Abstraction Techniques for Parameterized Verification

Muralidhar Talupur

November 2006
CMU-CS-06-169

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Edmund M. Clarke, Chair
Randal E. Bryant
Amir Pnueli, New York University
Jeannette M. Wing

This research was sponsored by the Gigascale Systems Research Center (GSRC), the Semiconductor Research Corporation (SRC), the Office of Naval Research (ONR), the Naval Research Laboratory (NRL), and the Army Research Office (ARO).

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring institutions, the U.S. Government, or any other entity.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
NOV 2006		**00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Abstraction Techniques for Parameterized Verification	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,Computer Science Department,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**278**	

Keywords: Formal methods, model checking, abstract interpretation, abstraction, parameterized systems, cache coherence protocols, mutual exclusion protocols.

To my parents.

Abstract

Model checking is a well known formal verification technique that has been particularly successful for finite state systems such as hardware systems. Model checking essentially works by a thorough exploration of the state space of a given system. As such, model checking is not directly applicable to systems with unbounded state spaces like parameterized systems. The standard approach for applying model checking to unbounded systems is to extract finite state models from them using conservative abstraction techniques. Properties of interest can then be verified over the finite abstract models.

In this thesis, we propose a novel abstraction technique for model checking parameterized systems . Parameterized systems are systems with replicated processes in which the number of processes is a parameter. This kind of replicated structure is quite common in practice. Standard examples of systems with replicated processes are cache coherence protocols, mutual exclusion protocols, and controllers on automobiles. As the exact number of processes is a parameter, the system is essentially an unbounded system. The abstraction technique we propose, called *environment abstraction*, tries to simulate the way a human designer thinks about systems with replicated processes. The abstract models we construct are easy to compute and powerful enough to verify properties of interest without giving any spurious counterexamples. We have applied this abstraction method to several well known parameterized systems like cache coherence protocols and mutual exclusion protocols to demonstrate its efficacy. Importantly, we show how to remove a commonly used, but severely restricting assumption, called the *atomicity* assumption, while verifying parameterized systems.

We also apply insights from environment abstraction in a slightly different setting, namely, that of systems consisting of identical processes placed on a network graph. Adapting principles from environment abstraction, we show how the verification of a system with a large network graph can be decomposed into verification of a collection of systems, each with a small constant sized network graph. As far as we are aware, ours is the first result to show that verification of systems with complex network graphs can be decomposed into smaller problems.

Acknowledgments

I would like to thank my advisor Prof. Edmund Clarke for providing me the opportunity to pursue my research interests. Not only did I benefit academically from him, but I also had a chance to learn valuable life lessons from him. His encouraging attitude towards his students, and his insistence on simplicity have significantly changed my view of things.

I would also like to thank my thesis committee members, Prof. Randal Bryant, Prof. Amir Pnueli, and Prof. Jeannette Wing, for their insightful comments and suggestions regarding my work. My discussions with Prof. Pnueli helped me concretize my ideas. Prof. Wing's thorough comments on the early drafts of my thesis were very useful.

It has been a pleasure to work with my collaborator and co-author Helmut Veith. His suggestions for improving the presentation of my work, including this thesis, have been invaluable.

My friends Himanshu Jain, Shuvendu Lahiri, Flavio Lerda, and Nishant Sinha gave me excellent company and I have gained a lot, academically and otherwise, from them. It was fun sharing my office with Owen Cheng, who also rescued me from technical difficulties more times than I can remember.

Contents

1 Introduction **1**

1.1 Introduction 1

1.1.1 Systems with Replicated Processes 7

1.1.2 Thesis Outline 12

2 Environment Abstraction **15**

2.1 Introduction 15

2.2 A Generic Framework for Environment Abstraction 18

2.2.1 Description of the Abstract System $\mathcal{P}^{\mathcal{A}}$. 22

2.3 Soundness 26

2.3.1 Simulation Modulo Renaming 26

2.3.2 Proof of Soundness 27

2.4 Trade-Off between Expressive Labels and Index Variables 32

2.5 Extending Environment Abstraction 36

2.5.1 Multiple Reference Processes 36

2.5.2 Adding Monitor Processes 38

2.6 Example of Environment Abstraction 41

2.6.1 Abstract Descriptions 42

2.7 Related Work 44

2.7.1 Predicate abstraction 46

2.7.2 Indexed Predicates 48

2.7.3 Three Valued Logical Analysis (TVLA) 51
2.7.4 Counter Abstraction 54
2.8 Conclusion 56

3 Environment Abstraction for Verification of Cache Coherence Protocols 59
3.1 Introduction 59
3.1.1 Cache Coherence Protocols 60
3.2 Discussion of Related Work 64
3.3 System Model for Cache Coherence Protocols 66
3.3.1 State Variables 67
3.3.2 Program Description for the Caches 68
3.3.3 Program Description for the Directory 69
3.3.4 Describing Real-Life Protocols 72
3.4 Environment Abstraction for Cache Coherence Protocols 74
3.4.1 Specifications and Labels 74
3.4.2 Abstract Model 75
3.5 Optimizations to Reduce the Abstract State Space 79
3.5.1 Eliminating Unreachable Environments 80
3.5.2 Redundancy of the Abstract Set Variables 82
3.6 Computing the Abstract Model 85
3.6.1 Cache Transitions 86
3.6.2 Directory Transitions 88
3.7 Experiments 91
3.8 Conclusion 96
3.9 Protocol Descriptions 97

4 Environment Abstraction for Verification of Mutex Protocols 115
4.1 Introduction 115
4.2 System Model for Mutual Exclusion Protocols 119

4.2.1 Local State Variables 119
4.2.2 Transition Constructs 120
4.3 Environment Abstraction for Mutual Exclusion Protocols 123
4.3.1 Specifications and Labels 123
4.3.2 Abstract Descriptions 124
4.4 Extensions for Fairness and Liveness 132
4.4.1 Abstract Fairness Conditions 137
4.4.2 Soundness in the Presence of Fairness Conditions 140
4.4.3 Proof of Soundness 141
4.5 Computing the Abstract Model 145
4.5.1 Case 1: Guarded Transition for Reference Process 146
4.5.2 Case 2: Guarded Transition for Environment Processes 149
4.5.3 Case 3: Update Transition for Reference Process 152
4.5.4 Case 4: Update Transition for Environment Processes 155
4.6 Experimental Results 158
4.7 Protocols and Specifications 159

5 Removing the Atomicity Assumption for Mutex Protocols 165
5.1 Introduction 165
5.2 Modeling Mutual Exclusion Protocols without Atomicity Assumption . . 167
5.3 Atomicity Assumption 171
5.4 Monitors for Handling Non-atomicity 175
5.4.1 Abstracting the Monitor Variables 184
5.5 Computing the Abstract Model 187
5.5.1 Case 1: Guarded Transition for Reference Process 187
5.5.2 Case 2: Guarded Transition for Environment Processes 192
5.5.3 Case 3: Update Transition for Reference Process 195
5.5.4 Case 4: Update Transition for Environment Processes 200
5.6 Experimental Results 203

6 Verification by Network Decomposition **205**
6.1 Introduction 205
6.2 Related Work. 209
6.3 Computation Model 210
6.4 Reductions for Indexed LTL\X Specifications 213
6.4.1 Existential 2-indexed LTL\X Specifications 214
6.4.2 Existential k-indexed LTL\X Specifications 224
6.4.3 Specifications with General Quantifier Prefixes 226
6.4.4 Cut-Offs for Network Topologies 228
6.5 Bounded Reductions for CTL\X are Impossible 229
6.6 Conclusion 231
6.7 Proofs of Lemmas 232
6.8 Connection Topologies for 2-Indices 244

7 Conclusion **251**
7.1 Summary 251
7.2 Extensions 254

Bibliography **257**

List of Figures

1.1 Counter example guided model checking loop. 5

1.2 Tool chain for environment abstraction. 9

3.1 Results for Cache Coherence Protocols 96

4.1 Abstraction Mapping. 117

4.2 Process 7 changes its internal state, but the abstract state is not affected. Thus, there is a self-loop around the abstract state. The abstract infinite path consisting of repeated executions of this loop has no corresponding concrete infinite path. 133

4.3 Running Times 159

4.4 Szymanski's Mutual Exclusion Protocol 160

4.5 Lamport's Bakery Algorithm 163

5.1 Evaluation of a Guard 168

5.2 Evaluation of a Wait condition 169

5.3 Evaluation of an Update 171

5.4 A possible execution trace of the system with three processes. 173

5.5 A more complicated trace of the system. 174

5.6 Execution trace seen from the "outside". 176

5.7 Update procedure for monitor variables pertaining to guarded transitions. 178

5.8 Procedure for updating monitor variables pertaining to guarded transitions. 179

5.9 Procedure for updating monitor variables pertaining to update transitions. 182

5.10 Function ω. 194
5.11 Function Ω_t. 195
5.12 Function Ω_b 196
5.13 Running Times for the bakery protocol. Bakery(A) and Bakery(NA) stand for the bakery protocol with and without the atomicity assumption 203

6.1 Network Graphs A, B, realizing two different characteristic vectors . . . 216
6.2 A system with grid like network graph with 9 nodes. 221
6.3 Connection topologies for the grid-like network graph. 222
6.4 An example of a 5-index connection topology 225
6.5 The Kripke structure K, constructed for three levels. The dashed lines indicate the connections necessary to achieve a strongly connected graph. 242

Chapter 1

Introduction

1.1 Introduction

Modern hardware and software systems are extremely large and intricate. Designing such systems is necessarily an error prone process because of their complexity. A significant percentage of development time is taken up in identifying bugs. Error finding is primarily accomplished through informal/incomplete techniques like testing and simulation. These techniques are incomplete in that they are not guaranteed to find all the bugs in the system. The few errors that escape testing and simulation can still undermine a system, leading to huge financial losses (Intel Floating point error [79]) or even potentially fatal consequences (the Ariane 5 disaster [50]) .

Formal verification techniques like model checking and theorem proving provide an

alternative to incomplete techniques. These techniques explore every possible behavior of a system model and thus find all the bugs in the model. While formal verification methods tend to be expensive (both time wise and labor wise), they are worth the effort put in. The SLAM project at Microsoft [4], which is one of the well known success stories of model checking, managed to exhaustively verify, against a set of properties, the device drivers in a Windows machine. It had been previously observed that most of the crashes of Windows systems occurred due to bugs in the device drivers that escaped detection using testing and simulation. The SLAM project succeeded in eliminating many of the subtle bugs responsible for system crashes using model checking. Thus, the latest versions of the Windows operating systems have benefitted significantly from this project. Model checking has been even more successful in the hardware industry. In fact, most chip design companies, such as Intel and AMD, have dedicated model checking teams as part of the development process. Spurred on by successes like these in the software industry and the hardware industry, there is an increasing adoption of formal verification methods as an integral part of system development.

The central question in formal methods is the following: given a model $\mathcal{M}$ and a property Φ, does the property Φ hold on system $\mathcal{M}$? Formally this is expressed as:

$$M \models \Phi?$$

Model checking, which is the formal verification technique considered in this thesis, works by a thorough exploration of the state space of a given system. The system $\mathcal{M}$ is usually given as a Kripke Structure and the property Φ is expressed in a temporal logic. Kripke structures are specified by tuples of the form (S, I, T, L) where

- S is a finite collection of states,
- I is the set of initial states,
- $T \subseteq S \times S$ is the transition relation,
- L is a labelling function that associates every state in S with a finite set of labels.

Essentially, a Kripke structure is a non-deterministic finite state transition system. Since we are interested in the evolution of a system, we need the notion of time to express properties of interest. These properties are expressed in temporal logic, usually CTL [30] or LTL [65]. Traditional temporal logics are interpreted over Kripke structures.

In recent years, a whole range of powerful model checkers have been developed starting with Ken McMillan's seminal Binary Decision Diagram (BDD) based model checker SMV [57]. BDD based model checkers represent sets of states in a symbolic fashion. The representation of sets of states as BDDs is usually compact, and they can be efficiently manipulated using the standard operations on BDDs [16]. Explicit state model checkers like SPIN [48], on the other hand, represent states explicitly. While explicit representation of states can end up being cumbersome (especially if the reachable state space is very large), the fact that we can examine individual states in detail allows for clever pruning of the search space. For highly parallel systems, techniques like symmetry reduction in conjunction with explicit state model checkers are among the best options, time and space wise, available [27]. In the last few years, the advent of powerful Boolean satisfiability solvers (or SAT solvers) has led to the development of a new class of model checkers. SAT based Bounded Model Checkers [8], which convert the model checking question into

a SAT problem, are extremely fast and very useful in finding bugs that can be reached in a small number of transitions (called *shallow* bugs). Interpolant based model checkers [60] and proof based abstraction [46; 59; 61] too make use of fast SAT solvers, and are currently the fastest for a wide range of problems [1].

All the different model checkers essentially perform a thorough exploration of the state space. As such, model checking cannot be applied to large real world systems directly. Successful application of model checking to complex pieces of code like device drivers depends on the use of *abstraction* methods. An abstraction method extracts a small finite state system, $\mathcal{A}$, called the *abstract* system, from a given large or infinite concrete system $\mathcal{C}$. The abstract system is usually a *conservative* abstraction of the concrete system, which means every behavior seen in $\mathcal{C}$ is also seen in $\mathcal{A}$. It can be shown that if a universal property – a property that talks about *all* paths of a system – holds on the abstract system then it will also hold on the concrete model (see [23] for the results that form the basis for abstraction). Thus, instead of model checking $\mathcal{C}$ directly, we can model check $\mathcal{A}$ and infer the properties satisfied by $\mathcal{C}$.

Creating abstract models involves balancing two conflicting aims:

- **Small Abstract Models.** The abstract model has to be small enough that we can model check it efficiently.

- **Precise Abstract Models.** The smaller the abstract system, the more behaviors it allows. For instance, if the transition relation were **true** (that is, there is a transition

[1] To decide whether $\mathcal{M} \models \Phi$ is computationally very hard and it is unlikely that any one method performs the best on all problems.

from every state to all the other states), the abstract model would be the smallest possible one and would allow every trace. Abstract systems that have too many extraneous traces lead to spurious counter examples – traces that violate the property but do not appear in the concrete system. Thus, while the abstract model should be small, it should also be precise. This latter condition tends to make the abstract system large.

When we model check the abstract model, there are two possible outcomes, as shown in Figure 1.1:

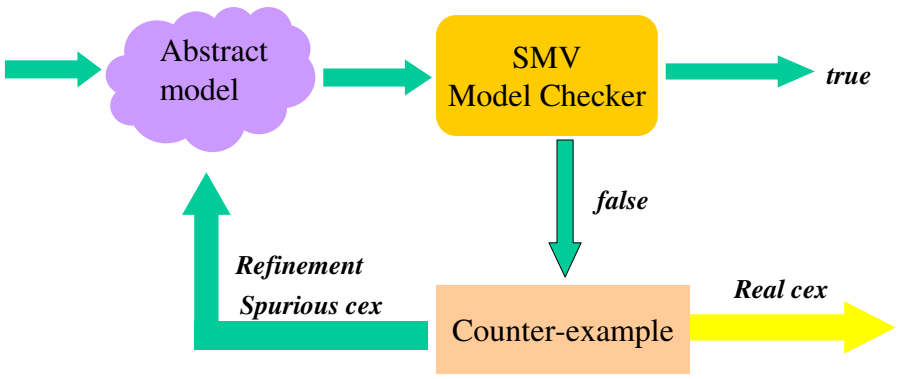


Figure 1.1: Counter example guided model checking loop.

(i) The model checker returns true, that is, the abstract model satisfies the universal

property Φ. In this case, the concrete model also satisfies the property Φ.

(ii) The model checker returns false, that is, the abstract model violates the universal property Φ. In this case, we can check if the counter example trace is a real counter example or a spurious counter example. If the trace is a real counter example, we have a valid counter example to Φ. Otherwise, we cannot say whether the concrete system satisfies Φ or not. In such a scenario, another abstract model, more refined than the previous one, is built and model checked. This process is continued until a definitive result is reached or the system capacity is exceeded.

In practice, it is never sufficient to build just one abstract model. It usually takes several abstract models – each more precise than the previous one – to reach a result. Since the question of whether a (possibly infinite) system satisfies a temporal property Φ is undecidable in general, the abstraction refinement loop is not guaranteed to terminate.

To extract useful abstract models, the abstraction technique must be domain specific. This is because the class of systems is too rich, including sequential software, concurrent protocols, and time triggered systems. The commonalities between these classes are not yet sufficiently understood that we can devise a general abstraction mechanism. All notable successes of model checking (in fact, of formal verification in general) have come from projects which have focused on a specific class of systems, for instance, the class of device drivers in the SLAM project. Following this trend, this thesis proposes a new abstraction technique for concurrent systems that have replicated components such as cache coherence protocols and mutual exclusion protocols. We have applied this abstraction technique to various real world examples to demonstrate its efficacy.

1.1.1 Systems with Replicated Processes

Many real world systems consist of concurrently executing replicated components. Classic examples of such systems are cache coherence protocols which consist of several processes (local caches) executing the exact same cache coherence protocol. That is, the same protocol is replicated at several different processes. As another example, consider controllers in an automobile that are connected via a common bus. The controllers themselves might be different with each controller performing a different function. All controllers use some set of rules, i.e., a protocol to access the bus in a safe and coordinated manner. This bus access protocol must be the same in all the controllers. Thus, if we consider the sub-system consisting of the bus access protocol, we again have an instance of the replicated structure. Replication is a widely occuring feature in real systems. In fact, any scenario in which a collection of processes are contending for a common resource will necessarily involve replication (of protocols/algorithms).

The main classes of replicated systems that researchers in formal verification have considered are cache coherence protocols, mutual exclusion protocols, and time triggered protocols. Such protocols are crucial parts of modern computer systems. Systems with replicated components/processes are usually designed to be correct no matter what the exact number of processes is. Systems with replicated components that have a parameterized number of processes are called *parameterized* systems. In general, systems can be parameterized not just by the number of processes but also by other parameters such as the size of the buffers available per communication channel, the width of the data path, and so on. All parameterized systems are essentially *unbounded systems*.

Applying model checking to such parameterized systems is challenging because they lack fixed state spaces. One way to formally reason about a parameterized system is to use model checking. In this approach, a finite state, conservative abstraction of the system is extracted and model checked. This is the approach followed by Pnueli et al. [66], Lahiri et al. [52; 53], Delzanno et al. [28; 29], Chou et al. [21], German and Sistla [43], Namjoshi [36], and Kahlon et al. [35]. The abstraction created is a conservative (or *sound*) abstraction. This means any universal property (a property that talks about *all* paths) that holds on the abstract model will also hold on the concrete model. The implication in the other direction does not usually hold, that is, if the universal property holds on the parameterized system then it may or may not hold on an abstract model. There are other model checking based techniques like Invisible Invariants [52; 53] and McMillan's Compositional Reasoning [62] which use model checking in a different fashion.

An alternate approach to verifying parameterized systems is to use theorem proving(We classify any technique that requires the users to supply lemmas about the system as theorem proving.). McMillan's Compositional Reasoning, mentioned earlier, is a good example in this class (model checking is used in this approach but the user has to come up with non trivial lemmas). Rushby et al. have used the PVS theorem prover to establish properties of certain clock synchronization algorithms (used in automobiles) and other systems with a parameterized number of replicated processes, see [49; 69].

One of the main contributions of this thesis is an abstraction technique, named *environment abstraction*, developed for reasoning about parameterized systems. Environment abstraction exploits the replicated structure of a parameterized system to make its verification easy. Ideas from this abstraction can be used even if the number of replicated pro-

cesses in a system is fixed. The essential principle is to create an abstraction that matches human reasoning closely. When a human designer creates a system with replicated processes, (s)he reasons about its correctness by focussing on the execution of one *reference* process and sees how the other processes might interfere with its execution. Following this idea, our abstraction maintains detailed information on the reference process and abstracts the other processes in relation to the reference process. The resulting abstraction is quite powerful and we believe it is the most natural abstraction (that is, it corresponds most closely to the abstraction humans use in reasoning about parameterized systems).

In the tradition of classical model checking, our approach provides an automated tool chain (shown in Figure 1.2).

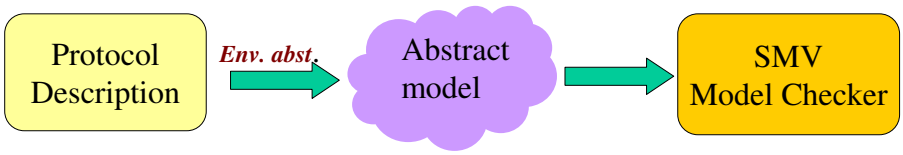


Figure 1.2: Tool chain for environment abstraction.

1. The behavior of the distributed algorithm/protocol is described in a suitable input language, cf. Section 3.3 and Section 4.2.

The user's role ends with inputting the protocol to the verification tool.

1. The environment abstraction tool extracts a finite state model from the protocol description, and puts the model in SMV format.

2. SMV verifies the specified properties.

We have used this abstraction based method to prove properties of well known cache coherence systems, mutual exclusion algorithms, and real time protocols.

Typically, handling liveness properties is much harder (theoretically) than handling safety properties. For instance, the Invisible Invariants method [64] requires significant additional work before it can handle liveness properties and the Indexed Predicates method [52; 53] cannot handle liveness properties at all. Informally, this is because verification of safety properties depends only on the reachable set of states, whereas verification of liveness properties depends also on the order in which the various states are reached. Ranking functions are needed to argue that desirable states are eventually reached. Finding such ranking functions is typically a non-trivial task.

In contrast, extending our method to handle liveness is very simple. Since our abstract model simulates the execution of one single process in precise detail and consequently, liveness properties of a single process are easy to reason about. We only need to rule out spurious loops introduced by the abstraction.

Importantly, other model checking based approaches to parameterized verification make the *atomicity* assumption while handling parameterized protocols. The atomicity assumption in essence states that, in a distributed system with several components, any component can know (or rather read) the state of all the other components instantaneously. This is quite unrealistic and simplifies a distributed protocol significantly. In this thesis, we describe a simple extension to remove the atomicity assumption. Note that the term *atomicity* is used in a different sense from the classical usage in distributed computing literature. In the latter usage, atomicity is used to qualify a *single* read or write operation. An atomic read or write operation is one which happens in an atomic time unit and thus, no other operation can interfere with its execution. Atomicity, as used in this thesis, qualifies a *set* of read/write operations.

The idea of looking at a system from the point of view of a reference process can be carried over into other settings as well. We consider systems with replicated processes which are arranged at the nodes of an underlying network graph. The processes communicate by passing tokens among themselves. If we are interested in checking two-process properties of such a system, we can show that it is enough to consider how the system *looks* from the point of view of pairs of processes. This result lets us decompose the verification problem of a system with a large network graph into verification of a collection of systems with small, constant sized network graphs. This network decomposition result lays the ground for reasoning about systems with network graphs and richer inter-process communication (such as complex leader election protocols).

1.1.2 Thesis Outline

The outline for the rest of the thesis is as follows. In the next chapter we present *environment abstraction* in general terms and derive its mathematical properties. We show that environment abstraction is sound for indexed temporal logic specifications in a very general framework, and discuss the relationship of our method to counter abstraction, canonical abstraction, and predicate abstraction. This chapter lays the foundation for our work on verification of parameterized systems with replicated processes. The same chapter also describes extensions to environment abstraction and considers some of the issues involved in applying this abstraction method to practical systems.

State-of-the-art architectures crucially rely on cache coherence protocols for increased performance. These protocols are extremely intricate and, usually, several processors run these protocols concurrently. Thus, ensuring the correctness of such protocols is a challenging problem and formal verification techniques are indispensable. Since the number of processors executing the cache protocol can vary, cache coherence verification is a classical example of the parameterized verification problem. In Chapter 3, we show how to apply environment abstraction for verifying cache coherence protocols. We first propose a simple programming language that allows us to model cache protocols at an *algorithmic* level. We then describe the precise abstract state space used in abstracting cache protocols. Environment abstraction as presented in Chapter 2 talks only about the general structure of the abstract state space. The precise form of the abstract states depends on the class of systems under consideration. Chapter 3 also deals with the crucial issue of how exactly we compute the abstract model. We have applied this method to verify safety properties of

several cache coherence protocols, including several variants of GERMAN's protocol and a modified version of the FLASH protocol. The language constructs used in describing cache coherence protocols are quite simple so that the essential principle behind the computation of the abstract model is easy to understand. It is for this reason that we consider cache coherence protocols as the first example.

In Chapter 4, we show how environment abstraction can be applied to mutual exclusion protocols, which exhibit complex inter-process communication. As with cache coherence protocols, we first describe a simple programming language that allows us to describe mutual exclusion protocols at an algorithmic level. The precise form of the abstract states is then described, followed by a section on how to compute the abstract model. We demonstrate the power of our approach by verifying Lamport's Bakery algorithm and Szymanski's mutual exclusion protocol. Note that in Chapter 4, we verify mutual exclusion protocols under the atomicity assumption.

In Chapter 5, we show how to verify mutual exclusion protocols without the atomicity assumption. The atomicity assumption, which says that any component can know the state of all the other components instantaneously, significantly reduces the complexity of a protocol. To handle protocols in full generality, without the atomicity assumption, we need to keep track of history information. We introduce monitor processes for this purpose and show how we can apply environment abstraction in presence of these monitor processes.

In Chapter 6, we consider a different system model, namely systems built around network graphs. For example, in routing protocols, the underlying topology of the system plays a crucial role. Similarly, in many wireless applications, the system performance de-

pends on how the different wireless entities are connected. Formal verification research has only now begun to address the problem of verifying these systems with complex network graphs. As a first step towards this larger problem, we consider systems consisting of a collection of identical processes arranged on the nodes of a network graph with very limited communication between the processes. We describe a new method to verify such networks of homogeneous processes that communicate by token passing. Given an arbitrary network graph and an indexed $LTL \setminus X$ property, we show how to decompose the network graph into multiple constant size networks, thereby reducing one model checking call on a large network to several calls on small networks. We thus obtain cut-offs for arbitrary classes of networks, adding to previous work by Emerson and Namjoshi on the ring topology [37]. Our results on $LTL \setminus X$ are complemented by a negative result that precludes the existence of reductions for $CTL \setminus X$ on general networks.

The last chapter concludes this thesis with a summary of contributions and possible extensions to the work presented here. We also discuss some of the outstanding challenges in parameterized verification.

Chapter 2

Environment Abstraction

2.1 Introduction

When a human engineer designs a hardware or software system, the correctness of the system, naturally, is among the main concerns of the designer. Although the reasoning of the designer is usually not available to the verification engineer in terms of assertions or proofs, the reasons for correctness are often reflected in the way a program is written. Knowledge of these implicit design principles can be systematically exploited for the construction of abstract models. For example, it is natural for us to assume that control flow conditions yield important predicates for reasoning about software, and that polygons are good approximations of numeric data that are human generated. Thus, the presence of a human engineer renders the analysis of hardware and software very different from the analysis of systems in physics, chemistry, or biology.

To pinpoint this difference, consider an example frequently discussed in the history of science, namely the Ptolemaic system in which the planet earth is surrounded by the sun. The persistance of Ptolemy's viewpoint over many centuries shows the intuitive reasoning which the human mind applies to complex systems: we tend to imagine systems with the human observer in the center. While a Ptolemaic viewpoint is known to be wrong (or, more precisely, infeasible) in physics, it naturally appears in the systems we *construct*. Consequently, the Ptolemaic viewpoint yields a natural abstraction principle for computer systems.

In this chapter, we explore a Ptolemaic viewpoint of concurrent systems to devise an abstraction method for concurrent systems with replicated processes which we call *environment abstraction*. Our systems are parameterized, i.e., the number of processes is a parameter, and all processes execute the same program. We write $\mathcal{P}(K)$ to denote a system with K processes [1]. We argue that during the construction of such a system, the programmer naturally imagines him/herself to be in the position of one *reference process*, around which the other processes – which constitute *the environment* – evolve. Thus, in many cases, an abstract model that describes the system from the viewpoint of a reference process contains sufficient information to reason about specifications of interest.

The goal of environment abstraction is to put this intuition into a formal framework. In environment abstraction, an abstract state is a description of a concrete system state from the point of view of a single reference process and its environment. The properties of the reference process are computed as if the process were chosen without loss of generality. Thus, verification results about the reference process generalize to all processes in the

[1] We will later also consider a finite number of non-replicated processes in addition.

system.

From a practical perspective, environment abstraction shares many properties with predicate abstraction as used in SLAM [4], BLAST [47], and MAGIC [19]:

- Environment abstraction computes a *finite-state abstract model* on which a standard model checker can verify a property. To verify an indexed temporal property $\forall x.\phi(x)$ on all parameterized models $\mathcal{P}(K)$, $K \geq 1$, the model checker just needs to verify the quantifier-free property $\phi(x)$ on a single abstract model $\mathcal{P}^{\mathcal{A}}$ which interprets the variable x. The model $\mathcal{P}^{\mathcal{A}}$ is obtained by a variation of existential abstraction that quantifies over the parameter K and the index variable x.

- Instead of computing the precise abstract model, environment abstraction *over-approximates* the abstract model. To this end, each statement of the concurrent program is approximated separately using decision procedures. Thus, similar to SLAM, BLAST, MAGIC, the abstract model used in the verification is an over-approximation of $\mathcal{P}^{\mathcal{A}}$.

The aim of this chapter is to describe environment abstraction from first principles. We derive environment abstraction from a few simple logical principles, and show its soundness for a large class of indexed $ACTL^{\star}$ properties. In addition, we put the method in perspective to other abstraction approaches such as Indexed Predicates, and TVLA's Canonical Abstraction.

2.2 A Generic Framework for Environment Abstraction

We consider parameterized concurrent systems $\mathcal{P}(K)$, where the parameter $K > 1$ denotes the number of replicated processes. The processes are distinguished by unique indices in $\{1, \ldots, K\}$ which serve as process ids. Each process executes the same program which has access to its process id. We do not make any specific assumptions about the processes, in particular we do not require them to be finite state processes.

Consider a system $\mathcal{P}(K)$ with a set S_K of states. Each state $s \in S_K$ contains the whole state information for each of the K concurrent processes, i.e., s is a vector

$$\langle s_1, \ldots, s_K \rangle$$

Technically, $\mathcal{P}(K)$ is a Kripke structure (S_K, I_K, R_K, L_K) where I_K is the set of initial states and R_K is the transition relation. We will discuss the labeling L_K for the states in S_K below.

Remark 1. While we consider systems composed solely of replicated processes, systems with a constant number of non-replicated processes, in addition to a set of replicated processes, can also be similarly handled. For such systems, each state is of the form $\langle s_1, \ldots, s_K, t \rangle$ where t is the combined state of all non-replicated components. With this minor change, the treatment presented below can be carried as is to this modified setting as well.

Process Properties. We will describe properties of $\mathcal{P}(K)$ using formulas with one free index variable x which denotes the index of a process. We will call such formulas *process*

properties, as they may or may not hold true for a process in a given state. For a process property $\phi(x)$, we write

$$s \models \phi(c)$$

to express that in state s, process c has property ϕ. We assume that for each state s and process c, we have either $s \models \phi(c)$ or $s \models \neg\phi(c)$.

Example 2.2.1. The following statements are sample process properties:

- ***"Process x has program counter position 5."***

  We will express this fact by the formula $\mathsf{pc}[x] = 5$. We may use this process property in all systems where the processes have a variable pc.

- ***"There exists a process $y \neq x$ where $\mathsf{pc}[y] = 5$."***

  This property is expressed by the quantified formula $\exists y \neq x.\mathsf{pc}[y] = 5$. Note that in this formula, only variable x is free. Intuitively, this property means that a process in the *environment of* x has program counter position 5. We shall therefore write $5 \in \mathsf{env}(x)$ to express this property.

- ***"Process x has program counter position 5, and there exist two other processes t_1 and t_2 in program counter position 1 such that the data variable d satisfies $d[x] < d[t_1] = d[t_2]$."***

  This property, too, can be expressed easily with two quantifiers and one free variable x as shown below

$$\exists t_1, t_2. \quad t_1 \neq t_2 \wedge x \notin \{t_1, t_2\} \wedge \mathsf{pc}[x] = 5 \wedge \mathsf{pc}[t_1] = 1$$

$$\wedge \mathsf{pc}[t_2] = 1 \wedge d[x] < d[t_1] \wedge d[t_1] = d[t_2]$$

Note that the labels discussed in the first two items are highly relevant in our applications and will be discussed below in detail.

Labels and Descriptions. In environment abstraction, we distinguish two sets of process properties that we use for different purposes:

(a) **Labels.** A label is a process property $l(x)$ that we use in a specification. The set of all labels is denoted by L. For example, for $l(x) = (\mathsf{pc}[x] = 10)$, we may write $\forall x.\ \mathbf{AG}\ \neg(\mathsf{pc}[x] = 10)$ to denote that no process reaches program counter 10. For a process c, a c-label is an instantiated formula $l(c)$ where $l(x) \in L$. We write $L(c)$ to denote the set of c-labels.

In the Kripke structure $\mathcal{P}(K)$, a state s has a label $l(c)$, if $s \models l(c)$, i.e.,

$$L_K(s) = \{l(c) : s \models l(c), c \in [1..K]\}.$$

(b) **Descriptions.** A description is a process property $\Delta(x)$ which typically describes not only the process, but also its environment, as in the second and the third items of Example 2.2.1. The set of all descriptions D is our abstract state space.

Intuitively, an abstract state $\Delta(x) \in D$ is an abstraction of a concrete state s if there exists a concrete process c which has property Δ, i.e., if $s \models \Delta(c)$. For example, the description $\mathsf{pc}[x] = 5$ represents all states s which have a process c whose pc variable equals 5. In our applications, the descriptions will usually be relatively large and intricate formulas.

Remark 2. Note that our process properties contain a free index variable x. While the name of the free index variable is immaterial, we have chosen to call it x as it makes the

presentation less cluttered. We also use x in other places, for example, in single index formulas $\forall x.\Phi(x)$. The usage should be clear from the context.

Soundness Requirements for Labels and Descriptions. We will need two requirements on the set D of descriptions and the set L of labels to make them useful as building blocks for the abstract model:

1. **Coverage.** For each system $\mathcal{P}(K)$,$K \geq 2$, each state s in S_K and each process c there is some description $\Delta(x) \in D$ which describes the properties of c, i.e.,

   $$s \models \Delta(c).$$

   The coverage property means that every concrete situation is reflected by some abstract state.

2. **Congruence.** For each description $\Delta(x) \in D$ and each label $l(x) \in L$ it holds that either

   $$\Delta(x) \rightarrow l(x)$$

   or

   $$\Delta(x) \rightarrow \neg l(x).$$

   In other words, the descriptions in D contain enough information about a process to conclude whether a label holds true for this process or not.

   The congruence property enables us to give natural labels to each state of the abstract system: An abstract state $\Delta(x)$ has the label $l(x)$ if $\Delta(x) \rightarrow l(x)$.

2.2.1 Description of the Abstract System $\mathcal{P}^{\mathcal{A}}$.

Given two sets D and L of descriptions and labels that satisfy the *coverage* and *congruence* criteria, the abstract system $\mathcal{P}^{\mathcal{A}}$ is a Kripke structure

$$\langle D, I^A, R^A, L^A \rangle$$

where each $\Delta(x) \in D$ has a label $l(x) \in L$ if $\Delta(x) \to l(x)$, i.e., $L^A(\Delta(x)) = \{l(x) : \Delta(x) \to l(c)\}$. Before we describe I^A and R^A, we can already state the following lemma about preservation of labels.

Lemma 2.2.2. *Suppose that $s \models \Delta(c)$. Then the following are equivalent:*

(i) The concrete state s has label $l(c)$.

(ii) The abstract state $\Delta(x)$ has label $l(x)$.

Proof. Assume that (i) but not (ii). Then by the congruence property, we have $\Delta(x) \to \neg l(x)$. Together with the assumption $s \models \Delta(c)$ of the lemma, we conclude that $s \models \neg l(c)$, which contradicts (i). The converse implication is trivial.

Note that the proof of the lemma requires the congruence property. □

This motivates the following abstraction function:

Definition 2.2.3. Given a concrete state s and a process c, the abstraction of s with reference process c is given by the set

$$\alpha_c(s) = \{\Delta(x) \in D : s \models \Delta(c)\}.$$

Note the following remarks on this definition:

- The coverage requirement guarantees that $\alpha_c(s)$ is always non-empty.
- If the $\Delta(x)$ are mutually exclusive, then $\alpha_c(s)$ always contains exactly one description $\Delta(x)$.
- Two processes c and d of the same state s will in general give rise to different abstractions, i.e., $\alpha_c(s) = \alpha_d(s)$ is in general not true.

Remark 3. In our application of environment abstraction to various distributed protocols, it is usually the case that the abstract descriptions $\Delta(x)$'s are mutually exclusive. Thus, given a state s and reference process c, $\alpha_c(s)$ will contain exactly one abstract description $\Delta(x)$. In such cases, we simply write $\alpha_c(s) = \Delta(x)$.

Now we define the *transition relation* of the abstract system by a variation of existential abstraction: R^A contains a transition between $\Delta_1(x)$ and $\Delta_2(x)$ if there exists a concrete system $\mathcal{P}(K)$, two states s_1, s_2 and a process r such that

1. $\Delta_1(x) \in \alpha_r(s_1)$,
2. $\Delta_2(x) \in \alpha_r(s_2)$, and
3. there is a transition from s_1 to s_2 in $\mathcal{P}(K)$, i.e., $(s_1, s_2) \in R_K$.

We note three important properties of this definition:

- We existentially quantify over K, s_1, s_2, and r. This is different from standard existential abstraction where we only quantify over s_1 and s_2. For fixed K and r, our definition is equivalent to existential abstraction.

- Both abstractions Δ_1 and Δ_2 use the same process r. Thus, the point of view of the abstraction is not changed in the transition.

- The process that actually makes the transition can be any process in $\mathcal{P}(K)$, it does not have to be r.

Finally, the set I^A of abstract initial states is the union of the abstractions of concrete states, i.e., $\Delta(x) \in I^A$ if there exists a system $\mathcal{P}(K)$ with state $s \in I_K$ and process r such that $\Delta(x) \in \alpha_r(s)$.

To summarize, $\mathcal{P}^{\mathcal{A}}$ is a Kripke structure (D, I^A, R^A, L^A) such that the set of abstract descriptions D satisfies the congruence and closure conditions with respect to the set of labels L and the transition relation R^A is defined in an existential fashion.

Remark 4. It will be convenient later on to represent the abstract descriptions as tuples. For example, if the abstract descriptions were all of the form

$$\pm P_1(x) \wedge \ldots \pm P_T(x), T > 1$$

where $P_1(x), \ldots, P_T(x)$ are some process properties and $\pm P_i(x)$ indicates that property $P_i(x)$ can appear negated or unnegated, then we can represent an abstract description $\Delta(x)$ as a tuple

$$\langle p_1, \ldots, p_T \rangle$$

where $p_i = 1 \Leftrightarrow \Delta(x) \Rightarrow P_i(x)$. That is, the value of each bit p_i reflects the polarity of the corresponding predicate $P_i(x)$ in $\Delta(x)$.

Single-Indexed Specifications and Soundness of Environment Abstraction. We consider an indexed temporal specification language where specifications have the form

$$\forall x.\phi(x).$$

Here, $\phi(x)$ is an ACTL* formula whose atomic formulas are labels in L. We say that $\mathcal{P}(K) \models \forall x.\phi(x)$ if for all $c \in \{1 \dots K\}$ we have $\mathcal{P}(K) \models \phi(c)$.

Despite the single index, this specification language is powerful, because the labels in L can talk about other processes. For example, using the label $5 \in \mathsf{env}(x)$ from Example 2.2.1 above, we can express mutual exclusion by the formula

$$\forall x.\mathbf{AG}\ (\mathsf{pc}[x] = 5) \rightarrow \neg(5 \in \mathsf{env}(x))$$

as well as many other properties. For a more thorough discussion of the expressive power of this language, see Section 2.4. In Section 2.5.1 we will also consider abstractions with multiple reference processes for specifications with multiple indices.

For environment abstractions with L and D that satisfy coverage and congruence, we have the following general soundness theorem.

Theorem 2.2.4 (Soundness of Environment Abstraction). *Let $\mathcal{P}(K)$ be a parameterized system and $\mathcal{P}^{\mathcal{A}}$ be its abstraction as described above. Then for single indexed $ACTL^\star$ specifications $\forall x.\phi(x)$ the following holds:*

$$\mathcal{P}^{\mathcal{A}} \models \phi(x) \quad \text{implies} \quad \forall K.\mathcal{P}(K) \models \forall x.\phi(x).$$

2.3 Soundness

We will now give a proof of correctness for Theorem 2.2.4. Before we give the proof of the soundness theorem we introduce some notation to simplify later proofs.

2.3.1 Simulation Modulo Renaming

Given a fixed process c, we write $\mathcal{P}_c(K)$ to denote the Kripke structure obtained from $\mathcal{P}(K)$ where L_K is restricted to *only those labels which refer to process* c. Thus, $\mathcal{P}_c(K)$ is labeled only with c-labels.

Fact 1. Let c be a process in $\mathcal{P}(K)$ and $\phi(x)$ be a temporal formula over atomic labels from L. Then

$$\mathcal{P}(K) \models \phi(c) \quad \text{if and only if} \quad \mathcal{P}_c(K) \models \phi(c).$$

This follows directly from the fact that the truth of $\phi(c)$ depends only on c-labels.

Our soundness proofs will require a simple variation of the classical abstraction theorem [23]. Recall that the classical abstraction theorem for ACTL* says that for ACTL* specifications ϕ and two Kripke structures K_1 and K_2 it holds that $K_1 \succeq K_2$ and $K_1 \models \phi$ together imply $K_2 \models \phi$. That is, if K_1 simulates K_2 then any $ACTL^*$ property satisfied by K_1 is also satisfied by K_2.

Definition 2.3.1 (Simulation Modulo Renaming). Let K be a Kripke structure, and c and d be processes. Then $K[c/d]$ denotes the Kripke structure obtained from K by replacing

each label of the form $l(c)$ by $l(d)$. Simulation modulo renaming $\preceq_{c/d}$ is defined as follows:

$$K_1 \preceq_{c/d} K_2 \qquad \text{iff} \qquad K_1[c/d] \preceq K_2.$$

Then $\preceq_{c/d}$ gives rise to a simple variation of the classical abstraction theorem:

Fact 2 (Abstraction Theorem Modulo Renaming). Let $\phi(x)$ be a temporal formula over atomic labels from L, and let K_1, K_2 be Kripke structures which are labelled only with c_1-labels and c_2-labels respectively.

If $K_2 \preceq_{c_2/c_1} K_1$ and $K_1 \models \phi(c_1)$, then $K_2 \models \phi(c_2)$.

Proof. First note that $K_2 \models \phi(c_2)$ is equivalent to $K_2[c_2/c_1] \models \phi(c_1)$: if the labels in the Kripke structure and the atomic propositions in the specification are consistently renamed, then the satisfaction relation does not change.

Thus, given that $K_2 \preceq_{c_2/c_1} K_1$ and $K_1 \models \phi(c_1)$, it is enough to show that $K_2[c_2/c_1] \models \phi(c_1)$. By the definition of $\preceq_{c/d}$, $K_2 \preceq_{c_2/c_1} K_1$ iff $K_2[c_2/c_1] \preceq K_1$ and by the classical abstraction theorem [23], $K_1 \models \phi(c_1)$ implies $K_2[c_2/c_1] \models \phi(c_1)$. This proves the abstraction theorem. □

2.3.2 Proof of Soundness

We will show that environment abstraction preserves indexed properties of the form $\forall x.\phi(x)$ where $\phi(x)$ is an ACTL* formula over atomic labels from L.

Step 1: Reduction to Simulation. Formally, we have to show that

$$\mathcal{P}^{\mathcal{A}} \models \phi(x) \quad \text{implies} \quad \forall K.\mathcal{P}(K) \models \forall x.\phi(x).$$

By the semantics of our specification language, this is equivalent to saying that for all $K > 1$,

$$\mathcal{P}^{\mathcal{A}} \models \phi(x) \quad \text{implies} \quad \forall c \in [1..K].\mathcal{P}(K) \models \phi(c).$$

Thus, we need to show that for all $K > 1$ and all processes $c \in [1..K]$

$$\mathcal{P}^{\mathcal{A}} \models \phi(x) \quad \text{implies} \quad \mathcal{P}(K) \models \phi(c).$$

Recall that $\mathcal{P}_c(K)$ is the Kripke structure obtained from $\mathcal{P}(K)$ that contains only c-labels. By Fact 1 we know that $\mathcal{P}(K) \models \phi(c)$ iff $\mathcal{P}_c(K) \models \phi(c)$. Thus, we need to show that for all $K > 1$ and for all $c \in [1..K]$

$$\mathcal{P}^{\mathcal{A}} \models \phi(x) \quad \text{implies} \quad \mathcal{P}_c(K) \models \phi(c).$$

Now, by the abstraction theorem modulo renaming (Fact 2), it suffices to show that

$$\boxed{\mathcal{P}_c(K) \quad \preceq_{c/x} \quad \mathcal{P}^{\mathcal{A}} \qquad \text{for all } K \text{ and } c \in [1..K]}$$

where $\preceq_{c/x}$ denotes simulation modulo renaming as defined previously.

We will now prove these simulations.

Step 2: Proof of Simulation. We will now show how to establish the simulation relation $\mathcal{P}_c(K) \preceq_{c/x} \mathcal{P}^{\mathcal{A}}$ between $\mathcal{P}_c(K)$ and $\mathcal{P}^{\mathcal{A}}$ for all $K > 1$ and $c \in [1..K]$. To this end, for each K and c, we will construct an intermediate abstract system $\mathcal{P}^{\mathcal{A}}_{c,K}$ such that

$\mathcal{P}_c(K) \preceq_{c/x} \mathcal{P}^{\mathcal{A}}_{c,K}$ *(Simulation 1)*

and

$\mathcal{P}^{\mathcal{A}}_{c,K} \preceq \mathcal{P}^{\mathcal{A}}$. *(Simulation 2)*

The required simulation then follows by transitivity of simulation. Intuitively, the intermediate model $\mathcal{P}^{\mathcal{A}}_{c,K}$ is the abstraction of the K-process non-parameterized system $\mathcal{P}(K)$ where the reference process c is fixed. Thus, $\mathcal{P}^{\mathcal{A}}_{c,K}$ is obtained from $P_c(K)$ by "classical" predicate abstraction. Note however that $\mathcal{P}^{\mathcal{A}}_{c,K}$ is a mathematical construction to show soundness of the abstract model $\mathcal{P}^{\mathcal{A}}$. In the implementation, we directly construct an approximation of $\mathcal{P}^{\mathcal{A}}$.

Construction of $\mathcal{P}^{\mathcal{A}}_{c,K}$. The abstract model

$$\mathcal{P}^{\mathcal{A}}_{c,K} = \langle D, I^A_{c,K}, R^A_{c,K}, L^A \rangle$$

is defined analogously to $\mathcal{P}^{\mathcal{A}}$ for the special case where K and c are fixed. Thus, $\mathcal{P}^{\mathcal{A}}_{c,K}$ is the abstract model of the concrete system $\mathcal{P}_c(K)$ with a fixed number K of processes and reference process c. More precisely, $\mathcal{P}^{\mathcal{A}}_{c,K}$ is defined as follows:

(a) The state space D is the same as in $\mathcal{P}^{\mathcal{A}}$.

(b) The set of initial states $I^A_{c,K}$ is the subset of the initial states I^A of $\mathcal{P}^{\mathcal{A}}$ for the special case of K and c. Thus, $I^A_{c,K}$ is given by those abstract states $\Delta(x)$ for which there exists a state s in $\mathcal{P}_c(K)$ such that $\Delta(x) \in \alpha_c(s)$.

(c) The transition relation $R^A_{c,K}$ is the subset of the transition relation R^A of $\mathcal{P}^{\mathcal{A}}$ for the special case of K and c. Thus, there is a transition from $\Delta_1(x)$ to $\Delta_2(x)$ in $R^A_{c,K}$ if and only if there are two states s_1, s_2 in $\mathcal{P}_c(K)$ such that $\Delta_1(x) \in \alpha_c(s_1)$, $\Delta_2(x) \in \alpha_c(s_2)$, and $(s_1, s_2) \in R$.

(d) The labeling function L^A is the same as in $\mathcal{P}^{\mathcal{A}}$.

Proof of Simulation 1. We need to show that $\mathcal{P}_c(K) \preceq_{c/x} \mathcal{P}^{\mathcal{A}}_{c,K}$, which by definition of $\preceq_{c/x}$ is equivalent to

$$\mathcal{P}_c(K)[c/x] \preceq \mathcal{P}^{\mathcal{A}}_{c,K}.$$

Consider the structure $\mathcal{P}_c(K)[c/x]$. This is just the K-process system $\mathcal{P}(K)$ restricted to the labels for process c, but because of the renaming the labels have the form $l(x)$ instead of $l(c)$. Thus, the labels of $\mathcal{P}_c(K)[c/x]$ are taken from the set L. Note that the labels of the abstract system $\mathcal{P}^{\mathcal{A}}$ are also taken from the set L. The proof idea below is similar to the construction of a simulation relation for existential abstraction.

Consider the relation

$$\mathcal{I} = \{\langle s, \Delta(x)\rangle \ : \ s \models \Delta(c), s \in S_K, \Delta(x) \in D\}.$$

We claim that $\mathcal{I}$ is a simulation relation between $\mathcal{P}_c(K)[c/x]$ and $\mathcal{P}^{\mathcal{A}}_{c,K}$:

1. Lemma 1 together with the renaming of c to x guarantees that for every tuple $\langle s, \Delta(x)\rangle \in \mathcal{I}$, the states s and $\Delta(x)$ have the same labels.

2. Consider a tuple $\langle s, \Delta(x)\rangle \in \mathcal{I}$. Assume that s has a successor state s', i.e., $(s, s') \in R_K$. We need to show that there exists an abstract state $\Delta'(x)$ such that

   (i) $(\Delta(x), \Delta'(x)) \in R^A_{c,K}$, and

   (ii) $\langle s', \Delta'(x)\rangle \in \mathcal{I}$.

To find such a $\Delta'(x)$, consider the abstraction $\alpha_c(s')$ of s', and choose some description $\Gamma(x) \in \alpha_c(s')$. By the coverage condition, $\alpha_c(s')$ is non-empty.

We will show by contradiction that each such $\Gamma(x)$ fulfills the properties (i) and (ii) menitioned above.

Property (i) Assume that $\Gamma(x)$ does not fulfill property (i), i.e., $(\Delta(x), \Gamma(x)) \notin R^A_{c,K}$. Then for all states s_1 and s_2 it must hold that whenever $\Delta(x) \in \alpha_c(s_1)$ and $\Gamma(x) \in \alpha_c(s_2)$ that *there is no transition* between s_1 and s_2. On the other hand, we assumed above that $\Delta(x) \in \alpha_c(s)$, $\Gamma(x) \in \alpha_c(s')$ and there is a transition from s to s'. Hence we have a contradiction.

Property (ii) Assume now that $\Gamma(x)$ does not fulfill property (ii), i.e, $\langle s', \Gamma(x)\rangle \notin \mathcal{I}$. By the definition of $\mathcal{I}$, this means that $s' \not\models \Gamma(c)$, and thus, $\Gamma(c) \notin \alpha_c(s')$. This gives us the required contradiction.

Thus, $\Delta'(x)$ can be chosen from among the descriptions in $\alpha_c(s')$.

3. Finally, the coverage property guarantees that for every initial state $s \in I_K$ there exists some $\Delta(x) \in I^A_{c,K}$ s.t. $\langle s, \Delta(x)\rangle \in \mathcal{I}$.

Proof of Simulation 2. By construction, $I^A_{c,K} \subseteq I^A$ and $R^A_{c,K} \subseteq R^A$. Therefore, $\mathcal{P}^{\mathcal{A}}$ is an over-approximation of $P^A_{c,K}$, and the simulation follows. □

Remark 5. Note that the coverage and congruence requirements for D and L are used in crucial parts of Simulation 1 in the soundness proof. Congruence is used in the proof of Lemma 2.2.2 which gives us property 1 of Simulation 1. Property 2 of Simulation 1

requires coverage to make sure that $\alpha_c(s')$ is non-empty. Property 3 of Simulation 1 also requires coverage to ensure the existence of an abstract initial state.

Remark 6. In the formulation above, we have not assumed that processes in $\mathcal{P}(K)$ execute synchronously or asynchronously. That is, our definitions are not affected by how the system evolves. We only assume that there is a global transition relation for $\mathcal{P}(K)$. Thus, results described above hold whether the processes in $\mathcal{P}(K)$ execute synchrounously or asynchronously. This fact will allow us to later augment $\mathcal{P}(K)$ by adding synchronously executing monitor processes.

2.4 Trade-Off between Expressive Labels and Index Variables

In this section we argue why a well-chosen set of labels L makes it often possible to use a single index variable. The Ptolemaic system view explains why we seldom find more than two indices in practical specifications: when we specify a system, we tend to track properties the reference process has in relation to other processes out there, one at a time. Thus, two-indexed specifications of the form

$$\forall x, y.\ x \neq y \rightarrow \phi(x, y)$$

often suffice to express the specifications of interest. Properties involving three processes at a time are typically complicated, as we need to consider a triangle of processes and their relationships.

In our work on verifying mutual exclusion and cache coherence protocols, we used two kinds of labels (see also Example 2.2.1):

- $\mathsf{pc}[x] = L$ and
- $L \in \mathsf{env}(x)$ which semantically stands for $\exists y \neq x.\mathsf{pc}[y] = L$.

Note that the label $\mathsf{pc}[x] = L$ refers only to process x whereas $L \in \mathsf{env}(x)$ also refers to the environment of x using a *hidden quantification*. This hidden quantification in the environment label gives surprising power to single-indexed specifications.

To see this, consider the standard mutual exclusion property. The classical way to specify mutual exclusion is expressed in a formula such as

$$\forall x, y.x \neq y \rightarrow \mathbf{AG}\ (\mathsf{pc}[x] = 5) \rightarrow (\mathsf{pc}[y] \neq 5).$$

It is easy to see that using the label $5 \in \mathsf{env}(x)$, we can express this specification by the logically equivalent single-indexed formula

$$\forall x.\mathbf{AG}\ (\mathsf{pc}[x] = 5) \rightarrow \neg(\exists y \neq x.\mathsf{pc}[y] = 5).$$

which is in turn equivalent to

$$\forall x.\mathbf{AG}\ (\mathsf{pc}[x] = 5) \rightarrow \neg(5 \in \mathsf{env}(x))$$

The difference between the three formulas is that in the first specification the index quantifiers are in prenex form, while in the second and third formula, the quantifier for y has been distributed inside the formula, and is hidden in the label $5 \in \mathsf{env}(x)$. Again, the

Ptolemaic viewpoint explains why such situations are likely to happen: in many specifications, we consider *our* process over time (i.e., using a temporal logic specification), but only at the individual time points we evaluate its relationship to other processes. Thus, a *time-local quantification* suffices.

The interplay between labels and index variables gives rise to interesting logical considerations that we will discuss briefly now.

Distributive Fragments of CTL and LTL. It is natural to ask when a double-indexed specification can be translated into a single-indexed specification as in the example above. Somewhat surprisingly, this question is related to previous work on temporal logic query languages [73; 74; 75]. A temporal logic query is a formula $\underline{\gamma}$ with one occurrence of a distinguished atomic subformula "?" (called a placeholder). Given $\underline{\gamma}$ and a formula ψ, we write $\underline{\gamma}[\psi]$ to denote the formula obtained by replacing $?$ with ψ. In [73; 74; 75], syntactic characterizations for CTL and LTL queries with the distributivity property

$$\underline{\gamma}[\psi_1 \wedge \psi_2] \leftrightarrow \underline{\gamma}[\psi_1] \wedge \underline{\gamma}[\psi_2].$$

are described. A template grammar for the distributive fragment of LTL is given in the appendix of [74].

The prototypical example of a distributive query is $\mathbf{AG}?$, and we have seen above that for **AG** properties, we can translate double indexed properties into single-indexed properties. As argued above, this translation actually amounts to distributing one universal quantifier inside the temporal formula.

Such a translation is possible for all specifications which are distributive with respect

to one index variable: consider a double-indexed specification $\forall x, y.\ x \neq y \rightarrow \phi(x, y)$ where all occurrences of y in ϕ are located in a subformula $\theta(x, y)$ of ϕ. Then we can write ϕ as a query $\underline{\gamma}[\theta]$. Now suppose that $\underline{\gamma}$ is distributive. On each finite $\mathcal{P}(K)$, the universal quantification reduces to a conjunction, i.e.,

$$\mathcal{P}(K) \models \forall x, y.\ x \neq y \rightarrow \underline{\gamma}[\theta(x, y)] \quad \text{iff} \quad \mathcal{P}(K) \models \forall x. \bigwedge_{1 \leq c \leq K, c \neq x} \underline{\gamma}[\theta(x, c)]$$

which by distributivity of $\underline{\gamma}$ is equivalent to

$$\mathcal{P}(K) \models \forall x.\ \underline{\gamma} \left[\bigwedge_{1 \leq c \leq K, i \neq x} \theta(x, c) \right]$$

and thus to

$$\mathcal{P}(K) \models \forall x.\ \underline{\gamma}\, [\, \forall y. x \neq y \rightarrow \theta(x, c)\,]\, .$$

For a suitable label $l(x) := \forall y.\ x \neq y \rightarrow \theta(x, y)$ this can be written as

$$\mathcal{P}(K) \models \forall x.\ \underline{\gamma}[l(x)].$$

For the important special case where $\theta(x, y)$ has the form $\mathsf{pc}[y] = L$, this is equivalent to

$$\mathcal{P}(K) \models \forall x.\ \underline{\gamma}[L \notin \mathsf{env}(x)].$$

While the characterization of distributive queries gives us a good understanding about the scope of single-indexed specifications, it is clear that not all two-indexed specifications can be rewritten with a single index. Consider, for example, the formula

$$\forall x, y. x \neq y \rightarrow \mathbf{AF}(\mathsf{pc}[x] = 5 \wedge \mathsf{pc}[y] = 5).$$

Here it is evidently not possible to move the quantifier inside. This can also be derived from the characterization in [74]. Consequently, this specification cannot be expressed with a single index.

In Section 2.5.1 we will show how to extend environment abstraction to multiple reference processes. Of course, having more reference processes will, in general, make the abstract model larger, and, thus, harder to analyze. This motivates the following approach to deal with two-indexed specifications σ:

1. Using the grammar characterizations of distributive queries, determine whether σ can be written with a single index.

2. Otherwise, use an abstraction with two reference processes, as described in Section 2.5.1.

2.5 Extending Environment Abstraction

In this section, we will describe a few easy extensions to environment abstraction.

2.5.1 Multiple Reference Processes

In the preceding sections, we focused on a framework for single-indexed specifications of the form $\forall x.\phi(x)$. Extending this framework to two reference processes is simple – essentially, we need to replace the free variable x in the process properties by a pair x, y, and carry this modification through all definitions and proofs. The generalization to more indices is straightforward, and left to the reader.

For the set D of descriptions, we will now use descriptions of the form $\Delta(x, y)$ which capture the state of two reference processes x, y and the environment around them. Thus, we can track the mutual relationship of two processes in greater detail. Similarly, we can extend the set of labels. The set L of labels is partitioned into unary labels L^1 of the form $l(x)$ and binary labels L^2 of the form $l(x, y)$. Note that, in practice, the single-indexed labels will usually suffice. A state s of system $\mathcal{P}(K)$ is labeled with $l(c)$ if and only if $s \models l(c)$. State s is labeled with $l(c, d)$ if and only if $s \models l(c, d)$.

The coverage and congruence requirements are generalized analogously:

1. **Coverage**. For each system $\mathcal{P}(K)$, each state s in $\mathcal{P}(K)$ and any two processes c, d there is some description $\Delta(x, y) \in D$ which describes the properties of c, d, i.e., $s \models \Delta(c, d)$.

2. **Congruence**. For each description $\Delta(x, y) \in D$ and each label $l(x, y) \in L^2$ it holds that either $\Delta(x, y) \rightarrow l(x, y)$ or $\Delta(x, y) \rightarrow \neg l(x, y)$. An analogous condition holds for labels in L^1.

Thus, we obtain a natural definition of the abstraction mapping:

Definition 2.5.1. Given a concrete state s and two processes c and d, the abstraction of s with reference processes c and d is given by the set

$$\alpha_{c,d}(s) = \{\Delta(x, y) \in D : s \models \Delta(c, d)\}.$$

The construction of the abstract model is analogous to the single index case. To indicate the number of reference processes in the abstract model, we write P_2^A for the ab-

stract model with two reference processes. Analogously to the single-index case, we attach labels to each state of $\mathcal{P}_2^A$ such that the abstract state $\Delta(x, y)$ has label $l(x, y)$ iff $\Delta(x, y) \rightarrow l(x, y)$.

Theorem 2.5.2 (Soundness of Double-Index Environment Abstraction). *Let $\mathcal{P}(K)$ be a parameterized system and P_2^A be its abstraction with two reference processes. Then for double indexed $ACTL^\star$ specifications $\forall x \neq y.\phi(x, y)$ the following holds:*

$$P_2^A \models \phi(x, y) \quad \text{implies} \quad \forall K.\mathcal{P}(K) \models \forall x \neq y.\phi(x, y).$$

The environment abstraction principle can be easily extended to incorporate more than two reference processes. As argued above, it is quite unlikely that a practical verification problem will require the use of three reference processes.

2.5.2 Adding Monitor Processes

Often times it is necessary to augment a given parameterized system $\mathcal{P}(K)$ by adding *non-interfering monitor* processes. Monitors are essentially synchronous processes (i.e., they execute at every step of $\mathcal{P}(K)$) that maintain history information regarding the processes in $\mathcal{P}(K)$. Addition of monitors gives more information about the evolution of the system. Thus, taking monitors into account during abstraction can give us better abstract models. A typical case where monitors are needed is for handling liveness properties. As we will see later in Section 4.4, environment abstraction, as described in the earlier

sections, is too coarse to handle liveness properties. This is because the abstraction can introduce spurious loops, which can lead to false negatives. These spurious abstract behaviors can be eliminated by augmenting the system $\mathcal{P}(K)$ with monitors and abstracting the augmented system. While the precise details of monitor processes are considered later in Chapter 4, we will consider here the theoretical basis for adding monitors.

Consider a parameterized system $\mathcal{P}(K)$ and assume that we augment it by adding a collection of identical monitor processes $M(1), \ldots, M(K)$. Each $M(i)$ is exactly the same as the other monitor processes except for its id. Denote the augmented parameterized system by $\mathcal{PM}(K)$. The states of $\mathcal{PM}(K)$ are given by tuples of the form $s_M \doteq \langle \mathcal{L}_1, \ldots, \mathcal{L}_K, \mathcal{M}_1, \ldots, \mathcal{M}_K \rangle$ where $\mathcal{L}_i$ is denotes local state of process $P(i)$ and $\mathcal{M}_i$ denotes the local state of the monitor process $M(i)$.

The results presented in Section 2.2 assume there is only one collection of replicated process. To make the results of Section 2.2 applicable, we can compose each $M(i)$ with the corresponding $P(i)$ to create a hybrid process $PM(i)$. The augmented system $\mathcal{PM}(K) \doteq \langle S_M, I_M, R_M, L_M \rangle$ is a parameterized system with $PM(i)$'s as the constituting processes. The set of labels L_M is usually the same as the set of labels L of $\mathcal{P}(K)$. To apply environment abstraction to $\mathcal{PM}(K)$ we just have to pick the appropriate set of abstract descriptions satisfying the congruence and coverage properties together with labels in L. Let D_M be a collection of abstract descriptions $\Delta_M(x)$ and α_M be the abstraction mapping from S_M to D_M such that D_M satisfies the coverage and congruence conditions with respect to the set of labels L.

Define the augmented abstract model $\mathcal{P}_M^{\mathcal{A}}$ in the usual fashion.

Definition 2.5.3 (Augmented Abstract Model). The abstract model $\mathcal{P}_M^{\mathcal{A}}$ of a parameterized system $\mathcal{PM}(K)$ is defined as the Kripke structure $(D_M, I_M^{\mathcal{A}}, R_M^{\mathcal{A}}, L_M^{\mathcal{A}})$ where

- D_M is the set of all augmented abstract descriptions
- $I_M^{\mathcal{A}}$, the set of initial abstract states, is the set of augmented abstract states $\hat{s}_M$ such that there exists a concrete initial state s_M of a concrete system $\mathcal{PM}(K)$, $K > 1$, and a process $p \in [1..K]$, such that $\alpha_{M_p}(s_M) = \hat{s}_M$.
- $R_M^{\mathcal{A}}$ is defined as follows: There is a transition from abstract state $\hat{s}_{M_1}$ to abstract state $\hat{s}_{M_2}$ if *there exist*
  (i) a concrete system $\mathcal{PM}(K), K > 1$ with a process p
  (ii) a concrete transition from concrete state s_{M_1} to s_{M_2} in $\mathcal{PM}(K)$

  such that $\alpha_{M_p}(s_1) = \hat{s}_{M_1}$ and $\alpha_{M_p}(s_2) = \hat{s}_{M_2}$.
- $\Delta_M(x)$ is labeled with $l(x) \in L$ if and only if $\Delta_M(x) \Rightarrow l(x)$.

Corollary 1. Let $\mathcal{PM}(K)$ be the augmented parameterized system corresponding to the parameterized system $\mathcal{P}(K)$. Let $\mathcal{P}_M^{\mathcal{A}}$ be the augmented abstract model as described above. Then, for any single indexed $ACTL^*$ specification $\forall x.\phi(x)$, where $\phi(x)$ is a formula over labels L, we have

$$\mathcal{P}_M^{\mathcal{A}} \models \phi(x) \Rightarrow \forall K > 1.\mathcal{PM}(K) \models \forall x.\phi(x)$$

Proof. This follows simply from Theorem 2.2.4. Note that we are using the fact that Theorem 2.2.4 holds whether the parameterized system $\mathcal{P}(K)$ executes asynchronously or not. □

Since we have assumed that the monitors are non-interfering, $\mathcal{PM}(K) \models \forall x.\phi(x)$ implies $\mathcal{P}(K) \models \forall x.\phi(x)$. Thus

$$\mathcal{P}_M^{\mathcal{A}} \models \phi(x) \Rightarrow \forall K > 1.\mathcal{PM}(K) \models \forall x.\phi(x) \Rightarrow \forall K > 1.\mathcal{P}(K) \models \forall x.\phi(x)$$

Remark 7. Note that the number of monitor processes is exactly the same as the number of processes. This does not reduce the generality of the results above for the following reasons: if the number of monitor processes is constant (i.e., independent of K) then they can be treated as one single non-replicated process. On the other hand if the number of monitors was a function of K then we can compose a set of monitors and processes (instead of one monitor and one process) to create composite processes. For example, suppose we had only $K/2$ monitors in the system $\mathcal{P}(K)$ with K processes. Then we can compose two processes and one monitor to create a larger composite process $P_M^2(i)$ and the augmented parameterized system is composed of $K/2$ such composite processes. Thus, our results will still be applicable.

2.6 Example of Environment Abstraction

We have thus far described environment abstraction in its most general terms. We have not indicated what descriptions to choose or what labels to use beyond specifying their general forms. In the following, we discuss, using an example, some of these issues which let us apply this abstraction method to practical systems.

2.6.1 Abstract Descriptions

Consider abstract descriptions of the form $\Delta(x)$ consisting of a single reference process. A description $\Delta(x)$ can provide very detailed information on process x and its environment. In our work on verifying mutual exclusion protocols (see Chapter 4), we found it useful to have descriptions $\Delta(x)$ of the following form:

$$\Delta(x) \doteq \mathsf{pc}[x] = L \wedge \exists y \neq x.E_1(x,y) \wedge \ldots \exists y \neq x.E_T(x,y), T \geq 1$$

Informally, the condition $\mathsf{pc}[x] = L$ describes the *control location* of the reference process x. Each of the conditions $\exists y \neq x.E_i(x,y)$ tells that there exists a process y in the *environment* of x satisfying a certain predicate $E_i(x,y)$ over the state variables of processes x, y. Each $E_i(x,y)$ itself is of the form

$$E_i(x,y) \doteq \pm R_1(x,y) \wedge \ldots \wedge \pm R_M(x,y) \wedge \mathsf{pc}[y] = L, M \geq 1$$

where each $R_i(x,y)$ is an atomic predicate relating the data variables of two processes x, y. The condition $\mathsf{pc}[y] = L$ says that process y is in control state L. That is, we take every possible cube over the atomic predicates $R_1(x,y), \ldots, R_M(x,y)$ (that is,every expression of the form $\pm R_1(x,y) \wedge \ldots \wedge \pm R_M(x,y)$ and conjoin them with every possible predicate of the form $\mathsf{pc}[y] = L$ to obtain the full set of $E_i(x,y)$ predicates. It is easy to see that every process y in the environment of a process x will satisfy one of the $E_i(x,y)$ predicates. It is also easy to see the set of descriptions as constructed above has the required coverage property: for all concrete systems $\mathcal{P}(K)$, each concrete s of $\mathcal{P}(K)$ and process $c \in [1..K]$, $s \models \Delta(c)$ for some description $\Delta(x)$.

The choice of the set of descriptions was dictated by the properties that we were inter-

ested in verifying, namely, two index safety properties of the form

$$\forall x, y.x \neq y \wedge (\mathsf{pc}[x] = \mathbf{crit} \Rightarrow \neg(\mathsf{pc}[y] = \mathbf{crit})$$

As discussed earlier, this property can be equivalently written as

$$\forall x.(\mathsf{pc}[x] = \mathbf{crit} \Rightarrow \neg(\exists y \neq x.\mathsf{pc}[y] = \mathbf{crit}))$$

Thus, the two indexed property is essentially composed of two kinds of labels

- $\mathsf{pc}[x] = L$, and
- $\exists y \neq x.\mathsf{pc}[y] = L$.

Observe first that only x occurs free in both types of labels. Further, each description $\Delta(x)$ either implies $\mathsf{pc}[x] = L$ or its negation. Similarly, each $\Delta(x)$ either implies $\exists y \neq x.\mathsf{pc}[y] = L$ or its negation. Thus, we also have the required congruence property. Thus, the set of descriptions we chose have both congruence and coverage properties required by our abstraction framework.

As an aside, if we let $\exists^k$ stand for the generalization of the usual existential quantifier $\exists$ meaning *there exist at least k different* elements, then our descriptions can be made even stronger. Instead of the descriptions above we can use

$$\Delta(x) \doteq \mathsf{pc}[x] = L \wedge \exists^k y \neq x.E_1(x, y) \wedge \ldots \exists^k y \neq x.E_T(x, y).$$

Instead of just telling us whether there is a process satisfying $E_i(x, y)$ these descriptions also tell us whether there are atleast k such processes or not. Note that this is quite close in spirit to *counting abstraction* which also *counts* processes satisfying certain conditions (though there is no notion of a reference process in counting abstraction).

2.7 Related Work

Verification of parameterized systems is well known to be undecidable, see [2; 76]. Nonetheless, many interesting approaches to this problem have been developed over the years, including the use of symbolic automata-based techniques [1; 10; 12; 51], invariant based techniques [3; 64], predicate abstraction [52], and symmetry [24; 31; 38; 39; 40]. Some of the earliest work on verifying parameterized systems includes the works by Browne et al [14; 15], German and Sistla [43], and Emerson and Sistla [38]. Papers that handle systems similar to the parameterized systems considered in this thesis are [3; 6; 7; 41; 42; 52; 53; 64; 66]. The paper [66] by Pnueli et al., which introduces the term *counter abstraction*, inspired our work.

Environment abstraction fits the *Abstract Interpretation* framework of Cousot and Cousot [26]. In the Abstract Interpretation framework one studies the effect of a program in an abstract domain instead of the concrete domain that the program is supposed to handle. The abstract domain is designed to be sound so that a property that holds in the abstract domain will also hold in the concrete domain. While this provides a general methodology, it provides no guidance on what abstract domain to choose. The choice of the abstract domain to consider is in fact the toughest question facing any Abstract Interpretation based method.

In the context of verifying software and hardware systems, several different alternatives have been proposed to construct abstract domains. Any such method must address two conflicting issues:

Generality: The abstract domains must be as widely applicable as possible. It defeats the propose of automated and efficient program analysis if the user has to figure out the abstract domain for each and every program separately. Thus, it is required that methods for constructing the abstract domains should not be too specific.

Usability: On the other hand, widely applicable but trivial abstract domains can be constructed quite easily. Such abstract domains are useless in proving interesting properties of a program under consideration. It is typically the case that the more widely a method (for constructing abstract domains) is applicable, the less powerful it is.

In this thesis, we are essentially proposing a new approach for constructing abstract domains. This approach is applicable to any system that has replicated components. For such systems, the abstract domain we consider has detailed information on one *reference* component and the rest of the components are considered in less detail and in relation to the reference component. It is our claim that this is the way a human designer thinks (when designing systems with replicated components), and, hence the abstract domains constructed according to this pattern will be powerful. It is to be noted that we have not specified all the details of the abstract domain as they necessarily depends on the specific class of programs under consideration. But following this general structure, we hope that filling in the details will be easy.

In the following sections we discuss some of the well known abstraction methods and how they relate to our work.

2.7.1 Predicate abstraction

This method, proposed by Graf and Saidi [70; 72] has, over the years, become one of the most widely used abstraction mechanisms for handling systems with large or unbounded state spaces. The basic idea of this approach is to consider the effect of the program on a set of (carefully chosen) predicates. The abstract domain consists of a set of predicates over the program variables. Assume that the set of predicates is $\mathcal{P} \doteq \{P_1, \ldots, P_n\}$. The abstract domain consists of all possible valuations $\langle b_1, \ldots, b_n \rangle$ of the predicates $P_1, \ldots, P_n$. Denote the set of concrete states by S and the set of abstract states by $\hat{S}$. We can then define the standard abstraction mapping α from S to abstract states $\hat{S}$ as follows. For any concrete state $s \in S$ and $\langle b_1, \ldots, b_n \rangle \in \hat{S}$

$$\alpha(s) \doteq \langle b_1, \ldots, b_n \rangle \text{ such that each } b_i = 1 \text{ iff } s \models P_i.$$

The corresponding concretization mapping γ from $\hat{S}$ to 2^S is then defined as

$$\gamma(\langle b_1, \ldots, b_n \rangle) \doteq \{s | \alpha(s) = \langle b_1, \ldots, b_n \rangle\}$$

Once the abstraction mapping is defined, the abstract model is described using the well-known existential definition: given two abstract states $\hat{s}_1, \hat{s}_2$ there is an abstract transition from $\hat{s}_1$ to $\hat{s}_2$ if **there exist** two concrete states s_1, s_2 **such that**

- $\alpha(s_1) = \hat{s}_1$,
- $\alpha(s_2) = \hat{s}_2$, and
- there is a concrete transition from s_1 to s_2.

It can be shown that the abstract model so defined is a conservative abstraction of the concrete system [70].

Predicate abstraction is a very general method. The main problem in applying predicate abstraction is in deciding what set of predicates to use. This is an active area of research and several heuristics are used to discover relevant predicates to use (for example the CEGAR loop [22]). In contrast, our method does provide a framework for constructing predicates.

There are some crucial differences between standard predicate abstraction and our method. Given a fixed set of predicates, each concrete state can map only to one abstract state in usual predicate abstraction. On the other hand, in our abstraction method, a concrete state can map to multiple abstract states depending on which process is chosen as the reference process.

Further, in standard predicate abstraction, the predicates typically involve the variables of the same process/program. In our approach, the predicates span multiple processes and relate the states of different components in the system. TVLA [71; 80] was the first work to identify the importance of such predicates and it has been successfully used to verify various multi-threaded systems and heap properties. We believe the use of predicates relating different processes/components within a system is a natural and powerful extension of standard predicate abstraction. Such predicates are required if one wants to verify multi process systems or reason about heap properties.

2.7.2 Indexed Predicates

The Indexed Predicates approach was proposed by Lahiri and Bryant [52; 53] to handle unbounded systems such as those with replicated processes or unbounded data structures. The invariants of such systems are usually quantified over the parameters and the indices of the various components of the system. Typically, the scope of the quantifiers contains complex formulas (which are themselves composed of smaller predicates containing free index variables). If one were to use standard predicate abstraction, discovering such invariants would involve predicates which are almost as complex as the invariants themselves. To get around this problem, the Indexed Predicates approach uses simple predicates which can contain free-index variables and tries to build complex quantified invariants from these *indexed predicates*. The invariants discovered using this method contain only universal quantifiers.

The Indexed Predicates starts with a set of predicates $\mathcal{P} \doteq \{P_1, \ldots, P_n\}$ which can contain free index variables from a set $\mathcal{X}$. As with standard predicate abstraction, the abstract state space $\hat{S}$ is just the set of all possible valuations $\langle b_1, \ldots, b_n \rangle$ of the atomic predicates in $\mathcal{P}$. The abstraction mapping function though is quite different. A concrete state s maps to an abstract state $\hat{s} \doteq \langle b_1, \ldots, b_n \rangle$ if for some valuation of the index variables in $\mathcal{X}$ the value of each predicate P_i in $\mathcal{P}$ matches the corresponding b_i in $\hat{s}$. More formally, let $v(\mathcal{X})$ denote some valuation of the index variables in $\mathcal{X}$. Then

$$\alpha(s) \doteq \{\hat{s} \in \hat{S} | \exists v(\mathcal{X}).(s \models_{v(\mathcal{X})} P_i \Leftrightarrow b_i)\}$$

where $s \models_{v(\mathcal{X})} P_i$ means state s satisfies predicate P_i with all the free index variables occuring in P_i fixed according to the valuation $v(\mathcal{X})$.

Since there are multiple possible valuations for variables in $\mathcal{X}$, a single concrete state can map to several different abstract states. Note that, in our method a single concrete state can also map to several different abstract states depending on which process is chosen as the reference process (the reference process in our method can be modeled as a free index variable). Thus, the abstraction mapping used in our method and in Indexed Predicates are essentially the same. But unlike our method, Indexed Predicates method defines a concretization function for a set of abstract states, not for a single abstract state. The concretization of a set of abstract states $\hat{C}$ is the set C of concrete states such that, for all valuation of the free index variables, every state $s \in S$ maps to some state $\hat{s} \in \hat{S}$. More formally, for a set $\hat{C}$ of abstract states

$$\gamma(\hat{C}) \doteq \{C \subseteq S | \forall s \in C . \alpha(s) \subseteq \hat{C}\}$$

The abstract reachability is carried out by defining a reachability function that operates on sets of abstract states instead of single abstract states. Denote the concrete transition relation by ρ. Then the abstract reachability function $\hat{\rho}$ is defined as:

$$\hat{\rho}(\hat{S}) \doteq \alpha(\rho(\gamma(\hat{S})).$$

Let $R, \hat{R}$ be the set of concrete and abstract reachable states. Then it can be shown that [53]

$$\alpha(R) \subseteq \hat{R}.$$

Thus, an over-approximation of the concrete reachable states can be found by doing a reachability analysis on the abstract model.

A crucial difference between our method and Indexed Predicates method is that we can define a concretization function that operates on individual abstract states instead of sets of abstract states. In our framework, the concretization function γ is defined as follows. For an abstract state $\hat{s}$

$$\gamma(\hat{s}) \doteq \{s \in S | \exists c. \alpha_c(s) = \hat{s}\}$$

where c is the *reference process*.

Because we can talk of concrete states corresponding to each abstract state, we can also define an abstract transition relation, not just an abstract reachability function operating on sets of states (as in Indexed Predicates. [2]) As a consequence, Indexed predicates method is not suited for handling liveness properties, which requires an abstract transition relation. Indexed Predicates method can verify only safety properties. In contrast, our approach can handle both safety and liveness properties.

In Indexed Predicates, the computation of the abstract reachable states is done symbolically by reducing each step of the reachability analysis to finding solutions of a quantified CLU formula (CLU logic is a subset of first order logic with uninterpreted functions [18]). Quantified CLU formulas are then solved by posing them as Boolean SAT problems [17]. Observe that this method of abstract reachability does not really exploit our knowledge of the concrete transition statements. On the flip side, the systems that Indexed Predicates can handle is limited in theory only by the availability of solvers for first order logic formulas.

In contrast, in our approach, we consider each statement of the protocol and compute an over-approximation of all the abstract transitions this can lead to. In doing this, we

[2] As an aside, we believe the Indexed Predicates method can also be generalized to have an abstract transition relation.

are able to exploit our knowledge of the transition statements whereas Indexed Predicates cannot. This means that, for each type of transition statement, we have to write by hand an over-approximation for it. We believe this is a novel feature of our approach: the user has to provide an over-approximation for each *type* of transition statement in the system. Since there are limited number of different types of transition statements (in our work on protocol verification we had around 6 types), this is a fairly easy task.

Another difference between the two approaches is that Indexed Predicates is a general framework much in the spirit of standard predicate abstraction and TVLA. It provides no guidance on what predicates to use. This problem is compounded by the fact that, unlike predicate abstraction, there is no possibility of applying the Counterexample Guided Abstraction loop to extract useful predicates. This is because there is no abstract transition relation in Indexed Predicates and consequently, no notion of an abstract trace. Our approach, in contrast, does provide a guideline for what predicates to use. Moreover, as we have an abstract transition relation, automatic predicate discovery guided by counterexample traces is also possible.

2.7.3 Three Valued Logical Analysis (TVLA)

The TVLA method proposed by Reps et al. [71; 80] is an abstract interpretation based approach for verifying safety properties of multi-threaded systems, and for doing shape analysis. This is a widely applicable method that uses the universe of first order logical structures as the abstract domain. To make verification of unbounded systems possible, they use the notion of *summarization*, which is similar to the idea of *counting abstraction*.

The essential idea is to represent the state of a system as a first order structure (called a *configuration*) consisting of objects and predicates over these objects. The objects in the first order structure can be used to represent threads and heap allocated data structures. The predicates can be used to represent relationships between the objects. For example, the fact that a pointer p points to thread t can be represented by using a predicate *pointsTo(p,t)*. Papers on TVLA were the first to observe that predicates spanning or relating multiple components of a system are essential if we want to reason about multi-threaded systems and heap properties.

Once a set of relevant predicates have been picked, the mapping from concrete states to abstract states is straight-forward. There is a one to one mapping from the threads and other components of the concrete system to the objects in the abstract domain. Further, the valuations of the different predicates are known from the concrete state being considered. If the number of threads and other components in the concrete system are bounded then the number of objects necessary in the abstract domain is also finite.

To handle the case where the concrete system can have an unbounded number of threads and other components, TVLA uses the notion of *summarization* which is essentially a form of counting abstraction. Suppose components $c_1, \ldots, c_n$ all satisfy the same set of unary predicates [3]. Then instead of mapping them to different objects $o_1, \ldots, o_n$ in the abstract domain, they are mapped to one abstract object $\hat{o}$. Thus, an unbounded number of concrete components can be summarized using a single abstract object $\hat{o}$. Observe that *summarization* of $o_1, \ldots, o_n$ into one abstract object $\hat{o}$ introduces uncertainty in the

[3]TVLA uses binary predicates to specify relationship between different components and unary predicates to specify properties of a particular component.

properties satisfied by $\hat{o}$. For instance, suppose object o_1 satisfies a certain binary predicate $Bin(o_1, t)$ for some fixed arguement t, but the rest of the objects do not. It is not clear in this case whether $\hat{o}$ should satisfy $Bin(\hat{o}, t)$ or not. To deal with situations like these, TVLA uses three-valued logic (hence the name) instead of the standard two-valued logic. Thus, a predicate P can take three values $0, 1/2$, and 1, where $0, 1$ have the usual meaning and $1/2$ denotes that P can have either value.

While *summarization* of similar objects is a powerful feature that lets TVLA deal with unbounded systems, it is sometimes necessary to track one object,say o_1, separately from other objects $o_2, \ldots, o_n$ even though they may have the same properties. For this sake, special unary predicates can be used to select some particular object as a special object and thus track its execution in detail. Such unary predicates used to distinguish individual objects are called *instrumentation* predicates.

It might seem that instrumentation predicates can be used to simulate our notion of a *reference process*, but that is not the case. The only thing that distinguishes a reference process from other processes in the system is its id. Thus, if we use instrumentation predicates to simulate the notion of a reference process, the predicates will have to refer to the process ids. This means that once a reference process is chosen by instrumentation predicates it cannot change. But, in our abstraction, the identity of the process that serves as the reference process may change from transition to transition. Thus, the notion of a reference process cannot be simulated using instrumentation predicates.

To explore the state space of a given system, TVLA starts with the initial set of abstract configurations each of which corresponds to some concrete initial state. The actions

of the concrete system *rewrite* the abstract configurations into new configurations. TVLA's model checker performs on-the-fly model checking by exploring new configurations until all the configurations are covered. Because of *summarization*, the set of abstract configurations is bounded, and the explicit exploration of the abstract domain will terminate. Thus, no abstract model is built up front on which model checking is performed. In contrast, in our method we build an abstract model up front.

The framework proposed by TVLA is extremely general; essentially any real world system can be handled by this framework. Consequently, no method for choosing the predicates can be specified and the central problem in predicate abstraction, namely what predicates to use, is left unsolved. For the examples considered in [81], the authors manually pick the predicates. In contrast, our method specifies a framework for what type of predicates to pick. In our case studies, the relevant predicates were constructed just by a syntactic exploration of the protocol code.

2.7.4 Counter Abstraction

Counter Abstraction is an intuitive method to use on parameterized systems, and it has been employed by various researchers in different contexts [5; 28; 34; 66]. Pnueli et al. [66], who coined the term counter abstraction, show how concurrent systems composed of symmetric and finite state processes can be handled automatically. The essential idea in counter abstraction is to have a counter C_i for each possible local state i of the processes. Counter C_i then counts the number of processes in state i in a given concrete system configuration. The counters are typically bounded by a small value so that the ab-

stract system consisting of the counters is finite state. Environment abstraction generalizes counter abstraction since the abstract descriptions $\Delta(x)$ can serve as counters. But, instead of counting simply the processes according to their local states, we count processes according to their local states *and* according to their relationship to the reference process. It is the latter feature that lets us handle systems in which each replicated process has infinite state space.

In a symmetric protocol, the identities of the processes cannot be used in the protocol code. For instance, a condition of the form

$$\textbf{forall} \quad j < i. \quad \Phi(j)$$

appearing in the code of process i breaks the symmetry because the process with id 1 will exhibit different behavior from process with id $m > 1$ (the condition is trivially true for process 1 and not so for other processes with ids greater than 1). Most real life systems are not symmetric, that is, the code for each process can make use of the process id. Thus, the verification of Szymanski's protocol in [66] requires manual introduction of new variables. Our method does not require each process to be finite state nor do we require the processes to be symmetric.

In [66], the notion of "all-but-one" counter abstraction is described. The idea here is to apply counter abstraction to all processes except one. By tracking one special process in detail, they are able to reason about single index liveness properties. It is important to note the following:

- In a symmetric protocol, any process can be chosen as the special process, it makes no difference in the abstraction.

- Further, the other processes in the system are abstracted (or counted) according to their local states alone, not based on their relationship to the special process. This is the crucial difference between our method and counter abstraction.

There are also important differences in how we compute the abstract model and how the abstract model is computed in [66]. In [66], the abstract model is computed precisely using symbolic techniques. In contrast, we over-approximate the abstract model by considering each transition statement of the protocol code.

Another approach that uses, among other things, counter abstraction is the method proposed by Henzinger et al. [45]. Like "all-but-one" abstraction of Pnueli et al. [66], Henzinger et al. also track one thread in detail (called the *main* thread). As with counter abstraction, the main thread does not serve as a reference process. The other threads in the system are abstracted independently of the main thread.

2.8 Conclusion

In this chapter we presented the mathematical principles underlying environment abstraction. This abstraction framework is designed specifically for systems with replicated components. Informally, this framework is built around the insight that when we humans reason about systems with replicated components we focus on one particular component while considering the other components only abstractly.

In this chapter, we assumed that the replicated components were processes. In general

the replicated components can vary. For instance, a memory bank can be treated as a collection of identical memory cells. Our method can be extended to all these instances as well.

It is crucial to distinguish two different issues that have been covered in this chapter: **(i)** "what abstract state space to consider?" and **(ii)** "how to build the abstract model?" or rather, "how to use this abstract state space to accomplish verification?". In answer to the first question we propose using an abstract state space of descriptions $\Delta(x)$. In answer to the second question, we propose constructing, up front, an over-approximate abstract model. It is not necessary for using environment abstraction that we build the abstract model up front. We can use an explicit state exploration as done by TVLA as well. However, we think that for protocols, which can usually be expressed using only a few types of basic constructs, our way of building the abstract model up front is the best possible choice.

In the next chapter, we instantiate environment abstraction in the context of cache coherence verification. We will cover all the issues raised in this chapter from descriptions to computing the abstract model.

Chapter 3

Environment Abstraction for Verification of Cache Coherence Protocols

3.1 Introduction

The performance advantages of multi-core shared memory architectures have created a strong industrial trend towards multi-core designs. Such state-of-the-art architectures crucially rely on caching mechanisms for increased performance. The increasing complexity of such systems is reflected in the intricate cache protocols they employ. As these cache coherence protocols are inherently parameterized, it is a challenging task to ensure their correctness by automatic verification methods. In this chapter, we show how to use the

environment abstraction to verify directory based cache coherence protocols – the most widely employed class of cache coherence protocols. We use this abstraction method to verify the standard safety property of several versions of the GERMAN's protocol and of a modified version of the FLASH protocol.

3.1.1 Cache Coherence Protocols

Caching mechanisms are ubiquitous in modern computer systems. Computer systems usually have several memory banks, each with different latency. To reduce the time needed to access data items and thus improve the performance, caching mechanisms are used to store frequently accessed data items in the fastest available memory bank.

Modern processors typically come with several levels of caches. A cache is a small memory bank that usually sits on the motherboard of a processor. Higher the physical distance of a cache, the higher the latency of that cache. A data item that is frequently used by the processor can be stored in one of its caches. When the data item is needed again, instead of going all the way to the main memory, the data item can be supplied from the cache itself.

While the availability of such caches dramatically increases the performance of a multi-processor system, care must be taken to prevent processors from accessing data items in an unsafe manner. For instance, two processors $P1$ and $P2$ might both have a data item d in their local caches. After performing some computations both the processors may decide to write back their local values of d to the main memory. If this activity is not coordinated properly, the value of d as determined by one of the processors will be

lost. In the presence of multiple data items, such loss can lead to computations that are not feasible in any legal execution of the processors. Thus, to coordinate the activities of different processors in a multiple processor system and to provide a consistent view of the memory to all the processors cache coherence protocols are used.

There are broadly two types of cache coherence protocols, namely *snoopy* and *directory based* protocols. The first class of protocols is broadcast based with no central coordination. The second type of protocols, the *directory* based protocols, are based on point to point communication and have centralized coordination. In snoopy protocols, all the processors (more precisely, their cache controllers) monitor the activities on the common system bus. Since every processor knows what data items the other processors are using, cache coherence can be achieved quite simply. In snoopy protocols, there is no centralized decision making. The actions of the local caches, which have full knowledge of other caches, are enough to ensure cache coherence. Snoopy protocols are typically used in systems which have a small number of processors.

Directory based protocols, on the other hand, use centralized decision making to ensure cache coherence. For each data item, one of the processors is designated as the *home* or the *directory* process. Requests by the processors to access a data item are sent to the home process for that item. The home process maintains detailed information about which processors are using the item and can respond appropriately to each request. Directory protocols are more widely used as they scale better [56].

A crucial issue in the design of cache protocols is the speed with which a data item is delivered to the requesting process. Depending on how this issue is handled, directory pro-

tocols are of two types: lazy and eager protocols. In lazy protocols, the directory process doesn't grant exclusive access to a requesting processor until it has received acknowledgements from other processors in the system that were sharing the data item and were sent invalidate messages. Eager protocols, on the other hand, do not wait for the acknowledgements. In our experiments, we considered an eager version of the FLASH protocol and a lazy version of the GERMAN protocol.

There is no consensus on which type of cache coherence protocols – snoopy or directory based – is better. While snoopy protocols tend to have lower latency, they require totally-ordered interconnect with a broadcast mechanism (usually a bus) connecting all the processors. Directory protocols do away with the interconnect in exchange for higher latency. In an informative article [56], Martin revisits this debate from a verification point of view.

There are multiple correctness issues to be considered while designing cache coherence protocols. The simplest correctness properties talk about the way a single data item is accessed (called *coherence* properties). For instance, all cache protocols require that a data item cannot be held in *exclusive* (or *dirty*) state while it is held in *shared* state by some other processor. It is also required that a requesting process will eventually get the data item. In our work, we have dealt with correctness properties involving only one data item. Cache properties involving multiple data items (called *consistency* properties) are usually complex and very hard to verify formally. For example, verifying whether all the executions that a cache protocol allows are legal under the chosen *memory consistency model* is a very hard problem. While there has been some effort to address this problem, it is far from being solved [20].

In this chapter, we will first formalize the system model for cache coherence protocols. Our model will contain one non-replicated process (the *central process*) representing the *home* processes, and an unbounded number of replicated processes (the *local processes*) representing the caches. The transitions executed by the caches are very simple, whereas the central directory can perform quite complex actions. For instance, the directory can keep pointer variables, which point to the caches, and modify the local states of the caches. We will describe a simple language for writing the transitions of local processes and the central process. The constructs used in this language ignore the low level implementation details and describe the protocol at an *algorithmic* level. In fact, these constructs correspond to the way system designers think about cache protocols.

We will then use the environment abstraction presented in the previous chapter to parameterically verify the safety property of cache coherence protocols.

Outline

In Section 3.3 we describe a modeling language that accounts for the specifics of cache coherence protocols, and in Section 3.4 we describe how to apply environment abstraction to verify cache coherence protocols. Section 3.5 describes a redundancy criterion for removing *set* variables which drastically reduces the size of the abstract models. Section 3.6 presents our approach to over-approximating the abstract model. The last two sections contain experimental results and conclusions.

In the rest of this chapter, we will, for the sake of simplicity, speak of "caches" and "directory" instead of "local processes" and "central process" respectively. We will consider

only coherence properties involving a single data item.

3.2 Discussion of Related Work

Parameterized verification of cache coherence protocols has received considerable attention, see [21; 28; 29; 52; 58; 64; 67; 68].

The papers closest to our approach are [67; 68] and [29]. Delzanno and Bultan [29] that describe a constraint based verification method for handling the safety and liveness properties of GERMAN's protocol. Their approach avoids the problem of handling variables which store cache ids and sets of cache ids by exploiting synchronization labels for actions. But, real protocols do not use such synchronizations mechanisms, which are unsuited to model cache coherence protocols. For example, when using such synchronization labels, staggered reception of messages by different caches (during a broadcast transition) cannot be modeled.

Pong and DuBois [67; 68] developed an explicit state model checking method that uses a technique very similar to counter abstraction to exploit the symmetry and homogeneity of cache coherence protocols. They handle snoopy protocols as well as directory based protocols. Note that neither [67; 68] nor [29] have the notion of a reference process. Consequently, in contrast to our approach, they cannot verify single index liveness properties. Furthermore, their abstraction explicitly considers the set variables. In our abstraction, we are able to eliminate the set variables from the abstract model, which drastically reduces

the size of the abstract model.

The *compositional method* of McMillan [58] uses compositional reasoning to handle infinite state systems including directory based protocols. This technique, which requires user intervention at various stages, has been applied to verify safety and liveness properties of the FLASH protocol. The paper [21] by Chou et al. presents a method along similar lines that was used to verify safety of FLASH and GERMAN's protocol. The *aggregated transactions* method pioneered by Park and Dill [63] is based on theorem proving, and has been used to verify directory based protocols such as the FLASH protocol. The essential idea behind this technique is to collect the various statements in the protocol code into a set of 7-8 high level transactions. The user has to provide proofs of correspondence between the high level transactions and the protocol code.

Pnueli et al. [64] show how to verify safety of GERMAN's cache coherence protocol. They do not verify liveness properties nor have they handled FLASH protocol.

Bingham et al. [9] describe a method for verifying infinite state systems that can be modelled as *Well Structured Transition Systems* or WSTS systems. WSTS systems are a well-studied class of infinite state systems for which the problem of reachability of error states is decidable (subject to some technicalities). They applied this method to GERMAN'S protocol and verified data coherence (that is, read on a data item returns the last written value).

3.3 System Model for Cache Coherence Protocols

Our system model reflects the structure of real-life cache coherence protocols. A typical cache coherence system contains several caches, one of which is designated as the *home* cache. The home cache maintains a directory and regulates the access to the data items for which it is responsible. Following [64], we will model the home cache as consisting only of the *directory* and call it the *directory* or directory process. Since the number of caches in the system is not fixed, cache coherence protocols are classical instances of parameterized systems. Note, however, that the presence of the directory in the system breaks the symmetry between the processes. Since we are concerned with coherence properties of a cache protocol, it is enough to consider only one data item. Thus, we will implicitly assume that there is only one data item in the system.

In our formal model, we consider asynchronous systems consisting of K caches running the same program P and one directory running a different program C. For given programs P and C, the system consisting of K caches and one directory is denoted by $\mathcal{P}(K)$. Each cache has a distinct id in the range $\mathsf{R} = \{1, \ldots, K\}$. As all caches are identical, their sets of variables are also named identically. When necessary, we will write $v(i)$ to refer to variable v of cache i.

The system $\mathcal{P}(K)$ is formally modelled as Kripke structure (S_K, I_K, R_K, L_K). The set of states S_K is given by tuples of the form $\langle \mathcal{L}_1, \ldots, \mathcal{L}_K, \mathcal{C} \rangle$, where each $\mathcal{L}_i$ is the local state of cache i and $\mathcal{C}$ is the state of the central process C. In the following sections, we will describe the state space the caches and the central directory. Then, we will define the transition relation R_K in terms of the transitions the caches and the central process take.

3.3.1 State Variables

The caches are essentially finite state machines, and thus each cache has one finite range control variable $\mathsf{pc_L}$ with range $\{1, \ldots, T\}$. Since multiple finite range variables can always be encoded as one variable, there is no loss of generality. In our implementation and in examples later in the chapter, we will, in fact, tacitly use multiple finite range variables.

The directory has three different kinds of variables, distinguished by the way they are used:

- The *control variable*, $\mathsf{pc_C}$, has finite range $\{1, \ldots, F\}$, $F \geq 1$, and represents the control locations of the directory.
- The *pointer variables*, $\mathsf{ptr}_1, \ldots, \mathsf{ptr}_b$, where $b \geq 1$, are used to store the ids of caches. Thus, in a system $\mathcal{P}(K)$, the range of the pointer variables is R.
- The *set variables* $\mathsf{set}_1, \ldots, \mathsf{set}_c$, where $c \geq 1$, are used to store sets of cache ids, and their range is the powerset 2^{R}.

Example. In GERMAN'S protocol, the variable *currclient* holds the id of the cache that the directory is currently communicating with. This variable is naturally modeled as a *pointer* variable. Similarly, GERMAN'S protocol has a list *sharlist* containing all caches that hold the data item in a *shared* state. This list is naturally modeled as a *set* variable.

A state of system $\mathcal{P}(K)$ is a tuple $\langle \mathcal{L}_1, \ldots, \mathcal{L}_K, \mathcal{C} \rangle$ where the $\mathcal{L}_i$ are the control locations of the caches, and $\mathcal{C}$ is the state of the directory,. The state of the directory, $\mathcal{C}$, is a valuation of the tuple $\langle \mathsf{pc_C}, \mathsf{ptr}_1, \ldots, \mathsf{ptr}_b, \mathsf{set}_1, \ldots, \mathsf{set}_c \rangle$.

We shall see below that the ptr variables are used solely to access the state of the caches. That is, no arithmetic or comparison on ptr variables is allowed. Similarly, set variables are used either to access caches or in membership queries (i.e., whether a cache belongs to the set or not). We assume that all variables are used in a *type-safe* way, that is, they are assigned or compared only against values from their ranges.

The *initial state* of the directory and the caches is given by a fixed valuation of all variables.

3.3.2 Program Description for the Caches

We will describe the transitions of the caches and the central process using a few high level constructs. Caches have very simple control flow structures, as they can move only from one control location to another. We can describe the cache transitions using the following transitions:

$$\mathsf{pc_L} = L_1^{\mathsf{L}} : \mathbf{goto}\ \mathsf{pc_L} = L_2^{\mathsf{L}}$$

The semantics of the transition is simple: a cache $P(i)$ in control location L_1^{L} can, at a nondeterministically chosen timepoint, change its state variable to L_2^{L}. The **goto** statement is deterministic in the sense that for each location L_1^{L}, there is at most one jump goal L_2^{L}.

Note that the state of a cache can also be changed by the directory, see Section 3.3.3.

3.3.3 Program Description for the Directory

The directory can execute more complex programs than the cache. In particular, it can execute a

- *simple action* to change its control variables, or
- *update action* to update its pointer or set variables, or
- *remote action* to change the state of a cache referenced by a pointer.

These *basic actions* reflect the operations used in a typical directory based cache coherence protocol. We will see that, of the above actions, the update action and the remote action depend on the state of the caches. However, only the remote action can change the state of a cache. Below we will define the actions in more detail.

A directory transition statement has the form

$$\textit{guard}: \ \textbf{do actions}\ A_1, A_2, .., A_k$$

where $A_1, \ldots, A_k$ are basic actions as described below, and *guard* is a condition of the form $\mathsf{pc_C} = L \wedge \Phi(\mathsf{ptr}, \mathsf{set})$. Here, L is a directory control location and $\Phi(\mathsf{ptr}, \mathsf{set})$ is a Boolean combination of expressions of the form $\mathsf{pc_L}[\mathsf{ptr}_i] = L^{\llcorner}$, $\mathsf{ptr} \in \mathsf{set}_i$, or $\mathsf{set}_i = \emptyset$.

The semantics of this statement is as follows:

1. If *guard* is true, then execute the actions $A_1, \ldots, A_k$.
2. The whole transition, including the evaluation of the guard, is executed atomically in one time step with actions $A_1, \ldots, A_k$ being executed in that order.

3. We will assume that the basic actions in a transition do not conflict with each other. In other words, no variable should be modified by more that one action. This implies that there is only one *simple action* per transition, that no ptr variable is updated by more than one action, that only one set variable is updated, and that remote actions are executed on different caches.

We will now describe the basic actions in more detail.

Simple Actions have the format **goto** $\mathsf{pc}_\mathsf{C} = L^\mathsf{C}$, where L^C is a directory control location. The semantics of this action is that the directory control variable pc_C is set to L^C.

Update Actions come in several formats:

- **assign** $\mathsf{ptr}_i = \mathsf{ptr}_j$ and **assign** $\mathsf{set}_i = \mathsf{set}_j$. The next value of ptr_i is set to the current value of ptr_j.

- **add** ptr_i **to** set_j and **remove** ptr_i **from** set_j. Add or remove the cache pointed to by ptr_i from set set_j.

- **pick** ptr_i **from** $\mathcal{S}^\mathsf{L}$, where $\mathcal{S}^\mathsf{L}$ is a list of (constant) cache control locations. The semantics of this action is that the variable ptr_i is *nondeterministically* made to point to one of the caches whose control location is in $\mathcal{S}^\mathsf{L}$. If there is no such cache, then ptr_i is unchanged.

Remote Actions have the form **remote** $\mathcal{V}$: **goto** $\mathsf{pc}_\mathsf{L} = L^\mathsf{L}$, where L^L is a cache control location and $\mathcal{V}$ is a pointer variable. This action enforces the new control location L on

the cache pointed to by $\mathcal{V}$. In general, the remote action can also have the form

remote $\mathcal{V}$: **map**

where **map** is a switch statement of the form:

$$
\begin{array}{l}
\textbf{switch } \mathsf{pc}_{\mathsf{L}}\{ \\
\qquad\qquad L_1^{\mathsf{L}} \;\; : \;\; \textbf{goto } \mathsf{pc}_{\mathsf{L}} = L_1^{\mathsf{L}'} \\
\qquad\qquad L_2^{\mathsf{L}} \;\; : \;\; \textbf{goto } \mathsf{pc}_{\mathsf{L}} = L_2^{\mathsf{L}'} \\
\qquad\qquad \ldots \\
\qquad\qquad \ldots \\
\qquad\qquad \}
\end{array}
$$

This action enforces the cache pointed to by $\mathcal{V}$ to execute the switch statement. The **remote** action is analogously defined for set variables. A **remote** action for a set variables forces all the caches in the set variable to execute the switch statement simultaneously. While GERMAN'S protocol does not require **remote** actions on set variables, FLASH protocol does.

The **remote** action is used to model the communication from central process to the local caches. For example, in GERMAN'S protocol, the central directory process sends invalidate message to all the caches present in the $invlist$ set variable one cache at a time. The central process first picks a cache present in $invlist$ by assigning a pointer variable

$temptr1$ appropriately. Then the central process writes an $invalid$ message to the incoming channel, $chan2$, of the cache pointed to by $temptr1$. Since, we model communication channels as internal variables of the caches, the effect of central process writing to $chan2$ can be accurately modelled as a **remote** action with the general switch statement.

3.3.4 Describing Real-Life Protocols

GERMAN'S protocol and the FLASH protocol can be naturally expressed in our protocol description language. These protocols share a common basic functionality: when a cache requests *shared* access to a data item, the directory grants the request if the data item is not held in exclusive state by any other cache. Otherwise, the directory sends a message to the cache having exclusive access to the data item to relinquish control over the data item. Subsequently, the directory grants *shared* access to the cache that issued the request. When a cache requests *exclusive* access to the data item, the directory grants the request if no other cache has any form of access to the data item. Otherwise, the directory sends messages to all caches having access to the data item to invalidate their local copies. The directory can either wait to receive acknowledgements from the caches (*lazy* operating mode) or grant exclusive access to the cache which issued the request (*eager* operating mode). In this thesis, we consider the FLASH protocol operating in *eager* mode and GERMAN'S protocol operating in *lazy* mode.

While the basic functionality of many cache coherence protocols essentially follows the above description, there are a lot of additional low level details that add to the complexity of a directory based protocol and need to be accounted for in our input language.

In a typical protocol, the caches communicate with the directory process using dedicated communication channels. The caches execute relatively independently of each other. Thus, the simple **goto** statements for caches suffice to model the transitions of the caches. The directory process usually maintains pointers to caches and also sets of caches. The pointer and set variables are used to receive and send messages to specific recipients.

Following other work in this area [29; 64], we assume that the communication channels between caches and the directory are of length 1. The communication channels are modeled using local variables of the caches. Since the directory can read and write to the local variables via the **remote** action, the local variables can simulate communication channels. For instance, in GERMAN's protocol we have a central transition statement:

$currcmd = empty \wedge read = yes \wedge chan1[currclient] = reqshar$:
do actions goto $read = no \wedge currcmd = reqshar$,
remote $currclient$: **goto** $chan1 = empty$

which shows how the directory communicates with a cache. Here, the pointer variable *currclient* points to a local process, and $chan1[currclient]$ is the variable that serves as a communication channel from the cache to the directory. Note also that there is more than one control variable in the directory, namely, *read*, and *currcmd*.

The above transition says that if there is a *reqshar* message in channel *chan1*, the directory process reads it by updating the variable *currcmd* using the **goto** action. After reading it, the directory removes the message from *chan1* using the **remote** action which sets *chan1* to *empty*. Broadcast actions can also be described succinctly using **remote** actions.

Note that in our language, the protocol is described at a high level without getting

into implementation details, reflecting the abstraction level at which designers think about protocols. This approach is consistent with the current trend towards synthesis of low level designs from reliable and easily verifiable high level designs.

The full descriptions of the FLASH protocol and GERMAN'S protocol are given in Section 3.9.

3.4 Environment Abstraction for Cache Coherence Protocols

In this section, we instantiate environment abstraction for verifying cache coherence protocols.

3.4.1 Specifications and Labels

Most properties of interest in parameterized systems refer to the *control locations*: for example, typical safety properties say that no two caches can hold the same data item in exclusive state at the same time. Usually we are interested in verifying such properties for *each* cache in the system, not for a *specific* cache. In this chapter, we will consider the two-indexed safety property

$$\forall x, y. x \neq y \wedge \mathsf{pc}[x] = \mathbf{crit} \Rightarrow \neg(\mathsf{pc}[y] = \mathbf{crit})$$

This can be equivalently written as a single index property

$$\forall x. (\mathsf{pc}[x] = \mathbf{crit} \Rightarrow \neg(\mathbf{crit} \in \mathsf{env}(x)))$$

To handle such specifications, the set of labels L we use will have labels of two types:

- $\mathsf{pc}_\mathsf{L}[x] = L$, and
- $L \in \mathsf{env}(x)$.

3.4.2 Abstract Model

As mentioned previously, we will represent abstract descriptions as tuples as this simplifies the presentation significantly. The abstract states will contain information about

- the internal state of the reference cache
- the internal states that occur in other caches, and
- the internal state of the directory.

Formally, an abstract state is a tuple

$$\hat{s} = \langle \mathbf{pc}_\mathsf{L}, e_1, \ldots, e_T \;;\; \mathbf{pc}_\mathsf{C}, \widehat{\mathsf{ptr}}_{\,1}, \ldots \widehat{\mathsf{ptr}}_{\,b}, \widehat{\mathsf{set}}_{\,1}, \ldots, \widehat{\mathsf{set}}_{\,c} \rangle$$

whose semantics we will explain in the following paragraphs.

First, and importantly, $\hat{s}$ describes the system from the viewpoint of the reference cache: $\mathbf{pc}_\mathsf{L}$ is the control location of the reference cache and each bit e_i tells whether some other cache is in control location i. Moreover, $\hat{s}$ contains information about the directory: $\mathbf{pc}_\mathsf{C}$ is the control location of the directory, and $\widehat{\mathsf{ptr}}_{\,i}$ and $\widehat{\mathsf{set}}_{\,i}$ are abstractions of the pointers and sets of the directory.

Thus, the variables have the following ranges: $\mathbf{pc}_{\mathsf{L}} \in \{1, \ldots, T\}$ is a cache control location, $\mathbf{pc}_{\mathsf{C}} \in \{1, \ldots, F\}$ is a directory control location, and the e_i are Boolean values representing the "environments". The bit e_i has value 1 if there exists a cache y different from x that is in control location i, i.e., "the environment of x contains a cache in control location i". This is expressed by the quantified formula

$$\mathcal{E}_i(x) \doteq \exists y \neq x.\mathsf{pc}_{\mathsf{L}}[y] = i$$

which we call the *environment predicate*. Note that an environment predicate $\mathcal{E}_L(x)$ and its corresponding bit e_L in the abstract state tell us if the atomic property $L \in \mathsf{env}(x)$ holds true in a state.

Concerning the pointers, it is important to note that in the abstract model, a pointer cannot refer to a cache, but only to an abstracted cache, i.e., an environment or the reference cache itself. Thus, we introduce the set $\{\mathsf{ref}\} \cup \{1, \ldots, T\}$ of *abstract locations*. The abstract locations are the possible values for the pointers in the abstract model. *An abstract pointer value $i \in \{1, \ldots, T\}$ means that the pointer refers to a cache in control state i, and an abstract pointer value* ref *means that the pointer refers to the reference cache.*

Analogously, the abstract set variables $\widehat{\mathsf{set}}_i$ range over the powerset $2^{\{1,\ldots,T\} \cup \{\mathsf{ref}\}}$ of the abstract locations.

Definition 3.4.1. Let s be a concrete state in a concrete system $\mathcal{P}(K)$, and consider a cache p in $\mathcal{P}(K)$. Then $\hat{s}$ is the abstraction of state s induced by cache p, in symbols

$$\alpha_p(s) = \langle \mathbf{pc}_{\mathsf{L}}, e_1, \ldots, e_T \; ; \; \mathbf{pc}_{\mathsf{C}}, \widehat{\mathsf{ptr}}_1, \ldots \widehat{\mathsf{ptr}}_b, \widehat{\mathsf{set}}_1, \ldots, \widehat{\mathsf{set}}_c \rangle$$

if the following conditions hold:

1. In state s, cache p is in control location $\mathbf{pc}_{\mathsf{L}}$, i.e.,

$$s \models \mathbf{pc}_{\mathsf{L}} = \mathsf{pc}_{\mathsf{L}}[p].$$

"The reference cache is in control location $\mathbf{pc}_{\mathsf{L}}$*."*

2. Each e_i is the truth value of the *environment predicate* $\mathcal{E}_i(x)$ for cache p, i.e.,

$$s \models \exists y \neq p.\mathsf{pc}_{\mathsf{L}}[y] = i \quad \text{iff} \quad e_i = 1.$$

"The environment contains a cache in control location i*. "*

3. The directory is in control location $\mathbf{pc}_{\mathsf{C}}$, i.e.,

$$s \models \mathsf{pc}_{\mathsf{C}} = \mathbf{pc}_{\mathsf{C}}.$$

"The directory is in control location $\mathbf{pc}_{\mathsf{C}}$*."*

4. Each pointer $\widehat{\mathsf{ptr}}_i$ has value $\mathrm{abs}(\mathsf{ptr}_i)$, where $\mathrm{abs}(\mathsf{ptr}_i)$ is the abstract location pointed to by ptr_i, i.e.,

$$\mathrm{abs}(\mathsf{ptr}_i) \doteq \begin{cases} \mathsf{ref} & \text{if } s \models \mathsf{ptr}_i = p \\ \mathsf{pc}_{\mathsf{L}}[\mathsf{ptr}_i] & \text{otherwise} \end{cases}.$$

"The i*-th pointer points to the abstract location* $\widehat{\mathsf{ptr}}_i$*."*

5. The sets $\widehat{\mathsf{set}}_i$ generalize the pointers in the natural way, i.e.,

$$\widehat{\mathsf{set}}_i \doteq \{\mathrm{abs}(q) : q \in \mathsf{set}_i\}.$$

"The i*-th set variable points to the set* $\widehat{\mathsf{set}}_i$ *of abstract locations."*

Before we can apply environment abstraction, we have to prove that the set of abstract states $S^{\mathcal{A}}$ and the set of labels L satisfy the coverage and congruence conditions.

Proposition 1. For the abstraction mapping α given above, the set of abstract states $S^{\mathcal{A}}$ satisfies the coverage condition.

Proof. Our abstract state space $S^{\mathcal{A}}$ consists of all possible tuples of the form $\langle \mathbf{pc}_{\mathsf{L}}, e_1, \ldots, e_T \ ; \ \mathbf{pc}_{\mathsf{C}}, \widehat{\mathsf{ptr}}_1, \ldots \widehat{\mathsf{ptr}}_b, \widehat{\mathsf{set}}_1, \ldots, \widehat{\mathsf{set}}_c \rangle$. This fact combined with our abstraction mapping defined above ensure that no matter what concrete state s and what process c we consider $\alpha_c(s) \in S^{\mathcal{A}}$. Thus, the coverage condition is trivially satisfied by our abstract state space. □

Proposition 2. For every label $l(x) \in L$ and every abstract state $\hat{s} \doteq \langle \mathbf{pc}_{\mathsf{L}}, e_1, \ldots, e_T \ ; \ \mathbf{pc}_{\mathsf{C}}, \widehat{\mathsf{ptr}}_1, \ldots \widehat{\mathsf{ptr}}_b, \widehat{\mathsf{set}}_1, \ldots, \widehat{\mathsf{set}}_c \rangle$, the abstract description $\Delta(x)$ corresponding to $\hat{s}$ either implies $l(x)$ or its negation. That is, $\Delta(x) \Rightarrow l(x)$ or $\Delta(x) \Rightarrow \neg l(x)$

Proof. Clearly, if the label $l(x)$ is of the form $\mathsf{pc}[x] = L$, then the abstract state $\Delta(x)$ either implies $l(x)$ or its negation.

In case $l(x)$ is of the form $L \in \mathsf{env}(x)$, then again $\Delta(x)$ implies $l(x)$ or its negation. This follows easily from the fact that each e_i indicates whether or not there is an environment process with control location i. If the bit e_L corresponding to control location L is 1 in the tuple corresponding to $\Delta(x)$ then $\Delta(x) \Rightarrow l(x)$. Otherwise, $\Delta(x) \Rightarrow \neg l(x)$. □

The abstract model $\mathcal{P}^{\mathcal{A}} \doteq (S^{\mathcal{A}}, I^{\mathcal{A}}, R^{\mathcal{A}}, L^{\mathcal{A}})$ is defined as in Section 2.2. The following corollary is then just an instantiation of Theorem 2.2.4

Corollary 2 (Soundness of Abstraction). Let $\mathcal{P}(\mathbf{N})$ be a parameterized cache coherence system and $\mathcal{P}^A$. Consider a control specification $\forall x.\phi(x)$. If $\mathcal{P}^{\mathcal{A}} \models \phi(x)$ then $\mathcal{P}(\mathbf{N}) \models \forall x.\phi(x)$.

From Environment Bits to Counters

To keep the presentation simple we have represented the variables e_i as bits which indicate whether there exists a cache in control location i. To make the abstraction more precise, the e_i can be easily generalized to *counters* of range, e.g., $\{0, 1, 2\}$, where 2 is called the *counter threshold*. Then $e_i = 0$ means that there is no cache $y \neq x$ in control location i, $e_i = 1$ means that there is exactly one cache $y \neq x$ in control location i, and $e_i = 2$ means that at least two caches $y \neq x$ are in control location i. All results in this chapter can be readily generalized to counter thresholds, and our tool also supports arbitrary counter thresholds.

3.5 Optimizations to Reduce the Abstract State Space

3.5.1 Eliminating Unreachable Environments

The abstract model as described so far has a environment bit e_L for each possible local state L of the caches. It may be the case that not all possible local states are indeed reachable and the corresponding abstract bits (or counters), which are redundant, can be eliminated. Our experiments in fact indicate that this kind of optimization achieves significant reduction in the size of the abstract model.

Finding the local reachable states can be done as follows. First note that the local state of a cache can change in two ways: 1) the cache executes a local **goto** action, or 2) the central process changes the state of the cache using a **remote** action. Considering the former case, if a local state s_1 is reachable and there is a local transition

$$\mathsf{pc_L} = s_1 : \textbf{goto } \mathsf{pc_L} = s_2$$

then local state s_2 is reachable as well. Thus, we add a transition (s_1, s_2) to a reachability relation R (the reachability relation R is initially empty).

For the latter case, consider a remote action:

$$\textbf{remote } \mathcal{V} : \textbf{map}$$

where **map** is a switch statement of the form

switch $\mathsf{pc_L}\{$

$$L_1^{\mathsf{L}} \quad : \quad \textbf{goto } \mathsf{pc_L} = L_1^{\mathsf{L}'}$$

$$L_2^{\mathsf{L}} \quad : \quad \textbf{goto } \mathsf{pc_L} = L_2^{\mathsf{L}'}$$

$$\ldots$$

$$\ldots$$

$$\}$$

Here $\mathcal{V}$ can be a pointer variable or a set variable. In case $\mathcal{V}$ is a pointer variable, we will say $\mathcal{V}$ can point to a local state s if a cache in state s can be pointed to by $\mathcal{V}$. Similarly, if $\mathcal{V}$ is a set variable, we will say $\mathcal{V}$ can point to a local state s if a cache in state s can belong to the set $\mathcal{V}$.

Now, if $\mathcal{V}$ can point to local state $s_1 = L_1^{\mathsf{L}}$ and s_1 is a reachable local state then the local state $s_2 = L_1^{\mathsf{L}'}$ is also reachable. Thus, we add (s_1, s_2) to R as well. By syntactically examining the protocol code, we can determine an over-approximation of all the local states that $\mathcal{V}$ can point to, as described below.

First consider the case where $\mathcal{V}$ is a pointer variable. Suppose $\mathcal{V}$ is the pointer variable ptr_i and the central process assigns $\mathcal{V}$ a value using an action of the form

$$\textbf{pick } \mathsf{ptr}_i \textbf{ from } \mathcal{S}^{\mathsf{L}}.$$

The pointer $\mathcal{V} = \mathsf{ptr}_i$ can point to any location in $\mathcal{S}^{\mathsf{L}}$. Finding the union of $\mathcal{S}^{\mathsf{L}}$'s from all actions that modify $\mathcal{V}$ gives us an over-approximation of the set of all caches locations

that $\mathcal{V}$ can point to. Call this set of locations S. For every, $s \in S$ we add (s, s'), where s' is the location that s is mapped to by **map**, to the reachability relation R.

In case $\mathcal{V}$ is a set variable the over-approximation of the set of location $\mathcal{V}$ can point to is computed as in Remark 8.

Once we have R, an over-approximation to the set of reachable local states is given by $R^*(\mathbf{init})$ where **init** is the initial state of the caches. It is enough to have counters corresponding to only these reachable locations.

3.5.2 Redundancy of the Abstract Set Variables

In this section we will describe how the set variables can be eliminated from many real protocols including GERMAN'S protocol and the FLASH protocol by a straightforward program analysis. In the following sections, we can therefore assume that no set variables are present. The evident motivation to eliminate the set variables is state explosion. Since each concrete set variable gives rise to an abstract set variable with domain $2^{\{1,\ldots,T,\mathsf{ref}\}}$, the abstract model may become prohibitively large.

Our method is based on the observation that in many real-life protocols, the following pattern occurs: whenever a cache is added to a set by an **add** action, then the *same transition* also contains a **remote** action which determines the control location of the cache (that is, when an **add** ptr_i to set_j action occurs a **remote** ptr_i ... action occurs as well). In practice, this means that whenever a cache is added to a list, it also receives a message. Similarly, each **remove** action is also accompanied by a **remote** action. Set variables following this pattern are in fact often redundant, that is, conditions involving sets can be

replaced by equivalent conditions on the local states of the caches. We will now describe how to determine if a set is redundant.

Let us fix a set variable set_j. Then we can partition the statements in the program D of the directory into three sets:

- T_j^{in} is the set of **remote** actions which occur together with an action of the form **add** ptr_i **to** set_j.

- T_j^{out} is the set of **remote** actions which occur together with an action of the form **remove** ptr_i **from** set_j.

- The remaining **remote** actions in the program are collected in the set T_j^{rest}.

Using these three sets T_j^{in}, T_j^{out} and T_j^{rest}, we will compute three sets of cache states R_j^{in}, R_j^{out}, R_j^{rest}. Intuitively, R_j^{in} will be the set of all states that a cache can have while it is a member of set_j. Similarly, R_j^{out} contains all states that occur in caches that are not members of set_j.

Given a set of cache states S, the set $r(S)$ is the set of all states reachable from states in S by local transitions (i.e., **goto**'s in the program of the cache) and by **remote** actions in T_j^{rest}. Note that for a given set S, $r(S)$ can be obtained by a simple syntactic computation on the program. With this notation, we can easily describe R_j^{in}, R_j^{out}, and R_j^{rest}.

- R_j^{rest} is the set of cache states reachable from the initial cache states, i.e., $R_j^{rest} = r(I_{\text{init}})$.

- R_j^{in} is computed as follows: We collect all jump goals of **remote** [1] actions in T_j^{in} into a set I_{in}. Then R_j^{in} is the set of cache states reachable from I_{in}, i.e., $r(I_{in})$.

- R_j^{out} is computed analogously to R_j^{in}, with I_{in} replaced by I_{out}, the set of jump goals in T_j^{out}.

If the sets R_j^{rest}, and R_j^{out} do not share any common elements with R_j^{in}, then the variable set_j is redundant in the sense of the following theorem:

Theorem 3.5.1. *Assume that $R_j^{rest} \cap R_j^{in} = \emptyset$ and $R_j^{out} \cap R_j^{in} = \emptyset$, and consider a global state s of a concrete system $\mathcal{P}(K)$ with a process p. Then*

$$s \models p \in \mathsf{set}_j \qquad \text{iff} \qquad s \models \mathsf{pc_L}[p] \in R_j^{in},$$

i.e., process p is contained in set_j iff its control location is in R_j^{in}.

Proof. Consider first the sets R_j^{rest} and R_j^{in}. Since $R_j^{rest} \cap R_j^{in} = \emptyset$ (that is, these two sets are mutually exclusive), a process can have state from R_j^{in} *only if* some central transition (of the directory) adds it to the variable set_j. Recall that we assumed that a process is put on a list simultaneously with being sent a message.

Further, since R_j^{in} and R_j^{out} are mutually exclusive, i.e., $R_j^{out} \cap R_j^{in} = \emptyset$, a process with a state in R_j^{in} must belong to the set variable set_j. Thus, a process belongs to set_j if and only if its state is in R_j^{in}. □

[1] Jump goals of a remote action are simply the control locations appearing after the **goto**'s in the remote action.

Remark 8. Note that, for the optimization presented in the previous section, an over-approximation to the local states that set_j can point to is given by R_j^{in}.

In the following sections we will assume that all the set variables appearing in a protocol are redundant according to the criterion presented in this section.

3.6 Computing the Abstract Model

In this section we describe how to extract an overapproximation of the abstract model $\mathcal{P}^{\mathcal{A}}$ from the program text. The main challenge arises from the fact that there are infinite number of concrete systems to consider. To solve this problem, we consider each transition statement of the program separately and over-approximate the set of abstract transitions it can lead to. This over-approximation can be expressed by an invariant on the current state and next state variables. The disjunction of all these invariants is the abstract transition relation. To keep the presentation simple, we will assume that set variables have been removed using the redundancy criteria presented previously.

The abstract transition relation $R^{\mathcal{A}}$ is computed as a series of transition invariants between current abstract state $\hat{s}$ and the next abstract state $\hat{s}'$. We consider each transition statement t appearing in the protocol code and find out what abstract transitions it can lead out. The set of abstract transitions corresponding to a concrete transition statement is described by a transition invariant $I(t)$. The abstract transition relation $R^{\mathcal{A}}$ is then given by

$$\bigvee_t I(t)$$

We first consider the case where t is a local transition statement of a cache and later consider the more complicated case where t is a central transition statement.

3.6.1 Cache Transitions

Recall that caches can only make simple transitions t of the form

$$\mathsf{pc}_{\mathsf{L}} = L_1^{\mathsf{L}} : \textbf{ goto } \mathsf{pc}_{\mathsf{L}} = L_2^{\mathsf{L}}.$$

This transition can be made either (i) by the reference cache or (ii) by one of the environment caches.

We will now give conditions when to include an abstract transition from
$\hat{s} = \langle \mathbf{pc}_{\mathsf{L}}, e_1, \ldots, e_T, \mathbf{pc}_{\mathsf{C}}, \widehat{\mathsf{ptr}}_1, \ldots, \widehat{\mathsf{ptr}}_b \rangle$ to
$\hat{s}' = \langle \mathbf{pc}'_{\mathsf{L}}, e'_1, \ldots, e'_T, \mathbf{pc}'_{\mathsf{C}}, \widehat{\mathsf{ptr}}'_1, \ldots, \widehat{\mathsf{ptr}}'_b \rangle$ corresponding to the transition statement t.

Case (i): Transition by reference cache. The local transition t is executed by the reference case. In this case, we require that

$$\mathsf{pc}_{\mathsf{L}} = L_1^{\mathsf{L}} \ \wedge \ \mathsf{pc}_{\mathsf{L}}{}' = L_2^{\mathsf{L}} \tag{3.1}$$

and all other variables are the same in $\hat{s}$ and $\hat{s}'$. Note that no abstract pointers of the directory need to be changed because the abstract pointers have a special value ref for the reference cache.

Case (ii): Transition by cache in environment. The local transition t is executed by an environment cache. In this case, we have the obvious condition that there is a cache in

state $L_1^{\llcorner}$ before the transition, and also a cache in $L_2^{\llcorner}$ after the transition:

$$e_{L_1^{\llcorner}} = 1 \ \wedge \ e'_{L_2^{\llcorner}} = 1. \tag{3.2}$$

Moreover, we have to make sure that the pointers of the directory are changed in accordance with the transition. Let **if** ϕ **then** α **else** β denote the formula $(\phi \wedge \alpha) \vee (\neg\phi \wedge \beta)$. Then, looping over all pointer variables ptr_1 to ptr_b, we include the condition below, which we denote by $\Lambda(L_1, L_2)$

$$\bigwedge_{1 \le i \le b} \textbf{if } \widehat{\mathsf{ptr}}_i \neq L_1^{\llcorner} \textbf{ then } \widehat{\mathsf{ptr}}_i = \widehat{\mathsf{ptr}}'_i \textbf{ else } \{$$
$$\textbf{if } e'_{L_1^{\llcorner}} = 0 \textbf{ then } \widehat{\mathsf{ptr}}'_i = L_2^{\llcorner} \textbf{ else } \widehat{\mathsf{ptr}}'_i \in \{L_1^{\llcorner}, L_2^{\llcorner}\} \ \}.$$

Intuitively, $\Lambda(L_1^{\llcorner}, L_2^{\llcorner})$ expresses the following: if the pointer does not point at $L_1^{\llcorner}$, then it remains unchanged. Otherwise, one of two things can happen after the transition. First, if there is no cache left in location $L_1^{\llcorner}$ i.e., $e'_{L_1^{\llcorner}} = 0$, then the cache referred to by the pointer must have moved, and thus, the pointer has to be updated to point to $L_2^{\llcorner}$. Second, if a cache is left in location $L_1^{\llcorner}$, then it is not clear which cache moved, and we over-approximate. Again, all other variables are the same in $\hat{s}$ and $\hat{s}'$.

The abstract invariant $I(t)$ corresponding to the transition statement t is given by the disjunction of 3.1 and 3.2.

Lemma 3.6.1. *Let s, s' be two states of a concrete system $\mathcal{P}(K)$. Let there be a transition from s to s' with process c executing a local transition. Then the abstract states $\alpha_c(s)$ and $\alpha_c(s')$ satisfy the invariant described by $I(t)$.*

Proof. The proof of this lemma follows simply from the way we constructed $I(t)$. □

3.6.2 Directory Transitions

Consider now the case where the transition statement t is a directory transition. Recall that the directory transitions have the form

$$\mathsf{pc}_{\mathsf{C}} = L \wedge \Phi(\mathsf{ptr}, \mathsf{set}) : \textbf{ do actions } A_1, .., A_k.$$

Each directory transition t will be translated into a condition

$$\mathbf{pc}_{\mathsf{C}} = L \ \wedge \ \hat{\Phi} \ \wedge \ \mathcal{I}^{A_1} \wedge \ldots \mathcal{I}^{A_k} \wedge \mathcal{R}$$

where the $\mathcal{I}^{A_i}$ are the abstract conditions corresponding to the actions A_i, and $\mathcal{R}$ constrains all the abstract variables not appearing elsewhere to be the same in $\hat{s}$ and $\hat{s'}$.

We will first show how to translate each basic action A_i into a condition $\mathcal{I}^{A_i}$.

- For the *simple* action **goto** L^{C} we obtain the natural condition $\mathbf{pc}'_{\mathsf{C}} = L^{\mathsf{C}}$.
- For the *update* action **assign** $\mathsf{ptr}_1 = \mathsf{ptr}_2$ we obtain the condition $\widehat{\mathsf{ptr}}'_1 = \widehat{\mathsf{ptr}}_2$.
- For the *update* action **pick** ptr **from** $\mathcal{S}^{\mathsf{L}}$ we obtain the condition

$$(\mathbf{pc}_{\mathsf{L}} \in \mathcal{S}^{\mathsf{L}} \wedge \widehat{\mathsf{ptr}}' = \mathsf{ref}) \vee (e_j = 1 \wedge j \in \mathcal{S}^{\mathsf{L}} \wedge \widehat{\mathsf{ptr}}' = j)$$
$$\vee(\mathbf{pc}_{\mathsf{L}} \notin \mathcal{S}^{\mathsf{L}} \wedge \textstyle\bigwedge_{j \in \mathcal{S}^{\mathsf{L}}} e_j = 0 \wedge \widehat{\mathsf{ptr}}' = \widehat{\mathsf{ptr}}).$$

  Intuitively, if the reference process has a control location from the set $\mathcal{S}^{\mathsf{L}}$ then, in the new state, ptr can point to the reference process. Thus, we have the disjunct $\mathbf{pc}_{\mathsf{L}} \in \mathcal{S}^{\mathsf{L}} \wedge \widehat{\mathsf{ptr}}' = \mathsf{ref}$. Alternatively, some environment process might have a

control location from the set $\mathcal{S}^{\mathsf{L}}$ and the pointer variable can point to it in the next state. Thus, we have the disjunct $e_j = 1 \wedge j \in \mathcal{S}^{\mathsf{L}} \wedge \widehat{\mathsf{ptr}}' = j$. Lastly, it is possible that none of the caches have control locations from the set $\mathcal{S}^{\mathsf{L}}$. In this case, the value of the pointer variable does not change. Hence we have the disjunct $\mathbf{pc}_{\mathsf{L}} \notin \mathcal{S}^{\mathsf{L}} \wedge \bigwedge_{j \in \mathcal{S}^{\mathsf{L}}} e_j = 0 \wedge \widehat{\mathsf{ptr}}' = \widehat{\mathsf{ptr}}$.

- For the remote action **remote** ptr : **goto** $\mathsf{pc}_{\mathsf{L}} = L^{\mathsf{L}}$ we obtain the condition

$$(\widehat{\mathsf{ptr}} = \mathsf{ref} \wedge \mathbf{pc}'_{\mathsf{L}} = L^{\mathsf{L}}) \tag{3.3}$$

$$\bigvee$$

$$\bigvee_{1 \le L \le T} (\widehat{\mathsf{ptr}} = L \wedge \widehat{\mathsf{ptr}}' = L^{\mathsf{L}} \wedge e'_{L^{\mathsf{L}}} = 1 \wedge \Lambda_{\mathsf{ptr}}(L, L^{\mathsf{L}})) \tag{3.4}$$

where $\Lambda_{\mathsf{ptr}}(L, L^{\mathsf{L}})$ is defined as

$$\bigwedge_{1 \le i \le b,\ \mathsf{ptr}_i \neq \mathsf{ptr}} \textbf{if } \widehat{\mathsf{ptr}}_i \neq L \textbf{ then } \widehat{\mathsf{ptr}}_i = \widehat{\mathsf{ptr}}'_i \textbf{ else } \{$$
$$\textbf{if } e'_L = 0 \textbf{ then } \widehat{\mathsf{ptr}}'_i = L^{\mathsf{L}} \textbf{ else } \widehat{\mathsf{ptr}}'_i \in \{L, L^{\mathsf{L}}\} \ \}.$$

Note that $\Lambda_{\mathsf{ptr}}(L, L^{\mathsf{L}})$ is similar to $\Lambda(L, L^{\mathsf{L}})$ defined in Section 3.6.1 except that pointer ptr is left unchaned.

The explanation for this abstract transition is quite simple. If the pointer ptr, which is used in the **remote** action, points to the reference process, then the control location of the reference process is changed to L^{L}. Thus, we have the disjunct $(\widehat{\mathsf{ptr}} = \mathsf{ref} \wedge \mathbf{pc}'_{\mathsf{L}} = L^{\mathsf{L}})$ shown in Equation 3.3.

To understand the second disjunct, shown in Equation 3.4, consider the case where ptr points to an environment process. Suppose the environment process is in environment e_L, that is, $\widehat{\mathsf{ptr}} = L$. Then the following hold:

– In the next state, the pointer variable points to a process in environment $e_{L^\llcorner}$ because the new state of the cache pointed to by ptr is $L^\llcorner$. Thus, we have the condition $\widehat{\mathsf{ptr}}' = L^\llcorner$

– In the next state, the environment $e_{L^\llcorner}$ is non-empty, that is, $e'_{L^\llcorner} = 1$. The environment e_L could be 0 or 1 in the next state.

– Since a process moves from environment e_L to $e_{L^\llcorner}$ pointer variables other than ptr must be updated according to the condition $\Lambda_{\mathsf{ptr}}(L, L^\llcorner)$

Putting all the above together, we have the condition

$$(\widehat{\mathsf{ptr}} = L \wedge \widehat{\mathsf{ptr}}' = L^\llcorner \wedge e'_{L^\llcorner} = 1 \wedge \Lambda_{\mathsf{ptr}}(L, L^\llcorner)).$$

The case where the **remote** action is of the more general form with **map** involving set variables is similar to the case described above.

Remark 9. Since the set variables are redundant, **add** and **remove** actions are irrelevant for the construction of the abstract model.

Assuming that the set variables are redundant in the sense of Section 3.5, the abstraction $\hat{\Phi}$ of the condition $\Phi(\mathsf{set}, \mathsf{ptr})$ is obtained by abstracting each atomic subformula:

- $\mathsf{pc}_\mathsf{L}[\mathsf{ptr}_i] = L^\llcorner$ is abstracted into $(\widehat{\mathsf{ptr}}_i = \mathsf{ref} \wedge \mathbf{pc}_\mathsf{L} = L^\llcorner) \vee \widehat{\mathsf{ptr}}_i = L^\llcorner$.
- $\mathsf{ptr}_i \in \mathsf{set}_j$ is abstracted into $(\widehat{\mathsf{ptr}}_i = \mathsf{ref} \wedge \mathbf{pc}_\mathsf{L} \in R_j^{in}) \vee (\widehat{\mathsf{ptr}}_i \neq \mathsf{ref} \wedge \widehat{\mathsf{ptr}}_i \in R_j^{in})$.
- $\mathsf{set}_j = \emptyset$ is abstracted into $\mathbf{pc}_\mathsf{L} \notin R_j^{in} \wedge \bigwedge_{s \in R_j^{in}} e_s = 0$. In other words, no cache should be in a state from R_j^{in}; hence $\mathbf{pc}_\mathsf{L} \notin R_j^{in}$, and all counters corresponding to states in R_j^{in} must be 0.

Lemma 3.6.2. *Let s, s' be two states of a concrete system $\mathcal{P}(K)$. Let there be a transition from s to s' with the directory process executing the statement t. Then the abstract states $\alpha_c(s)$ and $\alpha_c(s')$ for any process $c \in [1..K]$ together satisfy the transition invariant $I(t)$.*

Proof. The proof of this lemma follows simply from the way we constructed $I(t)$. □

3.7 Experiments

GERMAN'S cache coherence protocol [44; 66] and FLASH cache coherence protocol [63] are the two most widely studied cache coherence protocols. We applied our abstraction technique to several versions, including the standard, correct version, of the GERMAN'S protocol and a simplified version of the FLASH protocol.

GERMAN's protocol, which operates in *lazy* mode, has two set variables *sharlist* and *invlist*. Whenever a cache enters a shared state it is added to *sharlist*. The variable *sharlist* is redundant according to the criterion in Section 3.5. The variable *invlist* is used to send invalidate messages to caches which are in a shared state. Initially, *invlist* is set equal to *sharlist*. When an invalidate message is sent to a cache in *invlist*, it is removed from *invlist*. While *invlist* is not redundant according to the criterion of Section 3.5, a simple change makes our criterion applicable: instead of initializing *invlist* by assigning *sharlist* to it, we can add a cache to *invlist* whenever it is added to *sharlist*. This simple change makes the set variable *invlist* redundant, too. All the different versions of the GERMAN'S protocol that we verified had this modification [2].

[2]Alternatively, we can also create a stronger redundancy criterion for set variables, which will ensure that $invlist$ is redundant without any modification. But, the modification we introduced is minor and does

In addition to verifying the standard correct version of GERMAN'S protocol, we also tried our method on buggy versions of GERMAN'S protocol including one supplied by Steve German [44]. These buggy versions of the standard GERMAN'S protocol, referred to as BUGGY 1 and BUGGY 2, are described in the last section of this chapter. In addition, we also applied our method to a variant of GERMAN'S protocol which has four channels, instead of the usual three channels. We will refer to this version as GERMAN 4-CHAN.

For the FLASH protocol, we eliminated the local pointer variables from the caches. These local pointers are used to handle the *three-hop* case where the directory forwards the id of the cache requesting exclusive access to the cache already holding that data item in an exclusive state. For the three-hop case, we exploit the fact that at any point, for a given data item, there can be only one three-hop transaction going on. Thus, to reduce the state space, instead of storing a pointer at each cache, we store one pointer in the central directory. Hence, we can model the three-hop transaction as a **remote** action of the directory without changing the semantics of the three-hop transaction. While the modification in this case are significant, the resultant protocol is still quite complicated and retains enough similarity to the original protocol to justify calling it a variation of the FLASH protocol.

The safety property considered for all the protocols was

$$\forall x.\mathbf{AG}\ (\mathsf{pc_L}[x] = \text{excl} \Rightarrow (\text{excl} \notin \mathsf{env}(x) \wedge \text{shar} \notin \mathsf{env}(x))$$

i.e., if cache x holds the data item exclusively ($\mathsf{pc_L}[x] = \text{excl}$) then no other cache can hold the data item in shared or exclusive state. The results of our experiments are described below.

not change the protocol much.

Standard German's protocol. We first applied our method to the standard, correct version of GERMAN's protocol. We did not use the optimization to eliminate the counters for unreachable local states. For this unoptimized version, Cadence SMV took about 3 hours (11400 seconds) to verify the safety property. We then applied the optimization to eliminate the counters corresponding to unreachable local states. Instead of writing a procedure to find the unreachable states, we supplied a list of unreachable local states to the abstraction program manually as it is easy to figure out manually what local states are unreachable. For instance, for GERMAN'S protocol, it is easy to see that if the outgoing channel $chan3$ is carrying an $invack$ message then the cache state must be $invalid$. While the list of states we supplied may be not exhaustive, it still gives significant reduction in the abstract state space. With this optimization, SMV takes about 5 minutes to complete the verification. This running time compares favorably with other verification efforts involving GERMAN'S protocol, see for instance [3].

Version Buggy 1. In the BUGGY 1 version, after the directory grants exclusive access to a cache, it fails to set the $grantexcl$ variable to true. Thus, when another cache requests shared access, it gets the access even though the first cache holds it in exclusive state. We applied our abstraction (without the optimization to eliminate counters corresponding to unreachable states) and applied Cadence SMV's Bounded Model Checker. BMC takes around 15 mins to find the bug at depth 12 (that is, the bug is reached after 12 transitions have been executed by the cache coherence system).

Version Buggy 2. In the BUGGY 2 version, the directory grants a shared request even if $grantexcl$ variable is true. As with the previous version, we constructed the abstract model without using the optimization to remove counters for unreachable states. BMC

again takes under 15 mins to find the bug at depth 12.

German 4-Chan. In this variant of GERMAN'S protocol, there are four channels instead of the usual three channels. In specific, instead of just one incoming channel, there are two incoming channels, $chan2$ and $chan4$, for every cache. In the original version, the single incoming channel carries all three types of messages: $grantshar$, $grantexcl$ and $invalid$. In the four channel version, one of the incoming channels, $chan2$ carries $grantshar$ and $grantexcl$ messages while the other one, $chan4$, carries $invalid$ message. Having two incoming channels leads to the following subtle bug: cache 1 requests a shared access, and while this is being processed, it sends out another request. The first request is honored and cache 1 gets shared access (while the other request for shared access is still pending). Now the central process reads the second request from cache 1, and sends it another $grantshar$ message on $chan2$.

Immediately after this, another cache, say cache 2, requests exclusive access. Before granting exclusive access to cache 2, the central process sends out an invalidate message to all caches with shared access, including cache 1 on the second incoming channel $chan4$. Cache 1 reads the $invalid$ message on $chan4$ (while $chan2$ still has the $grantshar$ message) and transitions to $invalid$ state and sends an acknowledgement to the central process (on $chan3$). Once the central process sees all the acknowledgements, it grants exclusive access to cache 2. But, the $grantshar$ message is still present in $chan2$ of cache 1 and this leads cache 1 to transition to a shared state. Thus, cache 1 ends with shared access while cache 2 still has exclusive access.

We applied our abstraction method (with both the optimizations described in Sec-

tion 3.5) and used a BDD based model checker to find the bug. It took SMV 7 mins to find the bug at depth 15. BMC runs out of memory at depth 15. Note that BMC takes less than 15 mins for the two buggy versions because the counter example depth is only 12. For these buggy versions, BDD based model checker does not finish even after an hour (the abstract models for buggy versions were not optimized).

Flash protocol. We constructed an abstract model for FLASH protocol using both the optimizations described in Section 3.5. With *counter threshold* 1 (cf. Section 3.4.2), we get a spurious counterexample due to the three-hop case. The spurious counter example is as follows: suppose counter e_{excl} corresponds to an exclusive local state. Suppose now that the reference process requests exclusive access. The central process forwards this request to the environment process which is represented by the counter e_{excl}. After serving the request the environment process goes into an invalid state, and thus e_{excl} should become 0. But, since 1 stands for *many* in the abstract model, there is an abstract transtion that keeps e_{excl} as 1. This leads to the violation of the safety property.

To get rid of this spurious counterexample, we track counters corresponding to exclusive local states more carefully. We refine the abstract model by increasing the counter threshold to 2 for those environments where the cache is in the *exclusive* state. The resulting model is precise enough to prove the safety property. The model checking time was about 7 hours (25700 seconds).

The table shown in Figure 3.1 summarizes our experimental results.

Remark 10. For the protocols that do not satisfy the cache coherence property, the counterexamples always involve just two caches. For example, the version of GERMAN'S

Protocol	Optimizations	MC	Cex	Time
German(std)	1	BDD	No	3 hrs
German(std)	1, 2	BDD	No	5 mins
German(Buggy 1)	1	BMC	Yes (len=12)	15 mins
German(Buggy 2)	1	BMC	Yes (len=12)	15 mins
German(4-Chan)	1, 2	BDD	Yes (len=15)	7 mins
Flash	1, 2	BDD	No	7 hrs

Figure 3.1: Results for Cache Coherence Protocols

protocol with four channels has a bug involving only two caches. It seems to be the case that having just 3 caches might exhaust all the possibilities for a cache coherence protocol but this is hard to prove.

All the experiments were run on a 1.5 GHz machine with 3GB main memory. Since the time for extracting the abstract model is negligible compared to the model checking time, the reported times are runtimes of the model checker.

3.8 Conclusion

We have presented a natural application of environment abstraction that allows us to automatically verify complex cache coherence protocols. We first describe a high level description language to model such protocols. Our language is natural and facilitates easy protocol descriptions in the spirit of Lamport's TLA (although it is much more restricted).

In contrast to previous approaches [67], we use symbolic model checkers in this chapter.

To keep the computation feasible, we over-approximate the abstract transition system one statement at a time, similar to predicate abstraction for software. Moreover, we use the results of Section 3.5 to eliminate semantically redundant set variables, and thus to reduce the size of the abstract models.

In the next section, we present the descriptions of the protocols that we verified. In the next chapter we will consider the application of environment abstraction to mutual exclusion protocols.

3.9 Protocol Descriptions

A simplified version of the FLASH cache coherence protocol is shown in our input language below. The simplifications are as follows:

- In the original FLASH protocol, the directory (i.e. the central process) distinguishes between the home cache and the other caches. While our abstraction method can handle the full version, the current model checkers cannot handle the abstract model that is generated.

- The communication between the central process (directory) and local processes (the caches) is modelled by having two variables *chanin* and *chanout* per cache. These two variables serve as incoming and outgoing channels for each cache. Use of two variables implies that the communication buffers are bounded, in fact, are of size 1. This restriction is similar to the restriction seen in GERMAN's Protocol.

- For the three-hop case, we exploit the fact that at any point in time, for a given cache line, there can be only one three-hop transaction. This fact can be seen just by examining the code for the central process. So to reduce state space, instead of storing a pointer at each cache, we store one pointer at the central process (named *threeptr* in the model below). Since there is only one three-hop transaction and all the information on the caches involved is known, we model the three-hop transaction as part of the central process. This does not change the semantics of the three-hop transaction in any way, it is just a convenient representation in our modelling language.

The central process reads a message from a cache via a transition involving **pick** action. For example, the transition

$currcmd = empty \wedge read = no$: **Do Actions**
goto $currcmd = get \wedge read = yes$
pick temptr **from** $chanout = get$

says, if current command (CURRCMD) is empty, and nothing has been read (READ = no), then do the action **pick** temptr **from** $chanout = get$. This action sets the pointer *temptr* to some cache satisfying the condition $chanout = get$, that is, some cache which has sent a get message. Then the current command is set to get and READ is marked yes.

The expression $sharlist = \Phi$ indicates that the set $sharlist$ is empty. Finally, the statement **remote** $sharlist$ **goto** $chanin = inv$ denotes the action where all caches present in $sharlist$ get an invalidate (inv) message.

FLASH PROTOCOL

Local Process

Local Vars

CACHESTATE: inv, shar, excl;

INVMARKED: yes, no;

CHANIN: empty, put, putx, inv, NAK;

CHANOUT: empty, get, getx, invack;

Local Transitions

$cachestate = inv \wedge chanout = emtpy$: **goto** $chanout = get$

$cachestate = inv \wedge chanout = emtpy$: **goto** $chanout = getx$

$cachestate = shar \wedge chanout = emtpy$: **goto** $chanout = getx$

$cachestate = inv \wedge chanin = inv$: **goto** $invmarked = yes$

$cachestate = inv \wedge invmarked = no \wedge chanin = put$: **goto** $cachestate = shar \wedge chanin = empty$

$cachestate = inv \wedge invmarked = yes \wedge chanin = put$: **goto** $invmarked = no \wedge chanin = empty$

$cachestate = inv \wedge invmarked = no \wedge chanin = putx$: **goto** $cachestate = excl \wedge chanin = empty$

$cachestate = inv \wedge invmarked = yes \wedge chanin = putx$: **goto** $invmarked = no \wedge chanin = empty$

$cachestate = shar \wedge chanin = inv$: **goto** $cachestate = inv \wedge chanout = invack$

$cachestate = excl \wedge chanin = inv$: **goto** $cachestate = inv \wedge chanout = invack$

$cachestate = inv \wedge chanin = NAK$: **goto** $chanin = empty$

$cachestate = shar \wedge chanin = NAK$: **goto** $chanin = empty$

Central Process

Central vars

DIRTY: no, yes;

PENDING: no, yes;

HDPTR: ptr;

HDVALID: no, yes;

CURRCMD: empty, get, getx, invack;

THREEHOP: empty, get, get1, getx, getx1;

THREEHOPTR: ptr;

CHECKSHRLIST: no, yes;

SHARLIST: set;

TEMPTR: ptr;

Central Transitions

$currcmd = empty \wedge read = no$: **Do Actions**

goto $currcmd = get \wedge read = yes$

pick $temptr$ **from** $chanout = get$

$currcmd = empty \wedge read = no$: **Do Actions**

goto $currcmd = getx \wedge read = yes$

pick $temptr$ **from** $chanout = getx$

$currcmd = empty \wedge read = no$: **Do Actions**

goto $currcmd = invack \wedge read = yes$

pick $temptr$ **from** $chanout = invack$

$currcmd = empty \wedge read = yes$: **Do Actions**

goto $read = no$

remote $temptr$ **goto** $chanout = empty$

$currcmd = get \wedge read = no \wedge pending = no \wedge dirty = no \wedge chanout[temptr] = empty$:

Do Actions

goto $currcmd = empty$

remote $temptr$ **goto** $chanin = put$

$currcmd = get \wedge read = no \wedge pending = no \wedge dirty = yes \wedge chanout[temptr] = empty$:

Do Actions

goto $currcmd = empty \wedge threehop = get \wedge pending = yes$

assign $threehoptr = temptr$

$threehop = get \wedge cachestate[hdptr] = excl$: **Do Actions**

goto $threehop = get1$

remote $hdptr$ **goto** $cachestate = inv$

$currcmd = get \wedge read = no \wedge pending = yes \wedge chanout[temptr] = empty$: **Do Actions**

goto $currcmd = empty$

remote $temptr$ **goto** $chanin = NAK$

$currcmd = getx \wedge read = no \wedge pending = yes \wedge chanout[temptr] = empty$: **Do Actions**

goto $currcmd = empty$

remote $temptr$ **goto** $chanin = NAK$

$currcmd = getx \wedge read = no \wedge pending = no \wedge dirty = no \wedge hdvalid = no \wedge chanout[temptr] = empty$: **Do Actions**

goto $currcmd = empty$

remote $temptr$ **goto** $chanin = putx$

$currcmd = getx \wedge read = no \wedge pending = no \wedge dirty = yes \wedge chanout[temptr] = empty$: **Do Actions**

goto $currcmd = empty \wedge threehop = getx \wedge pending = yes$

assign $threehoptr = temptr$

$currcmd = getx \wedge read = no \wedge pending = no \wedge dirty = no \wedge hdvalid = yes \wedge chanout[temptr] = empty$: **Do Actions**

goto $currcmd = empty \wedge pending = yes$

remote $sharlist$ **goto** $chanin = inv$

remote $temptr$ **goto** $chanin = putx$

$threehop = getx \wedge cachestate[hdptr] = excl$: **Do Actions**

goto $threehop = getx1$

remote $hdptr$ **goto** $cachestate = inv$

$currcmd = invack \wedge read = no$: **Do Actions**

goto $currcmd = empty \wedge checksharlist = yes$

$checksharlist = yes \wedge sharlist = \Phi$: **Do Actions**

goto $chechsharlist = no \wedge pending = no$

STANDARD GERMAN'S PROTOCOL

Local Process

Local Vars

Cachestate: {invalid, shar, excl}

chan1: {Empty, reqshar, reqexcl}

chan2: {Empty, invalid, grantshar, grantexcl}

chan3: {Empty, Invack}

Local Transitions

$cachestate = invalid \wedge chan1 = empty$: **goto** $chan1 = reqshar$;

$cachestate = invalid \wedge chan1 = empty$: **goto** $chan1 = reqexcl$;

$cachestate = shar \wedge chan1 = empty$: **goto** $chan1 = reqexcl$;

$chan2 = invalid \wedge chan3 = empty$: **goto** $chan2 = empty \wedge chan3 = invack \wedge cachestate = invalid$;

$chan2 = grantshar$: **goto** $chan2 = empty \wedge cachestate = shar$;

$chan2 = grantexcl$: **goto** $chan2 = empty \wedge cachestate = excl$;

Central Process

Central Vars

exclgrant: {yes, no}

currcmd: {empty, reqshar, reqexcl}

currclient: ptr

sharlist: set

Invlist: set

read: {yes, no}

tmpread1: {no, yes}

temptr2: ptr

tmpread2: {no, yes}

temptr1: ptr

Central Transitions

$currcmd = empty \wedge read = no$: **Do Actions**

goto $read = yes$

pick $currclient$ **from** $\{local|$ *chan1*[$local$]=*reqshar* $\vee$ *chan1*[$local$]=*reqexcl*$\}$

$currcmd = empty \wedge read = yes \wedge chan1[currclient] = reqshar$: **Do Actions**

goto $read = no \wedge currcmd = reqshar$

remote $currclient$ **goto** $chan1 = Empty$

$currcmd = empty \wedge read = yes \wedge chan1[currclient] = reqexcl$: **Do Actions**

goto $read = no \wedge currcmd = reqexcl$

remote $currclient$ **goto** $chan1 = Empty$

assign $Invlist = sharlist$

$currcmd = reqshar \wedge grantexcl = no \wedge chan2[currclient] = empty$: **Do Actions**

goto $currcmd = empty$

remote $currclient$ **goto** $chan2 = grantshar$

Add $currclient$ **to** $sharlist$

$currcmd = reqexcl \wedge sharlist = \Phi \wedge chan2[currclient] = empty$: **Do Actions**

goto $currcmd = empty \wedge grantexcl = yes$

remote $currclient$ **goto** $chan2 = grantexcl$

Add $currclient$ **to** $sharlist$

$currcmd = reqshar \wedge tmpread1 = no \wedge grantexcl = yes$: **Do Actions**

goto $tmpread1 = yes$

pick $temptr1$ **from** $\{local | (local \in Invlist) \wedge chan2[local] = Empty\}$

$currcmd = reqexcl \wedge tmpread1 = no$: **Do Actions**

goto $tmpread1 = yes$

pick $temptr1$ **from** $\{local | (local \in Invlist) \wedge chan2[local] = Empty\}$

$currcmd = reqshar \wedge tmpread1 = yes$: **Do Actions**

goto $tmpread = no$

remote $temptr1$ **goto** $chan2 = invalid$

Remove $temptr1$ **from** $Invlist$

$currcmd = reqexcl \wedge tmpread1 = no$: **Do Actions**

goto $tmpread1 = yes$

remote $temptr1$ **goto** $chan2 = invalid$

Remove $temptr1$ **from** $Invlist$

$currcmd = reqshar \wedge tmpread2 = no \wedge grantexcl = yes$: **Do Actions**

goto $tmpread2 = yes$

pick $temptr2$ **from** $\{local | chan3[local] = invack\}$

$currcmd = reqexcl \wedge tmpread2 = no$: **Do Actions**

goto $tmpread1 = yes$

pick $temptr2$ **from** $\{local | chan3[local] = invack\}$

$currcmd = reqshar \wedge tmpread2 = yes$: **Do Actions**

goto $tmpread2 = no \wedge grantexcl = no$

remote $temptr2$ **goto** $chan3 = Empty$

$currcmd = reqexcl \wedge tmpread2 = yes$: **Do Actions**

goto $tmpread2 = no \wedge grantexcl = no$

remote $temptr2$ **goto** $chan2 = invalid$

Remove $temptr2$ **from** $sharlist$

BUGGY VERSIONS OF GERMAN'S PROTOCOL

As a sanity check, we created two buggy versions of GERMAN'S protocol to see if our method is able to catch the bugs. The buggy versions are described below.

Buggy version 1. In the first buggy version, after the directory grants exclusive access to a cache, it fails to set the grantexcl variable to true. That is, instead of the correct transition

$currcmd = reqexcl \wedge sharlist = \Phi \wedge chan2[currclient] = empty$: **Do Actions**

goto $currcmd = empty \wedge grantexcl = yes$

remote $currclient$ **goto** $chan2 = grantexcl$

Add $currclient$ **to** $sharlist$

we have the faulty version

$currcmd = reqexcl \wedge sharlist = \Phi \wedge chan2[currclient] = empty$: **Do Actions**

goto $currcmd = empty$

remote $currclient$ **goto** $chan2 = grantexcl$

Add $currclient$ **to** $sharlist$

Buggy version 2. For the second buggy version, the directory grants a shared request even if grantexcl variable is true (that is, some cache has been granted exclusive access). Thus, instead of the normal transition

$currcmd = reqshar \wedge grantexcl = no \wedge chan2[currclient] = empty$: **Do Actions**

goto $currcmd = empty$

remote $currclient$ **goto** $chan2 = grantshar$

Add $currclient$ **to** $sharlist$

we have

$currcmd = reqshar \wedge grantexcl = yes \wedge chan2[currclient] = empty$: **Do Actions**

goto $currcmd = empty$

remote $currclient$ **goto** $chan2 = grantshar$

Add $currclient$ **to** $sharlist$

GERMAN'S PROTOCOL WITH EXTRA CHANNELS (4-CHAN)

Local Process

Local Vars

Cachestate: {invalid, shar, excl}

chan1: {Empty, reqshar, reqexcl}

chan2: {Empty, grantshar, grantexcl}

chan3: {Empty, Invack}

chan4: {Empty, invalid }

Local Transitions

$cachestate = invalid \wedge chan1 = empty$: **goto** $chan1 = reqshar$;

$cachestate = invalid \wedge chan1 = empty$: **goto** $chan1 = reqexcl$;

$cachestate = shar \wedge chan1 = empty$: **goto** $chan1 = reqexcl$;

$chan4 = invalid \wedge chan3 = empty$: **goto** $chan2 = empty \wedge chan3 = invack \wedge cachestate = invalid$

$chan2 = grantshar$: **goto** $chan2 = empty \wedge cachestate = shar$

$chan2 = grantexcl$: **goto** $chan2 = empty \wedge cachestate = excl$

Central Process

Central Vars

exclgrant: {yes, no}

currcmd: {empty, reqshar, reqexcl}

currclient: ptr

sharlist: set

Invlist: set

read: {yes, no}

tmpread1: {no, yes}

temptr2: ptr

tmpread2: {no, yes}

temptr1: ptr

Central Transitions

$currcmd = empty \wedge read = no$: **Do Actions**

goto $read = yes$

pick $currclient$ **from** $\{local|$ *chan1*[$local$]=*reqshar* $\vee$ *chan1*[$local$]=*reqexcl*$\}$

$currcmd = empty \wedge read = yes \wedge chan1[currclient] = reqshar$: **Do Actions**

goto $read = no \wedge currcmd = reqshar$

remote $currclient$ **goto** $chan1 = Empty$

$currcmd = empty \wedge read = yes \wedge chan1[currclient] = reqexcl$: **Do Actions**

goto $read = no \wedge currcmd = reqexcl$

remote $currclient$ **goto** $chan1 = Empty$

Assign $Invlist = sharlist$

$currcmd = reqshar \wedge grantexcl = no \wedge chan2[currclient] = empty$: **Do Actions**

goto $currcmd = empty$

remote $currclient$ **goto** $chan2 = grantshar$

Add $currclient$ **to** $sharlist$

$currcmd = reqexcl \wedge sharlist = \Phi \wedge chan2[currclient] = empty$: **Do Actions**

goto $currcmd = empty \wedge grantexcl = yes$

remote $currclient$ **goto** $chan2 = grantexcl$

Add $currclient$ **to** $sharlist$

$currcmd = reqshar \wedge tmpread1 = no \wedge grantexcl = yes$: **Do Actions**

goto $tmpread1 = yes$

pick $temptr1$ **from** $\{local | Invlist[local] = in \wedge chan2[local] = Empty\}$

$currcmd = reqexcl \wedge tmpread1 = no$: **Do Actions**

goto $tmpread1 = yes$

pick $temptr1$ **from** $\{local | Invlist[local] = in \wedge chan2[local] = Empty\}$

$currcmd = reqshar \wedge tmpread1 = yes$: **Do Actions**

goto $tmpread = no$

remote $temptr1$ **goto** $chan4 = invalid$

Remove $temptr1$ **from** $Invlist$

$currcmd = reqexcl \wedge tmpread1 = no$: **Do Actions**

goto $tmpread1 = yes$

remote $temptr1$ **goto** $chan2 = invalid$

Remove $temptr1$ **from** $Invlist$

$currcmd = reqshar \wedge tmpread2 = no \wedge grantexcl = yes$: **Do Actions**

goto $tmpread2 = yes$

pick $temptr2$ **from** $\{local | chan3[local] = invack\}$

$currcmd = reqexcl \wedge tmpread2 = no$: **Do Actions**

goto $tmpread1 = yes$

pick $temptr2$ **from** $\{local | chan3[local] = invack\}$

$currcmd = reqshar \wedge tmpread2 = yes$: **Do Actions**

goto $tmpread2 = no \wedge grantexcl = no$

remote $temptr2$ **goto** $chan3 = Empty$

$currcmd = reqexcl \wedge tmpread2 = yes$: **Do Actions**

goto $tmpread2 = no \wedge grantexcl = no$

remote $temptr2$ **goto** $chan2 = invalid$

Remove $temptr2$ **from** $sharlist$

Chapter 4

Environment Abstraction for Verification of Mutex Protocols

4.1 Introduction

Given a set of contending processes, providing them mutually exclusive access to resources is among the most basic primitives that any computer system requires. As such, mutual exclusion protocols have received considerable attention in the distributed computing literature. These protocols are usually designed to be correct no matter what the exact number of processes running them. Thus, mutual exclusion protocols are classic examples of parameterized systems. Note that, in contrast to cache coherence protocols, in mutual exclusion protocols each individual process itself might have infinite state space as they

can have unbounded *data* variables in addition to finite control variables.

Several model checking based methods, including Indexed Predicates [52], Invisible Invariants [64], and counter abstraction [66], have been proposed to parameterically verify mutual exclusion protocols. The Indexed Predicates method [52; 53], as already mentioned, is a new form of predicate abstraction for infinite state systems. This method works only for safety properties and not for liveness properties.

The idea behind Invisible Invariants technique, introduced in a series of papers [3; 41; 42; 64], is to find an invariant for the parameterized system by examining concrete systems for low valuations of the parameter(s). In [3], a modified version of the Bakery algorithm is verified – the original Bakery algorithm is modified to eliminate unbounded data variables.

Pnueli et al. [66], who coined the term *counter abstraction*, show how systems composed of symmetric and finite state processes can be handled automatically. However, protocols that either break symmetry by exploiting knowledge of process ids or that have infinite state spaces require manual intervention. Thus, the verification of Szymanski's and the Bakery protocol in [66] requires manual introduction of new variables. All the three methods mentioned above make use of the atomicity assumption.

In this chapter, we will show how environment abstraction can be used to verify mutual exclusion protocols automatically under the atomicity assumption. Environment abstraction essentially addresses the two disadvantages of counter abstraction by generalizing the idea of counting: *since the state space is infinite, we do not count the processes in a given state as in traditional counter abstraction, but instead we count the number of processes*

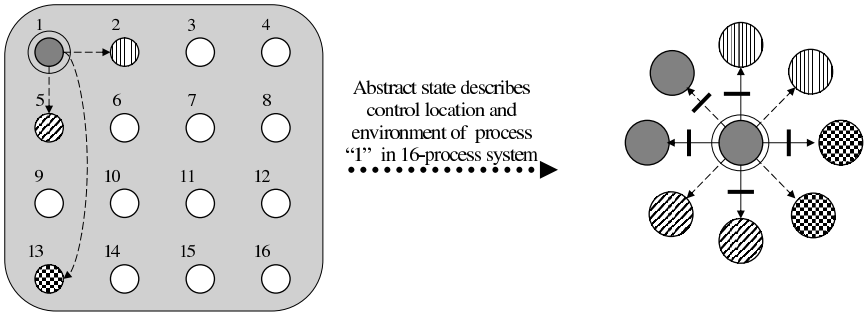


Figure 4.1: Abstraction Mapping.

satisfying a given predicate.

Figure 4.1 visualizes the intuition underlying environment abstraction. The grey box on the left hand side represents a concrete state of a system with 16 concurrent processes. The different colors of the disks/processes represent the internal states of the processes, i.e., the state of the control variables.

The star-shaped graph on the right hand side of Figure 4.1 represents an abstract state. The abstract state contains one distinguished process, the *reference process* x, which is at the center of the star. In this example, the reference process x represents process 1 of the concrete state. The disks on the circumference of the star represent the *environment* of the

reference process. *Intuitively, the goal of the abstraction is to embed the reference process x of the abstract state into an abstract environment as rich as the environment that process 1 has in the concrete state.* Thus, the abstract state represents the concrete state "from the point of view of process 1."

To describe the environment of a process, we need to consider the relationships which can hold between the data variables of two processes. We can graphically indicate a specific relationship between any two processes by a corresponding arrow between the processes; the form of the arrow (full, dashed, etc.) determines *which* relationship the two processes have. In Figure 4.1, we assume that we have only two relationships R_1 and R_2. For example, $R_1(x, y)$ might say that the local variable t of process x has the same value as local variable t in process y, while $R_2(x, y)$ might say that t has different values in processes x and y. Relationship R_1 is indicated by a full arrow, and R_2 is indicated by a dashed arrow. For better readability, not all relationships between the 16 processes are drawn.

Note that a single abstract state generally represents an infinite number of concrete states. Moreover, a given concrete state gives rise to several abstract states, each of which is induced by choosing a different possible reference process. For example, the concrete state in Figure 4.1 may induce up to 16 abstract states, one for each process.

Using the abstraction method described here, we have been able to verify automatically the safety and liveness properties of two well known mutual exclusion algorithms, namely Lamport's Bakery algorithm [54] and Szymanski's algorithm [77]. While safety and liveness properties of Szymanski's algorithm have been automatically verified with

atomicity assumption by Baukus et al. [6], this is the first time both safety and liveness of Lamport's Bakery algorithm have been verified (with the atomicity assumption) at this level of automation.

4.2 System Model for Mutual Exclusion Protocols

As in Section 2.2, we consider parameterized system $\mathcal{P}(K)$, $K > 1$, composed of (parameter) K processes. Unlike the system model for cache coherence protocols, there is no central process in the systems considered in this chapter. Technically, $\mathcal{P}(K)$ is a Kripke structure $\langle S_K, I_K, L_K, R_K \rangle$. The set of global states S_K and the global transition relation R_K are formed by composing the individual states and the transition relations of the K processes. Since mutual exclusion protocols are asynchronous system, the global transition relation R_K is the asynchronous composition of the individual the transition relations. In the following we will describe the state spaces and transition relations of the individual processes.

4.2.1 Local State Variables

Each process has two sets of variables: the control variables and the data variables. Intuitively, the two sets of variables serve different purposes. The control variables determine the internal control state of the process. Without loss of generality, we can assume that there is only one control variable pc per process. The set of data variables, $\mathbf{U} \doteq \{u_1, \ldots u_d\}$, contain actual data which can be read by other processes to calculate

their own data variables. We could also assume that there is only one data variable per process, but computation of the abstract model in presence of multiple data variables is different from the single data variable case. Hence, we consider the full general model.

We will usually refer to processes and their variables via their process ids. In particular, $\mathsf{pc}[i]$ and $u_k[i]$ denote the variables pc and u_k of the process with id i. A process can use the reserved expression slf to refer to its own process id. When a protocol text contains the variables pc or u_k without explicit reference to a process id, then this stands for $\mathsf{pc}[\mathsf{slf}]$ and $u_k[\mathsf{slf}]$ respectively. Note that all processes in a system $\mathcal{P}(K)$ are identical except for their ids. Thus, the process ids are the only means to break the symmetry between the processes.

A formula of the form $\mathsf{pc} = \mathrm{const}$ is called a *control assignment*. The range of pc is called the set of control locations.

Though we assume that there is only one control variable, in program texts we may take the freedom to use more than one finite range control variable as it makes the program better readable.

4.2.2 Transition Constructs

We will describe the transition relation of the processes in terms of two basic constructs, *guarded transitions* for the finite control, and the more complicated *update transitions* for modifying the data variables. A guarded transition has the form

$$\mathsf{pc} = L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } \mathsf{pc} = L_2 \textbf{ else goto } \mathsf{pc} = L_3$$

or shorter

$$L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2 \textbf{ else goto } L_3$$

where L_1, L_2, L_3 are control locations. In the guard $\forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ the variable otr ranges over the process ids of all other processes. The condition $\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ can be any formula involving the data variables of processes slf, otr and the pc variable of otr. The semantics of a guarded transition is straightforward: in control location L_1, the process evaluates the guard and changes to control location L_2 or L_3 accordingly.

Update transitions are needed to describe protocols such as the Bakery algorithm where a process *computes* a data value depending on all values that it can read from other processes. For example, the Bakery algorithm has to compute the maximum of a certain data variable (the "ticket variable") in all other processes. Thus, we define an update transition to have the general form

$$\begin{aligned} L_1 : \quad & \textbf{for all } \mathsf{otr} \neq \mathsf{slf} \quad \textbf{if } \mathcal{T}(\mathsf{slf}, \mathsf{otr}) \textbf{ then } u_k := \Phi(\mathsf{otr}) \\ & \textbf{goto } L_2 \end{aligned}$$

where L_1 and L_2 are control assignments, and $\mathcal{T}(\mathsf{slf}, \mathsf{otr})$ is a condition involving data variables of processes slf, otr. The semantics of the update transition is best understood in an operational manner: in control location L_1, the process scans over all the other processes (in a nondeterministically chosen order), and for each process otr, checks if the formula $\mathcal{T}(\mathsf{slf}, \mathsf{otr})$ is true. In this case, the process changes the value of its data variable

u_k according to $u_k := \Phi(\mathsf{otr})$, where $\Phi(\mathsf{otr})$ is an expression involving variables of process otr. Thus, the variable u_k can be reassigned multiple times within a transition. Finally, the process changes to control location L_2. *We assume that both guarded and update transitions are atomic, i.e., during their execution no other process makes a move.*

Example 4.2.1. As an example of a protocol written in this language, consider a parameterized system $\mathcal{P}(\mathbf{N})$ where each process P has one finite variable $\mathsf{pc} : \{1, 2, 3\}$ representing a program counter, one unbounded/integer variable $t : \mathbf{Int}$, and executes the following program:

$$
\begin{aligned}
1: &\quad \mathbf{goto}\ 2\\
2: &\quad \mathbf{if}\ \forall \mathsf{otr} \neq \mathsf{slf}.t[\mathsf{slf}] \neq t[\mathsf{otr}]\ \mathbf{then\ goto}\ 3\\
3: &\quad t := t[\mathsf{otr}] + 1; \quad \mathbf{goto}\ 1
\end{aligned}
$$

The statement $1: \ \mathbf{goto}\ 2$ is syntactic sugar for

$$\mathsf{pc} = 1: \ \mathbf{if}\ \forall \mathsf{otr} \neq \mathsf{slf}.\mathbf{true\ then\ goto}\ \mathsf{pc} = 2\ \mathbf{else\ goto}\ \mathsf{pc} = 1$$

Similarly, $3: \ t := t[\mathsf{otr}] + 1; \quad \mathbf{goto}\ = 1$ is syntactic sugar for

$$\mathsf{pc} = 3: \ \mathbf{if}\ \forall \mathsf{otr} \neq \mathsf{slf}.\mathbf{true\ then}\ t := t[\mathsf{otr}] + 1\ \mathbf{goto}\ \mathsf{pc} = 1.$$

This example illustrates that most commonly occurring transition statements in protocols can be written in our input language. □

Note that we have not specified the operations and predicates that are used in the conditions and assignments. Essentially, this choice depends on the protocols and the power

of the decision procedures used. For the protocols considered in this paper, we need linear order and equality on data variables as well as incrementation, i.e., addition by one. The last section of this chapter contains Szymanski's protocol and the Bakery protocol described in our input language.

4.3 Environment Abstraction for Mutual Exclusion Protocols

In this section, we show how to apply environment abstraction for mutual exclusion protocols. In Section 4.5, we will discuss how to actually compute abstract models.

To apply environment abstraction, we have to give the abstract descriptions and the abstraction mapping from the concrete states to the abstract states. We also have to prove that the abstract descriptions satisfy the coverage and congruence properties with respect to the set of labels we use. We consider these issues below.

4.3.1 Specifications and Labels

The typical properties that we are interested in verifying can be expressed as shown below.

- A single process liveness property can be written as

$$\forall x.\mathbf{AG}(\mathsf{pc}[x] = \mathbf{try} \Rightarrow \mathbf{F}(\mathsf{pc}[x] = \mathbf{crit}))$$

"For all processes x, the following holds: If process x is trying to enter the critical section then it eventually will."

- $$\forall x.\mathbf{AG}(\mathsf{pc}[x] = \mathbf{crit} \Rightarrow (\mathbf{crit} \notin \mathsf{env}(x)))$$

  "For all processes x the following invariant holds: If process x is in the critical section, then no other process is in the critial section"

Consequently, the set of labels L again has two types of labels:

- $\mathsf{pc}[x] = L$, and
- $L \in \mathsf{env}(x)$.

4.3.2 Abstract Descriptions

Technically, our descriptions reuse the predicates which occur in the control statements of the protocol description. Let S_L be the number of control locations in the program P. The internal state of a process x can be described by a predicate of the form

$$\mathsf{pc}[x] = L$$

where $L \in \{1..S_L\}$ is a control location.

In order to describe the relations between the data variables of different processes we collect *all* predicates $\mathcal{EP}_1(x, y), \ldots, \mathcal{EP}_r(x, y)$ which occur in the guards of the program. From now on we will refer to these predicates as the *inter-predicates* of the program. Since in most practical protocols, synchronization between processes involves only one or two data variables, the number of inter-predicates is usually quite small. The relationship

between a process x and a process y is now described by a formula of the form

$$R_i(x,y) \doteq \pm\mathcal{EP}_1(x,y) \wedge \ldots \wedge \pm\mathcal{EP}_r(x,y)$$

where $\pm\mathcal{EP}_i$ stands for $\mathcal{EP}_i$ or its negation $\neg\mathcal{EP}_i$. It is easy to see that there are 2^r possible relationships $R_1(x,y), \ldots, R_{2^r}(x,y)$ between x and y. In the example of Figure 4.1, the two relationship predicates R_1 and R_2 are visualized by full and dashed arrows.

Fact 3. The relationship conditions $R_1(x,y), \ldots, R_{2^r}(x,y)$ are mutually exclusive.

Before we explain the descriptions $\Delta(x)$ in detail, let us first describe the most important building blocks for the descriptions, which we call *environment predicates*. An environment predicate expresses that for process x we can find another process y which has a given relationship to process x and a certain internal state. The environment predicates thus have the form

$$\exists y.y \neq x \wedge R_i(x,y) \wedge \mathsf{pc}[y] = j.$$

An environment predicate says the following: *there exists a process y different from x whose relationship to x is described by the $\mathcal{EP}$ predicates in R_i and whose internal state is j.* There are $T := 2^r \times S_L$ different environment predicates; we name them $\mathcal{E}_1(x), \ldots, \mathcal{E}_T(x)$, and their quantifier-free matrices $E_1(x,y), \ldots, E_T(x,y)$. Note that each $E_k(x,y)$ has the form $y \neq x \wedge R(x,y) \wedge \mathsf{pc}[y] = L$.

Fact 4. If an environment process y satisfies an environment condition $E_i(x,y)$, then it cannot simultaneously satisfy any other environment condition $E_j(x,y)$, $i \neq j$.

Proof. Each environment condition $E_k(x, y)$ has the form $y \neq x \wedge R(x, y) \wedge \mathsf{pc}[y] = L$. Thus let

$$E_i(x, y) \doteq y \neq x \wedge R_i(x, y) \wedge \mathsf{pc}[y] = L_i$$

and

$$E_j(x, y) \doteq y \neq x \wedge R_j(x, y) \wedge \mathsf{pc}[y] = L_j$$

Since $E_i(x, y)$ and $E_j(x, y)$ are different, either $R_i(x, y)$ is different from $R_j(x, y)$ or $L_i \neq L_j$.

In the former case, by Fact 3, $R_i(x, y)$ and $R_j(x, y)$ are mutually exclusive. Thus, if process y satisfies $E_i(x, y)$ then it cannot satisfy $E_j(x, y)$.

In the latter case, if process y satisfies $E_i(x, y)$ then the control location of process y is L_i. Since $L_i \neq L_j$, process y cannot satisfy $E_j(x, y)$.

Hence, in both the cases we have shown that if process y satisfies environment condition $E_i(x, y)$ then it cannot satisfy any other environment condition $E_j(x, y)$.

□

Fact 5. Given a state s and two different processes c and d, there exists a unique environment condition $E_i(x, y)$ such that $s \models E_i(c, d)$.

Proof. Let L be the control location of process d in state s. Thus, $s \models \mathsf{pc}[d] = L$ holds.

Given processes c and d, for each inter-predicate $\mathcal{EP}_k(x, y)$ we have either $s \models \mathcal{EP}_k(c, d)$ or $s \not\models \mathcal{EP}_k(c, d)$. Consider the formula

$$F(x, y) \quad \doteq \quad y \neq x \wedge \mathsf{pc}[y] = L \wedge \bigwedge_{s \models \mathcal{EP}_k(c,d)} \mathcal{EP}_k(x, y) \wedge \bigwedge_{s \not\models \mathcal{EP}_k(c,d)} \neg \mathcal{EP}_k(x, y).$$

Clearly, $s \models F(c,d)$ by construction. Syntactically, $F(x,y)$ is identical to a unique environment condition. By Fact 4, processes c, d can satisfy at most one environment condition. □

Fact 6. Let $E_i(x,y)$ be an environment condition and $\mathcal{G}(x,y)$ be a boolean formula over the inter-predicates $\mathcal{EP}_1(x,y), \ldots, \mathcal{EP}_r(x,y)$ and predicates of the form $\mathsf{pc}[y] = L$. Then either $E_i(x,y) \Rightarrow \mathcal{G}(x,y)$ or $E_i(x,y) \Rightarrow \neg\mathcal{G}(x,y)$.

Proof. Since $E_i(x,y)$ has the form $\mathsf{pc}[y] = j \wedge R_k(x,y)$ where $R_k(x,y)$ is a min-term over the inter-predicates $\mathcal{EP}_1(x,y), \ldots, \mathcal{EP}_r(x,y)$, $E_i(x,y)$ enforces a unique truth value for all atomic subformulas of $\mathcal{G}(x,y)$. □

We are ready to return to the descriptions $\Delta(x)$. A description $\Delta(x)$ has the format

$$\mathsf{pc}[x] = i \quad \wedge \quad \pm\mathcal{E}_1(x) \wedge \pm\mathcal{E}_2(x) \wedge \cdots \wedge \pm\mathcal{E}_T(x), \qquad \text{where } i \in [1..S].$$

Intuitively, a description $\Delta(x)$ gives detailed information on the internal state of process x, and how the other processes are related to process x. Note the correspondence of $\Delta(x)$ to the abstract state in Figure 4.1: the control location i determines the color of the central circle, and the $\mathcal{E}_j$ determine the processes surrounding the central one.

Definition 4.3.1 (Abstraction Mapping). Let $P(K)$, $K > 1$, be a concrete system and $p \in [1..K]$ be a process. The abstraction mapping α_p induced by p *maps* a global state s of $\mathcal{P}(K)$ to an abstract state $\langle \mathbf{pc}, e_1, \ldots, e_T \rangle$ where $\mathbf{pc} =$ the value of $\mathsf{pc}[p]$ in state s and for all e_j we have $e_j = 1 \Leftrightarrow s \models \mathcal{E}_j(p)$.

We will now prove the coverage and congruence conditions that let us apply environment abstraction.

Lemma 4.3.2. *Consider a description $\Delta(x)$ and a label $l(x)$. Then either $\Delta(x) \Rightarrow l(x)$ or $\Delta(x) \Rightarrow \neg l(x)$*

Proof. Consider first the case where $l(x) \doteq \mathsf{pc}[x] = L$. Since $\Delta(x)$ is of the form

$$\mathsf{pc}[x] = L \quad \wedge \quad \pm\mathcal{E}_1(x) \wedge \pm\mathcal{E}_2(x) \wedge \cdots \wedge \pm\mathcal{E}_T(x)$$

it is easy to see that, in this case, $\Delta(x)$ either $\Delta(x) \Rightarrow l(x)$ or $\Delta(x) \Rightarrow \neg l(x)$.

Consider the second case where label $l(x) \doteq L \in \mathsf{env}(x)$. Recall that $L \in \mathsf{env}(x)$ is syntactic sugar for $\exists y.y \neq x.\mathsf{pc}[y] = L$. The description $\Delta(x)$ is of the form

$$\mathsf{pc}[x] = L \quad \wedge \quad \pm\mathcal{E}_1(x) \wedge \pm\mathcal{E}_2(x) \wedge \cdots \wedge \pm\mathcal{E}_T(x)$$

where each $\mathcal{E}_i(x)$ is of the form

$$\exists y.y \neq x \wedge R_j(x, y) \wedge \mathsf{pc}[y] = L_i$$

Consider all those environment conditions $\mathcal{E}_i(x)$ of the form

$$\exists y.y \neq x \wedge R_j(x, y) \wedge \mathsf{pc}[y] = L$$

That is, those environment conditions that require the other process y to be in control location L. Denote this set of environment conditions by $\mathcal{E}_L$.

Now

$$\Delta(x) \Rightarrow \bigvee_{\mathcal{E}_i(x) \in \mathcal{E}_L} \pm\mathcal{E}_i(x) \qquad (*)$$

where the polarity of each $\mathcal{E}_i(x)$ in the consequent is exactly as in the description $\Delta(x)$. Suppose at least one environment condition, say $\mathcal{E}_j(x)$, in $\mathcal{E}_L$ appears un-negated in the consequent of $(*)$. Then,

$$\Delta(x) \Rightarrow \mathcal{E}_j(x) \qquad (\dagger)$$

Since $\mathcal{E}_j(x)$ is of the form

$$\exists y.y \neq x \wedge R_k(x,y) \wedge \mathsf{pc}[y] = L$$

it follows that

$$\mathcal{E}_j(x) \Rightarrow \exists y.y \neq x \wedge \mathsf{pc}[y] = L.$$

Thus, we have

$$\Delta(x) \Rightarrow \exists y.y \neq x \wedge \mathsf{pc}[y] = L$$

in case at least one of the environment conditions in $\mathcal{E}_L$ appears un-negated in $\Delta(x)$.

For the other case, suppose none of the environment conditions in $\mathcal{E}_L$ appear unnegated in $\Delta(x)$. Then we have

$$\Delta(x) \Rightarrow \bigwedge_{\mathcal{E}_i(x) \in \mathcal{E}_L} \neg \mathcal{E}_i(x)$$

or equivalently,

$$\Delta(x) \Rightarrow \neg(\bigvee_{\mathcal{E}_i(x) \in \mathcal{E}_L} \mathcal{E}_i(x))$$

Now $\mathcal{E}_i(x)$ is of the form

$$\exists y.y \neq x \wedge R_k(x,y) \wedge \mathsf{pc}[y] = L$$

and where each $R_k(x,y)$ is of the form

$$R_k(x,y) \doteq \pm\mathcal{EP}_1(x,y) \wedge \ldots \wedge \pm\mathcal{EP}_r(x,y)$$

Since the set of relation predicates $R_k(x,y)$ is formed by taking all possible cubes over the inter-predicates $\mathcal{EP}_1(x,y), \ldots, \mathcal{EP}_T(x,y)$ it follows that

$$\bigvee R_k(x,y) = \mathbf{true}$$

This means at least one of the relation predicates must be **true**. Assume without loss of generality that $R_p(x, y)$ is **true**. Let the corresponding environment condition from $\mathcal{E}_L$ which involves $R_p(x, y)$ be $\mathcal{E}_p(x, y)$. Now

$$\Delta(x) \Rightarrow \neg(\bigvee_{\mathcal{E}_i(x) \in \mathcal{E}_L} \mathcal{E}_i(x))$$

which implies

$$\Delta(x) \Rightarrow \neg(\mathcal{E}_p(x))$$

that is

$$\Delta(x) \Rightarrow \exists y.y \neq x \wedge R_p(x, y) \wedge \mathsf{pc}[y] = L$$

Since $R_p(x, y)$ is **true**

$$\mathcal{E}_p(x) \doteq \exists y.y \neq x \wedge \mathsf{pc}[y] = L$$

So we have

$$\Delta(x) \Rightarrow \exists y.y \neq x \wedge \mathsf{pc}[y] = L$$

or equivalently

$$\Delta(x) \Rightarrow \neg l(x)$$

Thus, in case none of the environment conditions in $\mathcal{E}_L$ appear unnegated in $\Delta(x)$, $\Delta(x) \Rightarrow \neg l(x)$. Hence either $\Delta(x) \Rightarrow l(x)$ or $\Delta(x) \Rightarrow \neg l(x)$ and the lemma is proved. Note that, this lemma establishes the congruence property described in Section 2.2. □

Remark 11. Consider the full set of descriptions

$$\mathsf{pc}[x] = L \quad \wedge \quad \pm\mathcal{E}_1(x) \wedge \pm\mathcal{E}_2(x) \wedge \cdots \wedge \pm\mathcal{E}_T(x), \qquad \text{where } L \in [1..S].$$

Given any concrete state s and process c in a system $\mathcal{P}(K)$ it is clear that $s \models \Delta(c)$ for some description $\Delta(x)$. This is true simply because we take every possible conjunction of the predicates $\mathcal{E}_1(x), \ldots, \mathcal{E}_T(x)$ with every possible predicate $\mathsf{pc}[x] = L$. Thus, the coverage condition discussed in Section 2.2 holds for the set of descriptions given above.

The other property required to make environment abstraction sound, namely the congruence property is established by the above lemma. Thus, for the chosen set of abstract descriptions and labels, environment abstract is sound.

We will now represent descriptions $\Delta(x)$ by tuples of values, as usual in predicate abstraction. The possible descriptions $(*)$ only differ in the value of the program counter $\mathsf{pc}[x]$ and in where they have negations in front of the $\mathcal{E}(x)$ predicates. Denoting negation by 0 and absence of negation by 1, every description $\Delta(x)$ can be identified with a tuple $\langle \mathbf{pc}, e_1, \ldots e_T \rangle$ where **pc** is a control location, and each e_i is a boolean variable.

Example 4.3.3. Consider again the protocol shown in Example 4.2.1. There is only one inter-predicate $\mathcal{EP}_1(x, y) \doteq t[x] \neq t[y]$. Thus, we have two possible relationship conditions $R_1(x, y) \doteq t[x] = t[y]$ and $R_2(x, y) \doteq t[x] \neq t[y]$. Consequently, we have 6 different environment predicates:

$$\mathcal{E}_1(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 1 \wedge R_1(x, y) \qquad \mathcal{E}_4(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 1 \wedge R_2(x, y)$$

$$\mathcal{E}_2(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 2 \wedge R_1(x, y) \qquad \mathcal{E}_5(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 2 \wedge R_2(x, y)$$

$$\mathcal{E}_3(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 3 \wedge R_1(x, y) \qquad \mathcal{E}_6(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 3 \wedge R_2(x, y)$$

The abstract state then is a 7-tuple $\langle \mathbf{pc}, e_1, \ldots, e_6 \rangle$ where **pc** refers to the internal state of the reference process x. For each $i \in [1..6]$, the bit e_i tells whether there is an

environment process $y \neq x$ such that the environment predicate $\mathcal{E}_i(x)$ becomes true. $\square$

We build the abstract model $\mathcal{P}^{\mathcal{A}}$ exactly as in Section 2.2. Since the congruence and coverage conditions hold for the set of descriptions and labels we have chosen, we have the following corollary of Theorem 2.2.4:

Corollary 3 (Soundness of Abstraction). Let $\mathcal{P}(\mathbf{N})$ be a parameterized mutual exclusion system and $\mathcal{P}^{\mathcal{A}}$ be its abstraction. For an indexed property $\forall x.\Phi(x)$, where $\Phi(x)$ is a control condition, we have

$$\mathcal{P}^{\mathcal{A}} \models \Phi(x) \Rightarrow \forall K.\mathcal{P}(K) \models \forall x.\Phi(x).$$

4.4 Extensions for Fairness and Liveness

The abstract model that we have described, while sound, might be too coarse in practice to be able to verify liveness properties. The reason is two fold:

(i) Spurious Infinite Paths. Our abstract model may have infinite paths which cannot occur in any concrete system. Figure 4.2 shows one such instance, where a self-loop in the abstract model leads to a spurious infinite path. The two concrete states s_1 and s_2, such that s_1 transitions to s_2, map to the same abstract state $\hat{s}$, leading to a self-loop involving $\hat{s}$. This self-loop can lead to a spurious infinite path. Such spurious paths hinder the verification of liveness properties.

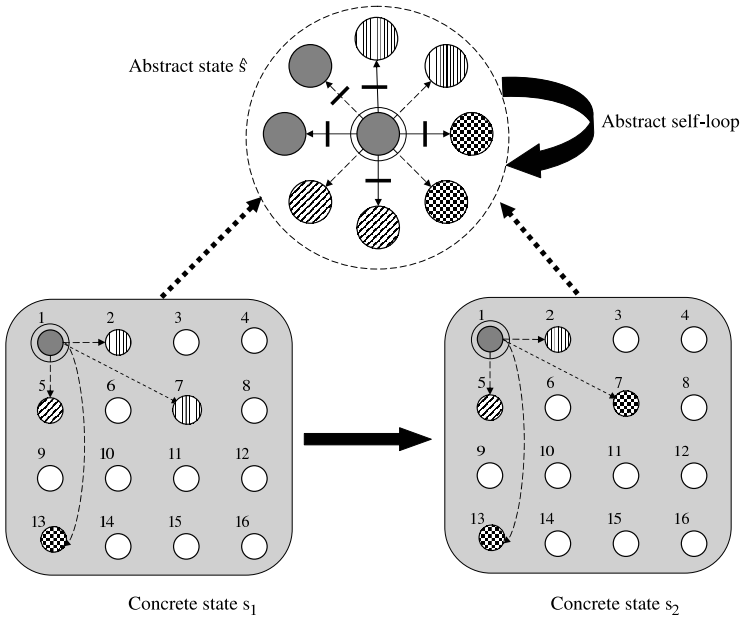


Figure 4.2: Process 7 changes its internal state, but the abstract state is not affected. Thus, there is a self-loop around the abstract state. The abstract infinite path consisting of repeated executions of this loop has no corresponding concrete infinite path.

(ii) Fairness Conditions. Liveness properties are usually expected to hold under some fairness conditions. A typical example of a fairness condition is that every process x must leave the critical section a finite amount of time after entering it. This is expressed formally by the fairness condition $\mathsf{pc}[x] \neq \mathbf{crit}$. In this work, we will consider fairness conditions $\mathsf{pc}[x] \neq L$, where L is a control location. Liveness properties are then expected to hold on *fair paths*: an infinite path in a concrete system $\mathcal{P}(K)$, $K > 1$ is *fair* only if, for each process i, the fairness condition $\mathsf{pc}[i] \neq L$ holds infinitely often.

Monitor Processes for Liveness

To handle these situations, we adapt a method developed by Pnueli et al. [66] in the context of counter abstraction to our environment abstraction. The extension to handle liveness essentially consists of adding monitor processes. We first augment the concrete system $\mathcal{P}(K)$ by adding monitor processes $M(1), \ldots, M(K)$ where each $M(i)$ has two sets of variables

- variables $from_1^i, \ldots, from_T^i$ where T is the number of different environments, and
- variables $to_1^i, \ldots, to_T^i$ where T is the number of different environments.

Intuitively, the $from^i$ and to^i variables keep track of the processes coming and going out of the environments $\mathcal{E}_1(i), \ldots, \mathcal{E}_T(i)$ as viewed from process i.

Updating Monitor Variables

Suppose the system $\mathcal{P}(K)$ is initially in state s_1 and some process j changes its state resulting in state s_2 for system $\mathcal{P}(K)$. Monitor process $M(i)$ then updates its variables as follows.

- *Case 1: A process $j \neq i$ changes its state*

  By Fact 4, we have uniquely defined environment predicates $\mathcal{E}_p(i)$ and $\mathcal{E}_q(i)$ such that $s_1 \models E_p(i,j)$ and $s_2 \models E_q(i,j)$. Thus monitor process $M(i)$ sets $from_p^i = \mathbf{true}$ and $to_q^i = \mathbf{true}$ in state s_2. The rest of the $from^i$ and to^i variables are set to $\mathbf{false}$ in state s_2.

- *Case 2: j = i and the process changes its state using update transition*

  For each environment condition $\mathcal{E}_p(i)$ such that there is a process y satisfying $E_p(i, y)$ in s_1, $from_p^i = \mathbf{true}$ in state s_2. For each environment condition $\mathcal{E}_p(i)$ such that there is a process z satisfying $E_p(i, z)$ in s_2, $to_p^i = \mathbf{true}$ in state s_2. In all other cases, the $from^i$ and to^i variables in s_2 are **false**.

- *Case 3: j = i and the process changes its state using a guarded transition*

  In this case, all the $from^i$ and to^i variables in s_2 are **false**.

Denote the system obtained by augmenting $\mathcal{P}(K)$ with monitor processes $M(1), \ldots, M(K)$ by $\mathcal{PM}(K)$. The states of $\mathcal{PM}(K)$ are given by tuples of the form $\langle \mathcal{L}_1, \ldots, \mathcal{L}_K, \mathcal{M}_1, \ldots, \mathcal{M}_K \rangle$ where $\mathcal{L}_i$ is denotes local state of process i and $\mathcal{M}_i$ denotes the local state of the monitor process $M(i)$. The augmented abstract states, given by tuples of the form $\langle \mathbf{pc}, e_1, \ldots e_T, from_1, \ldots, from_T, to_1, \ldots, to_T \rangle$, carry monitor information for reference process unchanged.

Definition 4.4.1 (Abstraction Mapping). Let $\mathcal{PM}(K)$, $K > 1$, be an augmented system and $p \in [1..K]$ be a process. The abstraction mapping α_p induced by p *maps* a global state s of $\mathcal{PM}(K)$ to an abstract state $\langle \mathbf{pc}, e_1, \ldots, e_T, from_1, \ldots, from_T, to_1, \ldots, to_T \rangle$ where

- $\mathbf{pc} =$ the value of $\mathsf{pc}[p]$ in state s
- for all e_j we have $e_j = 1 \Leftrightarrow s \models \mathcal{E}_j(p)$.
- $\forall j. \quad from_j = from_j^p$
- $\forall j. \quad to_j = to_j^p$

Intuitively, the $from$, and to variables keep track of the *immediate* history of an abstract state, that is, the last step by which the abstract state was reached.

Example 4.4.2. Referring to Figure 4.2, suppose process 7 in state s_1 satisfies the environment condition $E_i(1,7)$. Then, in the new augmented abstract state, variable $from_i$ will be set to **true** to indicate that a process satisfying environment condition $\mathcal{E}_i(1)$ made the move. Similarly, suppose that in the new concrete state s_2, process 7 satisfies the environment condition $E_j(1,7)$. Then in the new augmented abstract state, the variable to_j is set **true** to indicate that after the transition process 7 satisfies the new environment condition $\mathcal{E}_j(1)$.

Remark 12. Note that the abstract model does not retain the id of the process which was responsible for the transition (process 7 in this case). The abstract model only retains the environment predicates satisfied by the process before and after the transition. We are doing this for two reasons:

- During abstraction, all the processes except the *reference* process lose their identities.
- Remembering the environment predicate satisfied by the active process will give us a sufficiently precise abstraction to verify the properties of interest.

To recapitulate, using the $from$ and to variables we are able to keep track of the last step of the route by which an abstract state was reached.

For an augmented abstract state $\hat{s}$, we denote its projection consisting of only the pc and e_i variables by $\pi(\hat{s})$. The following notation is also useful: let s_1, s_2 be two concrete

states in a system $\mathcal{P}(K)$ such that there is a transition from s_1 to s_2. Denote by $s_1 \lhd s_2$ the index of the process whose local transition lead to the global transition from s_1 to s_2, e.g. process 7 in Figure 4.2. Recall that in an asynchronous system, only one process at a time changes its state, i.e., for each global transition, there exists a single process causing the transition.

The augmented abstract model $\mathcal{P}_a^{\mathcal{A}} \doteq (S_a^{\mathcal{A}}, I_a^{\mathcal{A}}, R_a^{\mathcal{A}}, L_a^{\mathcal{A}})$ of the augment parameterized system $\mathcal{PM}(K)$ is defined as in Section 2.5.2.

Note that coverage and congruence conditions for the augmented abstract descriptions are trivial to establish given that the original abstract descriptions satisfy both conditions. It follows from Section 2.5.2 that adding the additional monitor information does not affect the soundness of our abstract model. Thus, we have $\mathcal{P}_a^{\mathcal{A}} \models \Phi(x) \Rightarrow \mathcal{PM}(K) \models \forall x.\Phi(x)$ where $\Phi(x)$ is a control condition.

Corollary 4 (Soundness of Augmented Abstraction). Let $\mathcal{P}(\mathbf{N})$ be a parameterized system and $\mathcal{PM}(\mathbf{N})$ be an augmentation of $\mathcal{P}(\mathbf{N})$ with monitor processes as described above and $\mathcal{P}_a^{\mathcal{A}}$ be its augmented abstraction. For an indexed property $\forall x.\Phi(x)$, where $\Phi(x)$ is a control condition we have

$$\mathcal{P}_a^{\mathcal{A}} \models \Phi(x) \Rightarrow \forall K.\mathcal{PM}(K) \models \forall x.\Phi(x) \Rightarrow \forall K.\mathcal{P}(K) \models \forall x.\Phi(x)$$

4.4.1 Abstract Fairness Conditions

We will now show how to deal with the two problems mentioned in the beginning of this section, i.e., (i) spurious paths and (ii) fairness conditions.

Eliminating Spurious Infinite Paths.

Recall from the example in Figure 4.2 that due to the abstraction there may exist infinite spurious paths that do not have any corresponding concrete paths, in particular such paths where $from_i$ is true infinitely often but to_i is not. Such a path cannot correspond to any concrete path because:

- By definition, the variable $from_i$ is true if a process having satisfied $\mathcal{E}_i(x)$ in the previous state, does not satisfy $\mathcal{E}_i(x)$ in the current state.
- By definition, the variable to_i is true if a process having satisfied $\mathcal{E}_j(x)$ in the previous state, does satisfy $\mathcal{E}_i(x)$ in the current state.
- Each concrete system has only a finite number of processes.
- Thus, for a finite number of processes to make $from_i$ true infinitely often, it is necessary for to_i to be true infinitely often as well.

Therefore, to eliminate the spurious infinite paths arising from loops described above, we add for each $i \in [1..T]$ a *compassion condition* [66] $\langle from_i, to_i \rangle$ which says *if* $from_i = \mathbf{true}$ *holds infinitely often in a path, then* $to_i = \mathbf{true}$ *must hold infinitely often as well.* We will denote this set of fairness conditions by **F1**.

Adding Abstract Fairness Conditions.

Assume that in the concrete model each process has a fairness condition of the form $\mathsf{pc} \neq L$. This means that a process is not allowed to stay at control location L forever. To abstract

the concrete fairness condition we have to find two sets of abstract fairness conditions, one for the reference process I_A and the other for the environment processes.

Fairness conditions for the environment processes. The abstract model maintains the properties of the environment processes only in terms of the environment predicates. Thus, the concrete fairness conditions on the environment processes have to be translated to fairness conditions involving the environment predicates.

More precisely, given a fairness condition $\mathsf{pc} \neq L$, we need to consider those environment conditions that require the environment process to be in control location L, i.e., those environment conditions $E_i(x, y)$ where $E_i(x, y) \doteq R_j(x, y) \wedge \mathsf{pc}[y] = L$. For each such $E_i(x, y)$ we add the fairness condition

$$\neg(from_i = \mathbf{false} \wedge e_i = 1).$$

This condition excludes the cases where along an infinite path, the set of processes satisfying the environment condition $E_i(x, y)$ is non-empty (i.e., $e_i = 1$) and none of these processes ever changes its state (i.e., $from_i = \mathbf{false}$). We will denote this set of fairness conditions by **F2**

Fairness conditions for the reference process. The abstract fairness condition corresponding to the reference process is given by $\mathbf{pc} \neq L$. This expresses the requirement that the control location of the reference process is *not* L infinitely often. We will denote this set of fairness conditions by **F3**.

4.4.2 Soundness in the Presence of Fairness Conditions

Now we will show that adding these fairness conditions do not rule out any legitimate paths in the abstract model. Thus, the augmented abstracted model will be sound.

We are usually interested in verifying single index liveness properties of the form

$$\forall x.\mathbf{AG}(\phi(x) \rightarrow \mathbf{F}\psi(x)).$$

For example, for mutual exclusion protocols, the standard liveness property, which says if process is trying to get into the critical section it eventually will, can be written as

$$\forall x.\mathbf{AG}(\mathsf{pc}[x] = \mathbf{try} \rightarrow \mathbf{F}(\mathsf{pc}[x] = \mathbf{crit})).$$

The following theorem claims that, for single index liveness properties, the augmented abstract model that we constructed is sound.

Theorem 4.4.3 (Soundness of Abstraction). *Let $\mathcal{P}(K)$ be a parameterized system with fairness constraint $\mathsf{pc} \neq L$ and $\mathcal{P}_a^{\mathcal{A}}$ be its* augmented *abstraction using the abstract fairness and compassion conditions. Given the single-indexed liveness property $\forall x.\Phi(x)$, $\mathcal{P}_a^{\mathcal{A}} \models \Phi(x)$ under the abstract fairness conditions implies $\mathcal{P}(K) \models \forall x.\Phi(x)$ under the concrete fairness condition.*

The augmented abstraction thus obtained is precise enough to prove liveness properties for the two mutual exclusion protocols we considered. The following section gives a proof of the soundness theorem.

4.4.3 Proof of Soundness

Given concrete fairness condition $\mathsf{pc}[x] \neq L$, the augmented abstract model has three sets of fairness conditions:

F1. For each $i \in [1..T]$, the compassion condition $\langle from_i, to_i \rangle$ saying that if $from_i = \mathbf{true}$ infinitely often along an abstract path then $to_i = \mathbf{true}$ infinitely often as well.

F2. The fairness conditions $\neg(from_i = \mathbf{true} \wedge e_i = 1)$ for each i such that the environment condition $E_i(x, y)$ requires process y to be in control location L.

F3. The fairness condition $\mathbf{pc} \neq L$ requiring that the reference process satisfies the concrete fairness condition $\mathsf{pc}[x] \neq L$.

The proof of Theorem 4.4.3 relies on the following lemma.

Lemma 4.4.4. *Let $\mathcal{P}(K)$ be a concrete system with process c and let $\sigma \doteq g_0, g_1, \ldots$ be a fair path under the fairness constraint $\mathsf{pc}[x] \neq L$. Then the augmented abstract model $\mathcal{P}_a^{\mathcal{A}}$ has a path $\hat{\sigma} \doteq \hat{g}_0, \hat{g}_1, \ldots$ such that*

1. *for each $i \geq 0$, $\pi(\hat{g}_i) = \alpha_c(g_i)$, and*

2. *the abstract fairness conditions* **F1**, **F2**, **F3** *hold for $\hat{\sigma}$.*

Proof. The lemma claims that corresponding to every concrete fair path and a given reference process c there is an abstract fair path in the augmented abstract model. In other words, our abstract fairness conditions do not remove any fair paths.

Given a concrete fair path σ of system $\mathcal{P}(K)$ we construct an abstract fair path $\hat{\sigma}$ as follows:

- For the first state $\hat{g}_0$ we require $\pi(\hat{g}_0) = \alpha_c(g_0)$, and the $from_i$ and to_i variables can have any value.

- For each $\hat{g}_i$, $i > 0$ we require $\alpha_c(g_i) = \pi(\hat{g}_1)$, and the to_i, and $from_i$ are set according to the definition of the augmented abstract transition relation in Section 4.4.

Thus, item 1 of the lemma is satisfied by construction. The fact that $\hat{\sigma}$ is a valid trace in the abstract model also follows by construction.

We will now show that if σ is a fair path then $\hat{\sigma}$ satisfies the abstract fairness conditions as well. We consider the different ways in which $\hat{\sigma}$ might fail to satisfy the abstract fairness conditions and argue that each case leads to a contradiction.

Violation of F1. Assume towards a contradiction that $\hat{\sigma}$ violates the compassion condition $\langle from_k, to_k \rangle$ for some $k \in [1..T]$, i.e., there exists an $i \geq 0$ such that

- $from_k$ is **true** infinitely often in states $\hat{g}_i, \hat{g}_{i+1}, \ldots$ (†)

- but $to_k = \mathbf{false}$ in all the states $\hat{g}_i, \hat{g}_{i+1}, \ldots$ (*)

There are two cases in which $from_k$ holds **true** in a certain state $\hat{g}_l$:

(a) A process y satisfying enviroment condition $E_k(c, y)$ in g_{l-1} moves to a new environment in g_l.

(b) In state g_{l-1}, there is a process y satisfying the environment condition $E_k(c, y)$, and the reference process c makes an update transition from g_{l-1} to g_l.

For $from_k$ to be **true** infinitely often, either case (a) or case (b) has to hold infinitely often. We will show that both cases lead to a contradiction. First assume case (a). As there are only a finite number of processes, $from_k$ being true inifinitely often requires to_k to be true infinitely often as well. This contradicts (*).

In case (b) the $from_k$ is **true** in a state $\hat{g}_l$ because the reference process made an update transition and there was process y satisfying the environment $E_k(c, y)$ in state g_{l-1}. After such an update transition we again have two cases:

(b.1) There is a process y satisfying the condition $E_k(c, y)$ in state g_l, i.e., e_k is 1 in $\hat{g}_l$. In this case to_k is set to **true** by our definition of the augmented abstract transition relation in Section 4.4, or

(b.2) there is no process y satisfying the condition $E_k(c, y)$ in state g_1 i.e., $e_k = 0$.

The former case (b.1) immediately contradicts the assumption (*). In the latter case (b.2), if e_k continues to be 0, then, by definition, $from_k$ cannot be **true** again. This contradicts the assumption (†).

Thus, we have proved that the compassion condition $\langle from_i, to_i \rangle$ cannot be violated in the abstract trace $\hat{\sigma}$.

Violation of F2. Assume towards a contradiction that $\hat{\sigma}$ violates the fairness condition $\neg(from_k = \textbf{false} \wedge e_k = 1)$ where the environment condition $E_k(x, y)$ requires process y to be in control location L. That is, there exists an $i \geq 0$ such that $\hat{g}_j \models from_k =$

$\mathbf{false} \wedge e_k = 1$ for all $j \geq i$. In other words, in the concrete trace in all the states $g_j, g_{j+1}, \ldots$ of the concrete trace there is a process y satisfying the environment condition $E_k(c, y)$, and this process y never leaves the environment corresponding to $E_k(x, y)$. Since $E_k(x, y)$ requires process y to be in control location L, process y violates the concrete fairness condition $\mathsf{pc}[\mathsf{slf}] \neq L$, and thus, the assumption of the lemma.

Violation of F3. Assume towards a contradiction that $\hat{\sigma}$ violates the fairness condition $\mathbf{pc} \neq L$, i.e., there is an $i \geq 0$ such that for all $\hat{g}_j, j > i$, $\hat{g}_j \models \mathbf{pc} = L$ holds. This is possible only if process c stays in control location L after concrete state g_i, thus violating the concrete fairness condition, and thus, the assumption of the lemma.

We see that in all the three cases we are led to a contradiction. Consequently, the abstract trace $\hat{\sigma}$ does not violate any abstract fairness conditions. □

Theorem 4.4.3. Consider a single index liveness property $\forall x.\mathbf{AG}(\phi(x) \rightarrow \mathbf{F}\psi(x))$. Assume that

$$\mathcal{P}_a^{\mathcal{A}} \models \Phi(x)$$

under abstract fairness conditions, and assume towards a contradiction that there is a fair path $\sigma \doteq g_0, g_1, \ldots$ in system $\mathcal{P}(K)$ such that $\sigma \not\models \phi(c) \rightarrow \mathbf{F}\psi(c)$ for some process c. Thus, there exists an $i \geq 0$ such that $g_i \models \phi(c)$ and $g_j \not\models \psi(c)$ for all $j \geq i$. By Lemma 4.4.4 there is a *fair* abstract path $\hat{\sigma} \doteq \hat{g}_0, \hat{g}_1, \ldots$ such that for all k, $\pi(\hat{g}_k) = \alpha_c(g_k)$. By definition, $\hat{g}_i \models \phi(x)$ and for all $j \geq i$, $\hat{g}_j \not\models \psi(x)$. Thus, there is a fair path $\hat{\sigma}$ in the abstract model $\mathcal{P}_a^{\mathcal{A}}$ that violates the liveness property $\Phi(x)$, contradicting our assumption that $\mathcal{P}_a^{\mathcal{A}} \models \Phi(x)$. This completes our proof. □

4.5 Computing the Abstract Model

We have thus far presented the theoretical description of the abstract model and the properties it satisfies. We have not described how to actually obtain such an abstract model from a given parameterized system. In this section, we will show how to construct the abstract model.

Computing the abstract model is evidently complicated by the fact there is an infinite number of concrete systems. Further, it is well known that in predicate abstraction and related methods, computing the exact abstract model is computationally very expensive. Instead of finding the most precise abstract model, we find an over-approximation of the abstract model. We consider each concrete transition statement of the program separately and *over-approximate* the set of abstract transitions it can lead to. The union of these sets will be the abstract transition relation. A concrete transition can either be a guarded transition or an update transition. Each transition can be executed by the reference process or one of the environment processes. Thus, there are four cases to consider:

Active process is ...	guarded transition	update transition
... reference process	Case 1	Case 2
... environment process	Case 3	Case 4

We will show how we abstract in each of these cases and argue why the computed abstract transition is an *over-approximation*. Before we begin we recall the following facts:

Fact 7. For any two environment predicates $E_i(x, y)$ and $E_j(x, y)$, $i \neq j$ $E_i(x, y) \Rightarrow \neg E_j(x, y)$.

Fact 8. Given any formula $G(x, y)$ involving inter-predicates $\mathcal{EP}_1(x, y), \ldots, \mathcal{EP}_r(x, y)$ either $E_i(x, y) \Rightarrow \neg\mathcal{G}(x, y)$ or $E_i(x, y) \Rightarrow \mathcal{G}(x, y)$.

We now introduce some useful notation. The environment condition $E_i(x, y) \doteq y \neq x \wedge R_j(x, y) \wedge \mathsf{pc}[y] = L$ will be denoted by $E_{(j,L)}(x, y)$. The variables and formulas corresponding to this environment condition are referred to using the same subscript $\{j, L\}$, e.g., the corresponding environment predicate is referred to as $\mathcal{E}_{(j,L)}(x)$ and the corresponding abstract variable is $e_{(j,L)}$. The set of all environment conditions $E_{(j,L)}(x, y)$ is referred to as E_L.

4.5.1 Case 1: Guarded Transition for Reference Process

Consider first the case of guarded transitions being executed by the reference process. Consider the guarded transition statement t_G:

$$L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2 \textbf{ else goto } L_3$$

Suppose the reference process is executing this guarded transition statement. If at least one of the environment processes contradicts the guard $\mathcal{G}$ then the reference process transitions to control location L_3, i.e., the *else branch*. Otherwise, the reference process goes to L_2. We will now formalize the conditions under which the *if* and *else* branches are taken.

Definition 4.5.1 (Blocking Set for Reference Process). Let $\mathscr{G} \doteq \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ be

a guard. We say that an environment condition $E_i(x, y)$ blocks the guard $\mathscr{G}$ if $E_i(x, y) \Rightarrow \neg\mathcal{G}(x, y)$. The set $\mathcal{B}^x(\mathscr{G})$ of all indices i such that $E_i(x, y)$ blocks $\mathscr{G}$ is called the blocking set of the reference process for guard $\mathscr{G}$.

Note that by Fact 6, either $E_i(x, y) \Rightarrow \neg\mathcal{G}(x, y)$ or $E_i(x, y) \Rightarrow \mathcal{G}(x, y)$ for every environment $E_i(x, y)$. The intuitive idea behind the definition is that $\mathcal{B}^x(\mathscr{G})$ contains the indices of all environment conditions which enforce the *else branch*.

We will now explain how to represent the guarded transition t_G in the abstract model: we introduce an abstract transition from $\hat{s_1} = \langle \mathbf{pc}, e_1, .., e_T, from_1, .., from_T, to_1, .., to_T \rangle$ to $\hat{s_2} = \langle \mathbf{pc'}, e_1, .., e_T, from'_1, .., from'_T, to'_1, .., to'_T \rangle$ if

GR1. $\mathbf{pc} = L_1$, i.e., the reference process is in location L_1,

GR2. one of the following two conditions holds:

- *Then Branch:* $\forall i \in \mathcal{B}^x(\mathscr{G}).\ (e_i = 0)$ and $\mathbf{pc'} = L_2$, i.e., the guard is true and the reference process moves to control state L_2.
- *Else Branch:* $\neg\forall i \in \mathcal{B}^x(\mathscr{G}).\ (e_i = 0)$ and $\mathbf{pc'} = L_3$, i.e., the guard is false and the reference process moves to control state L_3.

GR3. and all the variables $from'_1, .., from'_T$ and $to'_1, .., to'_T$ are **false**, expressing that none of the environment processes changes its state.

Together, these three conditions can be viewed as an *transition invariant* $I^x(t_G)$ between the current and the next abstract states. The following fact shows that the set of

abstract transitions represented by $I^x(t_G)$ is indeed an over-approximation of the set of abstractions that t_G gives rise to.

Lemma 4.5.2. *Let s_1 be a state in a concrete system $\mathcal{P}(K)$, and suppose that a process c executes a guarded transition t_G which leads to state s_2. Then the abstract states $\hat{s}_1 \doteq \alpha_c(s_1)$ and $\hat{s}_2 \doteq \alpha_c(s_2)$ satisfy the invariant $I^x(t_G)$.*

Proof. Assume there is a guarded transition t_G from state s_1 to s_2 with the reference process c as the active process. There are two cases to consider:

- The concrete guard was true in state s_1 and process c's new control location in state s_2 is L_2. By Fact 6, each environment condition $E_i(x, y)$ either implies the guard condition $\mathcal{G}(x, y)$ or its negation. Any process y satisfying an environment $E_j(x, y), j \in \mathcal{B}^x(\mathscr{G})$ would block the guard $\mathcal{G}(c, y)$. In other words, for the *then branch* to be taken, in state s_1 every concrete process $y \neq c$ must have satisfied only environment conditions which are not mentioned in the blocking set $\mathcal{B}^x(\mathscr{G})$. Thus, the condition $\forall i \in \mathcal{B}^x(\mathscr{G}).e_i = 0$ is true in $\hat{s}_1$ and the abstract model transitions to state $\hat{s}_2$.

- The concrete guard was false in state s_1 and process c's new control location in state s_2 is L_3. By Fact 6, each environment condition $E_i(x, y)$ either implies the guard condition $\mathcal{G}(x, y)$ or its negation. For the *else-branch* to be taken, there must be at least one process y in state s_1 satisfying an environment $E_j(x, y), j \in \mathcal{B}^x(\mathscr{G})$ so that the guard $\mathcal{G}(c, y)$ evaluates to false. Thus, the abstract condition $\forall i \in \mathcal{B}^x(\mathscr{G}).e_i = 0$ is false for $\hat{s}_1$, and the abstract model still transitions to state $\hat{s}_2$.

□

4.5.2 Case 2: Guarded Transition for Environment Processes

Suppose that the guarded transition t_G

$$L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2 \textbf{ else goto } L_3$$

is executed by a concrete process y satisfying the environment condition $E_{(i,L_1)}(x, y)$. The active process thus switches from environment condition $E_{(i,L_1)}(x, y)$ to environment condition $E_{(i,L_2)}(x, y)$ or $E_{(i,L_3)}(x, y)$. Note that in a guarded transition, only the pc of the active process changes.

We will now define a *blocking set* for this environment condition $E_{(i,L_1)}(x, y)$ as follows. The difference from Definition 4.5.1 is that the guard for process y can be blocked either by the reference process or by another environment process. Therefore we need to distinguish two cases in the definition.

Definition 4.5.3 (Blocking Set for Environment $E_{(i,L_1)}(x, y)$). Let $\mathscr{G}(\mathsf{slf}) = \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ be a guard. We say that

1. An environment condition $E_j(x, z)$ blocks the guard for process y satisfying $E_{(i,L_1)}(x, y)$ if

   $$E_{(i,L_1)}(x, y) \wedge E_j(x, z) \Rightarrow \neg\mathcal{G}(y, z).$$

   Let $\mathcal{B}^1_{(i,L_1)}(\mathcal{G})$ be the set of all such indices j.

2. The control location L of the reference process x blocks the guard for process y satisfying $E_{(i,L_1)}(x,y)$ if

$$E_{(i,L_1)}(x,y) \wedge \mathsf{pc}[x] = L \Rightarrow \neg\mathcal{G}(y,x).$$

Let $\mathcal{B}^2_{(i,L_1)}(\mathcal{G})$ be the set of all such control locations L.

Note that we consider the guards $\mathcal{G}(y,z)$ and $\mathcal{G}(y,x)$ because y is the active process, i.e., y executes the transition. We define the *abstract guard* $\mathscr{G}^i$ for guard $\mathscr{G}(\mathsf{slf}) \doteq \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf},\mathsf{otr})$ and the environment condition $E_i(x,y)$ as follows:

$$\forall j \in \mathcal{B}^1_{(i,L_1)}(\mathscr{G}).(e_j = 0) \quad \wedge \quad \mathbf{pc} \notin \mathcal{B}^2_{(i,L_1)}(\mathscr{G}).$$

Since the transition starts in control location L_1 and the active process is an environment process, we will describe the abstract transition invariant $I^{i,L_1}_y(t_G)$ for each $E_{(i,L_1)}(x,y) \in E_{L_1}$ by a list of conditions as in Case 1. The abstract transition invariant for Case 2 will then be

$$I^y(t_G) \doteq \bigvee_{E_{(i,L_1)}(x,y) \in E_{L_1}} I^{i,L_1}_y(t_G).$$

Consider one such $E_{(i,L_1)}(x,y) \in E_{L_1}$. The abstract transition relation $I^{i,L_1}_y(t_G)$ has a transition from $\hat{s} = \langle \mathbf{pc}, e_1, \ldots, e_T, from_1, \ldots, from_T, to_1, \ldots, to_T \rangle$
to $\hat{s}' = \langle \mathbf{pc}', e'_1, \ldots, e'_T, from'_1, \ldots, from'_T, to'_1, \ldots, to'_T \rangle$ if the following conditions hold:

GE1. $e_{(i,L_1)} = 1$, that is, there is a process satisfying environment condition $E_{(i,L_1)}(x,y)$.

GE2. $from'_{(i,L_1)} = \mathbf{true}$, that is, the active process switches from environment condition $E_{(i,L_1)}(x,y)$ to some other environment condition.

GE3. $e'_{(i,L_1)} \in \{0,1\}$, that is, due to the active process switching, there may or may not remain a process satisfying the environment condition $E_{(i,L_1)}(x,y)$.

Depending on the value of the abstract guard, one of the following two cases holds:

1. The guard $\mathscr{G}^i$ is true, i.e., $\hat{s} \models \mathscr{G}^i$, and either
   - the *then branch* is taken, i.e.,$e'_{(i,L_2)} = 1$ and $to_{(i,L_2)} = \mathbf{true}$, or
   - the *Else branch* is taken, i.e., $e'_{(i,L_3)} = 1$ and $to_{(i,L_3)} = \mathbf{true}$.

     We will explain this case below.

2. The guard is false, i.e., $\hat{s} \not\models \mathscr{G}^i$, and the *Else branch* is taken, i.e., $e'_{(i,L_3)} = 1$, $to_{(i,L_3)} = \mathbf{true}$.

GE4. The rest of the e_j variables are the same in $\hat{s}$ and $\hat{s}'$.

GE5. The $from'_j$ and to'_j variables are set to $\mathbf{false}$ by default unless they are set to **true** by one of the above conditions.

The reason for the two *else-branches* is the fact that knowledge about a single process suffices to block the guard, while knowledge about all processes is necessary to make sure the guard is not blocked. The environment predicates $E_j(x,y)$ only contain accurate information about the relationship between the data variables of the reference process x and the data variables of environment process y. If it follows already from this partial information that the guard is violated then the *Else branch* is enforced. If however the

guard $\mathscr{G}^i$ is true, this may be due to lack of information in the abstract predicates, and we over-approximate the possible abstract transitions.

Note that Case 2 is different to Case 1, because, in Case 1, the reference process makes the guarded transition, while in Case 2, an environment process makes the transition. In case of the reference process, our abstraction maintains the relationship of its data variables to the other variables. In case of an environment process, we only know the relationship of its data variables to the reference process.

Lemma 4.5.4. *Let s_1 be a state in a concrete system $\mathcal{P}(K)$, and let c be a process used as reference process. Suppose that a process $d \neq c$ executes a guarded transition t_G that leads to state s_2. Then the abstract states $\alpha_c(s_1)$ and $\alpha_c(s_2)$ satisfy the invariant $I^y(t_G)$.*

Proof. This follows directly from the construction of the transition invariant $I^y(t_G)$. □

4.5.3 Case 3: Update Transition for Reference Process

Consider the case where the reference process is executing an update transition t_U:

$$L_1: \quad \textbf{for all } \mathsf{otr} \neq \mathsf{slf} \quad \textbf{if } \mathcal{T}(\mathsf{slf}, \mathsf{otr}) \textbf{ then } u_k := \Phi(\mathsf{otr})$$
$$\textbf{goto } L_2$$

Recall that each process has data variables $u_1, \ldots, u_d$. We denote the next state value of each variable u_m by u'_m.

When the reference process x changes the value of its data variables, the valuations of the environment predicates $\mathcal{E}_1(x) \ldots \mathcal{E}_T(x)$ will change. For a process y satisfying environment condition $E_i(x, y)$ we need to figure out the possible new environment conditions $E_j(x, y)$ that y might satisfy after the reference process x has executed the update transition. The set of possible new environment conditions for process y is called the *outset* O_i for condition $E_i(x, y)$. (Technically, the outset is the set of the indices of these environment conditions.) We will now explain how to compute the outset.

Denote by $\mathbf{T}(x, y)$ the formula

$$\mathcal{T}(x, y) \wedge u'_k[x] := \phi(y) \ \vee \ \neg\mathcal{T}(x, y) \wedge u'_k[x] := u_k[x].$$

We call $\mathbf{T}(x, y)$ the *update formula*. Given the update formula we find what possible inter-predicates involving $u'_k[x], u_k[y]$ can be true. Formally, the set $C^1(u_k)$ of these inter-predicates is given by all formulas $u_k[x] \prec u_k[y]$ where $\prec \in \{<, >, =\}$ such that

$$u'_k[x] \prec u_k[y] \wedge \mathbf{T}(x, y) \quad \text{is satisfiable.}$$

Thus, $C^1_{u_k}$ contains all possible ways $u_k[x]$ and $u_k[y]$ can relate to each other after the update. The possible relationships between $u_k[x]$ and $u_k[y]$ might change again when, in the course of evaluating the update transition, x repeatedly updates its u_k value by looking at other processes.

Suppose x looks at another process z and updates its $u_k[x]$ value again. We now find the set $C^2(u_k)$ of possible relationships between $u_k[x]$ and $u_k[y]$ *after* the new update involving process z under the assumption that a relation from $C^1(u_k)$ holds *before* the update.

Thus, the new set $C^2(u_k)$ of relationships is given by all formulas $u_k[x] \prec u_k[y]$ where $\prec \in \{<, >, =\}$, such that

$$u'_k[x] \prec u_k[y] \wedge \mathbf{T}(x, z) \wedge \psi(x, y) \text{ is satisfiable and } \psi(x, y) \in C^1(u_k).$$

Note that $C^1(u_k) \subseteq C^2(u_k)$ because the definition of $\mathbf{T}(x, z)$ allows the possibility that the value of $u_k[x]$ remains unchanged. We similarly compute sets $C^3(u_k), C^4(u_k) \ldots$ until a *fixpoint* is reached. Since the number of possible inter-predicates is finite, a fixpoint always exists; for simple inter-predicates involving $<, >$, and =, the fixpoint computation takes three iterations at the most. We denote this fixpoint by $C(u_k)$.

In the environment condition $E_i(x, y)$, let θ be the inter-predicate that describes the relation between $u_k[x]$ and $u_k[y]$. Consider the set of environment conditions $E_j(x, y)$ that are obtained from $E_i(x, y)$ by replacing θ by a formula in the fixpoint $C(u_k)$ – the indices of these environment conditions constitute the *outset* O_i of $E_i(x, y)$. Correspondingly, the *inset* $I_k \subseteq \{1..T\}$ for environment condition $E_k(x, y)$ consists of all j such that $k \in O_j$.

We denote the abstract transition invariant corresponding to the concrete update transition t_U by $I^x_{\text{up}}(t_U)$. $I^x_{\text{up}}(t_U)$ has a transition from the abstract state
$\hat{s} = \langle \mathbf{pc}, e_1, \ldots, e_T, from_1, \ldots, from_T, to_1, \ldots, to_T \rangle$ to the abstract state
$\hat{s}' = \langle \mathbf{pc}', e'_1, \ldots, e'_T, from'_1, \ldots, from'_T, to'_1, \ldots, to'_T \rangle$ if the following conditions hold:

UR1. $\mathbf{pc} = L_1$, i.e., the reference process is in control location L_1 before the transition.

UR2. $\mathbf{pc}' = L_2$, i.e., the reference process moves to control location L_2.

UR3. $\forall k \in [1..T].(e_k = 1 \Rightarrow \exists j \in O_k.e'_j = 1)$, i.e., if there was a process in environment $\mathcal{E}_k(x)$ before the transition then there must be a process in one of the outset

environments O_k of $E_k(x, y)$ after the transition.

UR4. $\forall j \in I_k.(e_j = 0 \Rightarrow e'_k = 0)$, i.e., if there is no process satisfying the *inset* environments I_k of environment $E_k(x, y)$ before the transition then after the transition there can be no process in environment $E_k(x, y)$.

UR5. $\forall k \in [1..T].(e'_k = 1 \Leftrightarrow to'_k = \mathbf{true})$. The variable to'_k indicates if after the transition there is a process satisfying $E_k(x, y)$.

UR6. $\forall k \in [1..T].(e_k = 1 \Leftrightarrow from'_k = \mathbf{true})$. The variable $from'_k$ indicates if before the transition there is a process satisfying $E_k(x, y)$.

Lemma 4.5.5. *Let s_1 be a state in a concrete system $\mathcal{P}(K)$, and suppose that a process c executes an update transition t_U which leads to state s_2. Then the abstract states $\alpha_c(s_1)$ and $\alpha_c(s_2)$ satisfy the invariant $I^x_{\mathrm{up}}(t_U)$.*

Proof. This follows directly from the construction of the invariant $I^x_{\mathrm{up}}(t_U)$. □

4.5.4 Case 4: Update Transition for Environment Processes

This case is quite similar to Case 3. Recall that $E_{(i,L_1)}(x, y)$ denotes the environment condition $y \neq x \wedge R_i(x, y) \wedge \mathsf{pc}[y] = L_1$. Consider the case where a generic process y satisfying environment $E_{(i,L_1)}(x, y)$ is executing an update transition t_U:

$$L_1 : \quad \textbf{for all } \mathsf{otr} \neq \mathsf{slf} \quad \textbf{if } \mathcal{T}(\mathsf{slf}, \mathsf{otr}) \textbf{ then } u_k := \Phi(\mathsf{otr})$$
$$\textbf{goto } L_2$$

After the update transition process y will have a new control location and also the relationship of its data variables to those of the reference process x will have changed. The *outset* $O_{(i,L_1)}$ for environment $E_{(i,L_1)}(x, y)$ will consist of all those environments $E_{(j,L_2)}(x, y)$ that process y may satisfy after the update transition. To compute the outset $O_{(i,L_1)}$ we proceed as follows. As in the previous case, we find a fixpoint $C(u_k)$ that contains the possible relationships between $u_k[x]$ and $u_k[y]$. The initial set of relationships $C^1(u_k)$ is the set of all $u_k[x] \prec u_k[y]$, $\prec \in \{<, >, =\}$ such that

$$\mathbf{T}(y, x) \wedge u_k[x] \prec u'_k[y] \text{ is satisfiable}$$

where $\mathbf{T}(y, x)$ is the update condition as defined in Section 4.5.3. Note that we consider $\mathbf{T}(y, x)$ (and not $\mathbf{T}(x, y)$ as in the previous section) because y is the active process. As y updates its u_k variable repeatedly, the relationship between $u_k[x], u_k[y]$ will also change. To compute all the possible relationships we use an approach similar to the fixpoint computation in Case 3. Thus, we find the set $C^2(u_k)$ of all $u_k[x] \prec u_k[y]$, $\prec \in \{<, >, =\}$ such that

$$\mathbf{T}(y, z) \wedge u_k[x] \prec u'_k[y] \wedge \psi(x, y) \text{ is satisfiable}$$

where $\psi(x, y) \in C^1(u_k)$. We similarly compute $C^3(u_k), C^4(u_k), \ldots$ until we reach a fixpoint $C(u_k)$.

In the environment condition $E_{(i,L_1)}(x, y)$, let θ be the inter-predicate that describes the relation between $u_k[x]$ and $u_k[y]$. Consider the set of environment conditions $E_{(j,L_2)}(x, y)$

that are obtained from $E_{(i,L_1)}(x,y)$ by replacing θ by a formula in the fixpoint $C(u_k)$ and replacing the condition $\mathsf{pc}[y] = L_1$ by $\mathsf{pc}[y] = L_2$ – the indices of these environment conditions, written as pairs (j, L_2), constitute the *outset* $O_{(i,L_1)}$ of $E_{(i,L_1)}(x,y)$.

Since the transition starts at control location L_1 and a generic process executes it, we will describe the abstract transition $I_y^{(i,L_1),(j,L_2)}(t_U)$ for each environment condition $E_{(i,L_1)}(x,y)$ and each $(j,L_2) \in O_{(i,L_1)}$. The abstract transition $I^y_{\mathrm{up}}(t_U)$ for Case 4 will be

$$\bigvee_{E_{(i,L_1)}(x,y)} \quad \bigvee_{(j,L_2)\in O_{(i,L_1)}} \quad I_y^{(i,L_1),(j,L_2)}(t_U).$$

As above, we will define $I_y^{(i,L_1),(j,L_2)}(t_U)$ by a list of conditions. $I_y^{(i,L_1),(j,L_2)}(t_U)$ has a transition from $\hat{s} = \langle \mathbf{pc}, e_1, \ldots, e_T, from_1, \ldots, from_T, to_1, \ldots, to_T \rangle$
to $\hat{s}' = \langle \mathbf{pc}', e'_1, \ldots, e'_T, from'_1, \ldots, from'_T, to'_1, \ldots, to'_T \rangle$ if the following conditions hold:

UE1. $\mathbf{pc} = \mathbf{pc}'$, i.e., the reference process does not move.

UE2. $e_{(i,L_1)} = 1$, i.e., there is a process in environment $E_{(i,L_1)(x,y)}$ before the transition.

UE3. $e'_{(j,L_2)} = 1$, i.e., there is a process in environment $E_{(j,L_2)(x,y)}$ after the transition.

UE4. $e'_l = e_l$ for $l \notin \{(i,L_1),(j,L_2)\}$, i.e., all the e variables except $e'_{(i,L_1)}$ and $e'_{(j,L_2)}$ remain the same.

UE5. $from'_{(i,L_1)} = \mathbf{true}$ and the rest of the $from'_l$ variables are **false**, i.e., only a process satisfying environment condition $E_{(i,L_1)}(x,y)$ moves, and no other process moves.

UE6. $to'_{(j,L_2)} = \mathbf{true}$ and the rest of the to'_l variables are **false**, i.e., only the environment condition $E_{(i,L_1)}(x, y)$ has a new process and no other environment condition has a new process.

Lemma 4.5.6. *Let s_1 be a state in a concrete system $\mathcal{P}(K)$, and let c be the process used as reference process. Suppose that a process $d \neq c$ executes an update transition t_U that leads to state s_2. Then the abstract states $\alpha_c(s_1)$ and $\alpha_c(s_2)$ satisfy the invariant $I^y_{\text{up}}(t_U)$.*

Proof. This follows directly from the construction of the invariant $I^y_{\text{up}}(t_U)$. □

4.6 Experimental Results

In most mutual exclusion protocols, the predicates appearing in the guards are simple linear expressions involving the $<, >$, and = operators. Thus, the decision problems that arise during abstraction are simple and are handled by our abstraction program internally. We verified the safety and liveness properties of Szymanski's mutual exclusion protocol and Lamport's bakery algorithm. These two protocols have an intricate combinatorial structure and have been used widely as benchmarks for parameterized verification. For safety properties, we verified that no two processes can be present in the critical section at the same time. For liveness, we verified the property that if a process wishes to enter the critical section then it eventually will.

Note that these protocols have been analyzed by other methods, but in most cases either the protocols have been simplified (in addition to the atomicity assumption) or the method cannot handle both protocols. Pnueli et al. [66] have verified Szymanski's and

	Inter-preds	Intra-preds	Reachable states	Safety	Liveness
Szymanski	1	8	$O(2^{14})$	0.1s	1.82s
Bakery	3	5	$O(2^{146})$	68.55s	755.0s

Figure 4.3: Running Times

the Bakery protocol using counter abstraction, but they manually introduce new auxillary variables. Lahiri and Bryant [53] verified the Bakery protocol but not Szymanski's protocol. Pnueli et al. [64] have verified a modified version of the Bakery protocol in which the unbounded *ticket* variable is replaced by a bounded variable. The method described in [6] can handle Szymanski's protocol but not the Bakery protocol because it has unbounded integer variables. A possible exception is regular model checking, but this method is very different from ours and encoding protocols as regular languages is a complex and error prone process.

We used the Cadence SMV model checker to verify the finite abstract model. The model checking times are shown in Figure 4.3. The abstraction time is negligible, less than 0.1s. Figure 4.3 also shows the number of predicates and the size of the reachable state space as reported by SMV. All experiments were run on a 2.4 GHz Pentium machine with 512 MB main memory.

4.7 Protocols and Specifications

The details of the two protocols that we verified are given below.

F = $\{pc\}, pc \in \{0, 1, 2, 3, 4, 5, 6, 7\}$

$pc = 0:$ **goto** $pc = 1$

$pc = 1:$ **if** $\forall \mathsf{otr} \neq \mathsf{slf}.pc[\mathsf{otr}] \in \{0, 1, 2, 4\}$ **then goto** $pc = 2$

else goto $pc = 1$

$pc = 2:$ **goto** $pc = 3$

$pc = 3:$ **if** $\forall \mathsf{otr} \neq \mathsf{slf}.pc[\mathsf{otr}] \notin 1, 2$ **then goto** $pc = 5$

else goto $pc = 4$

$pc = 4:$ **if** $\forall \mathsf{otr} \neq \mathsf{slf}.pc[\mathsf{otr}] \notin \{5, 6, 7\}$ **then goto** $pc = 4$

else goto $pc = 5$

$pc = 5:$ **if** $\forall \mathsf{otr} \neq \mathsf{slf}.pc[\mathsf{otr}] \notin \{2, 3, 4\}$ **then goto** $pc = 6$

else goto $pc = 5$

$pc = 6:$ **if** $\forall \mathsf{otr} > \mathsf{slf}.pc[\mathsf{otr}] \in \{0, 1, 2\}$ **then goto** $pc = 7$

else goto $pc = 6$

$pc = 7:$ **goto** $pc = 0$

Figure 4.4: Szymanski's Mutual Exclusion Protocol

Szymanski's mutual exclusion protocols written in our specification language is shown in Figure 4.4. This protocol has been taken from [66]. The protocol presented there has **wait** statements that, under the atomicity assumption, can be modeled by guarded statements. The transition

$$pc = 0 : \quad \textbf{goto } pc = 1$$

is syntactic sugar for the more complicated but equivalent guarded statement

$$pc = 0 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\ \textbf{true } \textbf{ then goto } pc = 1 \textbf{ else goto } pc = 1.$$

The property that we verified for Szymanski is

$$\forall x \neq y.\ \textbf{AG}\ .\neg(pc[x] = 7 \wedge pc[y] = 7)$$

and the liveness property that we verified is

$$\forall x.\ \textbf{AG}\ .(pc[x] = 1 \rightarrow \textbf{F}\ pc[x] = 7).$$

Note that $pc = 7$ corresponds to the critical state and $pc = 1$ corresponds to the trying state. The only inter-predicate is $x < y$, were x, y are index variables. As mentioned previously, the inter-predicates and the control assignments of the form $\mathsf{pc}[x] = L$ constitute all the predicates that occur in the protocol text.

Lamport's bakery algorithm is shown in Figure 4.5. The update transition

$$pc = 2 \wedge ch = 0 : \quad \textbf{update } t := 0 \textbf{ then goto } pc = 0 \wedge ch = 0$$

is syntactic sugar for

$$pc = 2 \wedge ch = 0 :\ \textbf{for all } \mathsf{otr} \neq \mathsf{slf}.\ (\textbf{if true then } t := 0) \textbf{ goto } pc = 0 \wedge ch = 0.$$

Note that here we have two finite variables pc and ch which together determine the control location. In Section 4.2 we have argued that without loss of generality we can have only one finite variable pc. In fact, we can easily write the Bakery protocol using just one finite variable pc with domain $\{0,1,2\} \times \{0,1\}$. Our implementation allows a protocol to have multiple finite variables. Thus, we did not have to rewrite the Bakery protocol before verifying it.

The variable ch indicates whether a process is updating its ticket variable t or not. A process updates its t value by choosing the maximum among all other t values and incrementing it by 1. In Lamport's original paper [54], a process i does the following check before entering the critical section:

for all $j \in [1..N]$

$L2$: **if** $ch[j] \neq 0$ **goto** $L2$ **else goto** $L3$

$L3$: **if** $t[j] > 0 \wedge ((t[\mathsf{otr}], \mathsf{otr}) \prec (t, \mathsf{slf}))$ **then goto** $L3$ **else goto crit**

crit

Here $(t[\mathsf{otr}], \mathsf{otr}) \prec (t, \mathsf{slf}))$ stands for $t[\mathsf{otr}] < t \vee (t[\mathsf{otr}] = t \wedge \mathsf{otr} < \mathsf{slf})$. Following the atomicity assumption discussed in Section 4.2, we model the **for** loop in the original Bakery algorithm as a guarded transition:

$$pc = 1 \wedge ch = 0 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.ch[\mathsf{otr}] = 0 \wedge \neq (t[\mathsf{otr}] > 0 \wedge ((t[\mathsf{otr}], \mathsf{otr}) \prec (t, \mathsf{slf})))$$

F = $\{pc, ch\}, pc \in \{0,1,2\},\ ch \in \{0,1\}$

$pc = 0 \wedge ch = 0 :$ **goto** $pc = 0 \wedge ch = 1$

$pc = 0 \wedge ch = 1 :$ **for all** $(\mathsf{otr} \neq \mathsf{slf})$. **if** $(t < t[\mathsf{otr}])$ **then** $t := t[\mathsf{otr}] + 1$

goto $pc = 1 \wedge ch = 0$

$pc = 1 \wedge ch = 0 :$ **if** $\forall \mathsf{otr} \neq \mathsf{slf}. ch[\mathsf{otr}] = 0 \wedge \neg(t[\mathsf{otr}] > 0 \wedge ((t[\mathsf{otr}], \mathsf{otr}) \prec (t, \mathsf{slf})))$

then goto $pc = 2 \wedge ch = 0$

else goto $pc = 1 \wedge ch = 0$

$pc = 2 \wedge ch = 0 :$ **update** $t := 0$ **goto** $pc = 0 \wedge ch = 0$

Figure 4.5: Lamport's Bakery Algorithm

The safety property that we verified is

$$\forall x \neq y.\ \mathbf{AG}\ .\neg((pc[x] = 2 \wedge ch[x] = 0) \wedge (pc[y] = 2 \wedge ch[y] = 0))$$

and the liveness property that we verified is

$$\forall x.\ \mathbf{AG}\ .((pc[x] = 0 \wedge ch[x] = 1) \rightarrow \mathbf{F}\ (pc[x] = 2 \wedge ch[x] = 0)).$$

Note that $pc = 2 \wedge ch = 0$ corresponds to the critical state, and $pc = 0 \wedge ch = 0$ corresponds to the trying state. The inter-predicates that we used are $x < y$, $t(x) < t(y)$, $t(x) = t(y)$, that is, all predicates appearing in the protocol code that compare variables of two different processes.

Chapter 5

Removing the Atomicity Assumption for Mutex Protocols

5.1 Introduction

In Chapter 4, we showed how environment abstraction can be applied to verify mutual exclusion protocols, like the Bakery protocol and Szymanski's protocol, completely automatically. But, the verification was carried out under the atomicity assumption. The atomicity assumption, in essence, says that any process in a distributed system consisting of a collection of processes can know the state of all the other processes instantaneously. As we will see in Section 5.3, this assumption is quite restrictive. In this chapter, we will show how this assumption can be removed with the help of non-interfering monitor

processes and thus verify mutual exclusion protocols in their full generality.

All the previous model checking based methods for parameterized verification have assumed atomicity to some extent. Counter abstraction [66] makes use of this assumption as does the work on Invisible Invariants [3; 41; 42; 64]. Removing the atomicity assumption in the latter method is theoretically possible but the reported experiments have made use of the atomicity assumption. The Indexed Predicates method [52; 53] too makes partial use of atomicity – the update transition appearing in the bakery protocol is assumed to happen atomically. As with the Invisible Invariants method, removing the atomicity assumption is theoretically possible in the Indexed Predicates method, but the cost of verification is probably high. The Inductive Method, presented in [62], is an exception to this trend. It has been applied to verify both safety and liveness of the Bakery algorithm without assuming atomicity. This approach however is not automatic as the user is required to provide lemmas and theorems to prove the properties under consideration. In contrast, our approach is a fully automatic procedure.

The outline for the rest of the chapter is as follows. In the next section, we will present the formal system model. In section 5.3, we will show, with the help of an example, why the atomicity assumption significantly reduces the complexity of a protocol. We will then discuss how monitors can be used to remove the assumption and show how to perform the abstraction in the presence of these monitor processes. In the last section, we will present experimental results to illustrate our method.

5.2 Modeling Mutual Exclusion Protocols without Atomicity Assumption

As before, we consider a parameterized system $\mathcal{P}(K)$ with K identical processes running asynchronously and communicating via shared variables. The state variables are exactly the same as in the model considered in Section 4.2. While we used only two transition constructs in the previous chapter, we will need three different transition constructs to describe mutual exclusion protocols in their full generality. We use *guarded transitions* and *wait transitions* for describing transitions involving only finite control, and the more complicated *update transitions* for transitions that modify data variables. Though guarded and update transitions are syntactically similar to their counterparts in Section 4.2, their semantics are quite different. The wait transition, as the name indicates, is used to model processes waiting for some global condition to happen before moving. The sections below describe the transitions in detail.

Guarded Transitions

A guarded transition has the form

$$\mathsf{pc} = L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } \mathsf{pc} = L_2 \textbf{ else goto } \mathsf{pc} = L_3$$

or shorter

$$L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2 \textbf{ else goto } L_3$$

$Obligations := \{1, .., K\} \setminus \{\mathsf{slf}\}$

Loop Forever{

1. **Pick** $\mathsf{otr} \in Obligations$

2. If $\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ then $Obligations := Obligations \setminus \{\mathsf{otr}\}$ else **Exit Loop** with **false**

3. If $Obligations$ is empty **Exit Loop** with **true** }

Figure 5.1: Evaluation of a Guard

where L_1, L_2, and L_3 are control locations. In the guard $\forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$, the variable otr ranges over the process ids of all other processes. The condition $\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ is any formula involving the data variables of processes slf, otr and the pc variable of otr. The semantics of a guarded transition is as follows. A process slf executing the transition first evaluates the guard $\forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ according to the pseudocode shown in Figure 5.1. In executing the loop, each line in the code is executed atomically. This is not a restricting assumption because each line is an internal action of a process.

The **then** branch is taken if the loop is exited with value **true** and pc is set to L_2. Otherwise, the **else** branch is taken and pc is set to L_3.

Wait Transitions

A wait transition has the form

$Obligations := \{1, .., K\} \setminus \{\mathsf{slf}\}$

Loop Forever{

1. **Pick** otr $\in Obligations$

2. If $\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ then $Obligations := Obligations \setminus \{\mathsf{otr}\}$

3. If $Obligations$ is empty **Exit Loop** }

Figure 5.2: Evaluation of a Wait condition

$$\mathsf{pc} = L_1 : \quad \textbf{wait till } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } \mathsf{pc} = L_2$$

or shorter

$$L_1 : \quad \textbf{wait till } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2$$

where L_1, L_2 are control locations. A process slf executing the transition first evaluates the guard $\forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ according to the loop shown in Figure 5.2. As with guarded transitions, each line of the pseudocode is executed atomically.

Note that unlike the loop for a guarded transition, the loop for a wait transition cannot be exited until the set $Obligations$ is empty. Once the loop is exited the process transitions to new control location L_2. Wait transitions are found often in protocols. This construct was not present in Chapter 4 because, under the atomicity assumption, the wait transition

$$L_1 : \quad \textbf{wait till } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2$$

is equivalent to the guarded transition

$$L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2 \textbf{ else goto } L_1$$

Update Transitions

Recall that update transitions are needed to describe protocols such as the Bakery algorithm where a process *computes* a data value depending on all values that it can read from other processes. Update transitions are syntactically of the form

$$\begin{aligned} \mathsf{pc} = L_1 : \quad & \textbf{for all } \mathsf{otr} \neq \mathsf{slf} \quad \textbf{if } \mathcal{T}(\mathsf{slf}, \mathsf{otr}) \textbf{ then } u_k := \Phi(\mathsf{otr}) \\ & \textbf{goto } \mathsf{pc} = L_2 \end{aligned}$$

or shorter

$$\begin{aligned} L_1 : \quad & \textbf{for all } \mathsf{otr} \neq \mathsf{slf} \quad \textbf{if } \mathcal{T}(\mathsf{slf}, \mathsf{otr}) \textbf{ then } u_k := \Phi(\mathsf{otr}) \\ & \textbf{goto } L_2 \end{aligned}$$

where L_1 and L_2 are control locations, and $\mathcal{T}(\mathsf{slf}, \mathsf{otr})$ is a condition involving data variables of processes slf and otr. The semantics of the update transition is best understood in an operational manner. A process slf executing the update transition first executes the loop shown in Figure 5.3. Each line in the pseudocode is executed atomically.

Once the loop is exited, the process transitions to control location L_2. In control location L_1, the process scans over all the other processes (in an arbitrary nondeterministically chosen order), and, for each process otr, checks if the formula $\mathcal{T}(\mathsf{slf}, \mathsf{otr})$ is true. In this case, the process changes the value of its data variable u_k according to $u_k := \Phi(\mathsf{otr})$, where $\Phi(\mathsf{otr})$ is an expression involving variables of process otr. Thus, the variable u_k can be reassigned multiple times within a transition.

Note that in the three loops above, process otr is chosen non-deterministically from the set $Obligations$. In real implementations, processes are usually evaluated in a fixed

$Obligations := \{1, .., K\} \setminus \{\mathsf{slf}\}$

Loop Forever{

1. **Pick** otr $\in Obligations$

2. If $\mathcal{T}(\mathsf{slf}, \mathsf{otr})$ then $u_k[\mathsf{slf}] := \Phi(\mathsf{otr})$

$Obligations := Obligations \setminus \{\mathsf{otr}\}$

3. If $Obligations$ is empty **Exit Loop** }

Figure 5.3: Evaluation of an Update

deterministic order. Since our semantics allows processes to be checked in any order, the protocols described in our language contain more behaviors than the actual implementations. Thus, correctness (involving $ACTL^*$ properties) of a protocol written in our language implies the correctness of the implementation as well.

Remark 13. In our system model, we do not consider how the loops described above are actually implemented. Clearly, implementing these loops will require additional state variables. We will treat such variables as invisible variables.

5.3 Atomicity Assumption

In this section, we discuss, with the help of a running example, how removing the atomicity assumption makes a protocol considerably more complex. Although the atomicity assumption simplifies a protocol considerably, powerful machinery is still required to prove protocols correct automatically.

Consider the following simple protocol in which each process has just one variable pc.

init: $\mathsf{pc} = 1 :$ $\quad \mathbf{goto}\ \mathsf{pc} = 2$

try: $\mathsf{pc} = 2 :$ $\quad \mathbf{if}\ \forall \mathsf{otr} \neq \mathsf{slf}.\mathsf{pc}[\mathsf{otr}] \neq 3\ \mathbf{then\ goto}\ \mathsf{pc} = 3\ \mathbf{else\ goto}\ \mathsf{pc} = 1;$

crit': $\mathsf{pc} = 3 :$ $\quad \mathbf{goto}\ \mathsf{pc} = 1;$

The state of each process is given by the valuation of its pc variable. If we assume that the transitions are all atomic, it is easy to see that this protocol ensures mutual exclusion. This is because the guard condition $\mathscr{G} \doteq \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{otr}, \mathsf{slf})$ where $\mathcal{G}(\mathsf{otr}, \mathsf{slf}) \doteq \mathsf{pc}[\mathsf{otr}] \neq 3$ evaluates to true only when no process is in state $\mathsf{pc} = 3$. While this simple protocol can ensure mutual exclusion under the atomicity assumption, it cannot do so under real life conditions, as we describe below.

Consider the concrete system $\mathcal{P}(3)$ with three processes $P(1), P(2)$, and $P(3)$. Figure 5.4 shows a possible execution sequence. Note that, in giving this sequence, we assume we have knowledge of the "insides" of a process: for example, steps like "$\mathcal{G}$ true of 2" are not visible. The only things visible are the pc and the data variables of a process.

The local states for each of the three processes are shown, and the executing process at each step is indicated by an arrow ($\leftarrow$). The *observation step* '$\mathcal{G}$ true of 2' appearing under the column for $P(1)$ denotes the step in which process $P(1)$ evaluates the guard

Process	P(1)	P(2)	P(3)
Initial States	pc= 1	pc= 1	pc=1
	pc= 2 ←	idle	idle
	idle	pc= 2←	idle
	$\mathcal{G}$ true of 2 ←	idle	idle
	idle	$\mathcal{G}$ true of 1 ←	idle
	$\mathcal{G}$ true of 3 ←	idle	idle
	idle	$\mathcal{G}$ true of 3 ←	idle
	pc= 3 ←	idle	idle
	idle	pc= 3 ←	idle

Figure 5.4: A possible execution trace of the system with three processes.

condition $\mathcal{G}$ for process $P(2)$ and concludes that it is satisfied. Observe that in the last state both $P(1)$ and $P(2)$ are in state $\mathsf{pc} = 3$, violating mutual exclusion. Consider a more complicated execution sequence, shown in Figure 5.5

Process	P(1)	P(2)	P(3)
Local States	pc= 1	pc= 1	pc=1
	pc= 2 $\leftarrow$	pc= 1	pc= 1
	pc= 2	pc= 1	pc= 2 $\leftarrow$
	pc= 2	pc= 1	$\mathcal{G}$ true of 1 $\leftarrow$
	pc= 2	pc= 1	$\mathcal{G}$ true of 2 $\leftarrow$
	$\mathcal{G}$ true of 3 $\leftarrow$	pc= 1	pc= 2
	pc= 2	pc= 1	pc= 3 $\leftarrow$
	$\mathcal{G}$ true of 2 $\leftarrow$	pc= 1	pc= 3
	pc= 2	pc= 2 $\leftarrow$	pc= 3
	pc= 2	$\mathcal{G}$ false of 3 $\leftarrow$	pc= 3
	pc= 3 $\leftarrow$	pc= 2	pc= 3
	pc= 3	pc= 1 $\leftarrow$	pc= 3

Figure 5.5: A more complicated trace of the system.

In this sequence, the actions are interleaved such that process $P(2)$ observes $P(3)$ while $P(3)$ is in state $\mathsf{pc} = 3$. Thus the guard $\mathscr{G}$ is false for 2. Process $P(1)$ sees $P(3)$

when it is in $\mathsf{pc} = 2$, thus $P(3)$ does not block $P(1)$. It is clear from these two examples that the interleaving of actions is crucial and adds considerable complexity to the protocol. In fact, under the atomicity assumption neither of the traces shown above are legal traces. In particular, the execution sequences where the *observation steps* of different processes are interleaved are excluded by the atomicity assumption. It is precisely because of such execution sequences that designing a distributed mutual exclusion protocol is challenging.

5.4 Monitors for Handling Non-atomicity

Recall that each process in our system has one pc variable and a collection of data variables. While it is clear that interleaving of observation steps add considerable complexity to the protocol, none of the variables used in our systems really tracks the state of these observations steps. For example, consider again the sample trace shown in Figures 5.4. Since the *observation steps* are hidden to the observers on the outside, the execution trace seen from the outside looks as shown in Figure 5.6.

The current state of process $P(i)$ gives us no information about how much of global condition it has finished evaluating and how much is still left. For example, at the state marked idle under column marked $P(1)$ in the figure above, we do not know much of the guard condition $\mathscr{G} \doteq \forall \mathsf{otr} \neq \mathsf{slf}.\mathsf{pc}[\mathsf{otr}] \neq 3$ has already been evaluated by process

Process	P(1)	P(2)	P(3)
Initial States	pc= 1	pc= 1	pc=1
	pc= 2 ←	idle	idle
	idle	pc= 2←	idle
	pc= 3 ←	idle	idle
	idle	pc= 3 ←	idle

Figure 5.6: Execution trace seen from the "outside".

$P(1)$. Thus, if we consider only the visible state of processes, comprising of pc and data variables, we have no way of knowing the truth or falsity of global conditions [1].

Fortunately, even when the *observation steps* are invisible, we can gather some information about the truth or falsity of the guards by looking at the state of each process. For this, we have to consider the previous states, in addition to the current states, of the processes. To this end, we will define a collection of monitor processes that track the evolution of the local states of the processes. These monitor processes are *non interfering* and are composed *synchronously* with concrete systems $\mathcal{P}(K)$. By synchronously composed we mean the following: every time a process in $\mathcal{P}(K)$ moves, all the monitor processes run simultanesouly and update their variables based on the current state of the processes in $\mathcal{P}(K)$. *Crucially, the construction of the monitor processes is not specific to any particular protocol.* In other words, for any mutual exclusion protocol we can automatically

[1]Note that, if we assume atomicity, the truth or falsity of guards can be known just by observing the current states of all processes.

construct the monitor processes defined below.

For each process $P(i)$ in $\mathcal{P}(K)$ we have a monitor process $M_g(i)$. The monitor process $M_g(i)$ has the following variables

- $K-1$ variables $mg(i,j), j \neq i$ one for each process $P(j)$, $j \neq i$ with range $\{\mathsf{clean}, \mathsf{dirty}, \mathsf{idle}\}$. These monitor variables are used to handle guarded transitions.

- Another set of $K-1$ variables $mu(i,j), j \neq i$ one for each process $P(j)$, $j \neq i$ with range $\{\mathsf{clean}, \mathsf{dirty}, \mathsf{idle}\}$. These variables are used to handle update transitions.

- In addition, there is one variable $em[i]$ with range $\{\mathsf{clean}, \mathsf{dirty}, \mathsf{idle}\}$.

Monitor variables have value idle if they are not in use. Usually, monitor variables transition to value dirty from value idle. Typically, a monitor variable being dirty indicates that certain actions are not possible (an exception to this is the value dirty of em variable, which actually permits more behaviors). After this the monitor variable may transition to value clean. This value for a monitor variable usually indicates that the monitor variable has seen enough history information to allow all behaviors. Sometimes a monitor variable can transition from value idle to clean directly. Once a variable has become clean, it will stay clean until it is reset to idle.

In the next two subsections we will describe how the monitor variables are updated by the monitor processes. We will also formalize the exact information that we gain from monitor processes.

Monitor Variables for Guarded Transition

The variable $mg(i, j)$ keeps track of process j and is updated as shown in the Figure 5.7.

The value of $mg(i, j)$ is computed as follows:

1. If process i is not evaluting any guard $mg(i, j) = \mathsf{idle}$.
2. If $mg(i, j) = \mathsf{idle}$ and process i is evaluating a guard with condition $\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ and $\mathcal{G}(i, j)$ is false then $mg(i, j) = \mathsf{dirty}$.
3. If process i is evaluting a guard with condition $\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ and $\mathcal{G}(i, j)$ is true then $mg(i, j) = \mathsf{clean}$.
4. Otherwise $mg(i, j)$ retains its value

Figure 5.7: Update procedure for monitor variables pertaining to guarded transitions.

Intuitively, the variable $mg(i, j)$ present in monitor $M_g(i)$ tracks whether process j entered any state that makes the guard condition $\mathcal{G}(i, j)$ true while process i is evaluating the guard $\mathscr{G}(\mathsf{slf}, \mathsf{otr}) \doteq \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$. In such a case, the monitor variable is clean. Otherwise it is dirty. Informally, the variable $mg(i, j)$ being dirty means that process j will *block* the guard $\mathscr{G}(\mathsf{slf}, \mathsf{otr})$ for process i.

This code is run by the monitor process after each step of the asynchronous system $\mathcal{P}(K)$. Note that, the monitor process does not intefere with the execution of $\mathcal{P}(K)$ in any way.

The variable $em(i)$ with range $\{\mathsf{clean}, \mathsf{dirty}, \mathsf{idle}\}$ is updated as shown in Figure 5.8.

The value of variable $em(i)$ is fixed as follows:

1. If process i is not evaluating any guard then $em = \mathsf{idle}$.
2. If process i is a evaluating a guard with condition $\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ and there exists a process $j \neq i$ such that $\mathcal{G}(i, j)$ is false then $em = \mathsf{dirty}$.
3. If process i is a evaluating a guard with condition $\mathcal{G}(\mathsf{slf}, \mathsf{otr})$, $em \neq \mathsf{dirty}$ and for all processes $j \neq i$ $\mathcal{G}(i, j)$ is true then $em = \mathsf{dirty}$.
4. Otherwise $em(i, j)$ retains its value.

Figure 5.8: Procedure for updating monitor variables pertaining to guarded transitions.

Intuitively, if any process $j \neq i$ was in a state that falsified the guard $\mathcal{G}(i, j)$ while process i was evaluating it, then $em(i)$ becomes dirty. It stays dirty until it is reset to idle. Against the general trend, value of em can go from idle to clean to dirty. In fact, the value dirty for em actually means more behaviors are possible.

Variable $em(i)$ tracks whether any process $j \neq i$ was in a state which makes $\mathcal{G}(i, j)$ false while $P(i)$ is evaluating $\mathcal{G}(i, j)$. If such a process exists, $em(i)$ is set to dirty. If $P(i)$ is not evaluating any guard, then $em(i)$ is set to the default value clean.

The information given by monitor processes can be used to decide – approximately– the truth or falsity of the guards. The following lemma formalizes the relation between the monitor variables and guards.

Lemma 5.4.1. *Let process i in a concrete system $P(K)$ be evaluating a guard with condition $\mathcal{G}(\mathsf{slf}, \mathsf{otr})$. Then we have the following:*

- *If process i concludes that the guard is true, then all monitor variables $mg(i,j)$, $j \neq i$, must be* clean.

- *If the process i concludes that the guard is false, then the variable $em(i)$ is* dirty.

Proof. This lemma follows trivially from the way we defined the monitor variables $mg(i,j)$, $j \neq i$ and $em(i)$. □

Monitors Variables for Update Transitions

Consider an update transition

$$L_1: \quad \textbf{for all } \mathsf{otr} \neq \mathsf{slf} \quad \textbf{if } \mathcal{T}(\mathsf{slf}, \mathsf{otr}) \textbf{ then } u_k := \Phi(\mathsf{otr})$$
$$\textbf{goto } L_2$$

This transition updates the variable u_k of the executing process and, thus, affects the mutual relationships between the u_k variables of the different processes. To predict what possible relations (more precisely predicates) hold between $u_k[i]$ and $u_k[j]$ after process i executes the above transition, we described an automatic procedure in Section 4.5. The fixpoint based computation presented in Section 4.5, assumes atomicity, that is, when process i is performing an update all the other processes stay fixed. Under this assumption, we can find a set $F(u_k[i], u_k[j])$ of all predicates of interest that can hold between $u_k[i]$ and $u_k[j]$ after the update.

But, without the atomicity assumption, the fixpoint computation is no longer valid. More precisely, if process j also performs an update operation on variable u_k while process

i is doing the same, then we cannot use the fixed point computation to predict which relationships hold between $u_k[i]$ and $u_k[j]$ after the update operation. In this case, we just say that the set of all possible relations between $u_k[i], u_k[j]$ is simply $\mathcal{F}(u_k[i], u_k[j])$, where $\mathcal{F}(u_k[i], u_k[j])$ is the set of all possible predicates of interest (usually syntactically picked from the protocol code).

Thus, if we knew that two processes were not updating the same variable t simultaneously, then we can better predict the set of possible relations after the update using the fixpoint computation. The $K-1$ variables $mu(i,1), \ldots, mu(i,i-1), mu(i,i+1), \ldots, mu(i,K)$ with range $\{\mathsf{clean}, \mathsf{dirty}, \mathsf{idle}\}$ try to track precisely this information: the variable $mu(i,j)$ tells us whether process j was updating the same data variable at the same time as process i. The value of the variable $mu(i,j)$, $j \neq i$ is computed as shown in Figure 5.9.

Intuitively, $mu(i,j)$ being clean indicates that, at some point when process i was updating its variable t, process j was also updating the same variable t. We can use the information contained in the monitor processes to abstract the concrete behaviors as follows:

- If there is a process j such that $mu(i,j)$ is clean then the valuation of a predicate involving $u_k[j]$ and $u_k[i]$ could be anything as $u_k[j]$ might have changed while $u_k[i]$ was being updated.

- If process j is such that $mu(i,j)$ is dirty then we know that $u_k[j]$ could not have changed while i was executing the update transition. It is possible to figure out the possible relationships, after the update, between $u_k[i]$ and $u_k[j]$ as described above.

The value of the variable $mu(i, j)$ is computed as follows

- If process i is not evaluating any update transiton, then $mu(i, j) =$ idle.
- If $mu(i, j) =$ idle, process i is evaluating an update transition involving variable t, and process j is not doing any update involving t, then $mu(i, j) =$ dirty.
- If both processes i and j are doing update transitions involving the same unbounded variable t, then $mu(i, j) =$ clean.
- Otherwise $mu(i, j)$ retains its value.

Figure 5.9: Procedure for updating monitor variables pertaining to update transitions.

The following lemma formalizes the relationship between the monitor variables and the update transitions.

Lemma 5.4.2. *Suppose process i is updating variable t in an update transition with the update expression* $\mathbf{T}(\mathsf{slf}, \mathsf{otr})$. *Let $F(u_k[i], u_k[j])$ be the fixpoint of predicates as computed in Section 4.5. If $mu(i,j) = \mathsf{dirty}$, then the set of predicates that hold between $u_k[i]$ and $u_k[j]$ after process i has finished the update transition is a subset of $F(u_k[i], u_k[j])$.*

Proof. The proof follows from the fact that $mu(i,j)$ is dirty only if process j was not updating its $u_k[j]$ variable while process i was updating its $u_k[i]$ variable. Thus, by the way we compute the fixpoint $F(u_k[i], u_k[j])$, it contains all the possible relationships between $u_k[i]$ and $u_k[j]$ after the update by process i. □

Thus, our lack of information about the invisible/hidden steps (used in evaluating guards and updates) can be overcome by making use of synchronously composed non interfering monitors and we can build a sound abstraction of the actual behaviors.

Remark 14. Note that a process can either execute a guarded transition or an update transition, but not both at the same time. Thus, instead of having two sets of variables, namely $mg(i,j), j \neq i$ and $mu(i,j), j \neq i$, we can just have one set of variables $m(i,j), j \neq i$ with range $\{\mathsf{clean}, \mathsf{dirty}\}$.

From now on, each monitor process $M_g(i)$ will have variables $\{m(i,j) | j \neq i\}$ and the variable $em(i)$.

5.4.1 Abstracting the Monitor Variables

As in Chapter 4, we start with descriptions of $\Delta(x)$ having the format

$$\mathsf{pc}[x] = i \quad \wedge \quad \pm\mathcal{E}_1(x) \wedge \pm\mathcal{E}_2(x) \wedge \cdots \wedge \pm\mathcal{E}_T(x), \qquad \text{where } i \in [1..S].$$

where the environment prediates $\mathcal{E}_i(x)$ are constructed as before. But, the abstract model constructed using descriptions $\Delta(x)$ given above will not have enough detail to verify a protocol without the atomicity assumption. Therefore, we augment our abstract states so that, in addition to the state of the reference process and its environment, they also contain the *history information* contained in the monitor processes. Our augmented abstract states will be of the form

$$\langle \mathbf{pc}, e_1, \ldots, e_T, t_1, \ldots, t_T, b_1, \ldots, b_T, te \rangle$$

where the variables $t_1, \ldots, t_T$ and te (called *trackers*) abstract the monitor variables of the reference process, $b_1, \ldots, b_T$ (called *backers*) abstract the monitor variables of the environment processes. We now describe how to abstract the different monitor variables.

Abstracting Trackers

Consider first the reference process x. Apart from the reference process, no other process is individually identifiable in the abstract state. Corresponding to each environment condition $\mathcal{E}_i$, we have an abstract variable t_i with range $\{\mathsf{clean}, \mathsf{dirty}, \mathsf{both}\}$ which abstracts the information present in the monitors. The value of t_i is computed as follows:

- If for all the processes y satisfying environment predicate $E_i(x, y)$ the variable $m(x, y) =$ clean, then $t_i =$ clean.

- If for all the processes y satisfying environment predicate $E_i(x, y)$ the variable $m(x, y) =$ dirty, then $t_i =$ dirty.

- Otherwise $t_i =$ both.

Given a concrete state s of a system $\mathcal{P}(K)$ with reference process x and an environment $\mathcal{E}_i$, the value of *tracker* t_i is uniquely determined. We denote the function from (s, x, i) to t_i by $\mathcal{F}^t$. This function will be used later on.

In addition, we have another variable te that abstracts the monitor variable $em(x)$. The value of te is exactly the same as value of $em(x)$.

Abstracting Backers

Trackers maintain history information that is relevant for the reference process. We also need to abstract the information present in the monitors for processes other than the reference process x. In particular, for each environment process y, we are interested in the monitor variable $m(y, x)$. As noted earlier, environment processes are grouped according to the environment condition they satisfy. For an environment $\mathcal{E}_i$, we maintain a variable bc_i that *combines* the $m(y, x)$ variables of all processes y in the environment $\mathcal{E}_i$. The value of bc_i is computed as follows:

- If for all processes y satisfying $E_i(x, y)$ we have $m(y, x) =$ clean, then $bc_i =$ clean.

- If for all processes y satisfying $E_i(x, y)$ we have $m(y, x) = \mathsf{dirty}$, then $bc_i = \mathsf{dirty}$.

- Otherwise $bc_i = \mathsf{both}$.

Given a concrete state s of a system $\mathcal{P}(K)$ with reference process x and an environment $\mathcal{E}_i$, the value of bc_i is uniquely determined. We can denote the function from (s, x, i) to bc_i by $\mathcal{F}^b$. This function will be used later on.

We will now define the abstraction mapping from augmented concrete states to augmented abstract states.

Definition 5.4.3 (Abstraction Mapping). Let $P(K)$, $K > 1$, be a concrete system and $p \in [1..K]$ be a process. The abstraction mapping α_p induced by p *maps* a global state s of $\mathcal{P}(K)$ to an abstract state $\langle \mathbf{pc}, e_1, \ldots, e_T, t_1, \ldots, t_T, b_1, \ldots, b_T, te \rangle$ where

- $\mathbf{pc} =$ the value of $\mathsf{pc}[p]$ in state s and for all e_j, we have $e_j = 1 \Leftrightarrow s \models \mathcal{E}_j(p)$.

- For all j we have $t_j = \mathcal{F}^t(s, p, j)$, and for all j, we have $b_j = \mathcal{F}^b(s, p, j)$

The corresponding augmented abstract model $\mathcal{P}^{\mathcal{A}}$ is defined as in Section 2.5.2. The set of labels is the same as the labels used in Section 4.2. From the coverage and congruence properties of the original abstract descriptions we can conclude that the same properties hold for the augmented abstract descriptions as well. Thus, the following corollary follows from Theorem 2.2.4.

Corollary 5 (Soundness of Augmented Abstraction). Let $\mathcal{P}(\mathbf{N})$ be a parameterized mutual exclusion system, $\mathcal{PM}(\mathbf{N})$ be an augmentation of $\mathcal{P}(\mathbf{N})$ with monitor processes

as described above, and $\mathcal{PM}^{\mathcal{A}}$ be its augmented abstraction. For an indexed property $\forall x.\Phi(x)$, where $\Phi(x)$ is a control condition we have

$$\mathcal{PM}^{\mathcal{A}} \models \Phi(x) \Rightarrow \forall K.\mathcal{PM}(K) \models \forall x.\Phi(x) \Rightarrow \forall K.\mathcal{P}(K) \models \forall x.\Phi(x)$$

5.5 Computing the Abstract Model

As in the atomicity case, we consider the following four cases for computing an over-approximation of a transition statement:

Active process is ...	guarded transition	update transition
... reference process	Case 1	Case 2
... environment process	Case 3	Case 4

Before we begin, we recall the notation introduced earlier in Section 4.5. The environment condition $E_i(x,y) \doteq y \neq x \wedge R_j(x,y) \wedge \mathsf{pc}[y] = L$ is denoted by $E_{(j,L)}(x,y)$. The corresponding environment predicate is referred to as $\mathcal{E}_{(j,L)}(x)$ and the corresponding abstract variable is $e_{(j,L)}$. The set of all environment conditions $E_{(j,L)}(x,y)$ is referred to as E_L.

5.5.1 Case 1: Guarded Transition for Reference Process

Let us now turn to Case 1 and consider the guarded transition t_G:

$$L_1: \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2 \textbf{ else goto } L_3$$

Suppose at least one of the trackers t_i, $i \in [1..T]$ is not clean. Then the reference process cannot conclude the guard is true. If all the trackers are clean, then we may conclude that the guard is true or false. Once reference process x ends up in a new control location, we have to appropriately assign new values to the trackers and the backers. To do this we need the following two definitions. The first definition is exactly the same as the one in Chapter 4, but we repeat it for the sake of completeness.

Definition 5.5.1 (Blocking Set for Reference Process). Let $\mathscr{G} \doteq \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ be a guard. We say that an environment condition $E_i(x, y)$ blocks the guard $\mathscr{G}$ if $E_i(x, y) \Rightarrow \neg\mathcal{G}(x, y)$. The set $\mathcal{B}^x(\mathscr{G})$ of all indices i such that $E_i(x, y)$ blocks $\mathscr{G}$ is called the blocking set of the reference process for guard $\mathscr{G}$.

Note that either $E_i(x, y) \Rightarrow \neg\mathcal{G}(x, y)$ or $E_i(x, y) \Rightarrow \mathcal{G}(x, y)$ holds for every environment $E_i(x, y)$.

Each environment $\mathcal{E}_i$ uniquely determines the control location of the processes satisfying it. We will assume, for simplicity, that there is only one transition starting at each control location [2]. Thus, each environment $\mathcal{E}_i$ has an unique guard or update expression associated with it. The following notion of *dependent environments for guarded transitions* is similar to the *blocking set* for the reference process.

Definition 5.5.2 (Dependent Set for Guards). Let $\mathsf{pc} = L$ be a control location of the reference process. The guard dependent set of L, $\mathcal{D}^g(L)$ contains all those environments whose associated guard $\mathscr{G} \doteq \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ are such that $E_i(y, x) \wedge \mathcal{G}(y, x) \wedge (\mathsf{pc}[x] = L)$ is satisfiable.

[2] Extension to the general case is simple.

Intuitively, the guard dependent set of a control location $\mathsf{pc} = L$ is the set of all those environments whose associated guards are such that the reference process x in control location $\mathsf{pc} = L$ does not contradict the guards. Thus, a process y present in any such environment *could* have seen the reference process x satisfy process y's guard. We define *update dependent sets* similarly.

Definition 5.5.3 (Dependent Set for Updates). Let $\mathsf{pc} = L$ be a control location of the reference process. The update dependent set of L, $\mathcal{D}^u(L)$, contains those environments whose associated update expression updates the same data variable as the update transition associated with L. If there is no update transition associated with $\mathsf{pc} = L$ then the set is empty.

We will now explain how to abstract the guarded transition t_G

$$L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2 \textbf{ else goto } L_3.$$

We will represent the set of abstract transition arising from this case by an invariant (between current and next states) $I^x_{t_G}$. The invariant, structured similar to the one in Section 4.5, will be presented in terms of three conditions **GR1**, **GR2**, **GR3**. The abstract model will have transition from $\hat{s_1} = \langle \mathbf{pc}, e_1, .., e_T, t_1 \ldots, t_T, bc_1, \ldots, bc_T, te \rangle$ to $\hat{s_2} = \langle \mathbf{pc'}, e_1, .., e_T, t'_1 \ldots, t'_T, bc'_1, \ldots, bc'_T, te' \rangle$ if

GR1. $\mathbf{pc} = L_1$, i.e., the reference process is in location L_1,

GR2. One of the following two conditions holds:

- *Then Branch:* $\forall i).(t_i = \mathsf{clean})$ and $\mathbf{pc}' = L_2$, i.e., the guard is true and the reference process moves to control state L_2.
- *Else Branch:* $\neg(\forall i.t_i = \mathsf{clean}) \vee (\forall i.t_i = \mathsf{clean} \wedge te = \mathsf{dirty})$ and $\mathbf{pc}' = L_3$, i.e., the guard is false and the reference process moves to control state L_3. Note that the condition $te = \mathsf{dirty}$ indicates that at least one tracker was dirty at some point in the past.

GR3 Assuming $\mathbf{pc}' = L$ where $L \in \{L_2, L_3\}$ the following conditions hold.

- If the transition associated with control location L is a guarded transition with $\mathscr{G} \doteq \forall \mathsf{otr} \neq \mathsf{slf}.\mathscr{G}(\mathsf{slf}, \mathsf{otr})$, then the following conditions hold.
  - If $i \in \mathcal{B}^x(\mathcal{G})$ or $e'_i = 0$ then $t'_i = \mathsf{clean}$. Else $t'_i = \mathsf{dirty}$, i.e., if $\mathcal{E}_i$ is a not blocking environment or if the environment is empty the corresponding tracker is clean. Otherwise, it is set to dirty as there is a process in a blocking environment.
  - If $i \in \mathcal{D}^g(L) \cup \mathcal{D}^u(L)$, then $bc'_i = \mathsf{clean}$ else $bc'_i = bc_i$. i.e., if the reference process does not block the guard associated with environment $\mathcal{E}_i$ or updates the same variable as the update transition associated with $\mathcal{E}_i$ then bc_i is set to clean, otherwise it is set to dirty.
  - If there exists an i such that $t'_i = \mathsf{dirty}$ then $te = \mathsf{dirty}$, i.e., the variable te is dirty if at least one of the trackers is dirty.
- If the transition associated with control location L is an update transition then the following conditions hold

- If $i \in \mathcal{D}^u(L)$ and $e'_i = 1$ then $t'_i =$ clean. Else $t'_i =$ dirty. That is, if environment e_i updates the same data variable as the reference process in control location L, then the tracker t_i must be set to clean otherwise t_i is set to dirty. This indicates that both the reference process and a process in e_i can change the same data variables simultaneously.
- If $i \in \mathcal{D}^g(L)$, then $bc'_i =$ clean else $bc'_i = bc_i$. That is, if control location L is such that the guard associated with e_i is not blocked by the reference process in control location L, then the backer bc_i is set to clean.
- $te' =$ clean. This is a default value for te as it is not really used for update transitions.

Similar to the concrete monitor variables, the value clean for trackers and backers is the most permissive –that is, if backers and trackers are clean, then the possible set of transitions is maximal. The value both is slightly more restrictive than clean: the environments corresponding to trackers and backers that are in the both state cannot be empty. The value dirty is the most restrictive. A tracker being dirty prevents the reference process from moving forward. Similarly, a backer being dirty prevents the processes of the corresponding environment from moving forward.

Lemma 5.5.4. *If states s_1 and s_2 in a concrete system $\mathcal{P}(K)$, $K > 1$ are such that $\alpha_c(s_1) = \hat{s}_1$ and $\alpha_c(s_2) = \hat{s}_2$ and there is a transition from s_1 to s_2 via process c executing a guarded transition t_G then $\hat{s}_1$ and $\hat{s}_2$ satisfy the transition invariant $I^x_{t_G}$.*

Proof. This follows simply from the way we constructed the invariant. □

Note that all we have done is to *translate* the lemmas listed in Section 5.4 in terms of the reference process and its environment. This is precisely where the power of this approach comes from. Constructing the abstract model is theoretically simple and it is easily extendible in case new constructs are allowed in the concrete protocols.

5.5.2 Case 2: Guarded Transition for Environment Processes

Suppose that the guarded transition t_G

$$L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2 \textbf{ else goto } L_3$$

is executed by a concrete process y satisfying the environment condition $E_{(i,L_1)}(x, y)$. The active process thus switches from environment condition $E_{(i,L_1)}(x, y)$ to environment condition $E_{(i,L_2)}(x, y)$ or $E_{(i,L_3)}(x, y)$. Note that in a guarded transition, only the pc of the active process changes.

We denote the abstract transition corresponding to this case by an invariant $I_i^y(t_G)$. We introduce an abstract transition from $\hat{s_1} = \langle \mathbf{pc}, e_1, .., e_T, t_1. \ldots, t_T, bc_1, \ldots, bc_T, te \rangle$ to $\hat{s_2} = \langle \mathbf{pc'}, e'_1, .., e'_T, t'_1. \ldots, t'_T, bc'_1, \ldots, bc'_T, te \rangle$ if the following conditions hold. For brevity we will represent the environment condition $\mathcal{E}_{(i,L_1)}$ by $\mathcal{E}_1$, $\mathcal{E}_{(i,L_2)}$ by $\mathcal{E}_2$, $\mathcal{E}_{(i,L_3)}$ by $\mathcal{E}_3$.

GE1. $e_1 = 1$, that is, there is an environment process in state control location L_1 [3].

[3]The requirement that, for each control location L, there be only one transition starting from L is being

GE2 One of the following two conditions holds:

- *Then Branch:* $bc_k \in \{\mathsf{clean}, \mathsf{both}\}$ and $e_2' = 1$, i.e., the guard is true and the reference process moves to control state L_2. e_1' can be 0 or 1.
- *Else Branch:* $e_3' = 1$, i.e., the guard is false and the environment process moves to control state L_3. e_1' can be 0 or 1.

GE3. $\mathbf{pc}' = \mathbf{pc}$. That is, the control location of the reference does not change.

GE4. Let the new control location of the environment process be $L \in \{L_2, L_3\}$. Denote the environment $E_{(i,L)}$ by E_j for the sake of brevity. The following conditions must hold:

- $t_j' = \omega(t_1, t_j)$. Function ω, described below, takes the current values of the trackers t_1, t_j and returns the new value for t_j.
- $t_1' = \Omega_t(t_1, e_1')$. Function Ω_t, described below, takes the current value of a tracker (or a backer) and the next state value of the corresponding environment and returns the next state value of the tracker (or the backer).
- $bc_1' = \Omega_t(bc_1, e_1')$.
- If the transition associated with L is a guarded transition then $bc_j' = \Omega_b(\mathcal{D}^g(L), bc_j)$. Function Ω_b finds the new value of backer bc_j as a function of the current value of the backer bc_j and the guard dependent set of control location L.

used here.

- If the transition associated with L is an update transition, then $bc'_j = \Omega_b(\mathcal{D}^u(L), bc_j)$.

Function $\omega(t_i, t_j)$ is shown in tabular form in Figure 5.10

Function $\omega(t_i, t_j)$ returns the new value of t_j given the current values of t_i and t_j.

t_j	t_i	$\omega(t_i, t_j)$
$t_j =$ clean	$t_i =$ dirty	$t'_j =$ both
$t_j =$ dirty	$t_i =$ dirty	$t'_j =$ dirty
$t_j =$ clean	$t_i =$ clean	$t'_j =$ clean
$t_j =$ dirty	$t_i =$ clean	$t'_j =$ both
$t_j =$ clean	$t_i =$ both	$t'_j =$ both
$t_j =$ dirty	$t_i =$ both	$t'_j =$ dirty
$t_j =$ both	-	$t'_j =$ clean

Figure 5.10: Function ω.

Informally, this new value of t_j should reflect the collective status of processes in the environment e_j. When a new process moves into the environment e_j, we can figure out the status of this new process by looking at the tracker value associated with its old environment. Depending on these two values, the current values of t_j and t_i, we can figure out the new value of t_j so that it reflects the collective condition of the processes in the environment e_j.

Function $\Omega_t(e'_i, t_i)$ is shown in Figure 5.11. The function code is self-explanatory as is the function $\Omega_b(Set, bc_j)$ given in Figure 5.5.2

Function $\Omega_t(e'_i, t_i)$ returns the new value of the tracker t_i given the current value of the tracker and the next value of the corresponding environment bit e_i.

- If $e'_i = 0$ then $t'_i = \mathsf{clean}$
- Otherwise
  - If $t_i = \mathsf{clean}$ then $t'_i = \mathsf{clean}$
  - If $t_i = \mathsf{dirty}$ then $t'_i = \mathsf{dirty}$
  - If $t_i = \mathsf{both}$ then $t'_i \in \{\mathsf{clean}, \mathsf{dirty}, \mathsf{both}\}$

Figure 5.11: Function Ω_t.

Lemma 5.5.5. *If states s_1 and s_2 in a concrete system $\mathcal{P}(K)$, $K > 1$ are such that $\alpha_c(s_1) = \hat{s}_1$ and $\alpha_c(s_2) = \hat{s}_2$, $c \in [1..K]$ and there is a transition from s_1 to s_2 via process $d \neq c$ executing a guarded transition t_G, then $\hat{s}_1$ and $\hat{s}_2$ satisfy the transition invariant $I_y(t_G)$.*

Proof. The proof of this lemma follows directly from the way we constructed $I_y(t_G)$. □

5.5.3 Case 3: Update Transition for Reference Process

Consider the case where the reference process is executing an update transition t_U:

Function Ω_b takes set of environment conditions and one backer as its arguments and returns the new value of the backer.

- If $E_j \in Set$ then $bc'_j = \mathsf{clean}$
- $E_j \notin Set$ then one of the following holds:
  - If $bc_j = \mathsf{clean}$ or $bc_j = \mathsf{both}$ then $bc'_j = \mathsf{both}$
  - If $bc_j = \mathsf{dirty}$ then $bc'_j = \mathsf{dirty}$

Figure 5.12: Function Ω_b

$$L_1: \quad \textbf{for all } \mathsf{otr} \neq \mathsf{slf} \quad \textbf{if } \mathcal{T}(\mathsf{slf}, \mathsf{otr}) \textbf{ then } u_k := \Phi(\mathsf{otr})$$
$$\textbf{goto } L_2$$

Recall that each process has data variables $u_1, \ldots, u_d$. We denote the next state value of each variable u_m by u'_m.

When the reference process x changes the value of its data variables, the valuations of the environment predicates $\mathcal{E}_1(x) \ldots \mathcal{E}_T(x)$ will change. For a process y satisfying environment condition $E_i(x, y)$, we need to figure out the possible new environment conditions $E_j(x, y)$ that y will satisfy after the reference process x has executed the update transition. Recall from Chapter 4 that the set of possible new environment conditions for process y satisfying the condition $E_i(x, y)$ is called the *outset* O_i. (Technically, the outset is the set of the indices of these environment conditions.) For sake of completeness, we will briefly

explain again how to compute the outset.

Case A.

The first case we need to consider is when process y does not update u_k while the reference process is updating its variable. Denote by $\mathbf{T}(x, y)$ the *update* formula $\mathcal{T}(x, y) \wedge u'_k[x] := \phi(y) \vee \neg\mathcal{T}(x, y) \wedge u'_k[x] := u_k[x]$. Given the update formula, we find what possible inter-predicates involving $u'_k[x], u_k[y]$ can be true. Formally, the set $C^1(u_k)$ of these inter-predicates is given by all formulas $u_k[x] \prec u_k[y]$ where $\prec \in \{<, >, =\}$ such that

$$u'_k[x] \prec u_k[y] \wedge \mathbf{T}(x, y) \quad \text{is satisfiable.}$$

The possible relationships between $u_k[x]$ and $u_k[y]$ might change when x repeatedly updates its u_k value by looking at other processes. Suppose x looks at another process z and updates its $u_k[x]$ value again. We now find the set $C^2(u_k)$ of possible relationships between $u_k[x]$ and $u_k[y]$ *after* the new update involving process z under the assumption that a relation from $C^1(u_k)$ holds *before* the update. Thus, the new set $C^2(u_k)$ of relationships is given by all formulas $u_k[x] \prec u_k[y]$ where $\prec \in \{<, >, =\}$ such that

$$u'_k[x] \prec u_k[y] \wedge \mathbf{T}(x, z) \wedge \psi(x, y) \text{ is satisfiable and } \psi(x, y) \in C^1(u_k).$$

Note that $C^1(u_k) \subseteq C^2(u_k)$ because the definition of $\mathbf{T}(x, z)$ allows the possibility that the value of $u_k[x]$ remains unchanged. We similarly compute sets $C^3(u_k), C^4(u_k) \ldots$ until a fixpoint, $C(u_k)$, is reached.

In the environment condition $E_i(x, y)$, let θ be the (unique) inter-predicate that describes the relation between $u_k[x]$ and $u_k[y]$. Consider the set of environment conditions

$E_j(x,y)$ that are obtained from $E_i(x,y)$ by replacing θ with a formula in the fixpoint $C(u_k)$: the indices of these environment conditions constitute the *outset* O_i of $E_i(x,y)$.

Correspondingly, the *inset* $I_k \subseteq \{1..T\}$ for environment condition $E_k(x,y)$ consists of all j such that $k \in O_j$.

Case B.

In the second case, process y is also updating its u_k variable. In this case, the set $C(u_k)$ is the set of all possible predicates involving $u_k[x]$ and $u_k[y]$. In other words, we cannot predict what the relationship between $u_k[x]$ and $u_k[y]$ is. The outset consisting of [the indices of] environments is then computed as described.

In the abstract model, to compute the outset for environment e_m we use **Case A** if the associated tracker t_m is dirty; otherwise we use **Case B**. Observe that, if the tracker is clean, more behaviors are possible.

Denote the set of abstract transitions corresponding to the concrete update transition (†) by $I_x(t_U)$. $I_x(t_U)$ has a transition from $\hat{s_1} = \langle \mathbf{pc}, e_1, .., e_T, t_1 \ldots, t_T, bc_1, \ldots, bc_T, te \rangle$ to $\hat{s_2} = \langle \mathbf{pc'}, e'_1, .., e'_T, t'_1 \ldots, t'_T, bc'_1, \ldots, bc'_T, te \rangle$ if the following conditions hold:

UR1. $\mathbf{pc} = L_1$, i.e., the reference process first is in control location L_1.

UR2. $\mathbf{pc'} = L_2$, i.e., the reference process moves to control location L_2.

UR3. $\forall k \in [1..T].(e_k = 1 \Rightarrow \exists j \in O_k.e'_j = 1)$, i.e., if there was a process in environment $\mathcal{E}_k(x)$ before the transition, then there must be a process in one of the outset environments O_k of $E_k(x,y)$ after the transition.

UR4. $\forall j \in I_k.(e_j = 0 \Rightarrow e'_k = 0)$, i.e., if there is no process satisfying the *inset* environments I_k of environment $E_k(x, y)$ before the transition then after the transition there can be no process in environment $E_k(x, y)$.

UR5. for each $k \in [1..T]$, the value of bc'_k is computed as follows

- if $e'_k = 0$ or $k \in \mathcal{D}^g(L_2) \cup \mathcal{D}^u(L_2)$ then $bc'_k = \mathsf{clean}$
- otherwise we have three cases:
  - if $\forall j \in I_k.bc_j = \mathsf{clean}$ then $bc'_k = \mathsf{clean}$
  - if $\forall j \in I_k.bc_j = \mathsf{dirty}$ then $bc'_k = \mathsf{dirty}$
  - if $\exists j \in I_k.bc_j = \mathsf{clean}$ and $\exists j \in I_k.bc_j = \mathsf{dirty}$ then bc'_k can take any value in $\{\mathsf{clean}, \mathsf{dirty}, \mathsf{both}\}$

UR6. For each $k \in [1..T]$ if $k \in \mathcal{D}(()L_2)$ then $t'_k = \mathsf{clean}$ else $t'_k = \mathsf{dirty}$ where $\mathcal{D}(L_2)$ is either $\mathcal{B}^x(\mathcal{G})$, if the transition associated with L_2 is a guarded transition with guard condition $\mathcal{G}$, or $\mathcal{D}(L_2)$ is $\mathcal{D}^u(L_2)$, if the transition associated with L_2 is an update transition.

Lemma 5.5.6. *If states s_1 and s_2 in a concrete system $\mathcal{P}(K)$, $K > 1$, are such that $\alpha_c(s_1) = \hat{s}_1$ and $\alpha_c(s_2) = \hat{s}_2$, with $c \in [1..K]$ and there is a transition from s_1 to s_2 via process c executing a guarded transition t_U then $\hat{s}_1$ and $\hat{s}_2$ satisfy $I_x(t_G)$.*

Proof. The proof of this lemma follows directly from the way we constructed $I_x(t_G)$. □

5.5.4 Case 4: Update Transition for Environment Processes

Consider the case where a generic process y satisfying environment $E_{(i,L_1)}(x,y)$ is executing an update transition t_U:

$$L_1 : \quad \textbf{for all } \mathsf{otr} \neq \mathsf{slf} \quad \textbf{if } \mathcal{T}(\mathsf{slf}, \mathsf{otr}) \textbf{ then } u_k := \Phi(\mathsf{otr})$$
$$\textbf{goto } L_2$$

After the update transition, process y will have a new control location and also the relationship of its data variables to those of the reference process x will have changed. Recall the notation $E_{(i,L_1)}(x,y)$ used to denote the environment condition $y \neq x \wedge R_i(x,y) \wedge \mathsf{pc}[y] = L_1$. The *outset* $O_{(i,L_1)}$ for environment $E_{(i,L_1)}(x,y)$ will consist of all those environments $E_{(j,L_2)}(x,y)$ that process y may satisfy after the update transition. To compute the outset $O_{(i,L_1)}$ we proceed as follows. As in the previous case we find a fixpoint $C(u_k)$ that contains the possible relationships between $u_k[x]$ and $u_k[y]$.

Case A.

Consider first the case where the reference process is not updating its variable u_k. The initial set of relationships $C^1(u_k)$ is the set of all $u_k[x] \prec u_k[y]$, $\prec \in \{<, >, =\}$ such that

$$\mathbf{T}(y,x) \wedge u_k[x] \prec u'_k[y] \text{ is satisfiable}$$

where $\mathbf{T}(y,x)$ is the update condition as defined in Section 5.5.3. Note that we consider $\mathbf{T}(y,x)$ (and not $\mathbf{T}(x,y)$ as in the previous section) because y is the active process. As

y updates its u_k variable repeatedly, the relationship between $u_k[x]$, and $u_k[y]$ will also change. To compute all the possible relationships, we use an approach similar to the fixpoint computation in Case 3. Thus, we find the set $C^2(u_k)$ of all $u_k[x] \prec u_k[y]$, $\prec \in \{<, >, =, \}$ such that

$$\mathbf{T}(y, z) \wedge u_k[x] \prec u'_k[y] \wedge \psi(x, y) \text{ is satisfiable}$$

where $\psi(x, y) \in C^1(u_k)$. We similarly compute $C^3(u_k), C^4(u_k), \ldots$ until we reach a fixpoint $C(u_k)$.

Case B.

Consider now the case where the reference process is also updating its u_k variable. In this case, $C(u_k)$ will consist of all possible relations on $u_k[x]$ and $u_k[y]$, denoting that we do not have enough information.

In the environment condition $E_{(i,L_1)}(x, y)$, let θ be the (unique) inter-predicate that describes the relation between $u_k[x]$ and $u_k[y]$. Consider the set of environment conditions $E_{(j,L_2)}(x, y)$ that are obtained from $E_{(i,L_1)}(x, y)$ by replacing θ by a formula in the fixpoint $C(u_k)$ and replacing the condition $\mathsf{pc}[y] = L_1$ by $\mathsf{pc}[y] = L_2$. The indices of these environment conditions constitute the *outset* $O_{(i,L_1)}$ of $E_{(i,L_1)}(x, y)$.

To compute the outset for an environment $e_{(i,L_1)}$, we will use **Case A** if the associated backer $bc_{(i,L_1)}$ is dirty. Otherwise, we use **Case B**. Note again that $bc_{(i,L_1)}$ being clean or both allows more behaviors.

Since the transition starts at control location L_1 and a generic process executes it, we will describe the abstract transition $I_y^{i,k}(t_U)$ for each environment condition $E_{(i,L_1)}(x, y)$

and each $k \in O_{(i,L_1)}$. The abstract transition $I_y(t_U)$ for Case 4 will be

$$\bigvee_{E_{(i,L_1)}(x,y)} \quad \bigvee_{k \in O_{(i,L_1)}} \quad I_y^{i,k}(t_U).$$

$I_y^{i,k}(t_U)$ has a transition from $\hat{s_1} = \langle \mathbf{pc}, e_1, .., e_T, t_1 \ldots, t_T, bc_1, \ldots, bc_T, te \rangle$ to $\hat{s_2} = \langle \mathbf{pc'}, e'_1, .., e'_T, t'_1 \ldots, t'_T, bc'_1, \ldots, bc'_T, te \rangle$ if the following conditions hold. For brevity we will represent the environment condition $\mathcal{E}_{(i,L_1)}$ by $\mathcal{E}_1$ and $\mathcal{E}_{(j,L_2)}$ by $\mathcal{E}_2$.

UR1. $\mathbf{pc} = \mathbf{pc'}$, i.e., the reference process does not move.

UR2. $e_1 = 1$, i.e., there is a process in environment $E_{(i,L_1)(x,y)}$ before the transition.

UR3. $e'_2 = 1$, i.e., there is a process in environment $E_{(j,L_2)(x,y)}$ after the transition.

UR4. The e variables except e'_1, e'_2 do not change, i.e., $e'_l = e_l$ for $l \notin \{(i, L_1), (j, L_2)\}$.

UR5. Assuming the new control location of the environment process that moved was $L \in L_2, L_3$, denote the environment $E_{(i,L)}$ by E_j. The following conditions must hold:

- $t'_2 = \omega(t_1, t_2)$
- $t'_1 = \Omega_t(e_1, t_1)$
- $bc'_1 = \Omega_t(e_1, bc_1)$
- If the transition associated with L is a guarded transition $bc'_j = \Omega_b(\mathcal{D}^g(L), bc_i, bc_j)$.
- If the transition associated with L is an update transition $bc'_j = \Omega_b(\mathcal{D}^u(L), bc_i, bc_j)$.

	Inter-preds	Intra-preds	Reachable states	Safety
Bakery (NA)	3	5	$O(2^{400})$	3800s
Bakery (A)	3	5	$O(2^{146})$	68.55s

Figure 5.13: Running Times for the bakery protocol. Bakery(A) and Bakery(NA) stand for the bakery protocol with and without the atomicity assumption

Lemma 5.5.7. *If states s_1 and s_2 in a concrete system $\mathcal{P}(K)$, $K > 1$, are such that $\alpha_c(s_1) = \hat{s}_1$ and $\alpha_c(s_2) = \hat{s}_2$, with $c \in [1..K]$ and there is a transition from s_1 to s_2 via process $d \neq c$, executing a guarded transition t_U then $\hat{s}_1$ and $\hat{s}_2$ satisfy $I_y(t_U)$.*

Proof. The proof of this lemma follows directly from the way we constructed $I_y(t_U)$. □

5.6 Experimental Results

We applied our abstraction method to the Bakery and Szymanski's protocols without the atomicity assumption. We were able to verify the safety property of the Bakery protocol, namely

$$\forall x.\forall y \neq x.\mathbf{AG}(\mathsf{pc}[x] = \mathbf{crit} \Rightarrow \mathsf{pc}[y] \neq \mathbf{crit})$$

in about 2 hours. The following table shows the run times and other statistics in the non-atomic case and the same verification carried out under the atomicity assumption

Note the enormous increase in the state space size once we remove the atomicity assumption. The increase in the model checking is equally dramatic. This again underlines

the significant reduction in complexity of protocols due to the atomicity assumption.

We were not able to verify the safety property of Szymanski's protocol. The correctness of Szymanski's protocol depends on the specific order in which a process looks at the other processes in the system. Szymanski's protocol is correct only if a process looks at the other processes in the increasing order of the index [55; 77]. The semantics we assigned to our guarded and update transitions was such that the order of processes was immaterial. Hence we cannot accurately model Szymanski's protocol in our input language.

We also applied our abstraction to the toy protocol described in Section 5.3. As expected, our method finds a trace violating the mutual exclusion protocol in under 5 mins.

Chapter 6

Verification by Network Decomposition

6.1 Introduction

Despite the big success of model checking in hardware and software verification, the classical approach to model checking can handle only finite state systems. Consequently, applying model checking techniques to systems involving unlimited concurrency, unlimited memory, or unlimited domain sizes, is a major challenge. Researchers have sought to address these issues by different verification methods including, among others, abstraction, regular model checking, static analysis, and theorem proving.

Many software and hardware systems, however, are described in terms of natural parameters and, for each concrete value of the parameters, the systems have a finite state

space. Verifying a property of a parameterized system amounts to verifying this property for all values of the parameters. Examples of parameterized systems include mutual exclusion protocols, cache coherence protocols, and multi-threaded systems.

While there has been considerable effort in verifying parameterized systems such as cache protocols and mutual exclusions, that have replicated but no underlying network graphs, there is little work on parameterized systems that have replicated process and underlying network graphs. Common examples of systems that are required to operate on arbitrary network topologies are network routing protocols. Leader election protocols, for example, are usually designed to operate no matter what the underlying network topology of the system. Verifying such systems is obviously complicated by the fact that the network graph can be arbitrary (in addition to the fact the network graph induces asymmetry in the system).

In a seminal paper, Emerson and Namjoshi [37] consider systems composed of identical asynchronous processes which are arranged in a ring topology and communicate by passing a Boolean token. For several classes of indexed $CTL^* \setminus X$ properties [15] they provide *cutoffs*, i.e., reductions to single systems of constant small size. Consequently, $CTL^* \setminus X$ properties over an *infinite class of networks* can be reduced to a *single model checking call*.

In this chapter, we extend the results of Emerson and Namjoshi from rings to *arbitrary classes of networks*. There are two modifications, however: first, our results hold true only for LTL$\setminus X$, and second, we introduce a more refined notion of cut-offs. The first restriction is necessary: We show in Section 4 that with $CTL \setminus X$ it is *impossible* to obtain

cut-offs for arbitrary networks.

The second modification actually provides an interesting new view on the notion of cut-offs: in order to verify the parametrized system, we are allowed to model check a *constant number* c of small systems whose network graphs have sizes bounded by a *constant* s. Then, the verification result for the parametrized system is a Boolean combination of the collected results for the small systems. We call such a reduction to a finite case distinction a (c, s)-bounded reduction.

Our main results can be summarized as follows:

- **Verification by Network Decomposition:** Verifying systems with fixed large network graphs G (e.g., concrete instantiations of a parametrized system) can be as challenging as verifying parameterized systems. Note that when $|Q|$ is the state space of the individual processes, then the state space of the whole network can be as high as $|Q|^n$, where n is the number of nodes. We show that the verification of an indexed LTL$\setminus X$ property φ for a system with network graph G can be achieved by an *efficiently computable* (c, s)-bounded reduction. For the important case of 2-indexed properties, it is sufficient to model check at most 36 networks of size 4.

- **Offline Verification:** In a scenario where φ is known in advance and the network G can change for different applications, we can first verify a constant number of small systems *offline*. Later, when we get to know the network graph G, the correctness of G with respect to specification φ can be verified *online* by simply evaluating a constant size Boolean function, regardless of the size of the processes.

Again, for 2-indexed properties, the offline computation involves at most 36 calls to the model checker for networks of size 4.

- **Cut-Offs:** For every class of networks $\mathbb{T}$ and k-indexed LTL$\setminus X$ property φ one can verify if φ holds on *all* networks in $\mathbb{T}$ by a (c, s)-bounded reduction, where c and s depend only on k.

  Depending on the complexity of the networks in $\mathbb{T}$, finding a suitable (c, s)-bounded reduction will in general still involve manual algorithm design. Similar to famous results about linear time algorithms for bounded tree-width [25], our proofs just guarantee the existence of small reductions.

Our results lay the foundation for reasoning about systems with arbitrary network graphs. While communication between the processes is simple, the results we obtain are non-trivial. In fact, we were surprised to discover that for $CTL \setminus X$ specification there are no cutoffs even for the simple communication model. The generalized notion of cutoffs we present will be crucial to reasoning about systems with more complicated communication.

This chapter is organized as follows: the next section contains the work closest to our work. In Section 3, we describe the system model in detail. Section 4 contains the main cutoff results. Section 5 shows that no cutoffs exist for $CTL \setminus X$. Finally, the conclusion in Section 5 briefly considers further performance enhancements for practical applications of our method.

6.2 Related Work.

Verification of parameterized systems is well known to be undecidable [2; 76]. Many interesting approaches to this problem have been developed over the years, including the use of symbolic automata-based techniques [1; 10; 12; 13; 51; 78], network invariants [3; 64], predicate abstraction [52; 53], and symmetry reduction [24; 31; 38; 39; 40]. In [11], cut-offs were used for the verification of systems sharing common resources, where the access to the resources is managed according to a FIFO-based policy.

In addition to [37] mentioned above, Emerson et al. have shown a large number of fundamental results involving cut-offs. The paper [33] by Emerson and Kahlon also considers LTL\X cut-offs for arbitrary network topologies with multiple tokens, but each token is *confined to two processes* which renders their model incomparable to ours. Other previous work by Emerson and Kahlon [32; 34; 35] consider other restricted forms of process interaction. Finally, [43] considers the verification of single index properties for systems with multiple synchronous processes.

Indexed temporal logic was introduced in [15]. The paper also considers identical processes arranged in ring topology.

The work that is closest in spirit to our negative results on $CTL^* \setminus X$ logic is the work by Browne, Clarke and Grumberg in [14] that shows how to characterize Kripke structures up to bisimilarity using fragments of CTL*. Our results show that even $CTL^* \setminus X$ with only two atomic propositions is sufficient to describe an infinite class of Kripke structures that are not bisimilar to each other. In other words, bisimilarity over the class of Kripke structures with two labels gives rise to an infinite number of equivalence classes.

6.3 Computation Model

Network Topologies. A *network graph* is a finite directed graph $G = (S, C)$ without self-loops, where S is the set of *sites*, and C is the set of *connections*. Without loss of generality we assume that the sites are numbers, i.e., $S = \{1, 2, \ldots, |S|\}$. A (network) *topology* $\mathbb{T}$ is a class of network graphs.

Token Passing Process. A single token passing process P (process) is a *labeled transition system* (Q, Σ, δ, I) such that:

- $Q = \widehat{Q} \times B$, where $\widehat{Q}$ is a finite, nonempty set and $B = \{0, 1\}$. Elements of Q will be called *local states*. The boolean component of a local state indicates the possession of the token. We say that a local state (q, b) holds the token if $b = 1$.
- $\Sigma = \Sigma_f \cup \Sigma_d \cup \{\mathsf{rcv}, \mathsf{snd}\}$ is the set of actions. The actions in Σ_d are token dependent actions, those of Σ_f are called token independent actions, and $\{\mathsf{rcv}, \mathsf{snd}\}$ are actions to receive and send the token. The sets Σ_f, Σ_d are mutually exclusive.
- $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, such that every $((q, b), a, (q', b')) \in \delta$ fulfills the following conditions:
  (a) A free transition does not change token possession: $a \in \Sigma_f \Rightarrow b = b'$
  (b) A dependent transition can execute only if the process possesses the token: $a \in \Sigma_d \Rightarrow b = b' = 1$
  (c) A receive establishes possession of token: $a = \mathsf{rcv} \Rightarrow b = 0, b' = 1$
  (d) A send revokes the possession of token: $a = \mathsf{snd} \Rightarrow b = 1, b' = 0$

- $I \subseteq Q$ is the set of initial states.

Topological Composition. Let $G = (S, C)$ be a network graph and $P = (Q, \Sigma, \delta, I)$ be a single token process. Then P^G denotes the concurrent system containing $n = |S|$ instances of P denoted by P_s, $s \in S$. The only synchronization mechanism between the processes is the passage of a token according to the network graph G. Formally, the system P^G is associated with a transition system $(\mathcal{Q}, \Delta, \mathcal{I})$ defined as follows:

- $\mathcal{Q} = \{(q_1, \ldots, q_n) \in Q^n \mid \text{exactly one of the } q_i \text{ holds the token}\}$.
- $\Delta \subseteq \mathcal{Q}^{2n}$ is defined as follows: a transition $(q_1, q_2, \ldots, q_n) \rightarrow (q'_1, q'_2, \ldots, q'_n)$ is in Δ in one of two cases:
  (a) **Asynchronous Transition:** there exist an index $j \in \{1, \ldots, n\}$ and an action $a \in \Sigma_f \cup \Sigma_d$ such that $(q_j, a, q'_j) \in \delta$, and for all indices $i \neq j$ we have $q_i = q'_i$. In other words, only process P_j makes a transition (different from a send or receive).
  (b) **Token Transition:** there exist a network connection $(j, k) \in C$ in the network graph, such that $(q_j, \mathsf{snd}, q'_j) \in \delta$, $(q_k, \mathsf{rcv}, q'_k) \in \delta$, and $q_i = q'_i$ for all indices i different from j, k.
- $\mathcal{I} = \{(q_1, \ldots, q_n) \in I^n \mid \text{exactly one of the } q_i \text{ holds the token}\}$.

An *execution path* is considered fair if and only if every process P_i receives and sends the token infinitely often. We assume that every system P^G that we consider has fair paths. An

immediate consequence of the fairness condition is that a system P^G can have fair paths only if G is strongly connected.

We shall use indexed temporal logics, which can refer explicitly to the atomic propositions of each process P_i, to specify properties of the compound systems. For each local state q in Q we introduce propositional variables $q(1), \ldots, q(n)$. The atomic proposition $q(i)$ says that process P_i is in state q. Thus, for a global state g we define

$$g \models q(i) \qquad \text{iff in global state } g, \text{ process } P_i \text{ is in state } q.$$

Starting from this definition for atomic propositions, we can easily define common temporal logics such as CTL or LTL in a canonical way. Throughout this paper, we will assume that the path quantifiers **A** and **E** quantify over fair paths. Further we assume that LTL formulas are implicitly quantified by **E**. This restriction simplifies our proofs but does not restrict generality.

Example 6.3.1. The formula $\mathbf{G}(q(1) \Rightarrow \mathbf{F}q(2))$ says that whenever process P_1 is in state q then process P_2 will be in state q sometime in the future.

For increased expressibility we permit that in an atomic formula $q(x)$ the process index x is a variable (called *index variable*) which can take any value from 1 to $|S|$, the total number of processes. Thus, x can refer to arbitrary processes. We shall write $\varphi(x_1, \ldots, x_n)$ to indicate that the temporal formula φ depends on the index variables $x_1, \ldots x_n$. We can substitute the index variables in a formula $\varphi(x_1, \ldots, x_k)$ by integer values $i_1, \ldots, i_k$ in the natural way, and denote the resulting formula by $\varphi(i_1, \ldots, i_k)$.

In addition to substitution by constants, we can also quantify over the index variables $x_1, \ldots x_n$ using a prefix of existential and universal quantifiers with the natural seman-

tics. Such formulas are called quantified temporal formulas. For example, the formula $\forall x \exists y.\varphi(x, y)$ means "For all processes x there exists a process y, such that the temporal formula $\varphi(x, y)$ holds." A formula without quantifier prefix is called *quantifier-free*. If all index variables in a formula are bound by quantifiers we say that the formula is *closed*, and *open* otherwise. The quantifier-free part of a quantified formula is called the *matrix* of a formula.

Example 6.3.2. The formula $\exists x, y.\mathbf{G}(q(x) \Rightarrow \mathbf{F}q(y))$ says that there exist two processes P_x and P_y, such that whenever process P_x is in state q then process P_y will be in state q some time in future.

The formal semantics of this logic is straightforward and is omitted for the sake of brevity.

Definition 6.3.3 (k-indexed Temporal Formula). Let $\mathcal{L}$ be a temporal logic. A k-indexed temporal formula is a formula whose matrix refers to at most k different processes, i.e., there are at most k different constant indices and index variables.

6.4 Reductions for Indexed LTL\X Specifications

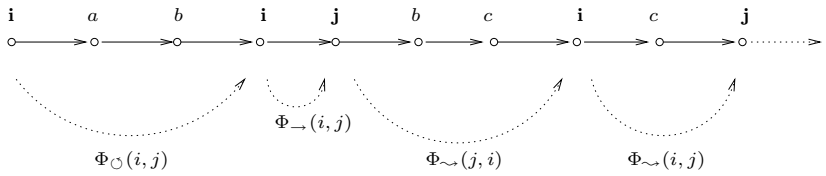
In this section, we will show how to reduce the model checking question $P^G \models \varphi$ to a series of model checking questions on smaller systems P^{G_i}'s where we can bound the size of the network graphs G_i as well as the number of the G_i's. For the sake of simplicity, we will start with the special case of 2-indexed existential LTL\X specifications, which can be readily generalized to the full case.

6.4.1 Existential 2-indexed LTL\X Specifications

In this section we show how to verify simple 2-indexed LTL\X properties of the form $\exists i, j.\varphi(i, j)$, where $i \neq j$. We will use the insights we obtain from this case to obtain the more general results later on.

Recall that 2-indexed properties are concerned only with properties of two processes in a given system. Our process communication model implies that two processes P_i and P_j can only affect each other by passing or receiving a token. Consequently, the synchronization between P_i and P_j crucially depends on the paths between sites i and j in the network graph. The following example is crucial to understanding the intuition behind our approach:

Example 6.4.1. The Figure below shows one path $\pi = i, a, b, i, j, b, c, i, c, j, \ldots$ that the token takes in a network graph.



Suppose that we are only interested in properties concerning the processes P_i and P_j, but not in processes P_a, P_b, P_c. Then only the sequence of the i's and j's in the path are of interest. Looking at π from left to right, we see four possibilities for what can happen between i and j: (1) P_i sends a token, and receives it back without P_j seeing it (formally, we will write $\Phi_{\circlearrowleft}(i, j)$ to denote this); (2) P_i passes the token *directly* to P_j ($\Phi_{\rightarrow}(i, j)$); (3) P_j sends the token to P_i through several intermediate sites ($\Phi_{\rightsquigarrow}(j, i)$); and (4) P_i sends

the token back to P_j through several intermediate sites ($\Phi_{\rightsquigarrow}(i,j)$). There are two more possibilities which do not occur in π: (5) $\Phi_{\rightarrow}(j,i)$ and (6) $\Phi_{\circlearrowleft}(j,i)$. The important insight is the following: **If we know which of these 6 cases can occur in a network graph G, then we have all information needed to reason about the communication between P_i and P_j.**

We will later construct small network graphs with 4 nodes where the sites i and j are represented by two distinguished nodes $site_1$ and $site_2$, while *all other* sites are represented by two "hub" nodes hub_1 and hub_2.

This example motivates the following definitions:

Definition 6.4.2 (Free Path). Let I be a set of indices, and π be a path in a network graph G. We say that π is I-free, if π does not contain a site from I.

We now define three kinds of path types that will be shown to capture all relevant token paths between two processes P_i and P_j.

Definition 6.4.3 (Connectivity, Characteristic Vectors). Let i,j be indices in a network graph G. We define three connectivity properties of the indices i,j:

$G \models \Phi_{\circlearrowleft}(i,j)$ "There is a $\{j\}$-*free* path from i to itself."

$G \models \Phi_{\rightsquigarrow}(i,j)$ "There is a path from i to j via a third node not in $\{i,j\}$."

$G \models \Phi_{\rightarrow}(i,j)$ "There is a direct edge from i to j."

Using the connectivity properties, we define an equivalence relation $\sim_2$ on network graphs: Given two network graphs G_1 and G_2 along with two pairs of indices a_1, b_1 and a_2, b_2, we

define

$$(G_1, a_1, b_1) \sim_2 (G_2, a_2, b_2)$$

iff for every $\Phi \in \{\Phi_{\circlearrowleft}, \Phi_{\rightsquigarrow}, \Phi_{\rightarrow}\}$,

$$G_1 \models \Phi(a_1, b_1) \iff G_2 \models \Phi(a_2, b_2) \text{ and}$$
$$G_1 \models \Phi(b_1, a_1) \iff G_2 \models \Phi(b_2, a_2)$$

If $(G_1, a_1, b_1) \sim_2 (G_2, a_2, b_2)$ we say that the indices a_1, b_1 in G_1 have the same connectivity as the indices a_2, b_2 in G_2.

The *characteristic vector* $v(G_1, a_1, b_1)$ is the 6-tuple containing the truth values of $G_1 \models \Phi_{\circlearrowleft}(a_1, b_1), G_1 \models \Phi_{\rightsquigarrow}(a_1, b_1), G_1 \models \Phi_{\rightarrow}(a_1, b_1)\, G_1 \models \Phi_{\circlearrowleft}(b_1, a_1), G_1 \models \Phi_{\rightarrow}(b_1, a_1)$, and $G_1 \models \Phi_{\rightsquigarrow}(b_1, a_1)$,

By definition it holds that $(G_1, a_1, b_1) \sim_2 (G_2, a_2, b_2)$ iff they have the same characteristic vectors, i.e., $v(G_1, a_1, b_1) = v(G_2, a_2, b_2)$. Since the number of characteristic vectors is constant, it follows that $\sim_2$ has finite index. The characteristic vectors can be viewed as representatives of the equivalence classes.

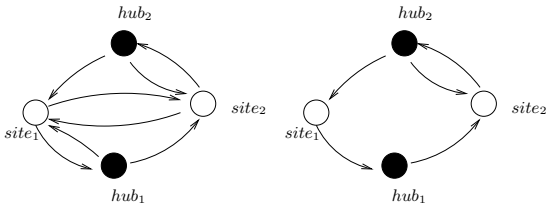


Figure 6.1: Network Graphs A, B, realizing two different characteristic vectors

Example 6.4.4. Consider the network graphs A, B of Figure 6.1. It is easy to see that $(A, site_1, site_2)$ has characteristic vector $(1, 1, 1, 1, 1, 1)$, i.e.,

$$A \models \quad \Phi_{\circlearrowleft}(site_1, site_2) \wedge \Phi_{\rightsquigarrow}(site_1, site_2) \wedge \Phi_{\rightarrow}(site_1, site_2) \wedge$$
$$\Phi_{\circlearrowleft}(site_2, site_1) \wedge \Phi_{\rightsquigarrow}(site_2, site_1) \wedge \Phi_{\rightarrow}(site_2, site_1)$$

and $(B, site_1, site_2)$ has characteristic vector $(0, 1, 0, 1, 1, 0)$, i.e.,

$$B \models \quad \neg\Phi_{\circlearrowleft}(site_1, site_2) \wedge \Phi_{\rightsquigarrow}(site_1, site_2) \wedge \neg\Phi_{\rightarrow}(site_1, site_2) \wedge$$
$$\Phi_{\circlearrowleft}(site_2, site_1) \wedge \Phi_{\rightsquigarrow}(site_2, site_1) \wedge \neg\Phi_{\rightarrow}(site_2, site_1).$$

Note that a network graph will in general have several characteristic vectors depending on the indices we consider. The set of characteristic vectors of a graph G can be efficiently computed from G in quadratic time. The crucial insight in our proof is that for two processes P_i and P_j, the connectivity between their indices i, j in the network graph determines the satisfaction of quantifier-free LTL\X properties $\varphi(i, j)$ over P^G:

Lemma 6.4.5 (2-Index Reduction Lemma). *Let G_1, G_2 be network graphs, P a process, and $\varphi(x, y)$ a 2-indexed quantifier-free LTL\X property. Let a_1, b_1 be a pair of indices on G_1, and a_2, b_2 a pair of indices on G_2. The following are equivalent:*

(a) $(G_1, a_1, b_1) \sim_2 (G_2, a_2, b_2)$*, i.e., a_1, b_1 and a_2, b_2 have the same connectivity.*

(b) $P^{G_1} \models \varphi(a_1, b_1)$ *iff* $P^{G_2} \models \varphi(a_2, b_2)$.

Proof of this lemma and other claims in this chapter have been moved to the last section for better readibility.

The lemma motivates the following model checking strategy: Given a (possibly complicated) network graph G_1 and two of its sites i, j, we can try to obtain a simpler network $G_2 := G_{(i,j)}$, with two special nodes $site_1$ and $site_2$ that have the same connectivity in G_2 as the indices i and j in G_1, and thus satisfies condition (a) of the lemma. For the case of two indices, we can always find such a network graph $G_{(i,j)}$ with at most 4 sites.

Proposition 3. For each graph G and indices i, j there exists a 4-node graph $G_{(i,j)}$ called the *connection topology of* i, j, having two special sites $site_1$ and $site_2$ such that

$$(G, i, j) \sim_2 (G_{(i,j)}, site_1, site_2).$$

In other words, the indices i and j in G have the same connectivity as the indices $site_1$ and $site_2$ in $G_{(i,j)}$.

Since $G_{(i,j)}$ satisfies condition (a) of Lemma 6.4.5, we obtain the following important consequence:

Corollary 6. Let $\varphi(i, j)$ be a 2-indexed quantifier-free LTL\X property. Then

$$P^G \models \varphi(i, j) \qquad \text{iff} \qquad P^{G_{(i,j)}} \models \varphi(site_1, site_2).$$

Thus, we have achieved a reduction from a potentially large network graph G to a 4-node network graph $G_{(i,j)}$. We will now show how to actually construct the connection topology $G_{(i,j)}$.

Construction of $G_{(i,j)}$. We construct the reduction graphs as follows. $G_{(i,j)}$ has four sites: $site_1, site_2, hub_1$, and hub_2. The sites $site_1$ and $site_2$ are called *primary sites*. They represent the sites of interest i and j. The other sites are called *hubs*, and they represent

the other nodes of the graph G. Let us describe in more detail the role of these different nodes. Recall that to satisfy Proposition 3, the sites $site_1$ and $site_2$ in $G_{(i,j)}$ should have the same connectivity as i, j in G. Therefore:

- If $\Phi_{\rightsquigarrow}(i, j)$ holds in G (i.e., there exists a path from i to j in G that goes through a third node), then $\Phi_{\rightsquigarrow}(site_1, site_2)$ has also to hold in $G_{(i,j)}$, i.e., there should exist in $G_{(i,j)}$ a path from $site_1$ to $site_2$ that goes through a third node. The site hub_1 will play the role of this "third node". Therefore, in this case, $G_{(i,j)}$ contains an edge from $site_1$ to hub_1, and from hub_1 to $site_2$.

- In the same manner, if $\Phi_{\circlearrowleft}(i, j)$ holds in G (i.e., there exists a path from i to itself in G that does not go through j), then $\Phi_{\circlearrowleft}(site_1, site_2)$ should also be true in $G_{(i,j)}$. As previously, this is ensured by considering the following edges: $(site_1, hub_1)$ and $(hub_1, site_1)$.

- Finally, if $\Phi_{\rightarrow}(i, j)$ holds in G (i.e., there exists a direct edge in G from i to j), then $G_{(i,j)}$ should also contain the edge $(site_1, site_2)$.

- The paths from j to i are treated in a symmetrical way.

For example, let H be a graph having as sites i, j, k, and l (among others), such that $v(H, i, j) = (1, 1, 1, 1, 1, 1)$, and $v(H, k, l) = (0, 1, 0, 1, 1, 0)$; then the graphs A and B of Example 6.4.4 correspond respectively to the reduction graphs $H_{(i,j)}$ and $H_{(k,l)}$.

Since our fairness assumption implies that the network is strongly connected, not all characteristic vectors actually occur in practice. A closer analysis yields the following bound:

Proposition 4. For 2 indices, there exist at most 36 connection topologies.

All the 36 connection topologies are shown in the Section 6.8.

Let us now return to the question of verifying properties of the form $\exists x, y.\varphi(x, y)$. Note that Corollary 6 only provides us with a way to verify one quantifier-free formula $\varphi(i, j)$. Given a system P^G, we define its 2-topology, denoted by $T_2(G)$, as the collection of all different connection topologies appearing in G. Formally,

Definition 6.4.6. Given a network graph $G = (S, C)$, the 2-topology of G is given by

$$T_2(G) = \{G_{(i,j)} \mid i, j \in S, i \neq j\}.$$

By Proposition 4, we know that $|T_2(G)| \leq 36$. Since we can express $\exists x, y.\varphi(x, y)$ as a disjunction $\bigvee_{i,j \in S} \varphi(i, j)$ we obtain the following result as a consequence of Corollary 6:

Theorem 6.4.7. *The following are equivalent:*

(i) $P^G \models \exists x, y.\varphi(x, y)$

(ii) There exists a connection topology $T \in T_2(G)$*, such that* $P^T \models \varphi(site_1, site_2)$.

Thus, we obtain the following reduction algorithm for model checking $P^G \models \exists x, y.\varphi(x, y)$:

1: Determine $T_2(G)$.

2: For each $T \in T_2(G)$, model check $P^T \models \varphi(site_1, site_2)$.

3: If one of the model checking calls is successful then output "true" else output "false".

Example 6.4.8.

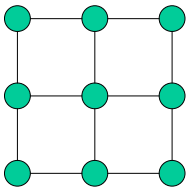


Figure 6.2: A system with grid like network graph with 9 nodes.

Consider a system P^G with a grid like network graph G shown in Figure 6.2. Assume that each edge of the network is bidirectional. To verify a 2-indexed $LTL \setminus X$ property $\exists x, y.\varphi(x, y)$ of this system, it is enough to consider two systems P^{G_1} and P^{G_2} with network graphs G_1, G_2 shown in Figure 6.3 and check $\varphi(site_1, site_2)$ on each of them.

If either system satisfies $\varphi(site_1, site_2)$ then $P^G \models \exists x, y.\varphi(x, y)$. Otherwise, it $P^G \not\models \exists x, y.\varphi(x, y)$.

Relation with Environment Abstraction

In this section we will consider the relationship between the decomposition presented here and environment abstraction presented in the earlier chapters. For ease of comparison, we will consider environment abstraction with single reference process and decompositions

Figure 6.3: Connection topologies for the grid-like network graph.

for two indexed properties.

First note that both the methods deal with properties of a fixed number of processes. In the case of environment abstraction, we considered primarily single index properties, that is, properties of one process and its environment. Here we consider double indexed properties, that is properties satisfied by two processes and their common environment. To build the abstract model in environment abstraction, we begin by asking how the system looks when viewed from the reference process. The environment of the reference process is captured using an appropriately chosen set of predicates. Our soundness theorem of Chapter 2 shows that the abstract model built using these predicates is sound and our experiments show that the abstract models are quite precise.

In this chapter too, we ask how the system looks like from the point of view of two processes. But this time, the environment around the two processes is described mainly in terms of the network topology. Note that the reduced system $P^{G_{(i,j)}}$ corresponding to processes i, j in a system P^G can be thought of as an abstraction of P^G. But, unlike usual abstractions, the set of properties (involving only processes i, j) satisfied by $P^{G_{(i,j)}}$

is exactly the same as the set of properties satisfied by P^G.

One way of looking at environment abstraction is to first consider the abstract models obtained by fixing the reference process (as we do in the proof of soundness). That is, for a given system $\mathcal{P}(K)$ consider the abstract models $\mathcal{P}_1^{\mathcal{A}}, \ldots, \mathcal{P}_K^{\mathcal{A}}$. If we can show, for each $i \in [1..K]$,

$$\mathcal{P}_i^{\mathcal{A}} \models \Phi(i)$$

then we can conclude that

$$\mathcal{P}(K) \models \forall x.\Phi(x)$$

But, it is not feasible to check each of the abstract models $\mathcal{P}_i^{\mathcal{A}}$ individually because there is no bound on K. So instead of verifying each abstract model separately, we create a new abstract model $\mathcal{P}^{\mathcal{A}}$ by combining all the individual models $\mathcal{P}_1^{\mathcal{A}}, \ldots, \mathcal{P}_K^{\mathcal{A}}$ to obtain an even more abstract model. By the existential abstraction principle, we have

$$\mathcal{P}^{\mathcal{A}} \models \Phi(x) \Rightarrow \forall i.\mathcal{P}_i^{\mathcal{A}} \models \Phi(i)$$

Thus, it is enough to verify the abstract model $\mathcal{P}^{\mathcal{A}}$.

In contrast, in this chapter, we take every possible pair of processes i, j, and construct the abstract model $P^{G_{(i,j)}}$ specific to each of them. But then, instead of grouping all these abstract models, we keep them separate and check each of them individually. This is possible because Proposition 4 guarantees that there are only 36 different possible reduction graphs (or abstract models). This could not be done in the case of environment abstraction, because we don't know apriori how many different individual abstract models are there nor do we know how to find them efficiently.

To summarize, the reduction presented here and the environment abstraction both involve describing the world around a fixed number of processes. Importantly, the results presented in this chapter amount to reductions, that is , the properties under consideration are preserved exactly. In contrast, in environment abstraction, the abstract model exhibits more behaviors than the concrete system.

6.4.2 Existential k-indexed LTL\X Specifications

We will now show how to generalize the results of the previous section to k-indexed properties. Throughout this section, we will write expressions such as $\bar{i}$ to denote k-tuples of indices, and $\bar{x}$ to denote k-tuples of variables. We will first adapt the notion of connectivity as follows. Let $\bar{i} = i_1, i_2 \ldots i_k$ be a sequence of indices, and $I = \{i_1, i_2 \ldots i_k\}$. Then we define the following connectivity properties:

$G \models \Phi_{\circlearrowleft}(x, I)$ "There is an $(I \setminus \{x\})$-*free* path from x to itself."

$G \models \Phi_{\rightsquigarrow}(x, y, I)$ "There is a path from x to y via a third node not in I."

$G \models \Phi_{\rightarrow}(x, y)$ "There is a direct edge from x to y."

By instantiating the variables x and y by the indices $i_1, \ldots, i_k$ in all possible ways, we obtain a finite number of different conditions which will describe all possible connectivities between the indices $i_1, \ldots, i_k$.

As in the previous section, we can define an equivalence relation $\sim_k$, where $(G_1, \bar{i}) \sim_k (G_2, \bar{j})$ iff the indices $\bar{i}$ have the same connectivity in G_1 as the indices $\bar{j}$ in G_2. Since the

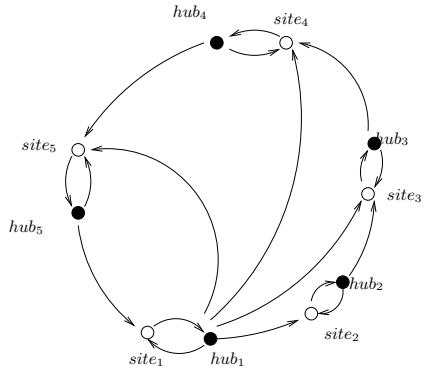


Figure 6.4: An example of a 5-index connection topology

number of conditions is bounded, $\sim_k$ is an equivalence relation of finite index, and we can describe each equivalence class by a characteristic vector $v(G, \bar{v})$. As in the previous section, we define the k-connection topologies, $G_{(i_1,i_2\ldots i_k)}$ of the processes $P_{i_1}, P_{i_2} \ldots P_{i_k}$ in G as the smallest graphs that preserve all the connectivity properties between the processes $P_{i_1}, P_{i_2} \ldots P_{i_k}$. The construction of the topology graphs is illustrated in Figure 6.4.

The unfilled nodes $site_1, \ldots, site_k$ in the graph are the primary sites. There is a ***hub*** site associated with each primary site. Moreover, there is an edge from each hub hub_j back to its primary $site_j$ if there is an $(I \setminus \{i_j\})$-***free*** path from i_j to itself. There is an edge from hub_j to $site_l$ if there is a path from i_j to i_l in G via a third node not in I, and there is an edge from $site_j$ to $site_l$ if there exists a direct edge (i_j, i_l) in G.

Analogous to the bounds on 2-connection topologies it can be shown that each k-connection topology has at most $2k$ processes and that there are at most $3^{k(k-1)}2^k$ distinct k-connection topologies. By an argument analogous to that of the previous section, we obtain the following corollary

Corollary 7. Let $\varphi(\bar{x})$ be a k-indexed quantifier-free LTL\X property. Then

$$P^G \models \varphi(\bar{i}) \qquad \text{iff} \qquad P^{G_{(\bar{i})}} \models \varphi(site_1, site_2, \ldots, site_k).$$

The notion of k-topology is defined analogously:

Definition 6.4.9. Given a network graph $G = (S, C)$ the k-topology of G is given by

$$T_k(G) = \{G_{(\bar{i})} \mid \bar{i} \in S^k, \text{all indices in } \bar{i} \text{ are distinct}\}.$$

Consequently, we obtain a model checking procedure from the following theorem, similar to the case of 2-indices:

Theorem 6.4.10. *The following are equivalent:*

(i) $P^G \models \exists \bar{x}.\varphi(\bar{x})$

(ii) There exists a connection topology $T \in T_k(G)$*, such that* $P^T \models \varphi(site_1, site_2, \ldots, site_k)$.

As mentioned before $|T_k(G)| \leq 3^{k(k-1)}2^k$.

6.4.3 Specifications with General Quantifier Prefixes

In this section we will show how to obtain reductions for k-indexed specifications with first order prefixes.

Let us for simplicity consider the 2-indexed formula $\Phi := \forall x \exists y.\varphi(x, y)$. Over a network graph $G = (S, C)$, $|S| = n$, it is clear that Φ is equivalent to $\wedge_{1 \leq i \leq n} \vee_{1 \leq j \leq n} \varphi(i, j)$. A naive application of Corollary 7 would therefore require n^2 calls to the model

checker, which may be expensive for practical values of n. In practice, however, we can bound the number of model checker calls by $|T_2(G)|$ since this is the maximum number of *different* connection topologies. We conclude that the n^2 model checker calls must contain repetitions. We can make sure that at most 36 calls to the model checker are needed. We obtain the following algorithm:

1: Determine $T_2(G)$.

2: For each $T \in T_2(G)$

3: model check $P^T \models \varphi(site_1, site_2)$

4: $g[T] := 1$ iff model checking successful, and 0 otherwise

5: Output $\bigwedge_{1 \leq i \leq n} \bigvee_{1 \leq j \leq n} g[G_{(i,j)}]$.

By simplifying the formula in line 5, we may further increase performance. The algorithm can be adapted for k indices in the obvious way. To state the main theorem of this section, we define (c, s)-bounded reductions, where c bounds the number of **c**alls to the model checker, and s bounds the **s**ize of the network graph.

Definition 6.4.11 ((c, s)-bounded Reduction). Let G, P be as above, and φ be a closed k-indexed formula with matrix $\varphi'(x_1, \ldots, x_k)$. Let Ψ denote a property of interest (e.g., the model checking property $''P^G \models \varphi''$). A (c, s)-bounded reduction of property Ψ is given by:

- a sequence of c *reduced* network graphs $G_i = (S_i, C_i), 1 \leq i \leq c$ such that $|S_i| \leq s$. called reduction graphs.

- a boolean function B over c variables $g_1, \ldots, g_c$, such that

$$\Psi \quad \text{iff} \quad B(g_1, \ldots, g_c) = 1 \quad \text{where} \quad g_i := 1 \text{ iff } G_i^P \models \varphi'(site_1, \ldots, site_k)$$

  In other words, property Ψ is decided by c calls to the model checker, where in each call the network graph is bounded by s.

Further, we say that a class $\mathcal{L}$ of specifications has (c, s) bounded reduction if for all network graphs G and any $\varphi \in \mathcal{L}$, the property $P^G \models \varphi$ has (c, s)-bounded reduction. We can now state our main result:

Theorem 6.4.12. *Let φ be any k-indexed LTL\X specification. Then the model checking problem "$P^G \models \varphi$" has polynomial-time*[1] *computable $(3^{k(k-1)}2^k, 2k)$-bounded reductions.*

In fact, the sequence of reduced network graphs is just the different k-connection topologies occurring in G. This implies that given k and network graph G, all k-indexed LTL\X specifications have the same reduction. Stated another way, LTL\X has $(3^{k(k-1)}2^k, 2k)$-bounded reduction.

6.4.4 Cut-Offs for Network Topologies

In this section, we prove the existence of cutoffs for network topologies, i.e., (infinite) classes of network graphs. We say that a class of network graphs has cutoff (c, s), if the question whether **all** the network graphs in this topology satisfy the specification has a (c, s)-bounded reduction.

[1] in the size of the network graph G

Definition 6.4.13 (Cut-off). Let $\mathbb{T}$ be a network topology, and $\mathcal{L}$ a class of specifications. $\mathbb{T}$ has a cut-off (c, s) for $\mathcal{L}$ if for all specifications $\varphi \in \mathcal{L}$ the property

$$\Psi := \quad \text{“}\, \forall G \in \mathbb{T} \,.\, P^G \models \varphi \,\text{”}$$

has a (c, s)-bounded reduction.

It is not hard to prove that a (c, s)-bounded reduction for a network graph translates to a cut-off for a network topology:

Theorem 6.4.14. *For k-indexed specifications, all network topologies $\mathbb{T}$ have $(2k, 3^{k(k-1)}2^k)$-bounded reductions.*

Note that the theorem does not provide us with an *effective* means to find the reduction; it does however guarantee that at least in principle we can always find a cutoff by investigating the topology $\mathbb{T}$.

6.5 Bounded Reductions for CTL$\setminus$X are Impossible

In this section, we show that indexed CTL$\setminus$ X formulas over two indices do not have (c, s)-bounded reductions. We will first show the following generic result about CTL$\setminus$ X:

Theorem 6.5.1. *For each number i there exists an CTL$\setminus$ X formula φ_i with the following properties:*

- *φ_i is satisfiable (and has a finite model).*
- *φ_i uses only two atomic propositions l and r.*

- *Every Kripke structure K where φ_i is true has at least i states.*
- *φ_i has the form $\mathbf{EF}\varphi_i'$.*

The result is true even when the Kripke structure is required to have a strongly connected transition relation.

Remark 15. This result is closely related to early results about characterizing Kripke structures up to bisimulation in [14]. The results in [14] give rise to the following proof idea for Theorem 6.5.1: Let $K_1, \ldots, K_n$ be all Kripke structures with 2 labels of size $\leq i$, and let $f_1, \ldots, f_n$ be CTL$\setminus$ X formulas which characterize them up to stuttering bisimulation. Consider now the formula $\varphi_i := \bigwedge_{1 \leq j \leq n} \neg f_j$. By construction every model of φ_i must have $> i$ states. At this point, however, the proof breaks down, because we do not know from the construction if φ_i is satisfiable at all. The natural way to show that φ_i has a model would be to prove that stuttering bisimulation over a 2-symbol alphabet has infinite index. This property however is a corollary to Theorem 6.5.1, and we are not aware of a proof in the literature.

For *properties involving only the presence of the token*, a system P^G, where $G = (S, C)$ essentially behaves like a Kripke structure with set of states S and transition relation C. To see this, consider a system P^G, where P is a trivial process which can always receive a token, and immediately send the token to a neighbor process. Let t_i and t_j be propositional formulas stating that the token is with process i and j respectively. Since the processes do not influence the path taken by the token, the token moves only according to the network graph G, and thus for each path on P^G there exists a corresponding path in G. Consequently, if a path on P^G satisfies a property without **X**, then the corresponding path

on G also satisfies this property. Now we can show by contradiction that indexed CTL$\setminus$ X cannot have bounded reductions. Suppose CTL$\setminus$X did have (c, s)-bounded reduction for some s. Then, by Theorem 6.5.1, we can always find a CTL$\setminus$X formula Φ such that the network graph underlying any system that satisfies Φ must have size at least $c + 1$. Thus CTL$\setminus$X does not have bounded reductions. Consequently, we also have the following corollary:

Corollary 8. There exists a network topology $\mathbb{T}$ for which 2-indexed CTL$\setminus$ X does not have cut-offs.

A detailed proof can be found in the last section of this chapter.

6.6 Conclusion

We have described a systematic approach for reducing the verification of large and parameterized systems to the verification of a sequence of much smaller systems. We will conclude this chapter with further considerations concerning the practical complexity of model checking.

For simplicity, let us again consider the case of 2-indexed properties. Suppose the processes P in our network have state space $|Q|$. Then our reduction requires to model check up to 36 network graphs with 4 sites, resulting in a state space of $|Q|^4$. Even this model checking problem may be too expensive in practice. By a close analysis of our proofs, it is however possible to reduce the state space even further to $O(|Q|^2)$.

It is easy to show that Lemma 6.4.5 will hold even when the *processes at the hubs*

are simple *dummy processes* containing two states whose mere task is to send and receive the token infinitely often. Consequently, the systems $P^{G_{(i,j)}}$ will have state space of size $2^2 \times |Q|^2$.

The results in this chapter on LTL$\backslash$X were derived assuming fairness condition on the systems. We can obtain similar reductions by removing this assumption. Doing away with fairness necessitates the consideration of two more path types other than the ones described in Section 6.4.1. Consequently, the topology graphs have more than 4 sites and also the number of different topology graphs increases.

6.7 Proofs of Lemmas

Proposition 4. *For* 2 *indices, there exist at most* 36 *connection topologies.*

Proof. By our fairness assumption, every connection topology must be strongly connected. This implies that the following conditions must hold:

- At least one of $\Phi_{\rightarrow}(i,j)$ or $\Phi_{\rightsquigarrow}(i,j)$ must be true.
- At least one of $\Phi_{\rightarrow}(j,i)$ or $\Phi_{\rightsquigarrow}(j,i)$ must be true.

This means in the characteristic vector of connection topology the following must hold:

- At least one of the second and third elements (corresponding to the connectivity properties discussed above) must be 1. This gives us three choices in picking the second and third elements of the vector.

- At least one of the fifth and sixth elements must be 1. This again gives us three choices in picking the fifth and the sixth elements of the vector.
- First and fourth elements can be either 0 or 1. This gives us four choices in picking the first and the fourth elements of the vector.

Consequently the number of different possible characteristic vectors is $3 \times 3 \times 4 = 36$. □

Lemma 6.4.5. *Let G_1, G_2 be network graphs, P a process, and $\varphi(x, y)$ a 2-indexed quantifier-free LTL\X property. Let a_1, b_1 be a pair of indices on G_1, and a_2, b_2 a pair of indices on G_2. The following are equivalent:*

(a) $(G_1, a_1, b_1) \sim_2 (G_2, a_2, b_2)$*, i.e., a_1, b_1 and a_2, b_2 have the same connectivity.*

(b) $P^{G_1} \models \varphi(a_1, b_1)$ *iff* $P^{G_2} \models \varphi(a_2, b_2)$.

We first define some notions which will be helpful in proving the lemma. Let P^G be a system with m processes

> An *execution trace* of the system P^G is a series of *global states* in such that there is a transition from every k^{th} state in the trace to the $(k+1)^{th}$ state.

Given a trace t, we will denote the n^{th} state in t by t^n.

> A *witness* in system P^G for a $LTL \setminus X$ formula $\varphi(i, j)$ (where P_i and P_j are two processes in G) is an execution trace of S that satisfies the $LTL \setminus X$ formula.

We now define the *projection* of an execution trace with respect to a set of indices $I = \{i_1, \ldots i_k\}$. First we describe the *collapse* of a trace with respect to I.

Definition 6.7.1. Given an execution trace t of a system P^G and a set of indices I, the *collapse* of t with respect to I is obtained by removing every global state, t^{n+1} in t such that $\forall i \in I. t^n(i) = t^{n+1}(i)$

Informally, a *collapse* of trace t is obtained by removing those global states from the trace which do not change the states of processes with indices I.

Definition 6.7.2. Given a collapsed trace t_c of P^G with respect to I the *projection* of t with respect to I is the series of states obtained by projecting each global state in t_c onto the processes in I.

Lemma 6.7.3. *If two execution traces, t_1 and t_2 have the same projection with respect to a set of processes, I, then the two traces satisfy exactly the same set of $LTL \setminus X$ properties over I.*

Proof. This follows from the semantics of $LTL \setminus X$ properties. □

Lemma 6.7.4. *A system P^G with two indices i and j satisfies an $LTL \setminus X$ property $\varphi(i, j)$ if and only if the system $P^{G_{(i,j)}}$ satisfies the property $\varphi(site_1, site_2)$.*

Proof. We will prove that if P^G satisifes a property $\varphi(i, j)$ then $P^{G_{(i,j)}}$ satisfies $\varphi(site_1, site_2)$. The proof for the other direction is exactly the same.

Consider any two-indexed $LTL \setminus X$ property $\varphi(i, j)$. Let system P^G satisfy $\varphi(i, j)$. Consider a witness, w, for $\varphi(i, j)$ in the system S. Obtain the projection, w_p, of w with

respect to indices i, j. Note that each state in w_p will be of the form (q_i, q_j) where q_i, q_j are local states of processes i, j respectively.

We will say a trace w' of $P^{G_{(i,j)}}$ *matches* w_p if the projection of w' with respect to indices $site_1$ and $site_2$ is isomorphic to w_p modulo renaming. Clearly, if a trace w' matching w_p exists in $P^{G_{(i,j)}}$ then the system $P^{G_{(i,j)}}$ satisfies the property $\varphi(site_1, site_2)$.

We will now construct a trace w' of $P^{G_{(i,j)}}$ that matches w_p. The state of process i in P^G will be matched by the state of process $site_1$ in $P^{G_{(i,j)}}$ and the state of process j will be matched by the state of process $site_2$. Consider the first state (q_i, q_j) of the trace w_p. Since (q_i, q_j) is the first state of the trace w_p, both q_i, q_j must be the initial local states. The first state of w' will then be $(q_{site_1}, q_{hub_1}, q_{site_2}, q_{hub_2})$ where q_{site_1}, q_{site_2} are initial local states. The hubs can be in any local state, so by default we require them to be in initial states as well.

The token could be held in three possible ways the state (q_i, q_j):

- By process i.
- By process j.
- By neither i nor j

In case the token is with process i then in w' the token will be with process $site_1$. In case the token is with process j then in w' the token will be with process $site_2$. In the last case, the first global state of w', $(q_{site_1}, q_{hub_1}, q_{site_2}, q_{hub_2})$, is such that token is with q_{hub_1} or q_{hub_2}. It is easy to see that the first state of w' thus constructed *matches* the first state of w_p.

Assume that we have been able to construct a prefix pf of w' which matches the prefix of w_p of length k. Denote the m^{th} state in w_p by w_p^m and the prefix of length m by m-prefix.

To extend the trace w' to $k+1$ states consider the states w_p^k and w_p^{k+1} of w_p. We have the following cases to consider:

- In going from w_p^k to w_p^{k+1} there is no change in the process holding the token. Assume, without loss of generality, that the difference between w_p^k and w_p^{k+1} is a change in the local state of process i. Consider w'^k, the k^{th} state of w'. Since w' matches the k-prefix of w_p, the states of process i in w_p^k and process $site_1$ in w'^k must match. This means whatever action i can take, the same action can be taken by $site_1$. Thus, we can extend w' to $k+1$ states by replicating the action of process i using process $site_1$.

- In w_p^k the token is with i and in w_p^{k+1} is with j. We can then infer that there must be a direct edge in G from process i to j, that is, $G \models \Phi_{\rightarrow}(i,j)$ must be true. Thus there must be a similar direct edge in $G_{(i,j)}$ from $site_1$ to $site_2$, that is in $G_{(i,j)} \models \Phi_{\rightarrow}(i,j)$ is true. And since the prefix pf of w' matches the k-prefix of w_p, in the last state of pf, w'^k, the token must be with process $site_1$. Further, the states of $site_1$ and $site_2$ in w'^k must be the same as the states of i and j (respectively) in w_p^k. Thus, we can extend w' with the state that is obtained by a token transfer from $site_1$ to $site_2$. Thus we have a prefix of w' that matches the $k+1$-prefix of w_p. The case where token is with j in w_p^k and with i in w_p^{k+1} is analogous.

- In w_p^k the token is with neither i nor j and in w_p^{k+1} it is with j. This case has the following three sub-cases

– In the k-prefix of w_p the token was last with j. That is, there is a state w_p^m, $m < k$ such that the token is with process j in w_p^m and in no state w_p^n, $m < n \leq k$ is the token with either process i or j.

  This implies that there a path from j to itself in G that does not go through i, that is $G \models \Phi_{\circlearrowleft}(j, i)$. Then there must be a similar path from $site_2$ to itself in $G_{(i,j)}$ which does not go through $site_1$, that is, $G_{(i,j)} \models \Phi_{\circlearrowleft}(site_2, site_1)$ must hold. Since pf matches k-prefix of w_p, the process that last had the token in pf must be $site_2$. In the last state l of pf the token is neither with $site_1$ or $site_2$. We can infer that the token must be with process hub_2 because $site_2$ can send token to either $site_1$ or hub_2 and the token is not with $site_1$. Then we can add a series of states to pf such that, at the end, token is transferred back to $site_2$ and the only process that changes the state in this series of states prior to token transfer is hub_2. This is always possible because of our assumption that each process can send and receive token infinitely often. Thus we now have prefix of w' that can match the first $k + 1$ states of w_P.

– In the k-prefix of w_p the token was last with i. This means that there a path from i to j in G that goes through a third process, that is $\Phi_{\rightsquigarrow}(i, j)$ must hold in G. Then there must be a similar path from $site_1$ to $site_2$ in $G_{(i,j)}$ that goes through hub_1. In the last state l of pf the token is with hub_1. To see this, note that $site_1$ can send token either to hub_1 or $site_2$ and in l the token cannot be with $site_2$ (otherwise pf will not match the k-prefix of w_p). As before, we can add a series of states to pf such that at the end token is with $site_2$ and the only process that changes state prior to token transfer is process hub_1.

– In the k-prefix of w_p the token was never with i or j. That is w_p^{k+1} is the first state where the token is with j. We have constructed pf such that it matches the k-prefix of w_p. Since there was no token transitions involving either i or j, all the transitions in pf must have been transitions local to i or j. Thus, it does not matter where the token was initially in pf. Since we are interested only in existential properties, we can construct pf such that the token is with process hub_2 in the last state l. This means that from l we can have a token transition from process hub_2 to $site_2$. Thus we can extend pf so that it matches $k+1$-prefix of w_p.

The case where the token is with neither i nor j in the state w_p^k and with i in the state w_p^{k+1} is analogous.

Thus, we can construct a trace of w' of S' which matches w_p.

Note that we have implicitly used the fairness assumption for P^G. The assumption is implicit in the fact there is always a $k+1^{th}$ state in pf that is to be matched.

□

Lemma 6.7.5. *Let P^{G_1} and P^{G_2} be two systems. Further let there be two processes indexed i and j in both G_1 and G_2. If for all two indexed $LTL \setminus X$ property $\varphi(i,j)$, $P^{G_1} \models \varphi(i,j) \Leftrightarrow P^{G_2} \models \varphi(i,j)$ then $G_{1_{(i,j)}} = G_{2_{(i,j)}}$.*

Proof. The proof strategy is the following. For each path-type, we will give a two-indexed $LTL \setminus X$ formula $\Psi(i,j)$ such that if $\Psi(i,j)$ holds on a system P^G then the associated path-type exists between i and j in the network G.

The three formulas are:

- For $\Phi_{\circlearrowleft}(i)$: $\mathbf{F}\ (t_i \wedge (\neg t_j\ \mathbf{U}\ \neg(t_i \vee t_j))\ \mathbf{U}\ t_i)$
- For $\Phi_{\rightsquigarrow}(i,j)$: $\mathbf{F}\ (t_i \wedge (\neg t_j\ \mathbf{U}\ \neg(t_i \vee t_j))\ \mathbf{U}\ t_j)$
- For $\Phi_{\rightarrow}(i,j)$: $\mathbf{F}\ (t_i \wedge (t_i\ \mathbf{U}\ t_j))$

It is easy to see that each of the three formulas implies the associated path type.

Now if P^{G_1} satisfies exactly those two-indexed properties as P^{G_2}, then the two systems must satisfy exactly the same *type* formulas. Hence $G_{1_{(i,j)}} = G_{2_{(i,j)}}$ □

Lemma 6.4.5. We first prove that $(i) \Rightarrow (ii)$. Assume that $(G_1, a_1, b_1) \sim_2 (G_2, a_2, b_2)$. Then we know that

$$G_{1_{(a_1,b_1)}} = G_{2_{(a_2,b_2)}}.$$

By Lemma 6.7.5,

$P^{G_1} \models \varphi(a_1, b_1)$

$\Leftrightarrow P^{G_{1_{(a_1,b_1)}}} \models \varphi(site_1, site_2)$

$\Leftrightarrow P^{G_{2_{(a_2,b_2)}}} \models \varphi(site_1, site_2)$

$\Leftrightarrow P^{G_2} \models \varphi(a_2, b_2).$

For the other direction, assume $P^{G_1} \models \varphi(a_1, b_1) \Leftrightarrow P^{G_2} \models \varphi(a_2, b_2)$. Now,

$$P^{G_1} \models \varphi(a_1, b_1) \Leftrightarrow P^{G_{1_{(a_1,b_1)}}} \models \varphi(site_1, site_2)$$

and

$$P^{G_2} \models \varphi(a_2, b_2) \Leftrightarrow P^{G_{2_{(a_2,b_2)}}} \models \varphi(site_1, site_2).$$

Thus

$$P^{G_{1_{(a_1,b_1)}}} \models \varphi(site_1, site_2) \Leftrightarrow P^{G_{2_{(a_2,b_2)}}} \models \varphi(site_1, site_2)$$

which implies, by Lemma 6.7.4, that $G_{1_{(a_1,b_1)}} = G_{2_{(a_2,b_2)}}$ and therefore$(G_1, a_1, b_1) \sim_2 (G_2, a_2, b_2)$.

□

Theorem 6.4.14 *For k-indexed specifications, all network topologies $\mathbb{T}$ have $(2k, 3^{k(k-1)}2^k)$-bounded reductions.*

Proof. Let φ be a k-indexed specification and $G_1, G_2, \ldots$ be an enumeration of the network graphs in $\mathbb{T}$. Since model checking for each graph $G_i \in \mathbb{T}$ is $(2k, 3^{k(k-1)}2^k)$-bounded regardless of the size of G_i, we obtain a sequence of Boolean functions B_i over the same variables $g_1, \ldots, g_{3^{k(k-1)}2^k}$. Consider now the (infinitary) conjunction $\mathcal{B} := \bigwedge_{i \geq 1} B_i$. By Corollary 7, the function $\mathcal{B}$ expresses that for all G_i we have $G_i^P \models \varphi$. It remains to show that Ψ is equivalent to a finite formula. Since $\mathcal{B}$ depends only on a finite number $(3^{k(k-1)}2^k)$ of Boolean variables, functional completeness of Boolean logic implies that $\mathcal{B}$ is equivalent to a finite formula of size at most $2^{3^{k(k-1)}2^k}$. □

Theorem 6.5.1. *For each number i there exists a CTL\X formula φ_i with the following properties:*

- *φ_i is satisfiable (and has a finite model).*

- *φ_i uses only two atomic propositions l and r.*
- *Every Kripke structure K where φ_i is true has at least i states.*
- *φ_i has the form $\mathbf{EF}\varphi_i'$.*

The result remains true, when the Kripke structure is required to have a strongly connected transition relation.

Proof. Our goal is to describe a formula φ_i using atomic propositions l and r whose models must have at least i states. We will construct a large conjunction $\bigwedge_{\psi\in\Gamma}\psi$, and describe which formulas to put in Γ. The idea is simple: Γ needs to contain i CTL$\backslash$X formulas which describe the existence of i different states. Then the formula $\mathbf{EF}\bigwedge_{\psi\in\Gamma}\psi$ will be the sought for φ_i.

Consider a Kripke structure K as in Figure 6.5:

- In Level 0, it contains two distinct states L, R labelled with l and r respectively. To express the presence of these states, we include the formulas, let $\psi_0^1 := (l \wedge \neg r)$ and $\psi_0^2 := (r \wedge \neg l)$, and include $\mathbf{EF}\psi_0^1$ and $\mathbf{EF}\psi_0^2$ into Γ.

  It is clear that $\mathbf{EF}\psi_0^1$ and $\mathbf{EF}\psi_0^2$ express the presence of two mutually exclusive states.

- In Level 1, K contains $2^2 - 1 = 3$ states, such that the first one has $\{L, R\}$-free paths to L and R, the second one an $\{L, R\}$-free path only to L, and the third one an $\{L, R\}$-free path only to R. The characteristic properties of level 1 states are expressed by formulas

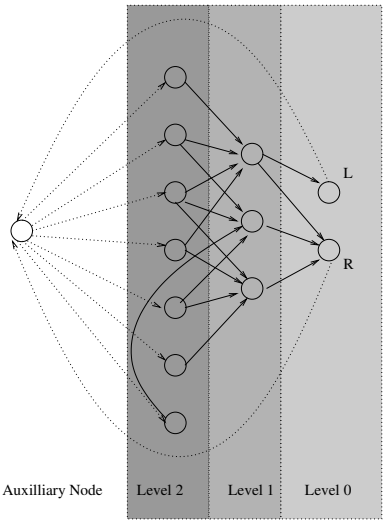


Figure 6.5: The Kripke structure K, constructed for three levels. The dashed lines indicate the connections necessary to achieve a strongly connected graph.

$\psi_1^1 := \mathbf{EF}^- \psi_0^1 \wedge \mathbf{EF}^- \psi_0^2$

$\psi_1^2 := \mathbf{EF}^- \psi_0^1 \wedge \neg\mathbf{EF}^- \psi_0^2$

$\psi_1^3 := \neg\mathbf{EF}^- \psi_0^1 \wedge \mathbf{EF}^- \psi_0^2$

where $\mathbf{EF}^- x$ denotes $\mathbf{E}(\neg l \wedge \neg r)\mathbf{U}x$, i.e., a variant of $\mathbf{EF}$ which forbids paths through L and R. To enforce the existence of the Level 1 states in the Kripke structure, we include $\mathbf{EF}\psi_1^1, \mathbf{EF}\psi_1^2, \mathbf{EF}\psi_1^3$ into Γ.

- In Level 2, K contains $2^3 - 1 = 7$ states, such that every state in level 2 can reach one of the 7 non-empty subsets of Level 1. The characteristic properties of Level 2 states can be expressed by formulas such as

  $\psi_2^1 := \mathbf{EF}^- \psi_1^1 \wedge \mathbf{EF}^- \psi_1^2 \wedge \mathbf{EF}^- \psi_1^3$

and

$\psi_2^2 := \neg \mathbf{EF}^- \psi_1^1 \wedge \mathbf{EF}^- \psi_1^2 \wedge \mathbf{EF}^- \psi_1^3$

, etc including ψ_2^3 to ψ_2^7. To enforce the presence of Level 2 states in the Kripke structure, we include the formulas $\mathbf{EF}\psi_2^i$ for $i = 1, \ldots, 7$ into Γ.

- In general, each Level k has at least $2^{k+1} - 1$ states that differ in their relationship to the states in Level $k - 1$. The presence of such states is expressed by formulas $\mathbf{EF}\psi_k^x$.

All these formulas are included into Γ until the requested number i of different states is reached. By construction, all properties required by the theorem are trivially fulfilled. In particular, Figure 6.5 demonstrates that there always exists a strongly connected model. □

The formula φ_i uses two labels l, r. To use the above theorem in the setting of systems with network graphs, we replace the labels l, r by atomic propositions t_x, t_y. Recall that an atomic proposition t_x states that the token is with process x. We will denote the modified formula by $\varphi_i(x, y)$. We have the following proposition as a consequence of the above theorem.

Corollary 9. If $P^G \models \exists x, y.\ \varphi_i(x, y)$, where $G = (S, C)$ then $|S| > i$.

Proof. Consider formula $\exists x, y.\varphi_i(x, y)$ and suppose, towards a contradiction, that there is a system P^G, $G = (S, C)$ where $|S| < i$ such that $P^G \models \exists x, y.\varphi_i(x, y)$. Then there exist indices a, b such that $P^G \models \varphi_i(a, b)$. We construct the Kripke structure $\mathcal{K}$ with state space S, transition relation C, initial state 1, and two atomic propositions t_a, t_b which hold

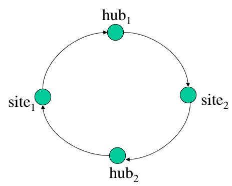
true on states a and b respectively. Note that since the formula $\varphi_i(a,b)$ is of the form $\mathbf{EF}\varphi_i'(a,b)$ and C is strongly connected, satisfaction of $\varphi_i(a,b)$ does not depend on the choice of the initial state. Since we know that for all paths on P^G, the corresponding paths on G preserve properties without **X**, it follows that $\mathcal{K} \models \varphi_i(a,b)$. By the above theorem, $\mathcal{K}$ must have at least i states, which contradicts our assumption that $|S| < i$. Thus, we have a proof by contradiction that if $P^G \models \exists x,y.\varphi_i(x,y)$ then the network graph G must have at least i nodes in it. □

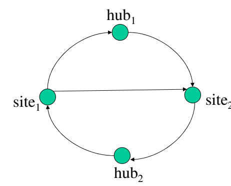
Corollary 8. *There exists a network topology $\mathbb{T}$ for which 2-indexed $CTL \setminus X$ does not have cut-offs.*

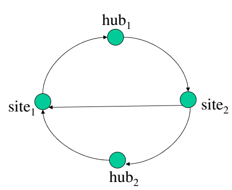
Proof. Let $\mathbb{T}$ be the class of strongly connected graphs. Then Corollary 9 tells us that $\exists x,y.\varphi_n(x,y)$ does not have a cut-off for $\mathbb{T}$. □

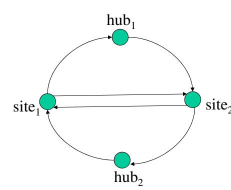
6.8 Connection Topologies for 2-Indices

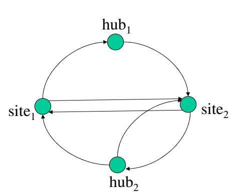
All the 36 possible connection topologies between two processes are presented below.

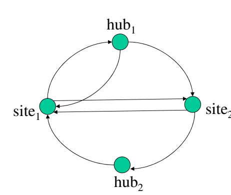

hub$_1$
site$_1$
site$_2$
hub$_2$

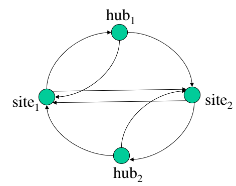

hub$_1$
site$_1$
site$_2$
hub$_2$


hub$_1$
site$_1$
site$_2$
hub$_2$

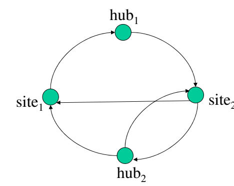

hub$_1$
site$_1$
site$_2$
hub$_2$


hub$_1$
site$_1$
site$_2$
hub$_2$

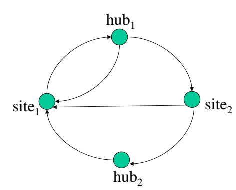

hub$_1$
site$_1$
site$_2$
hub$_2$



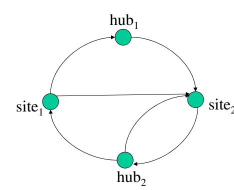
hub1
site1
site2
hub2



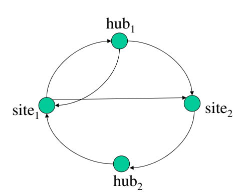
hub1
site1
site2
hub2



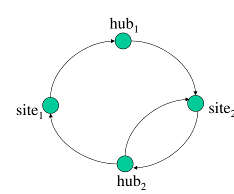
hub1
site1
site2
hub2



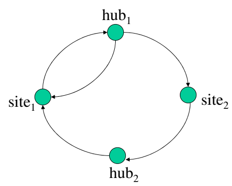
hub1
site1
site2
hub2

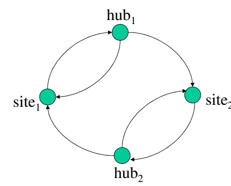


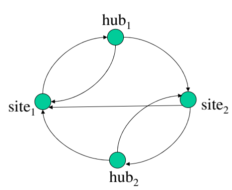
hub1
site1
site2
hub2

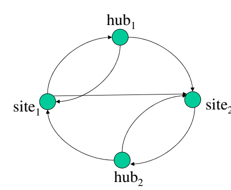


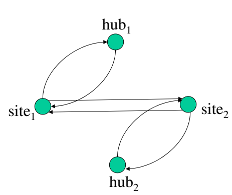
hub1
site1
site2
hub2

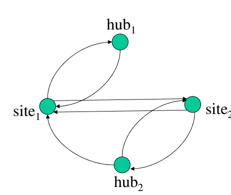

hub$_1$
site$_1$
site$_2$
hub$_2$

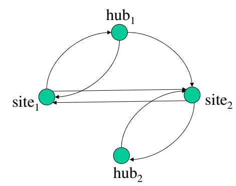

hub$_1$
site$_1$
site$_2$
hub$_2$

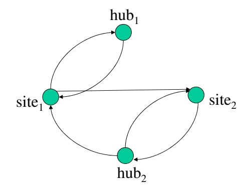

hub$_1$
site$_1$
site$_2$
hub$_2$

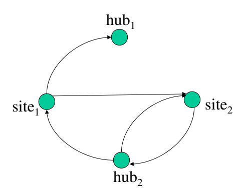

hub$_1$
site$_1$
site$_2$
hub$_2$

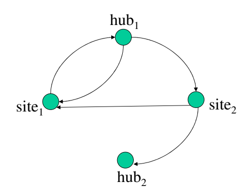

hub$_1$
site$_1$
site$_2$
hub$_2$

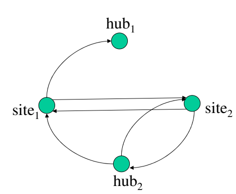

hub$_1$
site$_1$
site$_2$
hub$_2$

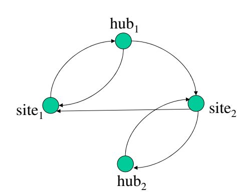

hub1
site1
site2
hub2

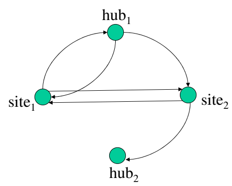

hub1
site1
site2
hub2

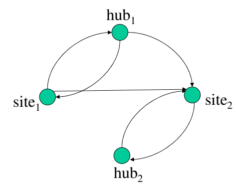

hub1
site1
site2
hub2

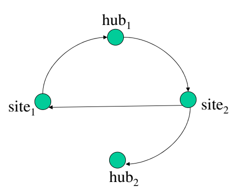

hub1
site1
site2
hub2

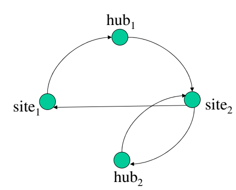

hub1
site1
site2
hub2

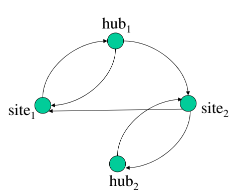

hub1
site1
site2
hub2

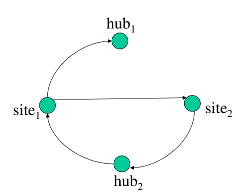

hub$_1$
site$_1$
site$_2$
hub$_2$

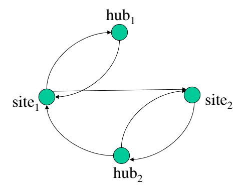

hub$_1$
site$_1$
site$_2$
hub$_2$

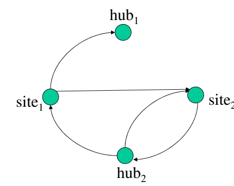

hub$_1$
site$_1$
site$_2$
hub$_2$

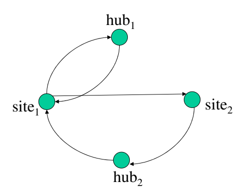

hub$_1$
site$_1$
site$_2$
hub$_2$

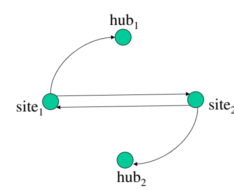

hub$_1$
site$_1$
site$_2$
hub$_2$

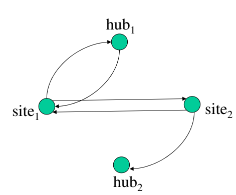

hub$_1$
site$_1$
site$_2$
hub$_2$

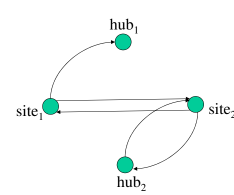

hub$_1$
site$_1$
site$_2$
hub$_2$


hub$_1$
site$_1$
site$_2$
hub$_2$


hub$_1$
site$_1$
site$_2$
hub$_2$


hub$_1$
site$_1$
site$_2$
hub$_2$


hub$_1$
site$_1$
site$_2$
hub$_2$


hub$_1$
site$_1$
site$_2$
hub$_2$

Chapter 7

Conclusion

7.1 Summary

This thesis presents an efficient abstraction technique to facilitate model checking of parameterized systems with replicated processes. All successful applications of model checking thus far have made use of domain specific abstraction techniques. Continuing this trend, we exploit the domain knowledge about parameterized systems to devise an appropriate abstraction method.

The problem of verifying parameterized systems is both challenging theoretically (because of their unboundedness) and very relevant practically (because many crucial components of real systems are parameterized). For example, in the recent years, verification of cache coherence protocols has become a very important problem in the hardware industry. All the modern multi-core architectures have very intricate cache coherence protocols,

and there are no rigorous techniques for their verification. Similarly, the number of controllers used in embedded applications, for example on automobiles, is also increasing, and, to facilitate efficient communication between the controllers, complex time triggered protocols are being developed. These protocols are also parameterized as the number of controllers can vary. As with cache coherence protocols, there are no efficient automated techniques for verifying these protocols. To model check complex protocols like these, we need efficient abstraction techniques.

In this thesis, we present an abstraction method called *environment abstraction* for verifying parameterized systems with replicated processes. The main insight in this technique is that, when a human designer reasons about a system with replicated components, (s)he tends to focus on a *reference* component and consider the *environment* around it. We formalize this insight and provide a rigorous framework for constructing such abstract models. The abstract models are quite precise and easy to construct. In most abstraction methods, liveness properties are more difficult to handle, even theoretically, than safety properties. Our method, however, has a simple extension to handle liveness properties. Finally, most automatic abstraction based methods for verifying parameterized systems use the atomicity assumption. In contrast, we are able to remove the atomicity assumption by adding monitor processes and, thus, verify protocols in their full generality. Our experiments with different cache and mutual exclusion protocols suggest that environment abstraction works extremely well in practice.

The insight of constructing an abstract model by considering one reference process and looking at the world around it can be generalized to different settings. Instead of just considering a collection of processes, each of which can talk with every other process,

we consider a richer model in which processes are arranged on the nodes of a network graph. Thus, in addition to replication of processes, we also have an arbitrary network graph to reason about. This problem structure is quite common in real life. For instance, network routing protocols, which route data through complicated networks of machines, have to function no matter what the structure of the network. Each of these machines runs the exact same routing protocol. While there has been work on verifying systems with replicated processes, there has not been much work on verifying systems with network graphs. In Chapter 6, we take the first steps towards verifying parameterized systems with network graphs. We consider the verification of two process properties and show how to decompose a system with a large network to a collection of systems with constant sized network graphs. The main idea is that it suffices to consider how the network looks from a pair of processes to figure out what properties the pair satisfies. It can also be shown that, for any pair of processes, there are only a finite number of possibilities for how the network around them can look like. The results presented in Chapter 6 also highlight an interesting contrast in the expressive power of LTL and CTL specifications. We show that, while decomposition of large network into smaller ones is possible for LTL specifications, it is not possible for CTL specifications. Informally, two process CTL specifications can encode information about the number of other processes in the system and, thus, decomposition is not possible for CTL properties.

7.2 Extensions

In this thesis, we considered parameterized systems with replicated processes. Environment abstraction is quite general and can be applied even when the replicated components are not processes or if there are multiple types of replication. For instance, we can think of a memory bank as a collection of identical components (ignoring the contents of the memory). Similarly, we can treat a collection of jobs waiting to be scheduled in a queue as an instance of replication (ignoring the specifics of the job). We believe this viewpoint will lead to useful abstractions.

The abstract model that we construct is doubly exponential in the number of local state variables. In real cache coherence protocols, the internal state of each cache can be quite complex and thus our method might fail. To get around this, the internal states of local caches themselves might have to be abstracted before applying environment abstraction. An interesting extension to our work would be to combine environment abstraction with standard abstraction for the internal states of the caches.

Our work in Chapter 6 lays the foundational results for the verification of parameterized systems with network graphs. While the system model does consider a network graph, the communication between the processes is very simple. An extension to our work would be to consider richer communication between processes. However, we suspect that the decomposition results may not exist even for LTL properties once we allow more than one token. It would be interesting to consider what restrictions to impose on the system model so that we can still obtain decomposition results.

The abstraction based approach we have presented for verification of parameterized

systems is just one possible approach. In most real world parameterized systems, it seems to be the case that, all possible two process behaviors are exhausted when the parameter value is just 4 or 5. If such a cutoff really exists, then parameterized verification is no different from ordinary verification. But, finding such cutoffs is very hard and no such cutoffs are currently known. A related idea is to determine cutoff on trace length: it would be extremely useful in practice if we can show that all interesting behaviors are exhibited by traces of length less than a certain cutoff c. For instance, we could use bounded model checkers, which are typically faster than the other types of model checkers, to explore the parameterized system up to depth c. If no bug is found, then our cutoff result ensures that the parameterized system is correct. While it seems such trace cutoffs must exist, no one has succeeded in finding them yet. Finding cutoff results is a challenging problem with significant practical impact.

Distributed and parallel systems are among the hardest systems for humans to reason about. Yet parallelism seems to afford the easiest route to scalability and increased performance. Consequently, highly parallel, distributed computer systems are becoming quite pervasive. Powerful verification techniques are required to ensure the correct functioning of these systems. Model checking, which performs an exhaustive search of the state space, seems ideally suited for verification of distributed systems. In this thesis, we have addressed the problem of model checking distributed protocols like cache coherence protocols and mutual exclusion and demonstrated that it is possible to efficiently and automatically model check such protocols in their full generality.

Bibliography

[1] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular Model-Checking made Simple and Efficient. In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR)*, 2002.

[2] K. Apt and D. Kozen. Limits for Automatic Verification of Finite State Concurrent Systems. *Information Processing Letters*, 15:307–309, 1986.

[3] T. Arons, A. Pnueli, S. Ruah, and L. Zuck. Parameterized Verification with Automatically Computed Inductive Assertions. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)*, 2001.

[4] Thomas Ball and Sriram K. Rajamani. Boolean Programs: A Model and Process for Software Analysis. Technical Report MSR-TR-2000-14, Microsoft Research Corporation, 2000.

[5] Tom Ball, Sagar Chaki, and Sriram Rajamani. Verification of Multi-Threaded Software Libraries. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, 2001.

[6] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S Systems to Verify Parameterized Networks. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2000.

[7] K. Baukus, Y. Lakhnech, and K. Stahl. Verification of Parameterized Protocols. In *Journal of Universal of Computer Science*, 2001.

[8] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic Model Checking using SAT Procedures instead of BDDs. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 317–320, 1999.

[9] Jesse Bingham and Alan Hu. Empirically Efficient Verification for a Class of Infinite State Systems. In *Proceedings of the 11th International Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2005.

[10] B. Boigelot, A. Legay, and P. Wolper. Iterating Transducers in the Large. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*. LNCS, Springer-Verlag, 2003.

[11] A. Bouajjani, P. Habermehl, and T. Vojnar. Verification of Parametric Concurrent Systems with Prioritized FIFO Resource Management. In *Proceedings of 14th International Conference on Concurrency Theory (CONCUR)*, 2003.

[12] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*. LNCS, Springer-Verlag, 2000.

[13] A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. LNCS, Springer-Verlag, 2002.

[14] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59:115–131, 1988.

[15] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about Networks with Many Identical Finite State Processes. *Information and Computation*, 81:13–31, 1989.

[16] Randal E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[17] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Deciding CLU Logic Formulas via Boolean and Pseudo-Boolean Encodings. In *Proceedings of International Workshop on Constraints in Formal Verification*, 2002.

[18] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*, 2002.

[19] Sagar Chaki, Edmund Clarke, ALex Groce, Somesh Jha, and Helmut Veith. Modular Verification of Software Components in C. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 385–395, 2003.

[20] Prosenjit Chatterjee, Hemanthkumar Sivaraj, and Ganesh Gopalakrishnan. Shared Memory Consistency Protocol Verification against Weak Memory Models: Refinement via Distributed Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*, 2002.

[21] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A Simple Method for Parameterized Verification of Cache Coherence Protocols. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2004.

[22] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, 2000.

[23] E. Clarke, O. Grumberg, and D. Long. Model Checking and Abstraction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL)*, 1992.

[24] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Model Checking. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV)*, 1993.

[25] Bruno Courcelle. Graph Rewriting: An Algebraic and Logic Approach. *Handbook of Theoretical Computer Science*, Volume B:459–492, 1990.

[26] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixponts. In *Proceedings of the 4th ACM Symposium of Principles of Programming Languages (POPL)*, pages 238–272, 1977.

[27] D. L. Dill. The Murphi Verification System. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV)*, pages 390–393, 1996.

[28] Georgio Delzanno. Automated Verification of Cache Coherence Protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, 2000.

[29] Giorgio Delzanno and Tevfik Bultan. Constraint-Based Verification of Client-Server Protocols. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 286–301, 2001.

[30] E. A. Emerson and J. Y. Halpern. Decision Procedures and Expressiveness in the Temporal Logic of Branching Time. *Journal of Computer and System Sciences*, 30:1–24, 1985.

[31] E. A. Emerson, J. Havlicek, and R. Trefler. Virtual Symmetry. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.

[32] E. A. Emerson and Vineet Kahlon. Reducing Model Checking of the Many to the Few. In *Proceedings of the 17th International Conference on Automated Deduction(CADE)*, pages 236–254, 2000.

[33] E. A. Emerson and Vineet Kahlon. Model Checking Large-Scale and Parameterized Resource Allocation Systems. In *Proceedings of the 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 251–265, 2002.

[34] E. A. Emerson and Vineet Kahlon. Model Checking Guarded Protocols. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 361–370, 2003.

[35] E. A. Emerson and Vineet Kahlon. Rapid Parameterized Model Checking of Snoopy Cache Protocols. In *Proceedings of the 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 144–159, 2003.

[36] E. A. Emerson and Kedar Namjoshi. Automatic Verification of Parameterized Synchronous Systems. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV)*, 1996.

[37] E. A. Emerson and Kedar S. Namjoshi. Reasoning About Rings. In *Proceedings of 22nd ACM Symposium on Principles of Programming Languages (POPL)*, 1995.

[38] E. A. Emerson and A. P. Sistla. Symmetry and Model Checking. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV)*, 1993.

[39] E. A. Emerson and A.P Sistla. Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata Theoretic Approach. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4, 1997.

[40] E. A. Emerson and R. Trefler. From Asymmetry to Full Symmetry. In *Proceedings of Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, 1999.

[41] Yi Fang, Nir Piterman, A. Pnueli, and L. Zuck. Liveness with Incomprehensible Ranking. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation*, 2004.

[42] Yi Fang, Nir Piterman, A. Pnueli, and L. Zuck. Liveness with Invisible Ranking. In *Proceedings of the 10th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2004.

[43] S. M. German and A. P. Sistla. Reasoning about Systems with Many Processes. *Journal of the ACM*, 39, 1992.

[44] Steven German. Cache Coherence Examples, 2006.

[45] Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. Race Checking with Context Inference. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[46] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from Proofs. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*, pages 232–244, 2004.

[47] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software Verification with Blast. In *Proceedings of the 10th SPIN Workshop on Model Checking Software*, pages 235–239, 2003.

[48] Gerard J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[49] J. Rushby. A Formally Verified Algorithm for Clock Synchronization Under a Hybrid Fault Model. In *13th ACM Symposium on Principles of Distributed Computing*, pages 304–313, Los Angeles, CA, 1994.

[50] Prof. J.L.Lions. Ariane 5: Flight 501 Failure. Report by the Inquiry Board. In *www.ima.umn.edu/∼arnold/disasters/ariane5rep.html*.

[51] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proceedings of the 9th International Conference on Computer Aided Verification(CAV)*, volume 1254 of *LNCS*, pages 424–435. Springer, June 1997.

[52] Shuvendu K. Lahiri and Randy Bryant. Constructing Quantified Invariants. In *Proceedings of the 10th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2004.

[53] Shuvendu K. Lahiri and Randy Bryant. Indexed Predicate Discovery for Unbounded System Verification. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, 2004.

[54] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, 1974.

[55] Z. Manna and A. Pnueli. An Exercise in the Verification of Multi – Process Programs. In *Beauty is Our Business*, 1990.

[56] Milo K. Martin. Formal Verification and its Impact on Snooping vs Directory Protocol Debate. In *Proceedings of the International Conference on Computer Design (ICCD)*, 2005.

[57] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Computer Science Department, 1992.

[58] Kenneth L. McMillan. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. In *Proceedings of Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 179–195, 2001.

[59] Kenneth L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 250–264, 2002.

[60] Kenneth L. McMillan. Applications of Craig Interpolants in Model Checking. In *Proceedings of the 11th International Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 1–12, 2005.

[61] Kenneth L. McMillan and Nina Amla. Automatic Abstraction without Counterexamples. In *Proceedings of the 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 2–17, 2003.

[62] Kenneth L. McMillan, Shaz Qadeer, and James B. Saxe. Induction in Compositional Model Checking. In *Proceedings of the 12th International Conference on Computer Aided Verfication (CAV)*, pages 312–327, 2000.

[63] Seungjoon Park and David L. Dill. Verification of Cache Coherence Protocols by Aggregation of Distributed Transactions. In *Theory of Computing Systems*, pages 355–376, 1998.

[64] A. Pnueli, S. Ruah, and L. Zuck. Automatic Deductive Verification with Invisible Invariants. In *Proceedings of the 7th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2001.

[65] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 946–67, 1977.

[66] Amir Pnueli, Jessica Xu, and Lenore Zuck. Liveness with $(0, 1, \infty)$ Counter Abstraction. In *Proceedings of the 14th International Conference on Computer Aided Verification 2002 (CAV)*, 2002.

[67] Fong Pong and Michel Dubois. Formal Verification of Complex Coherence Protocols Using Symbolic State Models. *Journal of the ACM*, 45(4):557–587, 1998.

[68] Fong Pong and Michel Dubois. Formal Automatic Verification of Cache Coherence in Multiprocessors with Relaxed Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):989–1006, 2000.

[69] J. Rushby. An Overview of Formal Verification for the Time-Triggered Architecture. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science*, 2002.

[70] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, 1997.

[71] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2002.

[72] H. Saidi and N. Shankar. Abstract and Model Check while you Prove. In *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 443–454, 1999.

[73] M. Samer and H. Veith. Validity of CTL Queries Revisiting. In *Proceedings of 12th Annual Conference of the European Association for Computer Science Logic (CSL)*, 2003.

[74] M. Samer and H. Veith. A Syntactic Characterization of Distributive LTL Queries. In *Proceedings of 31st International Conference on Automata, Languages and Programming (ICALP)*, 2004.

[75] M. Samer and H. Veith. Deterministic CTL Query Solving. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME)*, 2005.

[76] I. Suzuki. Proving Properties of a Ring of Finite State Machines. *Information Processing Letters*, 28:213–214, 1988.

[77] B. K. Szymanski. A Simple Solution to Lamport's Concurrent Programming Problem with Linear Wait. In *Proceedings of the International Conference on Supercomputing Systems (ICS)*, 1988.

[78] T. Touili. Widening Techniques for Regular Model Checking. In *Proceedings of the 1st Vepas Workshop*. Volume 50 of Electronic Notes in Theoretical Computer Science, 2001.

[79] V. Pratt. Anatomy of the Pentium Bug. In *TAPSOFT'95: Theory and Practice of Software Development*, pages 97–107. Springer Verlag, 1995.

[80] Eran Yahav. Verifying Safety Properties of Concurrent Java Programs using 3-Valued Logic. In *In the Proceedings of 18th ACM Symposium on Principles of Programming Languages (POPL*, 2001.

[81] Eran Yahav. *Property Guided Abstractions of Concurrent Heap Manipulating Programs*. PhD thesis, Tel-Aviv University, Israel, 2004.