctness and show by experiments with prototype distributed implementations that both the run times and memory usage scale up with the number of machines used. This means that larger state spaces can now be generated and reduced using cheap clusters of workstations. Most of the time, the reduced state space is small enough to further allow model checking by sequential tools.

Further, we approach the well known problem of decomposing a graph into its strongly connected components (Chapter 5). Since solving an arbitrary graph problem can usually be translated into solving the same problem for the graph's strongly connected components, the decomposition algorithms are applicable in all domains where graphs are used. We are studying it having in mind its application in verification (LTL/CTL model checking and branching bisimulation equivalence reduction). Our distributed message passing solution is based on a series of heuristics that work best for the special type of graphs representing state spaces. The prototype implementation shows promising results, though it has not yet been integrated with the tool for branching bisimulation reduction, nor has it actually been used in model checking.

Part II of the thesis addresses the question whether formal verification methods can be of help in understanding and designing distributed software architectures. We investigate this for the specific case of software architectures that use a shared dataspace to coordinate their distributed components.

We demonstrate the use of powerful process algebraic techniques in modeling and analysis of a simple shared dataspace software architecture with write and blocking non-destructive read as its only primitives (Chapter 7). Process algebra allows the description of the shared dataspace as a separate process, and thus a natural separation between the computation and coordination layers is achieved. We show that, due essentially to the nondestructive character of the read primitive, the dataspace can be implemented in a fully distributed manner, while keeping the global uniform view. Despite the restricted set of primitives, this simple architecture is functionally very expressive: any system requirements specification can be implemented on it.

Due to their simplicity and symmetric treatment of components, shared dataspace coordination architectures have been intensively studied and many variants have been implemented (Linda, Bonita, WCL, TSpaces, JavaSpaces). The implementations range from one central server (like JavaSpaces), to which all components address requests, to a full distribution, where every component has its own local copy of the repository (like Splice). From the verification point of view, it is interesting to understand how the different implementations affect the functionality and the performance of such a system. We investigate this in Chapter 8, where we take a unifying view and build a design framework that allows modeling and verification of shared dataspace systems with various sets of primitives and various degrees of distribution. The framework consists of a small specification language and tools that support verification and prototyping. Verification is done by translation to the more general specification language μCRL and prototyping is done by a transformation into distributed C programs.

Samenvatting

Over gedistribueerde verificatie en geverifieerde distributie

Dit proefschrift bestaat uit twee delen die de thema's *distributie* en *verificatie* gemeenschappelijk hebben.

Deel I gaat over de vraag of automatische verificatie gedistribueerd kan worden, zodat het mogelijk wordt om de rekenkracht en het geheugen van een heel netwerk van computers in te zetten bij het verifiëren van een systeem.

We onderzoeken dit in het specifieke geval van verificatie door middel van *enumerative model checking*. Daarbij wordt, ten behoeve van de verificatie van een systeem, eerst zijn hele toestandsruimte, een representatie van het systeemgedrag, gegenereerd. Vervolgens wordt deze toestandsruimte gereduceerd modulo een gedragsequivalentie die alle relevante eigenschappen bewaart. En tenslotte wordt de gereduceerde toestandsruimte geïnspecteerd om de gewenste eigenschappen te valideren, danwel tegenvoorbeelden te vinden. Het enige, maar belangrijke nadeel aan deze aanpak is de combinatorische explosie van de toestandsruimte: de omvang van de toestandsruimte groeit veel sneller dan de complexiteit van het gerepresenteerde systeem. Een recente manier om toch complexere systemen aan te kunnen, is door parallelle en gedistribueerde tools te bouwen. Deze maken het mogelijk om grotere toestandsruimten te bewerken en te verifiëren, en vergroten zo de toepasbaarheid van formele verificatietechnieken in realistische industriële systemen.

We zijn met name geïnteresseerd in het probleem van het reduceren van toestandsruimten modulo gedragsequivalenties. Daarvoor zijn in de literatuur zeer goede sequentiële oplossingen bekend, en ook enkele aanpassingen die geschikt zijn voor een parallel *shared memory* systeem. Wij presenteren in de hoofdstukken 3 en 4 zogenaamde *distributed memory* algoritmen voor de reductie van toestandsruimten modulo sterke en branching bisimulatie equivalentie. Ze maken gebruik van communicatie door middel van *message passing*, en daardoor speelt *latency* een essentiële rol (latency speelt nauwelijks een rol in een shared memory systeem). De drie gepresenteerde algoritmen zijn geïnspireerd op de sequentiële versies van Kanellakis en Smolka [KS83]. We be-

wijzen hun correctheid en laten, aan de hand van experimenten met prototypes van gedistribueerde implementaties, zien dat zowel de looptijd als het geheugengebruik proportioneel toenemen met het aantal gebruikte machines. Dat betekent dat grotere toestandsruimten kunnen worden gegenereerd en gereduceerd door gebruik te maken van goedkope clusters van werkstations. Meestal is de gereduceerde toestandsruimte klein genoeg voor model checking met sequentiële tools.

Verder behandelen we een aanpak van het bekende probleem van het decomponeren van een graaf in zijn *strongly connected components* (Hoofdstuk 5). Omdat het oplossen van een graafprobleem veelal kan worden gereduceerd naar het oplossen van het probleem voor zijn strongly connected components, hebben decompositie algoritmen toepassingen in alle gebieden waar grafen worden gebruikt. We bestuderen het probleem met in gedachten de toepassing in verificatie (LTL/CTL model checking en branching bisimulatie equivalentie reductie). Onze gedistribueerde message passing oplossing is gebaseerd op een reeks van heuristieken die zijn toegesneden op de speciale soort van grafen die toestandsruimten zijn. De prototype implementatie is veelbelovend, alhoewel zij nog niet is geïntegreerd met de tool voor branching bisimulatie reductie, noch daadwerkelijk is gebruikt in model checking.

Deel II gaat over de vraag of formele verificatiemethoden behulpzaam kunnen zijn bij het begrijpen en ontwerpen van gedistribueerde software architecturen. We onderzoeken dit met name voor software architecturen die een zogenaamde shared dataspace gebruiken om hun gedistribueerde componenten te coördineren.

We demonstreren het gebruik van krachtige procesalgebraïsche technieken bij het modelleren en analyseren van een simpele shared dataspace software architectuur met *write* en *blocking non-destructive read* als enige constructies (Hoofdstuk 7). Procesalgebra maakt het mogelijk om de shared dataspace als een apart proces te beschrijven, waardoor een natuurlijke scheiding van de berekenings- en coördinatielagen wordt bereikt. We laten zien dat, door het nondestructieve karakter van de *read* constructie, de shared dataspace volledig kan worden gedistribueerd, terwijl deze vanuit de componenten gezien globaal en uniform blijft. Ondanks het kleine aantal constructies is deze simpele architectuur functioneel zeer expressief: elke systeemspecificatie kan erop worden geïmplementeerd.

Vanwege hun eenvoud en symmetrische behandeling van componenten, zijn shared dataspace coördinatiearchitecturen uitgebreid bestudeerd en zijn er vele varianten geïmplementeerd (Linda, Bonita, WCL, TSpaces, JavaSpaces). De implementaties variëren van een enkele centrale server waaraan alle componenten verzoeken sturen (zoals in JavaSpaces), tot een volledig gedistribueerde implementatie, waarbij elke component zijn eigen lokale kopie van de dataspace heeft (zoals in Splice). Vanuit het oogpunt van verificatie is het interessant om te begrijpen hoe de verschillende implementaties de functionaliteit en performance van zo'n systeem beïnvloeden. We onderzoeken dit in Hoofdstuk 8, waar we een unificerende kijk op de zaak presente-

ren en een ontwerpomgeving opzetten die het mogelijk maakt om shared dataspace systemen met verschillende collecties van constructies en verschillende graden van distributie te modelleren en te verifiëren. Onze ontwerpomgeving bestaat uit een kleine specificatietaal en tools die de verificatie en het maken van prototypes ondersteunen. Verificatie gebeurt via een vertaling naar de meer algemene specificatietaal μ CRL en prototypes worden gemaakt door te vertalen naar gedistribueerde C programma's.

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. Kessler.** *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.** *Scheduler 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. Fábán.** *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. Chkhaev.** *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. Nedeia.** *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. Löb.** *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

F. Alkemade. *Evolutionary Agent-Based Economics.* Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station.* Faculty of Mathematics and Computer Science, TU/e.

2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17