Algorithms and Methodology for Scalable Model Checking

by

Shaz Qadeer

B. Tech. (Indian Institute of Technology Kanpur) 1994
M. S. (University of California at Berkeley) 1997

A dissertation submitted in partial satisfaction of the requirements for the degree of
Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Thomas A. Henzinger, Chair
Professor Robert K. Brayton
Professor John Steel

Fall 1999
The dissertation of Shaz Qadeer is approved:


Chair


Date


Date


Date


University of California at Berkeley


1999
Algorithms and Methodology for Scalable Model Checking

Copyright 1999
by
Shaz Qadeer
Abstract

Algorithms and Methodology for Scalable Model Checking

by

Shaz Qadeer

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Thomas A. Henzinger, Chair

Model checking algorithms for the verification of reactive systems proceed by a systematic
and exhaustive exploration of the system state space. They do not scale to large designs
because of the state explosion problem—the number of states grows exponentially with
the number of components in the design. Consequently, the model checking problem is
PSPACE-hard in the size of the design description. This dissertation proposes three novel
techniques to combat the state explosion problem.

One of the most important advances in model checking in recent years has been
the discovery of symbolic methods, which use a calculus of expressions to represent the
state space and the state sets encountered during state space exploration. Symbolic model
checking has proved to be quite effective for verifying hardware designs. Previously, symbolic
checking of temporal logic specifications was done by backward fixpoint reasoning with the
operator pre. Backward reasoning can be wasteful since unreachable states are explored,
and empirical studies have shown that evaluating pre on unreachable state sets is expensive.
We suggest the use of forward fixpoint reasoning based on the operator post. We show how
all linear temporal logic properties can be model checked by forward reasoning. In contrast
to backward reasoning, forward reasoning performs computations only on the reachable
states.

Heuristics that improve algorithms for application domains, such as symbolic
methods for hardware designs, are useful but not enough to make model checking feasible
on industrial designs. Currently, exhaustive state exploration is possible only on designs
with about 50-100 boolean state variables. Assume-guarantee verification attempts to com-
bat the state explosion problem by using the principle of “divide and conquer,” where the components of the implementation are analyzed one at a time. Typically, an implementation component refines its specification only when its inputs are suitably constrained by other components in the implementation. The assume-guarantee principle states that instead of constraining the inputs by implementation components, it is sound to constrain them by the corresponding specification components that are much smaller. We extend the assume-guarantee proof rule to deal with the case where the specification operates at a coarser time scale than the implementation. Using our model checker Mocha that implements this methodology, we verify VGI, a parallel DSP processor chip with 64 compute processors each containing ~800 state variables and ~30K gates.

Our third contribution is a systematic model checking methodology for verifying the abstract shared-memory interface of sequential consistency on multiprocessor systems with three parameters — number of processors, number of memory locations, and number of data values. Sequential consistency requires that some interleaving of the local temporal orders of read/write events at different processors be a trace of serial memory. Therefore, it suffices to construct a non-interfering serializer that watches and reorders read/write events so that a trace of serial memory is obtained. While in general such a serializer must be unbounded even for fixed values of the parameters — checking sequential consistency is undecidable! — we show that the paradigmatic class of snoopy cache coherence protocols has finite-state serializers. Therefore sequential consistency for fixed parameter values can be checked by language inclusion between finite automata. In order to reduce the arbitrary-parameter problem to the fixed-parameter problem, we develop a novel framework for induction over the number of processors. We show that under certain reasonable assumptions about the memory system, it is possible to conclude sequential consistency for any number of processors, memory locations, and data values by model checking two finite-state systems involving two processors each accessing a maximum of three locations, where each location stores at most two data values.

Professor Thomas A. Henzinger
Dissertation Committee Chair
To
Dr. Abdul Qadeer Siddiqi
and
Qamar Siddiqi
Contents

List of Figures vi
List of Tables vii
1 Introduction 1

2 Symbolic Model Checking with Forward Reasoning 7
  2.1 Definition of Pre and Post Logics ............................. 10
  2.1.1 Pre and post μ-calculi ...................................... 10
  2.1.2 Query logics .................................................. 14
  2.1.3 Equivalences on Kripke structures induced by pre and post logics 15
  2.2 Intersection of Pre and Post Logics ............................. 18
    2.2.1 In ......................................................... 18
    2.2.2 Out ..................................................... 30
  2.3 Hierarchy of Pre and Post Logics ................................ 32
  2.4 Discussion and Experimental Results ............................ 35
    2.4.1 Intersection of pre and post logics ....................... 35
    2.4.2 Union of pre and post logics ............................. 36
    2.4.3 Experimental results ..................................... 36

3 Assume-Guarantee Reasoning 38
  3.1 Transition constraints .......................................... 40
  3.2 Reactive modules .............................................. 47
  3.3 Verification of three-stage pipeline ........................... 62
  3.4 Verification of Tomasulo’s algorithm .......................... 66

4 Assume-Guarantee Reasoning with Sample 75
  4.1 The sample operator ............................................ 80
  4.2 Refinement checking with sample ................................ 83
  4.3 Verification of VGI ............................................. 87
    4.3.1 The Problem ............................................... 89
    4.3.2 The Proof ................................................ 91
    4.3.3 Discussion ................................................. 94
5 Verifying Parameterized Shared-Memory Systems .......................... 97
  5.1 I/O-processes ........................................................................ 102
  5.2 Parameterized memory systems ........................................ 108
  5.3 Sequential consistency ....................................................... 110
  5.4 Parameterized memory systems with fixed number of processors . 114
  5.5 Parameterized memory systems with arbitrary number of processors . 118
    5.5.1 Induction on the set of processors ............................... 118
    5.5.2 Reduction to a fixed number of memory locations and data values . 122
  5.6 Verification of a snoopy cache coherence protocol ................. 127

6 Conclusions ........................................................................... 131

Bibliography ............................................................................. 135

A Sliding window protocol .......................................................... 147
List of Figures

2.1 $K_1$ and $K'_1$ are post-bisimilar .............................................. 31
2.2 $K_2$ and $K'_2$ are post-bisimilar .............................................. 31
2.3 $K_3$ and $K'_3$ are post-bisimilar but not pre-bisimilar ...................... 32
2.4 $K_4$ and $K'_4$ are init-post-bisimilar but not post-bisimilar ............... 33

3.1 Specification of sender/receiver ................................................. 45
3.2 Implementation of sender/receiver .............................................. 46
3.3 Instruction set architecture ....................................................... 49
3.4 Pipeline stage 1 .................................................................. 51
3.5 Pipeline stages 2 and 3 ......................................................... 52
3.6 Pipeline output and stall ......................................................... 53
3.7 Specification of Sender ............................................................ 57
3.8 Specification of Receiver ......................................................... 58
3.9 Implementation of the sender ................................................... 59
3.10 Implementation of Receiver .................................................... 60

4.1 GCD Specification ................................................................. 78
4.2 GCD Implementation ............................................................. 78
4.3 VGI processor configuration with three input and two output queues ...... 90
4.4 Specification module for refinement check ...................................... 91

5.1 Snoopy cache coherence ........................................................... 128
5.2 Snoopy cache coherence invariant ............................................... 129
List of Tables

2.1 Experimental results for sliding window protocol .......................... 37
3.1 Size of lemmas in the proof of TOMASULO ................................. 74
Acknowledgements

I would like to acknowledge the contributions of Orna Kupferman and Sriram Rajamani who collaborated with me on the work described in this dissertation, and Freddy Mang whose unstinting efforts sustained the model checker MoCHA as a vehicle for my research. I am grateful to my advisors Robert Brayton and Thomas Henzinger for encouraging and guiding me during my stay at Berkeley, and for setting the standards of my research. Finally, I would like to acknowledge the influence of Rajeev Alur, Ken McMillan, Carl Pixley and Natarajan Shankar, who have encouraged and guided me, technically and otherwise, on numerous occasions.

The work in Chapter 2 is based on the paper “From pre-historic to post-modern symbolic model checking” [HKQ98], presented at the 10th International Conference on Computer Aided Verification 1998. The work in Chapter 3 is based on the paper “You assume, we guarantee: methodology and case studies” [HQR98], presented at the 10th International Conference on Computer Aided Verification 1998. The work in Chapter 4 is based on the papers “Assume-guarantee refinement between different time scales” [HQR99a], and “Formal specification and verification of a dataflow processor array” [HLQR99]. These papers were presented at the 11th International Conference on Computer Aided Verification 1999, and the IEEE/ACM International Conference on Computer Aided Design 1999, respectively. The work in Chapter 5 is based on the paper “Verifying sequential consistency on shared-memory multiprocessor systems” [HQR99b], presented at the 11th International Conference on Computer Aided Verification 1999.
Chapter 1

Introduction

This dissertation is concerned with the correctness problem for reactive systems [MP92]. A reactive system is composed of several processes operating and interacting with each other and their common environment, continuously and without termination, to produce a desired outcome. This desired outcome is called the specification of the system. An example of a reactive system is the sliding window protocol [Hol91] for reliably transmitting messages from one computer to another. The sliding window protocol consists of three processes — sender, receiver and channel. The protocol is supposed to interact continuously with its environment which produces and consumes messages, and to ensure that every message produced by the sender is eventually consumed by the receiver. Clearly we would like the protocol to work forever and hence non-termination is a desirable property.

Examples of reactive systems abound in the modern world. There are microprocessors and microcontrollers sitting in our desktop computers, in the network routers across the Internet, and in the switches of the telecommunications network. There is the operating system managing the hardware resources of the computer. There are protocols that make it possible for different parts of a distributed system like the Internet to talk to each other. The diverse and critical presence of such systems in the affairs of modern society makes it imperative that they work correctly. It is very difficult to design these systems correctly because of the combinatorial explosion of the possible interactions between the various components of the system and the environment. Hence there is a need for formally verifying such designs.

In formal verification, the system, also called the implementation is modeled mathematically and its specification is described in a formal language. The satisfaction relation
between the implementation and specification is then defined formally and checked. There are two broad approaches to formal verification of reactive systems — theorem proving and model checking. *Theorem proving* environments [BM79a, BM79b, GM93, GH93, ORR+96] typically use a very expressive logic for expressing the implementation and the specification. The implementation is expressed as a set of axioms, the specification is written down as a theorem to be proved in the axiomatic system and the verifier tries to discover a proof of the theorem according to the inference rules of the logic. Theorem proving has all the expressive power of higher order logic at its disposal and complex implementations and specifications can be easily expressed. But the logics used are mostly undecidable and theorem proving systems require a lot of interaction with the verifier. Although some automation has been introduced, the process of proof discovery remains largely manual and quite tedious. The lack of automation makes it difficult for theorem proving to be used in an industrial context.

*Model checking* [CE81, QS81] uses finite state machines as a model for the implementation. Either temporal logic [Pnu77] or nondeterministic finite state machines [MS72] are used for specification. If temporal logic is used as specification, the verification problem is called *property checking*. If nondeterministic finite state machines are used as specification, the verification problem is called *refinement checking*. Both property checking and refinement checking are decidable and the algorithms typically work by exploring the finite state space of the implementation. The automatic nature of model checking makes it attractive for practical use in the industry. The applicability of model checking is somewhat restricted though, for the following reasons.

1. A number of important systems are not finite-state; they are infinite-state or parameterized. Distributed algorithms [Lyn96] in an operating system that use integer variables are an example of infinite-state systems. Shared-memory multiprocessor protocols [ABM93, LLG+92, KOH+94, HP96a, SG97] that are supposed to work for an arbitrary number of processors and memory locations are examples of parameterized systems. Model checking uses finite automata for the implementation and therefore cannot directly verify such systems.

2. Even for finite-state systems, model checking algorithms do not scale. Reactive systems are typically described as composition of a number of smaller processes. The state space of a process $A$, expressed as a composition of processes $A_1, A_2, \ldots, A_n$
with state spaces of size $S_1, S_2, \ldots, S_n$ respectively, is $S_1 \times S_2 \times \cdots \times S_n$, which is exponential in $S_1 + S_2 + \cdots + S_n$ in the worst case. Since model-checking algorithms explore the state space of the implementation exhaustively, complexity is directly proportional to the size of the state space and consequently exponential in the size of system description. This exponential growth of the state space with the number of components in the system is called the state explosion problem. This problem has limited current model checking techniques to the verification of systems with 50-100 boolean state variables.

This dissertation has addressed both the problems mentioned above. Chapters 2, 3 and 4 discuss techniques to make model checking scalable on finite-state systems. Chapter 5 focuses on the verification of a particular class of parameterized systems, shared-memory multiprocessor protocols, and attempts to make the problem amenable to model checking.

There are two general classes of techniques for combating the state explosion problem in model checking. Type-1 techniques focus on improving algorithms, often developing heuristics that target specific application domains, such as symbolic methods [BCM+92, McM93] for synchronous hardware designs, and partial-order methods [Val90, GW91, Pel93, God96] for asynchronous communication protocols. Type-2 techniques focus on dividing the verification task at hand into simpler tasks, often making use of the compositional structure of both implementation and specification, such as assume-guarantee methods [MC81, Sta85, AL95, AH96, McM97] for proof decomposition. While type-1 techniques can be applied fully automatically and improve the efficiency of formal verification, they need to be complemented by type-2 techniques in order to make the approach fully scalable. Type-2 techniques, however, require substantial assistance from human verification experts, and their systematic application in nontrivial situations remains somewhat of a black art.

In Chapter 2, we discuss a new type-1 technique for property checking. Currently, there are two techniques for performing state space exploration — enumerative [HP96b, DiI96] and symbolic [McM93, HSV96]. In enumerative search, states and transitions are explored individually. A state is represented as a record and the explored states are stored in a table for bookkeeping. On the other hand, symbolic search uses a calculus of expressions [Koz83] to represent state sets and operations on them. State exploration is encoded as a fixpoint computation on these expressions. Both enumerative and symbolic
searches can be done either forward in which transitions from explored states are taken, or backward in which transitions to explored states are taken. Previously, symbolic checking of temporal logic specifications was done by backward fixpoint reasoning with the operator $pre$ [BCM+92]. For any state set $A$, this operator computes the state set $pre(A)$ that can reach $A$ in one step. Backward reasoning can be wasteful since unreachable states are explored, and empirical studies have shown that evaluating $pre$ on unreachable state sets is expensive [INH96]. We suggest the use of forward fixpoint reasoning based on the operator $post$. For any state set $A$, this operator computes the state set $post(A)$ that can be reached from $A$ in one step. We show how all linear temporal logic properties can be model checked by forward reasoning. In contrast to backward reasoning, forward reasoning performs computations only on the reachable states.

In Chapter 3, we discuss a type-2 technique called assume-guarantee reasoning. This technique has been implemented in the model checker MOCHA [AHM+98], that was developed in collaboration with other students in the group. In MOCHA, both implementation and specification are compositions of FSMs. The implementation refines the specification if every behavior of the implementation is a behavior of the specification. MOCHA works on the principle of “divide and conquer.” Instead of putting all implementation components together to get a flat representation, MOCHA analyzes one component at a time. Typically, an implementation component refines its specification only when its inputs are suitably constrained by other components in the implementation. The assume-guarantee principle states that instead of constraining the inputs by implementation components, it is sound to constrain them by the corresponding specification components, thereby avoiding the composition of large implementation FSMs. If the specification does not mention some of the inputs, then it can be enriched by defining those inputs abstractly. Thus, MOCHA utilizes the input design structure to decompose the refinement proof into smaller proof obligations that are within the capacity of the state exploration engine, while making sure that the decomposition is sound by the principle of assume-guarantee reasoning.

For some systems, it is easier to express the specification such that multiple steps of the implementation correspond to a single step of the specification. Consider a system in which process $A$ uses a rather complicated protocol taking several steps to send a message to process $B$ which takes some action based on it. A simple specification would just say that the message sent by $A$ reaches $B$, thereby collapsing several steps of the implementation into a single step of the specification. In Chapter 4, we introduce the sampling operator
that is used to relate an implementation to a specification operating at a coarser time scale. We generalize the assume-guarantee rule of Chapter 3 to incorporate the sample operator. The generalized rule has been implemented in Mocha and used to verify VGI, a parallel DSP processor chip designed by the Infopad group at Berkeley. The size of the chip was clearly beyond the capacity of existing model checking techniques—the chip has 64 compute processors each containing \(~\sim\)800 state variables and \(~\sim\)30K gates, and required sustained effort by several people over a few months to specify and verify it.

In Chapter 5, we develop a systematic model checking methodology for verifying abstract shared-memory interfaces on multiprocessor systems. The design of a correct and efficient shared memory is one of the most difficult tasks in the design of such systems [AG96]. The shared-memory interface is a contract between the designer and the programmer of the multiprocessor. All abstract memory models can be understood in terms of the fundamental \textit{serial-memory} model. A serial memory behaves as if there is a centralized memory that services read and write requests atomically such that a read to a location returns the latest value written to that location. \textit{Sequential consistency} [Lam79] offers a natural tradeoff between the flexibility afforded to the implementor and the complexity of the programmer's view of the memory. Sequential consistency requires that some interleaving of the local temporal orders of read/write events at different processors be a trace of serial memory. We develop a systematic methodology for proving sequential consistency for memory systems with three parameters—number of processors, number of memory locations, and number of data values. From the definition of sequential consistency it suffices to construct a non-interfering serializer [KP92] that watches and reorders read/write events so that a trace of serial memory is obtained. While in general such a serializer must be unbounded even for fixed values of the parameters—checking sequential consistency is undecidable! [AMP96]—we show that the paradigmatic class of \textit{snoopy cache coherence} protocols [HP96a] has finite-state serializers. Therefore sequential consistency for fixed parameter values can be checked by language inclusion between finite automata.

In order to reduce the arbitrary-parameter problem to the fixed-parameter problem, we develop a novel framework for induction over the number of processors. Classical induction schemas [KM89], which are based on process invariants that are inductive with respect to an implementation preorder that preserves the temporal sequence of events, are inadequate for our purposes, because proving sequential consistency requires the reordering of events. Hence we introduce \textit{merge invariants}, which permit certain reorderings of
read/write events. We show that under certain reasonable assumptions about the memory system, it is possible to conclude sequential consistency for any number of processors, memory locations, and data values by model checking two finite-state systems involving process and merge invariants: they involve two processors each accessing a maximum of three locations, where each location stores at most two data values.

In this dissertation, for each model checking technique we have used the simplest modeling language that can be used to described it. Chapter 2 describes a type-1 technique that works off the full state space of the system and does not make use of the its compositional structure. Hence, we use Kripke structures [Eme90], which are state transition graphs where the states are labeled with observations. Chapters 3 and 4 describe a type-2 technique that makes use of the structure of the design and tries to avoid building the full state space. Hence we use transition constraints with private and observable variables, and initial and transition predicates on them. Transition constraints are closed under composition and therefore they can be used to describe complex systems compositionally. Since they can easily be translated to Kripke structures, the technique in Chapter 2 is also applicable to them. Chapter 3 also uses a special class of transition constraints called reactive modules [AH96], which partitions the observable variables into inputs and outputs and also have executable non-blocking semantics. Chapter 5 describes a verification technique for shared-memory multiprocessors. Typically, the components in a shared memory multiprocessor synchronize by means of observable actions. A component performs an output action to synchronize with other components for which that action is an input action [Hoal85, Mi189]. Hence, we use I/O-processes that are state transition graphs with edges labeled by private, observable input, or observable output actions. I/O-processes are also closed under composition and can be used to describe a complex memory protocol compositionally. The technique of Chapter 2 is applicable, after simple modifications, also to I/O-processes.
Chapter 2

Symbolic Model Checking with Forward Reasoning

The discovery of symbolic model-checking methods [BCM+92] has been a very important development for achieving the goal of automatic verification. Traditional symbolic model-checking tools have been based on backward state traversal [McM93, HSV+96]. They compute expressions that represent state sets using, in addition to positive boolean operations, the functions \( \text{pre} \) and \( \overline{\text{pre}} \), which map a set of states to a subset of its predecessor states. Formally, given a set \( U \) of states, the set \( \text{pre}(U) \) contains the states for which there exists a successor state in \( U \), and the set \( \overline{\text{pre}}(U) \) contains the states all of whose successor states are in \( U \). By evaluating fixpoint expressions over boolean and pre operations, complicated state sets can be calculated. For example, to find the set of states from which a state satisfying a predicate \( p \) is reachable, the model checker starts with the set \( U \) of states in which \( p \) holds, and repeatedly adds to \( U \) the set \( \text{pre}(U) \), until no more states can be added. Formally, the model checker calculates the least fixpoint of the expression \( U = p \lor \text{pre}(U) \).

Binary decision diagrams (BDDs) [Bry86] have been used to represent the state transition graph of the system and the state sets encountered in evaluating these expressions. The use of BDDs has yielded model-checking tools that can handle very large state spaces [CGL94].

Specification logics can be broadly classified into two categories—branching-time and linear-time. Branching-time specifications are concerned with the branching nature of the system's state transition graph, whereas linear-time specifications ignore the branching structure and treat the state transition graph as a collection of paths of states. Symbolic
model-checking techniques were first applied to branching-time specifications by first translating them into fixpoint expressions [BCM+92] and then evaluating the expression on the state space of the system. Symbolic techniques were extended to linear-time specifications [CGH94] through the use of tableau construction. First, the automata representing the negation of the property, also called its tableau, is constructed. Then it is composed with the state transition graph of the system and the emptiness of the result is checked by evaluating a fixpoint expression.

As an alternative to symbolic model checking, in enumerative model checking states are represented individually. Traditional enumerative model-checking tools check linear-time specifications by forward state traversal [HP96b, Dil96]. There, the basic operation is to compute, for a given state, the list of successor states. Forward state traversal has several obvious advantages over backward state traversal. First, for operational system models, successor states are often easier to compute than predecessor states. Second, only the reachable part of the state space is traversed. Third, optimizations such as on-the-fly [GPVW95] and partial-order [Pel94] methods can be incorporated naturally. For example, in on-the-fly model checking, only those parts of the state space are traversed which are relevant for satisfying (or violating) the given specification.

Some of the advantages of forward state traversal can be easily incorporated into symbolic methods. For example, we may first compute the set of reachable states by symbolic forward state traversal, and then restrict backward state traversal to model checking of the reachable states. This method, however, is unsatisfactory; for example, it cannot find even a short error trace if the set of reachable states cannot be computed. We present a tighter, and more advantageous, integration of forward state traversal with symbolic methods. In symbolic forward state traversal, we replace the functions pre and \( \widehat{\text{pre}} \) by the functions post and \( \widehat{\text{post}} \), respectively, which map a set of states to a subset of its successor states. Formally, given a set \( U \) of states, the set \( \text{post}(U) \) contains the states for which there exists a predecessor state in \( U \), and the set \( \widehat{\text{post}}(U) \) contains the states all of whose predecessor states are in \( U \). Then, we evaluate fixpoint expressions over boolean and post operations on state sets. It has recently been shown that certain branching-time as well as linear-time specifications, such as response (i.e., \( \Box(p \rightarrow \Diamond q) \)), can be model checked by symbolic forward state traversal [INH96, IN97]. We attempt a more systematic study of what can and what cannot be model checked in this way. In particular, we show that all \( \omega \)-regular (linear-time) specifications (which include all LTL specifications) are amenable
to a symbolic forward approach, while some CTL (branching-time) specifications are not. Notice that while the symbolic LTL model checking method [CGH94] can be easily modified to use fixpoint expressions over post, it involves composing the automaton of the negation of the property with the system and results in search over an expanded state space. Our method, on the other hand, performs a fixpoint computation over the state space of the system only.

For a systematic study of the properties that can be checked with forward reasoning, we define post-\(\mu\), a fixpoint calculus that is based on post operations in the same way in which the traditional \(\mu\)-calculus, here called pre-\(\mu\), is based on pre operations [Koz83]. While pre-\(\mu\) expressions refer to the future of a given state in a model, post-\(\mu\) expressions refer to its past. Therefore, in stark contrast to the fact that every LTL and CTL specification has an equivalent expression in pre-\(\mu\), almost no LTL or CTL specification, including response, has an equivalent expression in post-\(\mu\). In order to compare pre and post logics, rather, we need to define query logics, whose formulas refer to a whole model, not an individual state. Query logics are based on the emptiness predicate \(E\). For a specification \(\phi\), which is true in some states of a model and false in others, the query \(E(\phi)\) is true in a model iff \(\phi\) is false in all states of the model. The query logic post-\(\mu_0\) contains all queries of the form \(E(\phi)\) and \(\neg E(\phi)\), for post-\(\mu\) expressions \(\phi\). On the positive side, we prove that every \(\omega\)-regular (Büchi) specification has an equivalent query in post-\(\mu_0\). As with the translation from Büchi automata to pre-\(\mu\) expressions [EL86, BC96], the translation from Büchi automata to post-\(\mu_0\) queries is linear and involves only fixpoint expressions of alternation depth two. Moreover, we show that every co-Büchi specification has an equivalent query in alternation-free post-\(\mu_0\), which can be checked efficiently (in linear time). On the negative side, we prove that there are CTL specifications that are not equivalent to any boolean combination of post-\(\mu_0\) queries.

Symbolic forward model checking combines the benefits of symbolic over enumerative state traversal with the benefits of forward over backward state traversal. In [INH96, IN97], the authors present experimental evidence that symbolic forward state traversal can be significantly more efficient than symbolic backward state traversal. Our preliminary experimental results confirm this observation. In addition, we give some theoretical justifications for the symbolic forward approach. We show that unlike enumerative forward model checking (which is traditionally based on depth-first state traversal) and unlike symbolic backward model checking, the symbolic forward approach guarantees a.s.a.p.
CHAPTER 2. SYMBOLIC MODEL CHECKING WITH FORWARD REASONING

error detection. Intuitively, if a model violates a safety specification, and the shortest error trace has length $m$, then the breadth-first nature of symbolic forward model checking ensures that the error will be found before any states at a distance greater than $m$ from the initial states are explored.

The remainder of this chapter is organized as follows. In Section 2.1 we define the logics $pre-\mu$ and $post-\mu$, and the query logics they induce. In Section 2.2, we translate Büchi automata into equivalent $post-\mu_0$ queries of alternation depth two, and co-Büchi automata into equivalent alternation-free $post-\mu_0$ queries. We also show that the translation guarantees a.s.a.p. error detection for safety specifications. In Section 2.3, we compare the distinguishing and expressive powers of the various pre, post, and query logics. Finally, in Section 2.4 we put our results in perspective and report on some experimental evidence for the value of symbolic forward model checking.

2.1 Definition of Pre and Post Logics

2.1.1 Pre and post $\mu$-calculi

The $\mu$-calculus is a modal logic augmented with least and greatest fixpoint operators [Koz83]. In this paper, we use the equational form of the propositional $\mu$-calculus, as in [CKS92]. The modalities of the $\mu$-calculus relate a set of states to a subset of its predecessor states. Therefore, we refer to the $\mu$-calculus by $pre-\mu$.

The formulas of $pre-\mu$ are defined with respect to a set $P$ of propositions and a set $V$ of variables. A modal expression is either $p$, $\neg p$, $X$, $\varphi \lor \psi$, $\varphi \land \psi$, $\exists \varphi$, or $\forall \varphi$, for propositions $p \in P$, variables $X \in V$, and modal expressions $\varphi$ and $\psi$. Let $I$ be a finite subset of the set of natural numbers. An equational block $B = \langle \lambda, \{X_i = \varphi_i \mid i \in I\} \rangle$ consists of a flag $\lambda \in \{\mu, \nu\}$ and a finite set of equations $X_i = \varphi_i$, where each $X_i$ is a variable, each $\varphi_i$ is a modal expression, and the variables $X_i$ are pairwise distinct. If $\lambda = \mu$, then $B$ is a $\mu$-block; otherwise $B$ is a $\nu$-block. Let the function flag map each block to its flag. For the equational block $B$, let $\text{vars}(B) = \{X_i \mid i \in I\}$ be the set of variables on the left-hand sides of the equations of $B$. A block tuple $B = \langle B_1, \ldots, B_n \rangle$ is a finite list of equational blocks such that the variable sets $\text{vars}(B_j)$, for $1 \leq j \leq n$, are pairwise disjoint. For the block tuple $B$, let $\text{vars}(B) = \bigcup_{1 \leq j \leq n} \text{vars}(B_j)$. For every variable $X \in \text{vars}(B)$, let $\text{expand}_B(X)$ be the modal expression on the right-hand side of the unique equation in $B$ whose left-hand
side is $X$. A pre-$\mu$ formula $\phi = (B, X_0)$ is a pair that consists of a block tuple $B$ and a variable $X_0 \in \text{vars}(B)$. The variable $X_0$ is called the root variable of $\phi$. The formula $\phi$ is a pre-$\mu$ sentence if every variable that occurs in some modal expression of $B$ is a member of $\text{vars}(B)$.

The semantics of a pre-$\mu$ formula is defined with respect to a Kripke structure and a valuation for the variables. A Kripke structure is a tuple $K = (P, W, R, L)$ that consists of a finite set $P$ of propositions, a finite set $W$ of states, a binary transition relation $R \subseteq W \times W$ total in both the first and second arguments (i.e., for every state $w \in W$, there is a state $w'$ such that $R(w, w')$ and there is a state $w''$ such that $R(w'', w)$), and a labeling function $L : W \to 2^P$ that assigns to each state a set of propositions. The set $P$ of propositions contains the distinguished proposition $\text{init}$; a state $w \in W$ is initial if $\text{init} \in L(w)$. We define four functions $\text{pre}$, $\text{pre}$, $\text{post}$ and $\text{post}$ from $2^W$ to $2^W$ as follows.

For any set $U \subseteq W$ of states, let

$$
\text{pre}(U) = \{ w \in W \mid \text{there exists a state } w' \in U \text{ with } R(w, w') \},
$$

$$
\text{pre}(U) = \{ w \in W \mid \text{for all states } w' \text{ with } R(w, w'), \text{ we have } w' \in U \},
$$

$$
\text{post}(U) = \{ w \in W \mid \text{there exists a state } w' \in U \text{ with } R(w', w) \},
$$

$$
\text{post}(U) = \{ w \in W \mid \text{for all states } w' \text{ with } R(w', w), \text{ we have } w' \in U \}.
$$

The functions $\text{pre}$ and $\text{post}$ can be defined on states $u \in W$ as $\text{pre}(u) = \text{pre}(\{u\})$ and $\text{post}(u) = \text{post}(\{u\})$ respectively.

A $K$-valuation for a set $V$ of variables is a function $\Gamma : V \to 2^W$ that assigns to each variable a set of states. If $\Gamma$ and $\Gamma'$ are $K$-valuations for $V$, and $V' \subseteq V$ is a subset of the variables, we write $\Gamma[\Gamma'/V']$ for the $K$-valuation for $V$ that assigns $\Gamma'(X)$ to each variable $X \in V'$, and $\Gamma(X)$ to each variable $X \in V \setminus V'$. Let $\mathcal{P}_{V,K}$ be the set of all $K$-valuations over $V$. We extend the set union $\cup$ and set intersection $\cap$ to $\mathcal{P}_{V,K}$, where they denote the pointwise set union and intersection, respectively. If, for example, $\Gamma_1(X) = S_1$ and $\Gamma_2(X) = S_2$, then $(\Gamma_1 \cup \Gamma_2)(X) = S_1 \cup S_2$ and $(\Gamma_1 \cap \Gamma_2)(X) = S_1 \cap S_2$. We define the pointwise set-containment relation $\subseteq$ over $\mathcal{P}_{V,K}$ as $\Gamma \subseteq \Gamma'$ iff for all $X \in V$, we have that $\Gamma(X) \subseteq \Gamma'(X)$. The tuple $\langle \mathcal{P}_{V,K}, \subseteq \rangle$ is a complete lattice. For any subset $Y \subseteq \mathcal{P}_{V,K}$, let $\text{glb}(Y)$ and $\text{lub}(Y)$ denote the greatest lower bound and the least upper bound of $Y$ respectively. Let $f$ be a function from $\mathcal{P}_{V,K}$ to $\mathcal{P}_{V,K}$. The function $f$ is monotonic if for all $\Gamma, \Gamma' \in \mathcal{P}_{V,K}$, if $\Gamma \subseteq \Gamma'$ then $f(\Gamma) \subseteq f(\Gamma')$. The function $f$ is said to be continuous if for all chains $\Gamma_0 \subseteq \Gamma_1 \subseteq \Gamma_2 \subseteq \ldots$, we have that $f(\text{lub}(\{\Gamma_i \mid i \geq 0\})) = \text{lub}(\{f(\Gamma_i) \mid i \geq 0\})$. Since
both $V$ and $K$ are finite, the set $\mathcal{P}_{V;K}$ is also finite. In that case, it can be shown easily that $f$ is monotonic iff $f$ is continuous. A valuation $\Gamma \in \mathcal{P}_{V;K}$ is a fixpoint of $f$ if $f(\Gamma) = \Gamma$.

**Theorem 2.1 (Knaster-Tarski)** Let $K$ be a Kripke structure, $V$ be a set of propositional variables and $f : \mathcal{P}_{V;K} \to \mathcal{P}_{V;K}$ be a monotonic function. Then, the greatest fixpoint of $f$ is $\text{lub}(\{\Gamma | \Gamma \subseteq f(\Gamma)\})$ and the least fixpoint of $f$ is $\text{glb}(\{\Gamma | f(\Gamma) \subseteq \Gamma\})$.

For all $i \geq 0$, define the function $f^i$ from $\mathcal{P}_{V;K}$ to $\mathcal{P}_{V;K}$ inductively as follows.

$$f^0(\Gamma) = \Gamma$$

$$f^{i+1}(\Gamma) = f(f^i(\Gamma))$$

Suppose $f$ is monotonic (and hence continuous). Let $\bot$ be the valuation that maps every variable to the empty set and let $\top$ be the valuation that maps every variable to $W$. Then the least fixpoint $\mu f = \bigcup_{i \geq 0} f^i(\bot)$ and the greatest fixpoint $\nu f = \bigcap_{i \geq 0} f^i(\top)$. This gives us a procedure for computing the least and greatest fixpoints of functions from $\mathcal{P}_{V;K}$ to $\mathcal{P}_{V;K}$ when $K$ is finite. Given a Kripke structure $K = \langle P, W, R, L \rangle$ and a $K$-valuation $\Gamma$ for a set $V$ of variables, every modal expression $\varphi$ over the propositions $P$ and the variables $V$ defines a set $\varphi^K(\Gamma) \subseteq W$ of states that satisfy $\varphi$. The definition proceeds inductively as follows.

- $p^K(\Gamma) = \{w \in W \mid p \in L(w)\}$.
- $(\neg p)^K(\Gamma) = \{w \in W \mid p \not\in L(w)\}$
- $X^K(\Gamma) = \Gamma(\mathcal{X})$.
- $(\varphi \lor \psi)^K(\Gamma) = \varphi^K(\Gamma) \cup \psi^K(\Gamma)$.
- $(\varphi \land \psi)^K(\Gamma) = \varphi^K(\Gamma) \cap \psi^K(\Gamma)$.
- $(\exists \varphi)^K(\Gamma) = \text{pre}(\varphi^K(\Gamma))$.
- $(\forall \varphi)^K(\Gamma) = \overline{\text{pre}}(\varphi^K(\Gamma))$.

Given $K$, every block tuple $\mathcal{B} = \langle B_1, \ldots, B_n \rangle$ over $P$ and $V$ defines a function $B^K$ from $\mathcal{P}_{V;K}$ to $\mathcal{P}_{V;K}$: inductively, if $n = 0$, then $B^K(\Gamma) = \Gamma$; if $B_1$ is a $\mu$-block, then $B^K(\Gamma)$ is the
least fixpoint of the function \( F^K_{B, \Gamma} \); if \( B_1 \) is a \( \nu \)-block, then \( B^K(\Gamma) \) is the greatest fixpoint of \( F^K_{B, \Gamma} \). The monotonic function \( F^K_{B, \Gamma} \) from \( P_{V, K} \) to \( P_{V, K} \) is defined by

\[
F^K_{B, \Gamma}(\Gamma')(X) = \begin{cases} 
\text{expand}(X)^K((B_2, \ldots, B_n)^K(\Gamma'[\Gamma'/\text{vars}(B_1)])) & \text{if } X \in \text{vars}(B_1), \\
((B_2, \ldots, B_n)^K(\Gamma'[\Gamma'/\text{vars}(B_1)]))(X) & \text{otherwise}.
\end{cases}
\]

Note that for a \( \premi \) sentence \( \phi = \langle B, X_0 \rangle \), the function \( B^K \) is a constant function. Given \( K \), the sentence \( \phi \) defines the set \( \phi^K = B^K(\Gamma)(X_0) \) of states (for any choice of \( \Gamma \)). For a state \( w \in W \) and a \( \premi \) sentence \( \phi \), we write \( w \models_K \phi \) if \( w \in \phi^K \). For a Kripke structure \( K \), we write \( K \models \phi \), and say that \( K \) satisfies \( \phi \), if there is an initial state \( w \) of \( K \) such that \( w \models_K \phi \).

The model-checking problem for \( \premi \) is to decide, given a Kripke structure \( K \) and a \( \premi \) sentence \( \phi \), whether \( K \models \phi \).

We now define the notion of alternation depth of a block tuple \( B \) as in [EL86, CKS92]. Given a block tuple \( B = \langle B_1, \ldots, B_n \rangle \), the block \( B_i \) depends on the block \( B_j \) if \( i \neq j \) and some variable that occurs in a modal expression of \( B_i \) is contained in \( \text{vars}(B_j) \). We write \( B_j \to B_i \) if \( B_i \) depends on \( B_j \). The transitive closure of \( \to \) is represented by \( \to^* \).

Define a function \( \text{label} \) from the set of blocks to \( \mathbb{N} \) by induction as follows.

\[
\text{label}(B_n) = 1, \\
\text{label}(B_i) = \max(Y_i \cup Z_i),
\]

where

\[
Y_i = \{1\} \cup \{\text{label}(B_j)|j > i \land \text{flag}(B_j) = \text{flag}(B_i) \land B_j \to^* B_i \to^* B_j\}, \quad \text{and}
\]

\[
Z_i = \{1\} \cup \{1 + \text{label}(B_j)|j > i \land \text{flag}(B_j) \neq \text{flag}(B_i) \land B_j \to^* B_i \to^* B_j\}.
\]

The alternation depth of \( B \) is defined to be \( \max(\{\text{label}(B_i)|1 \leq i \leq n\}) \). The block tuple \( B \) is said to be alternation-free if the alternation depth of \( B \) is one. The \( \premi \) sentence \( \phi = \langle B, X_0 \rangle \) is alternation-free if \( B \) is alternation-free.

**Example 2.1** Consider the \( \premi \) formula \( \varphi = \langle B_1, B_2 \rangle \), where \( B_1 \) is a \( \nu \)-block containing the single equation \( X_1 = \exists \Diamond X_2 \) and \( B_2 \) is a \( \mu \)-block containing the single equation \( X_2 = (p \land X_1) \lor \exists \Diamond X_2 \). This formula is equivalent to the linear temporal logic formula \( \Box \Diamond p \) Then \( \text{label}(B_2) = 1 \). Also \( B_1 \to B_2 \) and \( B_2 \to B_1 \). Therefore \( Y_1 = \{1\} \) and \( Z_1 = \{1, 2\} \), and \( \text{label}(B_1) = \max(\{1, 2\}) = 2 \). Thus we get that the alternation depth of \( \varphi \) is two. 

The logic \( \text{post}\-\mu \) is obtained from the logic \( \premi \) by replacing the future modal operators \( \exists \Diamond \) and \( \forall \Diamond \) by the past modal operators \( \exists \Box \) and \( \forall \Box \), with the interpretations

\footnote{Note that we work, for convenience, with the dual of the usual requirement that all initial states satisfy a \( \premi \) sentence.}
$(\exists \varphi^K(\Gamma) = \text{post}(\varphi^K(\Gamma))$ and $(\forall \varphi^K(\Gamma) = \text{pre}(\varphi^K(\Gamma))$. The semantics of \textit{post-} can alternatively be defined as follows. For a Kripke structure $K = \langle P, W, R, L \rangle$, define the Kripke structure $K^{-1} = \langle P, W, R^{-1}, L \rangle$, where $R^{-1}(w, w')$ iff $R(w', w)$. For a \textit{post-} sentence $\phi$, define $\phi^{-1}$ to be the \textit{pre-} sentence obtained from $\phi$ by replacing each occurrence of $\exists \varphi$ and $\forall \varphi$ by $\exists \varphi$ and $\forall \varphi$, respectively. Then, for every state $w$ of $K$, we have $w \models K \phi$ iff $w \models K^{-1} \phi^{-1}$.

\subsection{Query logics}

We define query logics that are based on \textit{pre-} and \textit{post-}. The sentences of \textit{pre-} refer to the future of a given state in a Kripke structure, and the sentences of \textit{post-} refer to its past. By contrast, the sentences of query logics, called \textit{queries}, refer to the whole structure and thus enable translations between pre and post logics. The query logics are obtained from \textit{pre-} and \textit{post-} by adding a predicate $\mathcal{E}$, called the \textit{emptiness predicate}, on sentences. For a logic $\mathcal{L}$, the query logic $\mathcal{L}_0$ contains the two queries $\mathcal{E}(\phi)$ and $\neg \mathcal{E}(\phi)$ for each sentence $\phi$ of $\mathcal{L}$. The query logic $\mathcal{L}_c$ is richer and its queries are constructed inductively as follows:

- $\mathcal{E}(\phi)$, where $\phi$ is a sentence of $\mathcal{L}$,
- $\neg \theta_1$ and $\theta_1 \lor \theta_2$, where $\theta_1$ and $\theta_2$ are queries of $\mathcal{L}_c$.

We sometimes refer to queries in $\mathcal{L}_0$ and $\mathcal{L}_c$ as \textit{sentences}. The satisfaction relation $\models$ for queries on a Kripke structure $K$ is inductively defined as follows:

- $K \models \mathcal{E}(\phi)$ iff for all states $s$ of $K$, we have $s \not\models \phi$,
- $K \models \neg \theta_1$ iff $K \not\models \theta_1$, and
- $K \models \theta_1 \lor \theta_2$ iff $K \models \theta_1$ or $K \models \theta_2$.

\textbf{Example 2.2} To see why query logics are needed to compare forward and backward reasoning, consider the problem of invariant checking. Given a set \textit{bad} of error states, we would like to check whether there is an initial state that has a path to a state in \textit{bad}. The property of having a path to a state in \textit{bad} is expressed by the \textit{pre-} sentence $\varphi = \langle B \rangle$, where $B$ is a $\mu$-block with a single equation $X = \textit{bad} \lor \exists \diamond X$. This sentence is equivalent to the CTL property $\exists \diamond \textit{bad}$. Clearly $\varphi$ talks about an event in the future whereas any
sentence in post-$\mu$, which uses only past operators, talks only about events in the past. Therefore no sentence of post-$\mu$ is equivalent to $\varphi$. The invariant checking problem can also be expressed by the pre-$\mu$ query $E(init \land \varphi)$. Consider the post-$\mu$ sentence $\varphi' = \langle B' \rangle$, where $B'$ is a $\mu$-block containing the single equation $X' = init \lor \exists \exists X'$. Then the invariant checking problem is also expressed by the post-$\mu$ query $E(\varphi' \land bad)$.

While our motivation for query logics is theoretical, for the purpose of comparing pre and post logics, query logics are also practical. This is because once the state set $\phi^K$ has been computed (either explicitly or implicitly, using BDDs), the evaluation of the query $E(\phi)$ requires constant time. Therefore, checking a query in pre-$\mu_e$ or post-$\mu_e$ is not harder than model checking pre-$\mu$ or post-$\mu$ sentences, respectively.

2.1.3 Equivalences on Kripke structures induced by pre and post logics

Let $K = \langle P, W, R, L \rangle$ and $K' = \langle P, W', R', L' \rangle$ be two Kripke structures with the same set of propositions. A relation $\beta \subseteq W \times W'$ is a pre-bisimilarity relation if for all states $w$ and $w'$, we have that $\beta(w, w')$ implies the following.

1. $L(w) = L'(w')$,
2. for every state $v$ with $R(w, v)$, there is a state $v'$ with $R'(w', v')$ and $\beta(v, v')$, and
3. for every state $v'$ with $R'(w', v')$, there is a state $v$ with $R(w, v)$ and $\beta(v, v')$.

Note that, in particular, $\beta(w, w')$ implies that either both $w$ and $w'$ are initial, or neither of them is initial. It is easy to see that pre-bisimilarity relations are closed under union. Thus, if $\beta_1$ and $\beta_2$ are pre-bisimilarity relations, so is $\beta_1 \cup \beta_2$. The pre-bisimilarity relation $\beta$ is a pre-bisimulation between $K$ and $K'$ if for all states $w \in W$, there is a state $w' \in W'$ such that $\beta(w, w')$, and for all states $w' \in W'$, there is a state $w \in W$ such that $\beta(w, w')$. The pre-bisimilarity relation $\beta$ is an init-pre-bisimulation between $K$ and $K'$ if for all initial states $w \in W$, there is an initial state $w' \in W'$ such that $\beta(w, w')$, and for all initial states $w' \in W'$, there is an initial state $w \in W$ such that $\beta(w, w')$. The relation $\beta \subseteq W \times W'$ is a post-bisimulation between $K$ and $K'$ if $\beta$ is a pre-bisimulation between $K^{-1}$ and $K'^{-1}$. The relation $\beta \subseteq W \times W'$ is an init-post-bisimulation between $K$ and $K'$ if $\beta$ is an init-pre-bisimulation between $K^{-1}$ and $K'^{-1}$.

**Proposition 2.2** ([BCG88]) Let $K$ and $K'$ be two Kripke structures. Then, for all $w$ and $w'$ the following statements are equivalent.
(1) There is a pre-bisimilarity relation $\beta$ between $K$ and $K'$ such that $\beta(w, w')$.

(2) For all pre-$\mu$ sentences $\phi$, we have that $w \models_K \phi$ iff $w' \models_{K'} \phi$.

The following are easy extensions of the above result for pre-$\mu$.

**Proposition 2.3** Let $K$ and $K'$ be two Kripke structures.

- There is an init-pre-bisimulation between $K$ and $K'$ iff for all sentences $\phi$ of pre-$\mu$, we have $K \models \phi$ iff $K' \models \phi$.

- The following three statements are equivalent:

  (1) There is a pre-bisimilarity between $K$ and $K'$.

  (2) For all queries $\theta$ of pre-$\mu_0$, we have $K \models \theta$ iff $K' \models \theta$.

  (3) For all queries $\theta$ of pre-$\mu_\infty$, we have $K \models \theta$ iff $K' \models \theta$.

**Proof:**

- $(\Rightarrow)$ Let $\beta$ be an init-pre-bisimulation between $K$ and $K'$. Consider a sentence $\phi$ of pre-$\mu$. Suppose $K \models \phi$. Then there is an initial state $w$ of $K$ such that $w \models_K \phi$. Since $\beta$ is an init-pre-bisimulation, there is an initial state $w'$ of $K'$ such that $\beta(w, w')$. From Proposition 2.2 we have that $w' \models_{K'} \phi$ and hence $K' \models \phi$. Similarly if $K' \models \phi$, we can show that $K \models \phi$.

- $(\Leftarrow)$ Suppose for all sentences $\phi$ of pre-$\mu$, we have $K \models \phi$ iff $K' \models \phi$. Consider an initial state $w$ of $K$. Suppose for all initial states $v$ of $K'$ there is a pre-$\mu$ sentence $\varphi_v$ such that $w \models_K \varphi_v$ and $v \not\models_{K'} \varphi_v$. Let $\psi = \bigwedge_v \varphi_v$. Then we get a contradiction because $K \models \psi$ but $K' \not\models \psi$. Therefore, there is an initial state $w'$ of $K'$ such that no pre-$\mu$ sentence can distinguish between $w$ and $w'$. From Proposition 2.2, there is a pre-bisimilarity relation $\beta_w$ between $K$ and $K'$ such that $\beta_w(w, w')$. Thus, for each initial state $w \in W$ we can get pre-bisimilarity relation $\beta_w$ and for each initial state $w' \in W'$ we can get pre-bisimilarity relation $\beta_{w'}$. We take the union of all such relations to get the desired init-pre-bisimulation between $K$ and $K'$.

- $(1) \Rightarrow (2)$ Suppose $\beta$ is a pre-bisimulation between $K$ and $K'$. Let $\phi$ be a pre-$\mu$ sentence. Suppose $K \models \mathcal{E}(\phi)$. Then for all states $v \in W$, we have that $v \not\models \phi$. Consider an arbitrary state $w' \in W'$. Since $\beta$ is a pre-bisimulation, there is a state
CHAPTER 2. SYMBOLIC MODEL CHECKING WITH FORWARD REASONING

\[ w \in W \text{ such that } \beta(w, u'). \text{ Since } w \not\models \phi \text{ we have from Proposition 2.2 that } u' \not\models \phi. \]

Therefore, we have that \( K' \models \mathcal{E}(\phi) \). Similarly, we can show that if \( K' \models \mathcal{E}(\phi) \) then \( K \models \neg\mathcal{E}(\phi) \). It easily follows that \( K \models \neg\mathcal{E}(\phi) \) iff \( K' \models \neg\mathcal{E}(\phi) \).

((2) \implies (3)) We prove this by an induction over the structure of the queries in pre-\( \mu_\mathcal{E} \).

The base case is supplied by (2). Suppose \( \theta_1 \) and \( \theta_2 \) are such that \( K \models \theta_1 \) iff \( K' \models \theta_1 \) and \( K \models \theta_2 \) iff \( K' \models \theta_2 \). Then we have that (a) \( K \models \neg\theta_1 \) iff \( K \not\models \theta_1 \) iff \( K' \not\models \theta_1 \) iff \( K' \models \neg\theta_1 \), and (b) \( K \models \theta_1 \lor \theta_2 \) iff \( K \models \theta_1 \) or \( K \models \theta_2 \) iff \( K' \models \theta_1 \) or \( K' \models \theta_2 \) iff \( K' \models \theta_1 \lor \theta_2 \).

((3) \implies (1)) Suppose for all queries \( \theta \) of pre-\( \mu_\mathcal{E} \), we have \( K \models \theta \) iff \( K' \models \theta \). In particular, this holds for all queries \( \theta \) of pre-\( \mu_0 \). Consider a state \( w \) of \( K \). Suppose for all states \( v \) of \( K' \) there is a pre-\( \mu \) sentence \( \varphi_v \) such that \( w \models_K \varphi_v \) and \( v \not\models_{K'} \varphi_v \).

Let \( \psi = \bigwedge_v \varphi_v \). Then we get a contradiction because \( K \not\models \mathcal{E}(\psi) \) but \( K' \models \mathcal{E}(\psi) \).

Therefore, there is a state \( w' \) of \( K' \) such that no pre-\( \mu \) sentence can distinguish between \( w \) and \( w' \). From Proposition 2.2, there is a pre-bisimilarity relation \( \beta \) between \( K \) and \( K' \) such that \( \beta(w, w') \). Thus, for each state \( w \in W \) we can get pre-bisimilarity relation \( \beta_w \) and for each state \( w' \in W' \) we can get pre-bisimilarity relation \( \beta_{w'} \). We take the union of all such relations to get the desired init-pre-bisimulation between \( K \) and \( K' \).

\[ \blacksquare \]

**Proposition 2.4** Let \( K \) and \( K' \) be two Kripke structures.

- There is an init-post-bisimulation between \( K \) and \( K' \) iff for all sentences \( \phi \) of post-\( \mu \), we have \( K \models \phi \) iff \( K' \models \phi \).

- The following three statements are equivalent:

  (1) There is a post-bisimulation between \( K \) and \( K' \).

  (2) For all queries \( \theta \) of post-\( \mu_0 \), we have \( K \models \theta \) iff \( K' \models \theta \).

  (3) For all queries \( \theta \) of post-\( \mu_\mathcal{E} \), we have \( K \models \theta \) iff \( K' \models \theta \).

**Proof:** The proof runs exactly along the lines of the proof of Proposition 2.3. \[ \blacksquare \]
2.2 Intersection of Pre and Post Logics

Of particular interest is the intersection of the query logics $pre-\mu_\mathcal{E}$ and $post-\mu_\mathcal{E}$. It contains the queries that can be specified in both $pre-\mu_\mathcal{E}$, which often is more convenient for specifiers, and in $post-\mu_\mathcal{E}$, which often is more efficient for symbolic model checking. In this section we show that essentially all linear properties lie in this intersection. On the other hand, there are simple branching properties that do not lie in the intersection.

2.2.1 In

Consider a Kripke structure $K = \langle P, W, R, L \rangle$. An observation of $K$ is a subset of the set of propositions $P$. An error trace of $K$ is a finite or infinite sequence of observations. A linear property of $K$ is a set of error traces.\footnote{Recall that we work, for convenience, in a setting that is dual to the one that considers linear properties to consist of all non-error traces.} Many useful linear properties, namely, the $\omega$-regular linear properties, can be specified by finite automata. A finite automaton $A = \langle P, S, S_0, S_F, r, \ell \rangle$ consists of a finite set $P$ of propositions, a finite set $S$ of states, a set $S_0 \subseteq S$ of initial states, a set $S_F \subseteq S$ of accepting states, a binary transition relation $r \subseteq S \times S$, and a labeling function $\ell : S \rightarrow 2^P$ that assigns to each state a set of propositions.\footnote{Note that while classical definitions of automata refer to an alphabet that labels the transitions between states, here we assume that the alphabet (the set $P$ of propositions) labels the states. This notational difference between the two approaches is technical.} The following definitions regarding paths apply equally to Kripke structures and automata. A path $\pi = u_0, u_1, \ldots$ of $K$ (resp. $A$) is a finite or infinite sequence of states such that for all $i \geq 0$, we have $R(u_i, u_{i+1})$ (resp. $r(u_i, u_{i+1})$). A reverse-path $\pi = u_0, u_1, \ldots$ of $K$ (resp. $A$) is a finite or infinite sequence of states such that for all $i \geq 0$, we have $R(u_{n-i}, u_i)$ (resp. $r(u_{n-i}, u_i)$). The path $\pi$ is initialized if $u_0$ is an initial state. We denote the concatenation of two paths $\pi_1$ and $\pi_2$ by $\pi_1 \cdot \pi_2$. For a finite path or reverse-path $\pi$, let $\text{last}(\pi)$ denote the last state in the path and let $|\pi|$ denote the length of the path. A finite path or reverse-path $\pi = u_0, u_1, \ldots, u_n$ is a cycle if $u_0 = u_n$. If $\pi_1 = u_0, u_1, \ldots, u_n$ is a finite path, $\pi_2 = v_0, v_1, \ldots, v_n$ is a cycle, and either $|\pi_1| = 0$ or $R(u_n, v_0)$ (resp. $r(u_n, v_0)$), then $u_0, u_1, \ldots, u_n \cdot (v_0, v_1, \ldots, v_{n-1})^\omega$ is an infinite path. By $\text{Inf}(\pi)$ we denote the set of states that appear in $\pi$ infinitely often. The labeling functions $L$ and $\ell$ are lifted from states to paths in the obvious way. Using the set $S_F$ of accepting states, we classify paths as accepting or rejecting. For example, we may say that a path is accepting if it visits a state...
from $S_F$. We consider several classifications, which depend on the interpretation we place on the automaton. We will get back to this point shortly.

With each finite automaton $A$ we associate a sentence $\exists A$ that is interpreted over a Kripke structure $K$ with the same propositions as $A$. The automaton $A$ models execution sequences that violate the property of interest. The model-checking problem for automata is to decide, given $K$ and $A$, whether $K \models \exists A$, where $K \models \exists A$ if there exist an initialized path $\pi_1$ of $K$ and an accepting initialized path $\pi_2$ of $A$ such that $L(\pi_1) = \ell(\pi_2)$. The observation sequence $L(\pi_1)$ is then called an error trace of $K$ with respect to $A$. Recall that the accepting paths of $A$ depend on the interpretation we place on the automaton $A$. We consider here three different interpretations: safety automata, Büchi automata, and co-Büchi automata.

For each interpretation we reduce the model-checking problem for automata to the model-checking problem for post-$\mu_0$, by translating automata into equivalent post-$\mu_0$ queries. The post-$\mu_0$ query $\theta$ is equivalent to the automaton $A$ if for every Kripke structure $K$, we have $K \models \exists A$ if $K \models \theta$.

In all translations, we will make use of the following. With each state $s$ of the automaton $A$, we associate variables $X_s$ and $X'_s$. In addition, we use the two variables $X$ and $X'$. In the following, let $V = \{X_s \mid s \in S\} \cup \{X'_s \mid s \in S\} \cup \{X, X'\}$. For each state $s$ of $A$, let $\gamma_s$ be a variable-free and modality-free expression that characterizes the state $s$ locally, namely, $\gamma_s = \bigwedge_{p \in \ell(s)} p \land \bigwedge_{p \not\in \ell(s)} \neg p$. Now, let $B_A$ be the following $\mu$-block, which consists of $|S| + 1$ equations, with $\text{vars}(B_A) = \{X_s \mid s \in S\} \cup \{X\}$:

$$\begin{align*}
X_s &= \begin{cases}
\gamma_s \land (\text{init} \lor \bigvee_{t \in \text{pre}(s)} \exists \Box X_t) & \text{if } s \in S_0, \\
\gamma_s \land \bigvee_{t \in \text{pre}(s)} \exists \Box X_t & \text{if } s \not\in S_0,
\end{cases} \\
X &= \bigvee_{f \in S_F} X_f.
\end{align*}$$

Note that the size of $B_A$ is linear in the size of $A$.

**Safety automata**

A safety property of a Kripke structure $K$ is a set of finite error traces. The regular safety properties can be specified by safety automata. A safety automaton is a finite automaton $A$ such that a path $\pi$ of $A$ is accepting if $\pi$ is a finite path and its last state is an accepting state of $A$. We show in Theorem 2.7 that the safety automaton $A$ is equivalent to the post-$\mu_0$ query $\theta_A = \neg \mathcal{E}(\langle B_A \rangle, X)$.
CHAPTER 2. SYMBOLIC MODEL CHECKING WITH FORWARD REASONING

If a finite error trace exists, we would like a model-checking algorithm to detect it as soon as possible. By evaluating the query \( \theta_A \) as follows (in the standard way), this can indeed be guaranteed. The evaluation of the \( \mu \)-block \( B_A \) over a Kripke structure \( K \) proceeds in iterations. Let \( X^K(i) \subseteq W \) denote the value of variable \( X \in vars(B_A) \) after the \( i \)-th iteration, and let \( \Gamma^K(i) \) denote the \( K \)-valuation that assigns to each variable in \( X \in vars(B_A) \) the value \( X^K(i) \). Initially, \( X^K(0) = \emptyset \) for all \( X \in vars(B_A) \). In all subsequent iterations, the value of each variable \( X \in vars(B_A) \) is updated according to the equation \( X^K(i + 1) = \text{expand}(X^K(G^K(i))) \). Since the modal expressions in \( B_A \) are monotonic, we have that \( \Gamma^K(0) \subseteq \Gamma^K(1) \subseteq \Gamma^K(2) \subseteq \ldots \). Therefore, if \( X^K(m) \neq \emptyset \) for some \( m \), then \( X^K(n) \neq \emptyset \) for all \( n > m \). Hence, we can detect that \( K \models \theta_A \) as soon as \( X^K(m) \) is nonempty. In other words, using symbolic forward state traversal, we will explore only states up to distance \( m \) from initial states.

**Lemma 2.5** Let \( A = \langle P, S, S_0, S_F, r, \ell \rangle \) be a safety automaton and \( K = \langle P, W, R, L \rangle \) be a Kripke structure. For all \( i \geq 0 \), \( s \in S \), and \( w \in W \), we have that \( w \in X^K(i) \) iff there is an initialized path \( \pi_1 \) of \( K \) ending in \( w \) and an initialized path \( \pi_2 \) of \( A \) ending in \( s \) such that \( L(\pi_1) = \ell(\pi_2) \) and \( |\pi_1| = |\pi_2| \leq i \).

**Proof:** We prove this by induction on \( i \).

**Base case:** We have that \( X^K(0) = \emptyset \) for all \( s \in S \). Also, the length of a path in \( K \) or \( A \) is always greater than 0. Hence, it is not possible to have paths \( \pi_1 \) and \( \pi_2 \) such that \(|\pi_1| = |\pi_2| \leq 0 \).

**Inductive case:** Assume the statement holds for \( i \). We prove it for \( i + 1 \). By the definition above, we have that

\[
X^K(i + 1) = \begin{cases} 
\gamma_s \land (\text{init} \lor \bigvee_{t \in \text{pre}(s)} \text{post}(X^K(i))) & \text{if } s \in S_0, \\
\gamma_s \land \bigvee_{t \in \text{pre}(s)} \text{post}(X^K(i)) & \text{if } s \not\in S_0.
\end{cases}
\]

**Case 1.** \( s \not\in S_0 \) Suppose \( v \in X^K(i + 1) \). From the definition of \( X^K(i + 1) \), we have that \( L(v) = \ell(s) \), and there is \( t \in \text{pre}(s) \) and \( u \in X^K(i) \) such that \( R(u, v) \). From the induction hypothesis, there is an initialized path \( \pi_1 \) in \( K \) ending in \( u \) and an initialized path \( \pi_2 \) in \( A \) ending in \( t \) such that \( L(\pi_1) = \ell(\pi_2) \) and \(|\pi_1| = |\pi_2| \leq i \). Extend \( \pi_1 \) by \( v \) and \( \pi_2 \) by \( s \) to get the desired initialized paths. Suppose there are initialized paths \( \pi_1 \) of \( K \) ending in \( v \) and \( \pi_2 \) of \( A \) ending in \( s \) such that \(|\pi_1| = |\pi_2| \leq i + 1 \) and \( L(\pi_1) = \ell(\pi_2) \). Since \( s \) is not an initial state, \(|\pi_2| > 1 \). Let \( u \) be the predecessor of
v in π₁ and t be the predecessor of s in π₂. Then there are initialized paths π'₁ in K ending in u and π'₂ in A ending in t such that \( L(\pi'_1) = \ell(\pi'_2) \) and \( |\pi'_1| = |\pi'_2| \leq i \). From the induction hypothesis, we get that \( u \in X^K_i(s) \). Using the definition of \( X^K_s(i+1) \), we conclude that \( v \in X^K_i(s+1) \).

**Case 2.** \( s \in S_0 \) Suppose \( v \in X^K_i(s) \). If \( v \in (\gamma_s \wedge \text{init})^K \) then \( v \) is an initial state and \( L(v) = \ell(s) \). We get \( \pi_1 = v \) and \( \pi_2 = s \) and \( |\pi_1| = |\pi_2| \leq i+1 \). If \( v \notin (\gamma_s \wedge \text{init})^K \), we have that \( L(v) = \ell(s) \), and there is \( t \in \text{pre}(s) \) and \( u \in X^K_i(t) \) such that \( R(u, v) \). We follow the same reasoning as in Case 1. Suppose there are initialized paths \( \pi_1 \) of K ending in \( v \) and \( \pi_2 \) of \( A \) ending in \( s \) such that \( |\pi_1| = |\pi_2| \leq i+1 \) and \( L(\pi_1) = \ell(\pi_2) \). If \( |\pi_1| = 1 \), then \( v \in (\gamma_s \wedge \text{init})^K \) and hence \( v \in X^K_i(s) \). If \( |\pi_1| > 1 \), we follow the same reasoning as in Case 1.

**Corollary 2.6** Let \( A = \langle P, S, S_0, S_F, r, \ell \rangle \) be a safety automaton and \( K = \langle P, W, R, L \rangle \) be a Kripke structure. Let \( \Gamma \in \mathcal{P}_V \). For all \( s \in S \) and \( w \in W \), we have that \( w \in (B_A)^K(\Gamma)(X_s) \) iff there is an initialized path \( \pi_1 \) of K ending in \( w \) and an initialized path \( \pi_2 \) of \( A \) ending in \( s \) such that \( L(\pi_1) = \ell(\pi_2) \).

The following theorem guarantees that if there is an error trace of length \( m \), then we will find it in \( m \) iterations.

**Theorem 2.7** For every safety automaton \( A \), an equivalent alternation-free post-\( \mu_0 \) query \( \theta_A = \neg E(\langle B_A, X \rangle) \) can be constructed in linear time. Further, for every Kripke structure \( K \), if the shortest error trace in \( K \) with respect to \( A \) has length \( m \), then \( X^K(m+1) \neq \emptyset \), where \( X \) is the root variable of \( \theta_A \).

**Proof:** Let \( K \) be a Kripke structure and \( \Gamma \in \mathcal{P}_V \). Suppose \( K \models \exists A \) and the shortest error trace of \( K \) with respect to \( A \) has length \( m \). Then there is an initialized path \( \pi_1 \) of \( K \) and an initialized accepting path \( \pi_2 \) of \( A \) such that \( L(\pi_1) = \ell(\pi_2) \) and \( |\pi_1| = |\pi_2| = m \). Let the path \( \pi_1 \) end in state \( w \) and the path \( \pi_2 \) end in state \( s \). From Lemma 2.5, we have that \( w \in X^K_i(m) \) and therefore \( w \in X^K_i(m+1) \). From the monotonicity property, we get that \( \langle B_A \rangle^K(\Gamma)(X_s) \neq \emptyset \). Therefore \( K \models \theta_A \). Suppose \( K \models \theta_A \). Then \( \langle B_A \rangle^K(\Gamma)(X_s) \neq \emptyset \). This means that \( \langle B_A \rangle^K(\Gamma)(X_s) \neq \emptyset \) for some \( s \in S_F \). Therefore, we have that \( w \in X^K_i(s) \) for some \( i \). An application of Lemma 2.5 gives us that \( K \models \exists A \).
Example 2.3 Consider the LTL formula $\psi = \Diamond p$. A computation $\pi$ satisfies $\psi$ iff $p$ always holds along $\pi$. The safety automaton $A_\psi$ in the figure accepts exactly all the computations that satisfy $\psi$.

![Safety Automaton Diagram]

The $\mu$-block $B_A$ contains the equations:

$$X_s = \neg p \land (init \lor \exists s X_s).$$

$$X_t = p \land (init \lor \exists s X_s).$$

$$X = X_t.$$ 

Büchi automata

Safety automata cannot specify infinite error traces. For that, we use Büchi automata. A Büchi automaton $A$ is a finite automaton such that a path $\pi$ of $A$ is accepting if $\text{Inf}(\pi) \cap S_F \neq \emptyset$; that is, some accepting state of $A$ occurs infinitely often in $\pi$. It is well-known [EL86, Dam94, BC96] that for every Büchi automaton $A$, there exists a pre-$\mu_0$ query $\theta_A$ such that for every Kripke structure $K$, we have $K \models \exists A$ iff $K \models \theta_A$. We now show that there exists also a post-$\mu_0$ query $\theta_A$ with the same property, thereby proving that the model-checking problem for Büchi automata lies in the intersection of pre-$\mu_0$ and post-$\mu_0$. We define two equational blocks: a $\nu$-block $B_1$ and a $\mu$-block $B_2$. The block $B_1$ contains the following $|S_F| + 1$ equations, with $\text{vars}(B_1) = \{X'_f \mid f \in S_F\} \cup \{X'\}$:

$$X'_f = X_f \land \bigvee_{t \in \text{pre}(f)} \exists s X'_t,$$

$$X' = \bigvee_{f \in S_F} X'_f.$$

The block $B_2$ contains an equation for each state $s \in S \setminus S_F$, defined by

$$X'_s = \gamma_s \land \bigvee_{t \in \text{pre}(s)} \exists X'_t.$$

Then $\theta_A = \neg \mathcal{E}(\langle B_1, B_2, B_A \rangle, X')$. The translation is linear in the size of the Büchi automaton. Also, the equational blocks $B_1$ and $B_2$ depend on each other and the alternation
depth of $\theta_A$ is two. Since Büchi automata are expressively equivalent to the $\omega$-regular languages, the query logic post-$\mu_0$ can specify all $\omega$-regular properties.

It is instructive to compare the translation of $A$ to pre-$\mu_0$ [BC96]. The translation contains three equational blocks — a $\mu$-block $C_1$, a $\nu$-block $C_2$, and a $\mu$-block $C_3$. The block $C_1$ contains a single equation

$$X'_f = \bigvee_{s \in S_0} X'_s.$$

The block $C_2$ contains an equation for each state $s \in S_F$, defined by

$$X'_f = \gamma_f \land \bigvee_{t \in post(f)} \exists X'_t.$$

The block $C_3$ contains an equation for each state $s \in S \setminus S_F$, defined by

$$X'_s = \gamma_s \land \bigvee_{t \in post(s)} \exists X'_t.$$

Then $\theta_A = \neg E(\langle C_1, C_2, C_3, X'_f \rangle)$. Again, the translation is linear in the size of the automaton and has an alternation depth of two. Hence, translation of LTL formulas to either post-$\mu_0$ or pre-$\mu_0$ has the same complexity.

**Lemma 2.8** Let $K$ be a Kripke structure and $\Gamma \in \mathcal{P}_{V,K}$. Then for all $s \in S \setminus S_F$ and $w \in W$, we have that $w \in \langle B_2 \rangle^K(\Gamma)(X'_f)$ iff there is $f \in S_F$, $u \in \Gamma(X'_f)$, a path $\pi_1$ in $K$ from $u$ to $w$, and a path $\pi_2$ in $A$ from $f$ to $s$ such that $L(\pi_1) = \ell(\pi_2)$.

**Proof:** Let $K$ be a Kripke structure. For all $s \in S \setminus S_F$, let $reach(s) \subseteq W$ be the set of states such that $w \in reach(s)$ iff there is $f \in S_F$, $u \in \Gamma(X'_f)$, a path $\pi_1$ in $K$ from $u$ to $w$, and a path $\pi_2$ in $A$ from $f$ to $s$ such that $L(\pi_1) = \ell(\pi_2)$. Then, we have to prove that for all $s \in S \setminus S_F$ and $w \in W$, we have that $w \in \langle B_2 \rangle^K(\Gamma)(X'_s)$ iff $w \in reach(s)$. In the following, let $s \in S \setminus S_F$ and $w \in W$.

$(\Longrightarrow)$ Recall that $\langle B_2 \rangle^K(\Gamma')$ is the greatest fixpoint of the function $F_{\langle B_2 \rangle,K}^r$ defined below.

$$F_{\langle B_2 \rangle,K}^r(\Gamma')(X) = \begin{cases} E(s, \Gamma') & \text{if } X = X'_s \text{ for some } s \in S \setminus S_F, \\ \Gamma'(X) & \text{otherwise,} \end{cases}$$

where $E(s, \Gamma') = \gamma_s \land (\bigvee_{t \in pre(s) \cap S_F} \text{post}(\Gamma(X'_t)) \lor \bigvee_{t \in pre(s) \cap S \setminus S_F} \text{post}(\Gamma'(X'_t)))$. Consider an evaluation $\Gamma'$ such that $F_{\langle B_2 \rangle,K}^r(\Gamma') \subseteq \Gamma'$. Define a new valuation $\Delta$ as follows.

$$\Delta(X) = \begin{cases} \Gamma'(X'_s) \cap \text{reach}(s) & \text{if } X = X'_s \text{ for some } s \in S \setminus S_F, \\ \Gamma'(X'_s) & \text{otherwise.} \end{cases}$$
CHAPTER 2. SYMBOLIC MODEL CHECKING WITH FORWARD REASONING

We show that \( F^K_{(B_2), \Gamma} \Delta \subseteq \Delta \). We need to show that if \( s \in S \setminus S_F \) then \( F^K_{(B_2), \Gamma} \Delta(X') \subseteq \Delta(X') \). Suppose \( w \in F^K_{(B_2), \Gamma} \Delta(X') \) for some \( s \in S \setminus S_F \). Since \( \Delta \subseteq \Gamma' \) and \( F^K_{(B_2), \Gamma} \) is monotonic, we have that \( F^K_{(B_2), \Gamma} \Delta \subseteq F^K_{(B_2), \Gamma} \Gamma' \). Therefore, \( w \in F^K_{(B_2), \Gamma} \Gamma'(X') \subseteq \Gamma'(X') \).

**Case 1.** There is \( t \in \text{pre}(s) \cap S_F \) and \( v \in \Gamma(X') \) such that \( R(v, w) \). Because of the paths \( \pi_1 = v \cdot w \) and \( \pi_2 = t \cdot s \), we have that \( w \in \text{reach}(s) \).

**Case 2.** There is some \( t \in \text{pre}(s) \cap S \setminus S_F \) and \( v \in \Delta(X') \) such that \( R(v, w) \). Since \( v \in \Delta(X') \) we have that \( v \in \text{reach}(t) \) also. Hence, there is \( f \in S_F \), \( u \in \Gamma(X') \), a path \( \pi_1 \) in \( K \) from \( u \) to \( v \), and a path \( \pi_2 \) in \( A \) from \( f \) to \( t \) such that \( \ell(\pi_1) = \ell(\pi_2) \). Let \( \pi_1 \) be obtained from \( \pi_1 \) by appending \( w \) and \( \pi_2 \) be obtained from \( \pi_2 \) by appending \( s \). This gives us that \( w \in \text{reach}(s) \).

Thus we get that \( w \in \Gamma'(X') \) and \( w \in \text{reach}(s) \). Therefore \( w \in \Delta(X') \). From Theorem 2.1, the least fixpoint \( (B_2)^K(\Gamma) \) of \( F^K_{(B_2), \Gamma} \) is equal to \( \text{glb}(\{\Gamma' \mid F^K_{(B_2), \Gamma} \Gamma' \subseteq \Gamma\}) \). This clearly means that if \( s \in S \setminus S_F \) then \( (B_2)^K(\Gamma)(X') \cap \text{reach}(s) = (B_2)^K(\Gamma)(X') \). In other words, we have that \( (B_2)^K(\Gamma)(X') \subseteq \text{reach}(s) \). Hence \( w \in \text{reach}(s) \).

\[(\Leftarrow)\] Suppose there is \( f \in S_F \), \( u \in \Gamma(X') \), a path \( \pi_1 \) in \( K \) from \( u \) to \( w \), and a path \( \pi_2 \) in \( A \) from \( f \) to \( s \) such that \( \ell(\pi_1) = \ell(\pi_2) \). Suppose that \( u \notin (B_2)^K(\Gamma)(X') \). Consider the smallest \( n \) such that \( \pi_2(n) \in S \setminus S_F \) and \( \pi_1(n) \notin (B_2)^K(\Gamma)(X') \) where \( t = \pi_2(n) \). Since \( \pi_2(0) = f \in S_F \), we have that \( n > 0 \). Let \( t' = \pi_2(n - 1) \). Then either \( t' \in S_F \) or \( \pi_1(n - 1) \notin (B_2)^K(\Gamma)(X') \). In either case, since \( (B_2)^K(\Gamma) \) is a fixpoint we get that \( \pi_1(n) \notin (B_2)^K(\Gamma)(X') \) which is a contradiction. Thus we get that \( w \in (B_2)^K(\Gamma)(X') \).

**Theorem 2.9** Let \( A \) be a B"uchi automaton. Then, we have that the post-\( \mu \) query \( \theta_{A} = \neg \mathcal{E}(\langle B_1, B_2, B_A \rangle, X') \) is equivalent to \( A \). The query \( \theta_{A} \) can be constructed in linear time and its alternation depth is two.

**Proof:** Let \( K \) be a Kripke structure. Let \( B \) be the block tuple \( \langle B_1, B_2, B_A \rangle \). We need to show that \( K \models \exists A \) if \( K \models \neg \mathcal{E}(\langle B, X' \rangle) \). Let \( \Gamma \in \mathcal{P}_{V,K} \). Since \( \langle B, X' \rangle \) is a sentence, the greatest fixpoint \( B^K(\Gamma) \) of \( F^K_{B,G} \) is independent of \( \Gamma \).

\[(\Rightarrow)\] Suppose \( K \models \exists A \). Then there is an initialized path \( \pi_1 \) in \( K \) and an accepting initialized path \( \pi_2 \) in \( A \) such that \( \ell(\pi_1) = \ell(\pi_2) \). Since both \( K \) and \( A \) are finite, there are \( m \) and \( n \) such that \( 0 \leq m < n \), \( \pi_1(m) = \pi_2(n) \), and \( \pi_2(m) = \pi_2(n) \in S_F \). Therefore, we
have initialized paths $\pi'_1 = \pi_1(0), \pi_1(1), \ldots, \pi_1(m-1) \cdot (\pi_1(m), \pi_1(m+1), \ldots, \pi_1(n-1))^\omega$ in $K$ and $\pi'_2 = \pi_2(0), \pi_2(1), \ldots, \pi_2(m-1) \cdot (\pi_2(m), \pi_2(m+1), \ldots, \pi_2(n-1))^\omega$ in $A$ such that $L(\pi'_1) = L(\pi'_2)$. Define the function $\text{before}$ on $[m, n)$ as follows.

$$\text{before}(k) = \begin{cases} 
  n-1 & \text{if } k = m, \\
  k-1 & \text{if } k > m.
  \end{cases}$$

For all $s \in S_F$, let $\text{add}(s) = \{\pi_1(k) | m \leq k < n \text{ and } \pi'_2(k) = s\}$. Define the valuation $\Delta$ as follows.

$$\Delta(X) = \begin{cases} 
  \text{add}(s) & \text{if } X = X'_s \text{ for some } s \in S_F, \\
  \emptyset & \text{otherwise}.
  \end{cases}$$

Recall that $B^K(\Gamma)$ is the greatest fixpoint of the function $F^K_{\text{BG}}$ defined as follows.

$$F^K_{\text{BG}}(\Gamma')(X) = \begin{cases} 
  (X_f \land (\lor_{t \in \text{pre}(f)} \exists \otimes(X'_i)))^K(\langle B_2, B_A \rangle^K(\Gamma')) & \text{if } X \in \text{vars}(B_1), \\
  \langle B_2, B_A \rangle^K(\Gamma')(X) & \text{otherwise}.
  \end{cases}$$

Notice that $\Gamma$ does not appear on the right hand side of the function because the tuple $\langle B_1, B_2, B_A \rangle$ does not contain any free variables. We have the following facts.

1. $\langle B_2, B_A \rangle^K(\Gamma')(X_s) = \langle B_A \rangle^K(\Gamma')(X_s)$ for all $s \in S$.

2. $\langle B_2, B_A \rangle^K(\Gamma')(X'_s) = \Gamma'(X'_s)$ for $s \in S_F$.

3. $\langle B_2, B_A \rangle^K(\Gamma')(X'_s) = \langle B_2 \rangle^K(\Gamma')(X'_s)$ for $s \in S \setminus S_F$.

Thus we can simplify the above definition to the following.

$$F^K_{\text{BG}}(\Gamma')(X) = \begin{cases} 
  \exists_{f \in S_F} \Gamma'(X'_f) & \text{if } X = X', \\
  E(f, \Gamma') & \text{if } X = X'_f \text{ for some } f \in S_F, \\
  \langle B_2, B_A \rangle^K(\Gamma')(X) & \text{otherwise}.
  \end{cases}$$

where

$$E(f, \Gamma') = \langle B_A \rangle^K(\Gamma'(X_f) \land (\lor_{t \in \text{pre}(f) \cap S_F} \text{post}(\Gamma'(X'_i)) \lor \lor_{t \in \text{pre}(f) \cap S \setminus S_F} \text{post}(\langle B_2 \rangle^K(\Gamma'(X'_i)))).$$

We now show that $\Delta \subseteq F^K_{\text{BG}}(\Delta)$ for any valuation $\Gamma$. All we need to show is that if $f \in S_F$ and $u \in \text{add}(f)$ then $u \in F^K_{\text{BG}}(\Delta)(X'_f)$. Suppose $u \in \text{add}(f)$ for some $f \in S_F$. Then for some $m \leq k < n$, we have that $f = \pi_2(k)$ and $u = \pi_1(k)$. There are
initialized paths $\sigma_1$ in $K$ to $u$ and $\sigma_2$ in $A$ to $f$ such that $L(\sigma_1) = \ell(\sigma_2)$. Hence, from Lemma 2.6 we have that $u \in \langle B_A \rangle^K(\Delta)(X_f)$. Let $s = \pi_2(before(k))$. If $s \in S_F$ then $v = \pi_1(before(k)) \in \Delta(X'_f)$. Then there is $s \in \text{pre}(f) \cap S_F$ such that $u \in \text{post}(\Delta(X'_f))$. Therefore $u \in F^K_{\mathcal{E}}(\Delta)(X'_f)$. If $s \in S \setminus S_F$ then there is a path $\sigma_1 = \pi_1(k), \pi_1(k+1), \ldots, \pi_1(n-1), \pi_1(m), \ldots, \pi_1(before(k))$ in $K$ and a path $\sigma_2 = \pi_2(k), \pi_2(k+1), \ldots, \pi_2(n-1), \pi_2(m), \ldots, \pi_2(before(k))$ in $A$ such that $L(\sigma_1) = \ell(\sigma_2)$. Therefore, from Lemma 2.8 we have that $v = \pi_1(before(k)) \in \langle B_2 \rangle^K(\Delta)(X'_f)$. Then there is $s \in \text{pre}(f) \cap S \setminus S_F$ such that $u \in \text{post}(\langle B_2 \rangle^K(\Delta)(X'_f))$. Therefore $u \in F^K_{\mathcal{E}}(\Delta)(X'_f)$. Thus, we get that $\Delta \subseteq F^K_{\mathcal{E}}(\Delta)$. From Theorem 2.1, we get that the greatest fixpoint $B^K(\Gamma)$ of $F^K_{\mathcal{E}}(\Delta)$ is equal to $\text{lub}(\langle \Gamma' \rangle \subseteq F^K_{\mathcal{E}}(\Delta))$. Therefore we have that $\Delta \subseteq B^K(\Gamma)$. This means that there is some $f \in S_F$ such that $B^K(\Gamma)(X'_f) \neq \emptyset$. Therefore $B^K(\Gamma)(X'_f) \neq \emptyset$ and $K \models \theta_A$.

($\Leftarrow$) Suppose $K \models \neg \mathcal{E}(\langle B, X' \rangle)$. We incrementally construct infinite reverse-paths $\pi_1$ in $K$ and $\pi_2$ in $A$ maintaining the invariants that (1) $L(\pi_1) = \ell(\pi_2)$, and (2) if last($\pi_1$) = $w$ and last($\pi_2$) = $s$ then $s \in S_F$ and $w \in B^K(\Gamma)(X'_f)$. Since $B^K(\Delta) \neq \emptyset$, there is $w \in B^K(\Delta)(X'_f)$ for some $f \in S_F$. Also $w \in B^K(\Gamma)(X'_f)$ means that $w \in B^K(\Gamma)(X'_f)$. Therefore $L(w) = \ell(f)$. Let $\pi_1 = w$ and $\pi_2 = f$ initially. Since $w \in B^K(\Gamma)(X'_f)$, there is a $t \in \text{pre}(f)$ and $v \in B^K(\Gamma)(X'_f)$ such that $R(v, w)$.

Case 1. $t \in S_F$ Augment $\pi_1$ by appending $v$ and $\pi_2$ by appending $t$.

Case 2. $t \in S \setminus S_F$ We have that $B^K(\Gamma) = \langle B_2 \rangle^K(B^K(\Gamma))$. Therefore we get that $v \in \langle B_2 \rangle^K(B^K(\Gamma))(X'_f)$. From Lemma 2.8, we get that there are $u \in W$, $f' \in S_F$, a path $\pi'_1$ in $K$ from $u$ to $v$, and a path $\pi'_2$ in $A$ from $f'$ to $t$ such that $u \in B^K(\Gamma)(X'_{f'})$ and $L(\pi'_1) = \ell(\pi'_2)$. Augment $\pi_1$ by appending $\pi'_1$ after reversing it and $\pi_2$ by appending $\pi'_2$ after reversing it.

The above process can be repeated ad infinitum to get infinite reverse-paths $\pi_1$ in $K$ and $\pi_2$ in $A$ such that $L(\pi_1) = \ell(\pi_2)$ and there are infinitely many occurrences of states in $S_F$ on $\pi_2$. Since both $K$ and $A$ are finite, there are $m$ and $n$ such that $0 \leq m < n$, $\pi_1(m) = \pi_1(n)$ and $\pi_2(n) \in S_F$. Therefore, we have reverse-paths $\pi'_1 = \pi_1(0), \pi_1(1), \ldots, \pi_1(m-1) \cdot (\pi_1(m), \pi_1(m+1), \ldots, \pi_1(n-1))^\omega$ in $K$ and $\pi'_2 = \pi_2(0), \pi_2(1), \ldots, \pi_2(m-1) \cdot (\pi_2(m), \pi_2(m+1), \ldots, \pi_2(n-1))^\omega$ in $A$ such that $L(\pi'_1) = \ell(\pi'_2)$. Consider the paths $(\pi_1(m), \pi_1(n-1), \pi_1(n-2), \ldots, \pi_1(m+1))^\omega$ in $K$ and $(\pi_2(m), \pi_2(n-1), \pi_2(n-2), \ldots, \pi_2(m+1))^\omega$ in $A$. Let $\pi_2(m) = f \in S_F$. We know that $\pi_1(m) \in B^K(\Gamma)(X'_f)$ and therefore $\pi_1(m) \in B^K(\Gamma)(X_f)$. From Theorem 2.7, we get that there are initialized paths $\sigma_1$ in
CHAPTER 2. SYMBOLIC MODEL CHECKING WITH FORWARD REASONING 27

$K$ to $\pi_1(m)$ and $\sigma_2$ in $A$ to $\pi_2(m)$ such that $L(\sigma_1) = \ell(\sigma_2)$. We get initialized paths $\pi'_1 = \sigma_1(0), \sigma_1(1), \ldots, \sigma_1(last(\sigma_1) - 1), (\pi_1(m), \pi_1(n - 1), \pi_1(n - 2), \ldots, \pi_1(m + 1))^\omega$ in $K$ and $\pi'_2 = \sigma_2(0), \sigma_2(1), \ldots, \sigma_2(last(\sigma_2) - 1), (\pi_2(m), \pi_2(n - 1), \pi_2(n - 2), \ldots, \pi_2(m + 1))^\omega$ in $A$ such that $L(\pi'_1) = \ell(\pi'_2)$.

In particular, since all sentences of the linear temporal logic LTL can be translated to Büchi automata [VW94], Theorem 2.9, together with [EL86], implies that all LTL sentences lie in the intersection $pre-\mu_0 \cap post-\mu_0$. Hence, LTL model checking can proceed by symbolic forward state traversal. Since the translation from LTL to Büchi automata involves an exponential blow-up, the translation from LTL to $post-\mu_0$ is also exponential.

**Example 2.4** Consider the LTL formula $\psi = \Box \Diamond p$. A computation $\pi$ satisfies $\psi$ iff $p$ holds infinitely often along $\pi$. The Büchi automaton $A_\psi$ in the figure accepts exactly all the computations that satisfy $\psi$.

$$A_\psi :$$

$$\begin{array}{c}
p \\
s \rightarrow t \\
t \rightarrow s \\
s \rightarrow p \\
p \rightarrow s \\
\end{array}$$

The $\nu$-block $B_1$ contains the equations

$$X'_s = X_s \land (\exists \Box X'_s \lor \exists \Box X'_t).$$

$$X'_t = X'_s.$$  

The $\mu$-block $B_2$ contains the single equation $X'_t = \neg p \land (\exists \Box X'_s \lor \exists \Box X'_t)$. The $\mu$-block $B_A$ contains the equations

$$X_s = p \land (init \lor \exists \Box X_s \lor \exists \Box X_t).$$

$$X_t = \neg p \land (init \lor \exists \Box X_s \lor \exists \Box X_t).$$

**Co-Büchi automata**

Recall that the translation from Theorem 2.9 results in formulas of alternation depth two. It has been recently argued [KV98a] that a linear property given by a co-
CHAPTER 2. SYMBOLIC MODEL CHECKING WITH FORWARD REASONING 28

Büchi automaton can be translated into an alternation-free pre-$\mu_0$ query. Consequently, the model checking of linear properties that are specified by co-Büchi automata requires time that is only linear in the size of the Kripke structure. We now show that every co-Büchi automaton $A$ can also be translated into an equivalent alternation-free post-$\mu_0$ query $\theta_A$, thereby proving that the model-checking problem for co-Büchi automata lies in the intersection of alternation-free pre-$\mu_0$ and alternation-free post-$\mu_0$. A co-Büchi automaton $A$ is a finite automaton such that a path $\pi$ of $A$ is accepting if $\text{Inf}(\pi) \subseteq S_F$; that is, all the non-accepting states of $A$ occur in $\pi$ only finitely often. We define an equational $\nu$-block $B_3$ that contains the following $|S_F| + 1$ equations, with $\text{vars}(B_3) = \{X'_f \mid f \in S_F\} \cup \{X'\}$:

$$X'_f = X_f \land \bigvee_{t \in \text{pre}(f) \cap S_F} \exists X'_t,$$

$$X' = \bigvee_{f \in S_F} X'_f.$$

Then, $\theta_A = \neg \mathcal{E}(\langle \langle B_3, B_A \rangle, X' \rangle)$. Note that $\theta_A$ is alternation-free and is linear in the size of $A$.

**Theorem 2.10** Let $A$ be a co-Büchi automaton. Then, we have that the post-$\mu_0$ query $\theta_A = \neg \mathcal{E}(\langle \langle B_3, B_A \rangle, X' \rangle)$ is equivalent to $A$. The query $\theta_A$ can be constructed in linear time and its alternation depth is one.

**Proof:** Let $K$ be a Kripke structure. Let $B$ be the block tuple $\langle B_3, B_A \rangle$. We need to show that $K \models \exists A \iff K \models \neg \mathcal{E}(\langle B, X' \rangle)$. Let $\Gamma \in \mathcal{P}_{V,K}$. Since $\langle B, X' \rangle$ is a sentence, the greatest fixpoint $B^K(\Gamma)$ of $F^K_{B,\Gamma}$ is independent of $\Gamma$.

$(\Rightarrow)$ Suppose $K \models \exists A$. Then there is an initialized path $\pi_1$ in $K$ and an accepting initialized path $\pi_2$ in $A$ such that $L(\pi_1) = \ell(\pi_2)$. Since both $K$ and $A$ are finite, there are $m$ and $n$ such that $0 \leq m < n$, $\pi_1(m) = \pi_1(n)$ and $\pi_2(m) = \pi_2(n) \in S_F$. Therefore, we have initialized paths $\pi'_1 = \pi_1(0), \pi_1(1), \ldots, \pi_1(m-1) \cdot (\pi_1(m), \pi_1(m+1), \ldots, \pi_1(n-1))^\omega$ in $K$ and $\pi'_2 = \pi_2(0), \pi_2(1), \ldots, \pi_2(m-1) \cdot (\pi_2(m), \pi_2(m+1), \ldots, \pi_2(n-1))^\omega$ in $A$ such that $L(\pi'_1) = \ell(\pi'_2)$. Define the function $\text{before}$ on $[m, n]$ as follows (exactly as in the proof of Theorem 2.9).

$$\text{before}(k) = \begin{cases} 
n - 1 & \text{if } k = m, \\
k - 1 & \text{if } k > m.
\end{cases}$$

$^4$The results in [KV98a] refer to sentences of the form $\forall A$, for deterministic Büchi automata $A$. Since an $\omega$-regular language can be specified by a deterministic Büchi automaton iff its complement can be specified by a co-Büchi automaton, the corresponding result for $\exists A$, for co-Büchi automata $A$, follows by duality.
For all $s \in S_F$, let $\text{add}(s) = \{ \pi_1(k) \mid m \leq k < n \text{ and } \pi_2(k) = s \}$. Define the valuation $\Delta$ as follows.

$$
\Delta(X) = \begin{cases} 
\text{add}(s) & \text{if } X = X'_s \text{ for some } s \in S_F, \\
\emptyset & \text{otherwise}.
\end{cases}
$$

Recall that $B^K(\Gamma')$ is the greatest fixpoint of the function $F^K_B(\Gamma')$ defined as follows.

$$
F^K_B(\Gamma')(X) = \begin{cases} 
(X_f \land (\forall_{t \in \text{pre}(f) \cup S_F} \exists \Theta)(X'_f)^K(\langle B_A \rangle^K(\Gamma'))^K) & \text{if } X \in \text{vars}(B_3), \\
\langle B_A \rangle^K(\Gamma')(X) & \text{otherwise}.
\end{cases}
$$

Notice that $\Gamma$ does not appear on the right hand side of the function because the tuple $\langle B_3, B_A \rangle$ does not contain any free variables. We have that $\langle B_A \rangle^K(\Gamma')(X'_s) = \Gamma'(X'_s)$ for $s \in S_F$.

Thus we can simplify the above definition to the following.

$$
F^K_B(\Gamma')(X) = \begin{cases} 
\forall_{f \in S_F} \Gamma'(X'_f) & \text{if } X = X'_f, \\
\langle B_A \rangle^K(\Gamma')(X_f) \land \\
\forall_{t \in \text{pre}(f) \cap S_F} \text{post}(\Gamma'(X'_t)) & \text{if } X = X'_f \text{ for some } f \in S_F, \\
\langle B_A \rangle^K(\Gamma')(X) & \text{otherwise}.
\end{cases}
$$

We now show that $\Delta \sqsubseteq F^K_B(\Delta)$ for any valuation $\Gamma$. All we need to show is that if $f \in S_F$ and $u \in \text{add}(f)$ then $u \in F^K_B(\Delta)(X'_f)$. Suppose $u \in \text{add}(f)$ for some $f \in S_F$. Then for some $m \leq k < n$, we have that $f = \pi_2(k)$ and $u = \pi_1(k)$. There are initialized paths $\sigma_1$ in $K$ to $u$ and $\sigma_2$ in $A$ to $f$ such that $L(\sigma_1) = \ell(\sigma_2)$. Hence, from Lemma 2.6 we have that $u \in \langle B_A \rangle^K(\Delta)(X_f)$. Let $s = \pi_2(\text{before}(k))$. Since $s \in S_F$ then $v = \pi_1(\text{before}(k)) \in \Delta(X'_s)$. Then there is $s \in \text{pre}(f) \cap S_F$ such that $u \in \text{post}(\Delta(X'_s))$. Therefore $u \in F^K_B(\Delta)(X'_f)$.

Thus, we get that $\Delta \sqsubseteq F^K_B(\Delta)$. From Theorem 2.1, we get that the greatest fixpoint $B^K(\Gamma)$ of $F^K_B(\Gamma')$ is equal to $\text{lub}(\langle \Gamma' \mid \Gamma' \sqsubseteq F^K_B(\Gamma') \rangle)$. Therefore we have that $\Delta \sqsubseteq B^K(\Gamma)$.

This means that there is some $f \in S_F$ such that $B^K(\Gamma')(X'_f) \neq \emptyset$. Therefore $B^K(\Gamma')(X_f) \neq \emptyset$ and $K \models \theta_A$.

$(\Leftarrow)$ Suppose $K \models \neg \mathcal{E}(\langle B, X' \rangle)$. We incrementally construct infinite reverse-paths $\pi_1$ in $K$ and $\pi_2$ in $A$ maintaining the invariants that (1) $L(\pi_1) = \ell(\pi_2)$, (2) for all $k < |\pi_2|$ we have that $\pi_2(k) \in S_F$, and (3) if $w = \text{last}(\pi_1)$ and $s = \text{last}(\pi_1)$ then $w \in B^K(\Gamma)(X'_f)$. There are $f$ and $w$ such that $f \in S_F$ and $w \in B^K(\Gamma)(X'_f)$. Since $w \in B^K(\Gamma)(X'_f)$, we have that $w \in B(\Gamma)(X_f)$. Therefore $L(w) = \ell(f)$. Let $\pi_1 = w$ and
\[ \pi_2 = f \] initially. Since \( w \in B^K(\Gamma)(X'_f) \), there is a \( t \in \text{pre}(f) \cup S_F \) and \( v \in B^K(\Gamma)(X'_f) \) such that \( R(v, w) \). Augment \( \pi_1 \) by appending \( v \) and \( \pi_2 \) by appending \( t \). The above process can be repeated ad infinitum to get infinite reverse-paths \( \pi_1 \) in \( K \) and \( \pi_2 \) in \( A \) such that \( L(\pi_1) = \ell(\pi_2) \) and for all \( k \) we have that \( \pi_2(k) \in S_F \). Since both \( K \) and \( A \) are finite, there are \( m \) and \( n \) such that \( 0 < m < n \) and \( \pi_1(m) = \pi_1(n) \). Therefore, we have reverse-paths \( \pi'_1 = \pi_1(0), \pi_1(1), \ldots, \pi_1(m-1) \cdot (\pi_1(m), \pi_1(m+1), \ldots, \pi_1(n-1))^{\omega} \) in \( K \) and \( \pi'_2 = \pi_2(0), \pi_2(1), \ldots, \pi_2(m-1) \cdot (\pi_2(m), \pi_2(m+1), \ldots, \pi_2(n-1))^{\omega} \) in \( A \) such that \( L(\pi'_1) = \ell(\pi'_2) \).

Consider the paths \( (\pi_1(m), \pi_1(n-1), \pi_1(n-2), \ldots, \pi_1(m+1))^{\omega} \) in \( K \) and \( (\pi_2(m), \pi_2(n-1), \pi_2(n-2), \ldots, \pi_2(m+1))^{\omega} \) in \( A \). Let \( \pi_2(m) = f \in S_F \). We know that \( \pi_1(m) \in B^K(\Gamma)(X'_f) \) and therefore \( \pi_1(m) \in B^K(\Gamma)(X'_f) \). From Theorem 2.7, we get that there are initialized paths \( \sigma_1 \) in \( K \) to \( \pi_1(m) \) and \( \sigma_2 \) in \( A \) to \( \pi_2(m) \) such that \( L(\sigma_1) = \ell(\sigma_2) \). We get initialized paths \( \pi'_1 = \sigma_1(0), \sigma_1(1), \ldots, \sigma_1(\text{last}(\sigma_1) - 1) \cdot (\pi_1(m), \pi_1(n-1), \pi_1(n-2), \ldots, \pi_1(m+1))^{\omega} \) in \( K \) and \( \pi'_2 = \sigma_2(0), \sigma_2(1), \ldots, \sigma_2(\text{last}(\sigma_2) - 1) \cdot (\pi_2(m), \pi_2(n-1), \pi_2(n-2), \ldots, \pi_2(m+1))^{\omega} \) in \( A \) such that \( L(\pi'_1) = \ell(\pi'_2) \).

**Example 2.5** Consider the LTL formula \( \psi = \Diamond p \). A computation \( \pi \) satisfies \( \psi \) iff \( p \) always holds after a certain point along \( \pi \). The co-Büchi automaton \( A_\psi \) in the figure accepts exactly all the computations that satisfy \( \psi \).

\[
A_\psi: \quad \begin{array}{c}
\text{p} \\
\text{s} \\
\text{t}
\end{array}
\]

The \( \nu \)-block \( B_3 \) contains the equations

\[
\begin{align*}
X'_s &= X_s \land \exists \Theta X_s, \\
X &= X'_s.
\end{align*}
\]

### 2.2.2 Out

We now show that there exist branching temporal-logic specifications that cannot be model checked by evaluating post-\( \mu \mathcal{E} \) queries. A post-\( \mu \mathcal{E} \) query \( \theta \) is equivalent to a pre-\( \mu \) sentence \( \phi \) if for every Kripke structure \( K \), we have \( K \models \phi \) iff \( K \models \theta \).
Proposition 2.11 There exist pre-μ sentences (in fact, CTL sentences) that have no equivalent post-μE queries.

Proof: We give examples of pre-μ formulas that do not have equivalent post-μE queries. In all cases, we give post-bisimilar Kripke structures such that the formula can distinguish between them. From Proposition 2.4, no post-μE query can distinguish between post-bisimilar Kripke structures, and we are done.

1. Consider the formula \( \phi_1 = \langle \langle \mu, \{X = \exists \Box p \land \exists \Box q\} \rangle, X \rangle \), which is equivalent to the CTL sentence \( \exists \Box p \land \exists \Box q \). It distinguishes between the post-bisimilar Kripke structures \( K_1 \) and \( K'_1 \) in Figure 2.1. Indeed, \( K_1 \) satisfies \( \phi_1 \) but \( K'_1 \) does not satisfy \( \phi_1 \).

2. Consider the formula \( \phi_2 = \langle \langle \mu, \{X_1 = (r \land X_2 \land X_3) \lor \exists \Box X_1\} \rangle, \mu, \{X_2 = p \lor \exists \Box X_2\}, \mu, \{X_3 = q \lor \exists \Box X_3\} \rangle, X_1 \rangle \), which is equivalent to the CTL sentence \( \exists \Box (r \land \exists \Box X_1 \lor \exists \Box X_2 \lor \exists \Box X_3) \).
\[ \exists \Diamond p \land \exists \Diamond q \]. It distinguishes between the post-bisimilar Kripke structures \( K_2 \) and \( K_2' \) in Figure 2.2. Indeed, \( K_2 \) satisfies \( \phi_2 \) but \( K_2' \) does not satisfy \( \phi_2 \).

3. Consider the formula \( \phi_3 = \langle \langle \nu, \{X = \exists \Diamond p \land \exists \Diamond X\} \rangle, X \rangle \), which is equivalent to the CTL sentence \( \exists \Box \exists \Diamond p \). It distinguishes between the post-bisimilar Kripke structures \( K_3 \) and \( K_3' \) in Figure 2.3. Indeed, \( K_3 \) satisfies \( \phi_3 \) but \( K_3' \) does not satisfy \( \phi_3 \).

Interestingly, while the CTL sentence \( \exists \Box \exists \Diamond p \) has no equivalent post-\( \mu \) query, the CTL sentence \( \exists \Box \exists \Diamond p \), which is equivalent to the pre-\( \mu \) sentence \( \langle \langle B_1, B_2 \rangle, X_1 \rangle \) with \( B_1 = \langle \nu, \{X_1 = X_2 \land \exists \Diamond X_1\} \rangle \) and \( B_2 = \langle \mu, \{X_2 = p \lor \exists \Diamond X_2\} \rangle \), and which is not equivalent to any LTL sentence [CD88], does have an equivalent query in post-\( \mu \). The query is \( \neg \mathcal{E}(\langle \langle B_3, B_4 \rangle, X_1 \rangle) \), with \( B_3 = \langle \nu, \{X_1 = p \land X_2, X_2 = X_3 \land \exists \Diamond X_2\} \rangle \) and \( B_4 = \langle \mu, \{X_3 = \text{init} \lor \exists \Diamond X_3\} \rangle \).

\section*{2.3 Hierarchy of Pre and Post Logics}

Let \( \mathcal{L}_1 \) and \( \mathcal{L}_2 \) be two logics whose sentences are interpreted over Kripke structures. The logic \( \mathcal{L}_2 \) is as expressive as the logic \( \mathcal{L}_1 \) if for every sentence \( \phi_1 \) of \( \mathcal{L}_1 \), there is a sentence \( \phi_2 \in \mathcal{L}_2 \) such that for every Kripke structure \( K \), we have \( K \models \phi_1 \) iff \( K \models \phi_2 \). The logic \( \mathcal{L}_2 \) is more expressive than \( \mathcal{L}_1 \) if \( \mathcal{L}_2 \) is as expressive as \( \mathcal{L}_1 \) but \( \mathcal{L}_2 \) is not as expressive as \( \mathcal{L}_2 \). The logic \( \mathcal{L}_2 \) is as distinguishing as the logic \( \mathcal{L}_1 \) if for all Kripke structures \( K \) and \( K' \), if there is a sentence \( \phi_1 \) of \( \mathcal{L}_1 \) such that \( K \models \phi_1 \) but \( K' \not\models \phi_1 \), then there is a sentence \( \phi_2 \) of \( \mathcal{L}_2 \) such that \( K \models \phi_2 \) but \( K' \not\models \phi_2 \). Finally, the logic \( \mathcal{L}_2 \) is more distinguishing than \( \mathcal{L}_1 \) if
$L_2$ is as distinguishing as $L_1$ but $L_1$ is not as distinguishing as $L_2$. In this section, we study the distinguishing and the expressive powers of \textit{pre-$\mu$} and \textit{post-$\mu$} and the query logics they induce.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure2_4.png}
\caption{$K_4$ and $K'_4$ are init-post-bisimilar but not post-bisimilar}
\end{figure}

**Proposition 2.12** The distinguishing powers of \textit{pre} and \textit{post} logics are summarized in the figure below. An arrow from logic $L_1$ to logic $L_2$ indicates that $L_1$ is as distinguishing as $L_2$. A line without arrow indicates incomparability.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure2_4.png}
\end{figure}

**Proof:** Proposition 2.3 implies that the distinguishing powers of \textit{pre-$\mu_0$} and \textit{pre-$\mu_\infty$} coincide. Similarly, proposition 2.4 implies that the distinguishing powers of \textit{post-$\mu_0$} and \textit{post-$\mu_\infty$} coincide. In order to prove the incomparability results, we show that the four relations init-pre-bisimulation, init-post-bisimulation, pre-bisimulation, and post-bisimulation are all distinct. Recall that there may be states in a Kripke structure that are not reachable from an initial state, as there may be states from which no initial state is reachable. Consider the Kripke structures $K_4$ and $K'_4$ appearing in Figure 2.4. There is an init-pre-bisimulation and an init-post-bisimulation between $K_4$ and $K'_4$, but no pre-bisimulation or post-bisimulation. Hence, pre-bisimulation is more distinguishing than init-pre-bisimulation and post-bisimulation is more distinguishing than init-post-bisimulation. Now consider the Kripke structures $K_3$ and $K'_3$ appearing in Figure 2.3. There is a post-bisimulation and an
init-post-bisimulation between $K_3$ and $K'_3$, but no pre-bisimulation or init-pre-bisimulation. Also, there is a pre-bisimulation and an init-pre-bisimulation between $K_3^{-1}$ and $K'_3^{-1}$ but no post-bisimulation or init-post-bisimulation. Hence, pre-bisimulation and post-bisimulation as well as init-pre-bisimulation and init-post-bisimulation are incomparable.

We make the following observations about the distinguishing power hierarchy of pre and post logics.

- If we restrict attention to Kripke structures in which all the states are reachable from some state satisfying $\text{init}$, then the hierarchy of pre logics collapses. Formally, suppose that $K \models E(\neg \exists \Diamond \text{init})$. Then for every formula $\phi$ of pre-$\mu$, we have that $K \models E(\phi)$ iff $K \not\models \exists \Diamond \phi$.

- If we restrict attention to Kripke structures in which all the states can reach some state satisfying $\text{init}$, then the hierarchy of post logics collapses. Formally, suppose that $K \models E(\neg \exists \Diamond \text{init})$. Then for every formula $\phi$ of post-$\mu$, we have that $K \models E(\phi)$ iff $K \not\models \exists \Diamond \phi$.

Proposition 2.13 The expressive powers of pre and post logics are summarized in the figure below. An arrow from logic $\mathcal{L}_1$ to logic $\mathcal{L}_2$ indicates that $\mathcal{L}_1$ is as expressive as $\mathcal{L}_2$. A line without arrow indicates incomparability.

Proof: It is easy to see that if a logic $\mathcal{L}_2$ is not as distinguishing as a logic $\mathcal{L}_1$, then $\mathcal{L}_2$ is not as expressive as $\mathcal{L}_1$. Therefore, most of our expressiveness results follow from the corresponding results about distinguishability. In addition, as a Kripke structure $K$ satisfies a sentence $\phi$ iff $K$ satisfies the query $\neg E(\text{init} \land \phi)$, the query logics pre-$\mu_0$ and post-$\mu_0$ are more expressive than pre-$\mu$ and post-$\mu$, respectively. In order to prove the advantage of the full query logics pre-$\mu_\varepsilon$ and post-$\mu_\varepsilon$ over its subsets pre-$\mu_0$ and post-$\mu_0$, it is easy to see that no query of the query logics pre-$\mu_0$ and post-$\mu_0$ is equivalent to the query $E(p) \lor E(q)$.
CHAPTER 2. SYMBOLIC MODEL CHECKING WITH FORWARD REASONING

2.4 Discussion and Experimental Results

2.4.1 Intersection of pre and post logics

While previous works presented symbolic forward state-traversal procedures for model checking some isolated linear and branching properties [INH96, IN97], we attempted to study more systematically the class of properties that can be model checked using both symbolic forward and backward state traversal. In particular, we showed that all \( \omega \)-regular linear properties (which includes all properties expressible in LTL) fall into this class, while some simple branching properties (expressible in CTL) do not. Our proof goes through a translation of \( \omega \)-regular linear properties to the a query logic that uses the post operator on sets of states. Thus, given a Kripke structure \( K \) and an automaton \( A \) for the property, we translate \( A \) to a query in \( post-\mu_0 \) and evaluate the query in \( K \). Alternatively, one can check \( K \) with respect to the formula \( \exists A \) by searching for an accepting path in the intersection \( K \times A \). It follows that when the type of \( A \) is such that the accepting paths of \( K \times A \) can be specified using \( post-\mu_0 \), forward symbolic model checking of \( K \) with respect to \( \exists A \) is possible. In particular, when \( A \) is a Büchi automaton, the accepting paths of \( K \times A \) can be specified by a formula of the form \( \Box \Diamond p \), which is equivalent to a \( post-\mu_0 \) query of alternation depth two, and when \( A \) is a co-Büchi automaton, the required formula is of the form \( \Diamond \Box p \), which is equivalent to a \( post-\mu_0 \) query of alternation depth one.

We showed that every query that can be specified in both \( pre-\mu_0 \) and \( post-\mu_0 \) cannot distinguish between structures that are both \( pre \)-bisimilar and \( post \)-bisimilar. The exact characterization of the intersection \( pre-\mu_0 \cap post-\mu_0 \) remains open. In [GK94], the authors identified a set of temporal-logic sentences called equi-linear. In particular, a \( pre-\mu \) sentence is equi-linear if it cannot distinguish between two Kripke structures with the same language (i.e., observation sequences that correspond to initialized paths). Clearly, all LTL sentences are equi-linear. However, some CTL sentences that have no equivalent LTL sentence are also equi-linear. For example, it is shown in [GK94] that while the CTL sentence \( \exists \Box \exists \Diamond p \) is not equi-linear, the CTL sentence \( \exists \Box \exists \Box p \) is equi-linear. Motivated by the examples from Section 2.2.2, we conjecture that equi-linearity precisely characterizes the properties that can be model checked using both symbolic forward and backward state traversal. Formally, we conjecture that a \( pre-\mu \) sentence is equi-linear iff there is an equivalent \( post-\mu_0 \) query.
2.4.2 Union of pre and post logics

In this chapter, we primarily think of post-$\mu_{e}$ as a language for describing symbolic model-checking procedures for temporal-logic specifications. Furthermore, we have focused on specification languages that contain only future temporal operators. Since LTL with past temporal operators is not more expressive than LTL without past operators [LPZ85], every LTL+past sentence can also be translated into an equivalent post-$\mu_{0}$ query. In addition, post-$\mu$ also permits the easy evaluation of branching past temporal operators that cannot be evaluated using pre-$\mu$. For example, the sentence $\forall \Box\left(grant \to (\neg init) \vee \overline{\Box req}\right)$, where $\overline{\Box}$ is a past version of the “weak-until” operator [MP92], specifies that grants are given only upon request. Assuming a branching interpretation for past temporal operators [KP95], this sentence has an equivalent post-$\mu_{0}$ query, but no equivalent pre-$\mu_{e}$ query; that is, it can be model checked by symbolic forward state traversal but not by symbolic backward state traversal.

While the intersection $pre_{-}\mu_{e} \cap post_{-}\mu_{e}$ identifies the queries that can be model checked by both forward and backward symbolic state traversal, it is the “union” $(pre_{-}\mu \cup post_{-}\mu)_{e}$ that identifies the queries that can be model checked at all symbolically, by mixed forward and backward state traversal. The logics $pre_{-}\mu_{e}$ and $post_{-}\mu_{e}$ are strict subsets of $(pre_{-}\mu \cup post_{-}\mu)_{e}$. For example, the $pre_{-}\mu \cup post_{-}\mu$ sentence $init \land \exists o p_{1} \land \exists o p_{2} \land \exists q_{1} \land \exists q_{2}$ is not expressible in either $pre_{-}\mu_{e}$ or $post_{-}\mu_{e}$. Furthermore, it is the alternation-free fragment of $(pre_{-}\mu \cup post_{-}\mu)_{e}$ that identifies the queries that can be model checked efficiently. Thus it is also of interest to ask which temporal logics can be translated into the (alternation-free) union of pre and post query logics. Such temporal logics can have both future and past temporal operators. In particular, it is easy to see that every CTL+past sentence (under the branching interpretation for past) has an equivalent query in the alternation-free fragment of $(pre_{-}\mu \cup post_{-}\mu)_{0}$.

2.4.3 Experimental results

In our experiments, we performed BDD-based symbolic model checking on a parameterized sliding-window protocol, described in Appendix A, for the reliable transmission

---

5By the union $pre_{-}\mu \cup post_{-}\mu$ we refer to the logic with all four modal operators $\exists \Box$, $\forall \Box$, $\exists \Box$, and $\forall \Box$. It has, of course, strictly more sentences than the union of the sets of $pre_{-}\mu$ and $post_{-}\mu$ sentences.

6In fact, not only can model checking algorithms be extended from $pre_{-}\mu$ to $(pre_{-}\mu \cup post_{-}\mu)_{e}$ without extra cost, the satisfiability problem for the union is also not harder than the satisfiability problem for either $pre_{-}\mu$ or $post_{-}\mu$ [Var98].
of packets over an unreliable channel. The parameter to the protocol is WINDOW, the number of outstanding unacknowledged messages at the sender end. In the protocol, the messages are modeled as boolean values. We checked whether all computations of the protocol satisfy the partial specification $\phi$, which states that if the produced message $msgP$ toggles infinitely often at the sender end, then so does the consumed message $msgC$ at the receiver end. Formally, the specification $\phi$ is given by the LTL sentence

$$\Box \Diamond (msgP \leftrightarrow \bigcirc \neg msgP) \rightarrow \Box \Diamond (msgC \leftrightarrow \bigcirc \neg msgC).$$

We note that this sentence cannot be handled by the methods presented in [INH96, IN97].

In the table below we list the running times (in seconds) for different values of WINDOW for checking $\phi$ using VIS [BHSV+96] for both symbolic forward and backward state traversal. The quantity within the parentheses is the number of boolean variables used to encode the state space of the protocol. It is folk wisdom in symbolic model checking that using don’t-care minimization based on unreachable states can dramatically improve the running times. So we also applied first symbolic forward state traversal to compute the set of reachable states and then symbolic backward state traversal for model checking, using the unreachable states as don’t cares. These results are shown in the last column. A dash indicates an unsuccessful verification attempt where the model checking run exceeded our time limit. In the future, we hope to compare our approach also against enumerative forward state-traversal methods for LTL model checking.

<table>
<thead>
<tr>
<th>WINDOW</th>
<th>Forward</th>
<th>Backward</th>
<th>Reach-optimized backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 (30)</td>
<td>18</td>
<td>222</td>
<td>91</td>
</tr>
<tr>
<td>3 (45)</td>
<td>300</td>
<td>4584</td>
<td>-</td>
</tr>
<tr>
<td>4 (50)</td>
<td>5231</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2.1: Experimental results for sliding window protocol
Chapter 3

Assume-Guarantee Reasoning

In this chapter, we focus on the verification problem of refinement checking when the specification is a more abstract design. We refer to the design being analyzed as the implementation. The refinement-checking problem is PSPACE-hard in the size of the implementation description and in the state space of the specification [MS72, KV98b]. Not surprisingly, algorithms for refinement checking are exponential in the size of the implementation description and doubly exponential in the size of the specification description. To combat this state explosion problem, we divide the verification task at hand into simpler tasks by making use of the compositional structure of both implementation and specification under the paradigm of assume-guarantee reasoning. We have developed a formal-verification tool, called MOCHA [AHM+98], which is based on the system description language of reactive modules [AH96]. Reactive modules permit the modular and hierarchical description of heterogeneous systems, and have been designed explicitly to support compositional techniques such as assume-guarantee reasoning. We illustrate our methodology on a simple three-stage pipeline and also describe our verification of a complex hardware circuit that implements Tomasulo’s algorithm for out-of-order execution.

We now briefly outline our methodology, which approaches a refinement-checking problem of the form $P_1 \parallel P_2 \preceq Q$ (where $\preceq$ is the trace-containment relation) by introducing abstraction and witness modules. Suppose that the state space of the implementation $P_1 \parallel P_2$ is too large to be handled by exhaustive search algorithms. A naïve compositional approach would attempt to prove both $P_1 \preceq Q$ and $P_2 \preceq Q$, and then conclude $P_1 \parallel P_2 \preceq Q$. Though sound, the naïve approach often fails in practice, because $P_1$ usually refines $Q$ only in a suitable constraining environment, and so does $P_2$. Hence we construct a suitable
CHAPTER 3. ASSUME-GUARANTEE REASONING

constraining environment $A_2$ for $P_1$, and similarly $A_1$ for $P_2$. Since $A_1$ describes the aspects of $P_1$ that are relevant to constraining $P_2$, and similarly $A_2$ is an abstract description of $P_2$, the two new modules $A_1$ and $A_2$ are called abstraction modules. By assume-guarantee reasoning, we conclude $P_1 \parallel P_2 \preceq Q$ from the two proof obligations $P_1 \parallel A_2 \preceq A_1 \parallel Q$ and $A_1 \parallel P_2 \preceq Q \parallel A_2$.

Traditionally, the size of the implementation has been viewed as the main source of complexity for the refinement-checking problem. In our approach, we shift the focus to the size of the gap between the implementation and the specification. As an extreme case, if we are given two identical copies of a design, we ought to be able to verify that one is a valid refinement of the other, no matter how large the designs. We want the success rate of our methodology to increase if the designer invests effort in structuring the implementation and specification so as to expose more commonality between them. Abstraction modules form an intermediate layer between the implementation and the specification, and thus provide a systematic way of reducing the gap. In our case studies, we found that abstraction modules generally take the form of abstract definitions for hidden implementation variables. When composed with the original specification, which often specifies only relationships between primary inputs and outputs, the abstraction modules yield a richer specification that is closer to the implementation. Constructing good abstraction modules requires manual effort. Once constructed, our methodology automatically makes effective use of the abstraction modules to decompose the refinement check.

Even if the state space of the implementation becomes manageable as a result of proof decomposition, each remaining refinement check, say $P' = P_1 \parallel A_2 \preceq A_1 \parallel Q = Q'$, is still PSPACE-hard in the size of the specification state space. However, for the special case that all variables of $Q'$ are also present in $P'$ (in this case, we say that $Q'$ is projection refinable by $P'$), the refinement check reduces to a transition-invariant check, which verifies that every move of $P'$ can be mimicked by $Q'$. The complexity of this procedure is linear on the state spaces of both $P'$ and $Q'$. If $Q'$ is not projection refinable by $P'$, our methodology advocates the introduction of a witness module $W$, which makes explicit how the hidden variables of the specification $Q'$ depend on the state of the implementation $P'$. Then $Q'$ is projection refinable by $P' \parallel W$, and it suffices to prove $P' \parallel W \preceq Q'$ in order to conclude $P' \preceq Q'$. The construction of witness modules also requires manual effort, but whenever the specification $Q'$ simulates the implementation $P'$, a suitable witness can be found.

Related work. Assume-guarantee rules for various formalisms can be found in
Abstraction modules have appeared before as refinement maps [McM97]. Witnesses have appeared in various guises and forms (homomorphisms, refinement mappings, simulation relations, etc.) in different works [Lam83, LT87, AL91, BBL92, CGL92, KUR94, LV95, McM97]. Our choice of case studies is not new either; other correctness proofs for Tomasulo’s algorithm can be found in [DP97, McM98]

3.1 Transition constraints

A variable has a name and a type associated with it. For any set of variables \( V \) let \( V' \) denote the set of new variables whose names are obtained from the names of variables in \( V \) by putting \( \_{}' \) at the end with the type remaining unchanged. For example, if \( V = \{a, b\} \) with \( a \) and \( b \) being of types integer and boolean respectively, then \( V' = \{a', b'\} \) where \( a' \) and \( b' \) have the same types as \( a \) and \( b \) respectively. A valuation for a set of variables \( V \) is a function that maps every variable in \( V \) to a value of appropriate type. For example the function \( f \) on \( V \) such that \( f(a) = 5 \) and \( f(b) = \text{false} \) is a valuation for \( V \). A transition constraint \( A \) is a 4-tuple \( \langle \text{Priv}(A), \text{Obs}(A), \text{I}(A), \text{U}(A) \rangle \) with the following components:

- A finite set \( \text{Priv}(A) \) of private variables and a finite set \( \text{Obs}(A) \) of observable variables, such that \( \text{Priv}(A) \cap \text{Obs}(A) = \emptyset \). The set \( \text{Var}(A) \) is the union of \( \text{Priv}(A) \) and \( \text{Obs}(A) \).

- A initial predicate \( \text{I}(A) \) over \( \text{Var}(A)' \) and an update predicate \( \text{U}(A) \) over \( \text{Var}(A) \cup \text{Var}(A)' \).

A state of a transition constraint \( A \) is a valuation for \( \text{Var}(A) \). A state \( s \) of \( A \) is initial if it satisfies the initial predicate of \( A \). Given two states \( s \) and \( t \), we write \( s \rightarrow_A t \) if the update predicate of \( A \) evaluates to true, when its unprimed variables are assigned values from \( s \) and its primed variables are assigned values from \( t \). A trajectory of \( A \) is a finite sequence \( s_0, \ldots, s_n \) of states such that (1) \( s_0 \) is an initial state of \( A \), and (2) for \( i < n \), we have \( s_i \rightarrow_A s_{i+1} \). The states that lie on trajectories are called reachable. An observation of \( A \) is a valuation for \( \text{Obs}(A) \). If \( s \) is a valuation to a set of variables and \( P \subseteq \text{Var}(A) \), we use \( [s]_P \) to denote the valuation restricted to the variables in \( P \). We also use \( [s]_A \) to denote \( [s]_{\text{Obs}(A)} \). For a state sequence \( \overline{s} = s_0, \ldots, s_n \), we write \( [\overline{s}]_A = [s_0]_A, \ldots, [s_n]_A \) for the corresponding observation sequence. If \( \overline{s} \) is a trajectory of \( A \), then the projection \( [\overline{s}]_A \) is called a trace of \( A \). Let \( Y \subseteq \text{Var}(A) \) for some constraint \( A \). Then \( A \) is said to be non-blocking for \( Y \) if (1) for all valuations \( \Gamma \) of \( Y \) there is an initial state \( t \) of \( A \) such that
\[ [t]_Y = \Gamma, \text{ and (2) for all states } s \text{ of } A \text{ and valuations } \Gamma \text{ of } Y' \text{ there is a state } t \text{ of } A \text{ such that } s \to_A t \text{ and } [t]_Y = \Gamma. \]

**Parallel composition.** The composition operation combines two transition constraints into a single transition constraint whose behavior captures the interaction between the two components. Two transition constraints \( A \) and \( B \) are *compatible* if their sets of private variables is disjoint. Given two compatible transition constraints \( A \) and \( B \), the composition \( A \parallel B \) is the transition constraint \( \langle \text{Priv}(A) \cup \text{Priv}(B), \text{Obs}(A) \cup \text{Obs}(B), \mathcal{I}(A) \land \mathcal{I}(B), \mathcal{U}(A) \land \mathcal{U}(B) \rangle. \)

The notion that two transition constraints describe the same system at different levels of detail is captured by the refinement relation. The transition constraint \( B \) is *refinable* by transition constraint \( A \) if every observable variable of \( B \) is an observable variable of \( A \). The transition constraint \( A \) *refines* the transition constraint \( B \), written \( A \preceq B \), if (1) \( B \) is refinable by \( A \), and (2) for every trajectory \( \bar{s} \) of \( A \), the projection \( [\bar{s}]_B \) is a trace of \( B \).

**Witness constraints.** Let \( A \) and \( B \) be two transition constraints. The problem of checking if \( A \preceq B \) is PSPACE-hard in the state space of \( B \). However, the refinement check is simpler in the special case in which all variables of \( B \) are observable. The module \( B \) is *projection refinable* by the module \( A \) if (1) \( B \) is refinable by \( A \), and (2) \( B \) has no private variables. If \( B \) is projection refinable by \( A \), then every variable of \( B \) is observable in both \( A \) and \( B \). Therefore, checking if \( A \preceq B \) reduces to checking if for every trajectory \( \bar{s} \) of \( A \), the projection \( [\bar{s}]_B \) is a trajectory of \( B \). According to the following theorem, this can be done by a transition-invariant check, whose complexity is linear in the state spaces of both \( A \) and \( B \).

**Theorem 3.1 (Projection refinement)** Let \( A \) and \( B \) be two transition constraints where \( B \) is projection refinable by \( A \). Then \( A \preceq B \) iff (1) if \( s \) is an initial state of \( A \), then \( [s]_B \) is an initial state of \( B \), and (2) if \( s \) is a reachable state of \( A \) and \( s \to_A t \), then \( [s]_B \to_B [t]_B \).

**Proof:** \((\Rightarrow)\) Since \( B \) is projection refinable by \( A \), we have that for every trajectory \( \bar{s} \) of \( A \) the projection \( [\bar{s}]_B \) is a trajectory of \( B \). (1) Suppose \( s \) is an initial state of \( A \). Then \( s \) is a trajectory of \( A \). Therefore \( [s]_B \) is a trajectory of \( B \) and \( [s]_B \) is an initial state of \( B \). (2) Suppose \( s \) is a reachable state of \( A \). Then there is a trajectory \( s_0, s_1, \ldots, s, t \) of \( A \) ending in \( s \). Since \( s \to_A t \) the sequence \( s_0, s_1, \ldots, s, t \) is also a trajectory of \( A \). Therefore \( [s_0]_B, [s_1]_B, \ldots, [s]_B, [t]_B \) is a trajectory of \( B \) and we get \( [s]_B \to_B [t]_B \).
(⇐=) We show that for every trajectory $\pi$ of $A$, the sequence $[s]_B$ is a trajectory of $B$ by induction on the length of $\pi$.

**Base step.** Suppose $\pi = s_0$ is a trajectory of $A$. Then $[s_0]_B$ is an initial state of $B$. Therefore $[s]_B$ is a trajectory of $B$.

**Inductive step.** Suppose $\pi = s_0, s_1, \ldots, s_n$ is a trajectory of $A$ for some $n > 0$. Then $\pi' = s_0, s_1, \ldots, s_{n-1}$ is a trajectory of $A$, state $s_{n-1}$ is reachable state of $A$ and $s_{n-1} \rightarrow_A s_n$. From the induction hypothesis, we have that $[s_0]_B, [s_1]_B, \ldots, [s_{n-1}]_B$ is a trajectory of $B$. Moreover $[s_{n-1}]_B \rightarrow_B [s_n]_B$. Therefore $[s_0]_B, [s_1]_B, \ldots, [s_{n-1}]_B, [s_n]_B$ is a trajectory of $B$.

We make use of this theorem as follows. Suppose that $B$ is refinable by $A$, but not projection refinable. This means that there are some private variables in $B$. Define $B''$ to be the constraint obtained by making every private variable of $B$ an observable variable. If we compose $A$ with a constraint $W$ whose observable variables include the private variables of $B$, then $B''$ is projection refinable by the composition $A||W$. Moreover, if $W$ does not constrain $A$, that is, it is non-blocking on those observable variables that are also observable variables of $A$, then $A||W \preceq B''$ implies $A \preceq B$. Such a constraint $W$ is called a witness to the refinement $A \preceq B$. The following theorem states that in order to check refinement, it is sufficient to first find a non-blocking witness constraint and then check projection refinement.

**Theorem 3.2 (Witness transition constraints)** Let $A$ and $B$ be two transition constraints such that $B$ is refinable by $A$. Let $W$ be a transition constraint such that (1) $W$ is compatible with $A$, (2) $\text{Priv}(B) \subseteq \text{Obs}(W)$, and (3) $W$ is non-blocking on $\text{Obs}(W) \cap \text{Obs}(A)$. Then $B''$ is projection refinable by $A||W$, and if $A||W \preceq B''$ then $A \preceq B$.

**Proof:** $B''$ by definition does not have any private variables. Since $B$ is refinable $A$ we have that $\text{Obs}(B) \subseteq \text{Obs}(A)$. Therefore $\text{Obs}(B'') = \text{Obs}(B) \cup \text{Priv}(B) \subseteq \text{Obs}(A) \cup \text{Obs}(W) = \text{Obs}(A||W)$.

Suppose $A||W \preceq B''$. Let $P = \text{Obs}(W) \cap \text{Obs}(A)$. We show that for every trajectory $\pi$ of $A$, there is a trajectory $\pi'$ of $A||W$ such that $[\pi]_A = [\pi']_A$. Since $A||W \preceq B''$, we can conclude that $A \preceq B$. We do induction on the length of $\pi$.

**Base step.** Let $\pi = s_0$. Since $W$ is non-blocking on $P$ there is an initial state $s'_0$ of $W$ such that $[s'_0]_P = [s_0]_P$. Therefore there is an initial state $t_0$ of $A||W$ such that $[t_0]_{\text{Var}(A)} = s_0$. 


CHAPTER 3. ASSUME-GUARANTEE REASONING

Inductive step. Let \( \pi = s_0, s_1, \ldots, s_n \) for some \( n > 0 \). From the induction hypothesis, there is a trajectory \( t_0, t_1, \ldots, t_{n-1} \) of \( A || W \) such that for all \( i < n \) we have that \([s_i]_A = [t_i]_A\). Let \( s = [s_{n-1}]_{\forall r(W)}\). Since \( W \) is non-blocking on \( P \), there is a state \( s' \) of \( W \) such that \( s \rightarrow_W s' \) and \([s']_P = [s_n]_P\). Therefore there is a state \( t \) of \( A || W \) such that \( t_{n-1} \rightarrow_A || W t \) and \([t]_{\forall r(A)} = s\). Thus we get a trajectory \( \bar{t} = t_0, t_1, \ldots, t_{n-1}, t \) of \( A || W \) such that \([\bar{s}]_A = [\bar{t}]_A\).

Assume-guarantee reasoning. Consider the problem of proving \( A \preceq B \) where \( A \) is the composition \( A_1 || A_2 || \ldots || A_n \) of \( n \) transition constraints and \( B \) is the composition \( B_1 || B_2 || \ldots || B_m \) of \( m \) transition constraints. Typically, the state spaces of the components of \( A \) are large and the state spaces of the components of \( B \) are small. The state space of \( A \) is the product of the state spaces of \( A_1, \ldots, A_n \) and consequently quite large. Therefore it is impractical to explore the state space of \( A \) directly to prove \( A \preceq B \). We would like to decompose the proof into smaller proof obligations that reason locally about the various components in \( A \) constrained by a suitable environment containing components mostly from \( B \). We prove below a theorem that lets us do this decomposition.

We use \( A^\tau \) to denote the transition constraint with the same variables as \( A \), which restricts their valuations only up to \( \tau \) update rounds. Formally, a state sequence \( \pi = s_0, \ldots, s_n \) is a trajectory of \( A^\tau \) if (1) \( n \leq \tau \) and \( \pi \) is a trajectory of \( A \), or (2) \( n > \tau \) and \( s_0, \ldots, s_\tau \) is a trajectory of \( A \). Note that if \( \tau < 0 \), then every state sequence is a trajectory of \( A^\tau \).

Theorem 3.3 (Assume-guarantee rule for constraints [McM98]) Suppose \( A = A_1 || A_2 || \ldots || A_n \) and \( B = B_1 || B_2 || \ldots || B_m \) are transition constraints. Let \( \prec \) be a partial order on the components of \( B \), let \( Z(B_k) = \{ B_j | B_j \prec B_k \} \), and let \( Z^C(B_k) = \{ B_j | B_j \not\in Z(B_k) \} \). For each \( B_k \), let \( C_k \) be some composition of transition constraints from \( A \), let \( D_k \) be some composition of transition constraints from \( Z(B_k) \), and let \( E_k \) be some composition of transition constraints from \( Z^C(B_k) \). If \( C_k^\tau || D_k^\tau || E_k^{\tau-1} \preceq B_k^\tau \) for all \( 1 \leq i \leq m \) and \( \tau \in \mathbb{N} \), then \( A \preceq B \).

Proof: Let \( B = \{ B_1, B_2, \ldots, B_m \} \). We extend the order \( \prec \) on the set \( B \) to the set \( B \times \mathbb{N} \) as follows. We say that \( \langle B_j, \tau' \rangle < \langle B_k, \tau \rangle \) iff \( B_j < B_k \) or \( B_j = B_k \) and \( \tau' < \tau \). It is not too difficult to see that \( < \) is a partial order on \( B \times \mathbb{N} \). We write \( A \models \langle B_k, \tau \rangle \) if all traces of \( A \) of length \( \tau + 1 \) are traces of \( B_k \). We show by well-founded induction on the partial order \( < \) that \( A \models \langle B_k, \tau \rangle \) for all \( 1 \leq k \leq m \) and for all \( \tau \in \mathbb{N} \).
Suppose \( A \models \langle B_j, \tau' \rangle \) for all \( \langle B_j, \tau' \rangle \preccurlyeq \langle B_k, \tau \rangle \). Then we show that \( A \models \langle B_k, \tau \rangle \).

Consider a trace \( \sigma \) of \( A \) of length \( \tau + 1 \).

1. Since \( \sigma \) is a trace of \( A \), we have that \( \sigma \) is a trace of \( A_i \) and hence a trace of \( A_i^\tau \) for all \( 1 \leq i \leq n \).

2. From the induction hypothesis, we have that \( \sigma \) is a trace of \( B_j \) and hence of \( B_j^\tau \) for all \( B_j \in Z(B_k) \).

3. **Case 1.** If \( \tau = 0 \) then \( \sigma \) is a trace of \( B_j^{\tau-1} \) for all \( 1 \leq j \leq m \).

   **Case 2.** If \( \tau > 0 \) then there is a trace \( \sigma' \) of \( A \) of length \( \tau - 1 \) and a state \( s \) of \( A \) such that \( \sigma = \sigma'[s]_A \). From the induction hypothesis, we have that \( \sigma' \) is a trace of \( B_j \) and hence of \( B_j^{\tau-1} \) for all \( 1 \leq j \leq m \). Therefore \( \sigma \) is a trace of \( B_j^{\tau-1} \) for all \( 1 \leq j \leq m \).

We know that \( C_k^{\tau} \| D_k^{\tau} \| E_k^{\tau-1} \preceq B_k^{\tau} \). Since \( \sigma \) is a trace of \( C_k^{\tau} \), \( D_k^{\tau} \) and \( E_k^{\tau-1} \), we get that \( \sigma \) is a trace of \( B_k^{\tau} \) and hence a trace of \( B_k \). 

**Example 3.1** Consider a simple communication protocol modeled by two constraints — a sender and a receiver of messages. To simplify the example, we have abstracted out the messages being transmitted and focus only on the signaling between the two. The specification of the protocol is given by \( \text{SndSpec} \| \text{RcvSpec} \), where \( \text{SndSpec} \) and \( \text{RcvSpec} \) are transition constraints shown in Figure 3.1. The \( \text{SndSpec} \) constraint has observable variables \text{ack} \ and \text{snd} \, and a private variable \text{wait}. When a message is produced during some round, the variable \text{snd} \ is set to \text{true} \, and reset to \text{false} \ in the next round. Further, the variable \text{wait} \ is set to \text{true} \, to denote that \( \text{SndSpec} \) is waiting for an \text{ack}. When \text{wait} \ is \text{true} \ and \( \text{SndSpec} \) receives an \text{ack}, \text{wait} \ is reset to \text{false}. The \( \text{RcvSpec} \) constraint has observable variables \text{snd} \ and \text{ack}, \, and a private variable \text{pending}. When \( \text{RcvSpec} \) receives a message (indicated by \text{snd} \ being \text{true} \), it records the event by setting \text{pending} \ to \text{true}. It then waits for an arbitrary amount of time, and then resets \text{wait} \ to \text{false}, simultaneously sending an \text{ack}. The implementation of the protocol is given by \( \text{SndImpl} \| \text{RcvImpl} \), where \( \text{SndImpl} \) and \( \text{RcvImpl} \) are transition constraints shown in Figure 3.2. The constraint \( \text{SndImpl} \) differs from \( \text{SndSpec} \) in its response to an incorrect \text{ack}. If \( \text{SndImpl} \) receives an \text{ack} \ when it is not waiting for one, it just goes into an error state, and behaves nondeterministically from then on. In a similar situation, \( \text{SndSpec} \) just ignores the offending \text{ack}. Thus \( \text{SndImpl} \not\models \text{SndSpec} \). Similarly, \( \text{RcvImpl} \) differs from \( \text{RcvSpec} \), if it receives a message when \text{pending} \ is \text{true}. Consequently \( \text{RcvImpl} \not\models \text{RcvSpec} \). However, we can apply the assume-guarantee rule in Theorem 3.3 in
constraint $\text{SndSpec}$
observable $\text{snd}, \text{ack}: \text{bool}$
private $\text{wait}: \text{bool}$
init
$\neg \text{wait}' \land \neg \text{snd}'$
update
$(\neg \text{wait} \lor (\text{wait} \land \text{ack} \land \neg \text{wait}')) \land (\neg \text{wait} \land \text{wait}') = \text{snd}'$
endconstraint

constraint $\text{RevSpec}$
observable $\text{ack}, \text{snd}: \text{bool}$
private $\text{pending}: \text{bool}$
init
$\neg \text{pending}' \land \neg \text{ack}'$
update
$(\neg \text{pending} \land \text{snd} \land \text{pending}') \lor \text{pending}) \land (\text{pending} \land \neg \text{pending}') = \text{ack}'$
endconstraint

Figure 3.1: Specification of sender/receiver
constraint $SndImpl$
  observable $snd, ack: bool$
  private $wait, error: bool$

init
  $\neg wait' \land \neg error' \land \neg snd'$

update
  $(\neg wait \land error' = error) \lor$
  $(wait \land ack \land \neg wait' \land error' = error) \lor$
  $(\neg wait \land ack \land error') \lor$
  $(error \land error' = error))$
  \land
  $(\neg wait \land wait') = snd'$
endconstraint

constraint $RevImpl$
  observable $ack, snd: bool$
  private $pending, error: bool$

init
  $\neg pending' \land error' \land \neg ack'$

update
  $(\neg pending \land snd \land pending' \land error' = error) \lor$
  $(pending \land error' = error) \lor$
  $(pending \land snd \land error'))$
  \land
  $(pending \land \neg pending' = ack') \lor error)$
endconstraint

Figure 3.2: Implementation of sender/receiver
the following way. Let $<$ be the empty partial order on the set $SndSpec$, $RcvSpec$. Then, we can show that $SndImpl^{\tau} || RcvSpec^{\tau-1} \leq SndSpec^{\tau}$ and $RcvImpl^{\tau} || SndSpec^{\tau-1} \leq RcvSpec^{\tau}$ for all $\tau$. Thus we conclude that $SndImpl || RcvImpl \preceq SndSpec || RcvSpec$.

Transition constraints, though simple, are not convenient for describing complex systems because they do not differentiate between input and output variables and they cannot be simulated. In the next section, we describe a special type of transition constraints called reactive modules that address this issue. We will then use Theorem 3.3 to derive a similar assume-guarantee rule for reactive modules and describe our verification methodology based on reactive modules.

### 3.2 Reactive modules

Reactive modules [AH96] are a special class of transition constraints with two special properties: (1) differentiation between inputs and outputs, and (2) executable non-blocking semantics. A reactive system is invariably described as a composition of smaller systems that communicate with other through inputs and outputs. Typically a component does not control its inputs but depending on the received input makes internal decisions that may result in different outputs. For expressing certain kinds of properties, it is necessary to be able to make the distinction between the uncontrollable input nondeterminism and the controllable internal nondeterminism. Reactive modules provide this distinction by partitioning the observable variables into external variables and interface variables. We denote the interface and external variables of a module $P$ by $Intf(P)$ and $ExtI(P)$ respectively. External variables are updated by the environment and can be read by the module, and interface variables are updated by the module and can be read by the environment. The interface and private variables of a module are called controlled variables. During the design phase of a complex reactive it is very important to be able to simulate or run the design on test inputs in order to get intuition into its behavior. Reactive modules provide simulation capability through their executable semantics.

The state of a reactive module changes again in a sequence of rounds. For the external variables, the values in the initial and update rounds are left unspecified (i.e., chosen nondeterministically). For the controlled variables, the values in the initial and update rounds are specified by (possibly nondeterministic) guarded commands. In each update round, the new value of a controlled variable may depend on the (latched) values
of some variables from the previous round. In addition, in each round, the initial (or new) value of a controlled variable may depend on the initial (or new) values of some other variables from the same round; such a dependency between the values of variables within a single round is called an \emph{await dependency}. In order to avoid inconsistent specifications, the await dependencies must be acyclic. In reactive modules, the acyclicity restriction is enforced statically, by partitioning the controlled variables into \emph{atoms} that can be ordered such that in each round, the initial (or new) values for all variables of an atom can be determined simultaneously from the initial (or new) values of the external variables and the variables of earlier atoms.

Each round, therefore, consists of several subrounds—one for the external variables, and one per atom. Each atom has an \emph{initial command}, which specifies the possible initial values for the variables of the atom, and an \emph{update command}, which specifies the possible new values for the variables of the atom within each update round. In the update command, unprimed occurrences of variables refer to the latched values from the previous round; in both the initial and update commands, primed occurrences of variables refer to the initial (or new) values from the same round.

\textbf{Example 3.2} Consider the simple instruction set architecture defined by the reactive module \textit{ISA} of Figure 3.3. The module \textit{ISA} has five external variables (inputs)—the operation \textit{op}, the immediate operand \textit{inp}, the source registers \textit{src1} and \textit{src2}, and the destination register \textit{dest}. There are two interface variables (outputs)—the value \textit{out} of a \textit{STORE} instruction, and a boolean variable \textit{stall}, which indicates if the current inputs have been accepted. If the value of \textit{stall} is \textit{true} in a round, then no instruction is processed in that round, and the environment is supposed to produce the same instruction again in the next round. Finally, there is one private variable—the register file \textit{isaRegFile}.

A round of the module \textit{ISA} consists of four subrounds. In the first subround of each update round, the environment chooses an operation, operands, and a destination, by assigning values to the external variables. In the second subround, the atom \textit{ISASTall} decides nondeterministically if the current inputs are processed, by setting \textit{stall} to \textit{true} or \textit{false}. The third subround belongs to the atom \textit{ISARegFile}. If the updated value of \textit{stall} is \textit{false}, then the current instruction is processed appropriately. If the operation is \textit{AND} or \textit{OR}, it is performed on the source registers and the result is placed into the destination register. If the operation is \textit{LOAD}, the immediate operand is placed into the destination register. The fourth subround belongs to the atom \textit{ISAOOut}. If the updated value of \textit{stall}
module ISA
  
  external op, inp, src1, src2, dest

interface out, stall

private isaRegFile

atom ISAS Stall controls stall
  init update
    [] true → stall' := nondet

atom ISARegFile controls isaRegFile
  init
    [] true → ∀ i. isaRegFile'[i] := 0
  update
    [] ¬stall' ∧ op' = LOAD → isaRegFile'[dest'] := inp'
    [] ¬stall' ∧ op' = AND → isaRegFile'[dest'] := isaRegFile[src1'] ∧ isaRegFile[src2']
    [] ¬stall' ∧ op' = OR → isaRegFile'[dest'] := isaRegFile[src1'] ∨ isaRegFile[src2']

atom ISAOut controls out
  init update
    [] ¬stall' ∧ op' = STORE → out' := isaRegFile[dest']

Figure 3.3: Instruction set architecture

is \textit{false} and the current operation is \textit{STORE}, then \textit{out} is updated to the contents of the destination register from the previous round. Since both atoms \textit{ISARegFile} and \textit{ISAOut} wait, in each update round, for the new value of \textit{stall}, they must be executed \textit{after} the atom \textit{ISAS Stall}, which produces the new value of \textit{stall}. However, there are no await dependencies between the atoms \textit{ISARegFile} and \textit{ISAOut}, and therefore the third and fourth subroutines of each update round can be interchanged.

Let \( P \) be a module. It can be interpreted as a transition constraint \( C(P) \), which has the same set of traces as \( P \). The observable variables of \( C(P) \) are the observable variables of \( P \), and the private variables of \( C(P) \) are the private variables of \( P \). The initial and update predicates of \( C(P) \) can be derived from the initial and update commands of \( P \), respectively. The initial predicate of \( C(P) \) is is defined to be \textit{true} in states \( s \) such that \( s \) can result from the initial round of \( P \). In similar spirit, the update predicate of \( C(P) \) is defined to be true in a state pair \( (s, t) \) such that \( P \) can start with state \( s \) and change to state \( t \) by the execution of an update round. The \textit{awaits} relation on the variables of \( P \) is denoted by \( \succ_P \) and defined
as $y \triangleright_P x$ iff $y$ is controlled by an atom that awaits $x$. Let $Y$ be a subset of the set of variables of $P$. Then $Y$ is \textit{awaits-closed} in $P$ if for all $y \in Y$ and for all $x \in \text{Var}(P)$, we have that (1) if $y \triangleright_P x$ then $x \in Y$, and (2) if $x$ is controlled by the same atom as $y$ then $x \in Y$. The non-blocking semantics of modules gives us that a module is non-blocking for an awaits-closed subset of its set of variables.

\textbf{Fact 3.1} Let $P$ be a module and $Y \subseteq \text{Var}(P)$ be awaits-closed in $P$. Then $P$ is non-blocking for $Y$.

The module $Q$ is \textit{refinable} by module $P$ if (1) every interface variable of $Q$ is an interface variable of $P$, and (2) every external variable of $Q$ is an observable variable of $P$. The module $P$ \textit{refines} the module $Q$, written $P \preceq Q$, if (1) $Q$ is refinable by $P$, and (2) for every trajectory $\bar{s}$ of $P$, the projection $[\bar{s}]_Q$ is a trace of $Q$.

\textbf{Fact 3.2} Let $P$ and $Q$ be modules such that $Q$ is refinable by $P$. Then $C(Q)$ is refinable by $C(P)$ and $P \preceq Q$ iff $C(P) \preceq C(Q)$.

\textbf{Parallel Composition.} The rules for compatibility of modules are more stringent than in the case of transition constraints. Two modules $P$ and $Q$ are \textit{compatible} if (1) the controlled variables of $P$ and $Q$ are disjoint, and (2) the await dependencies between the variables of $P$ and $Q$ are acyclic. If $P$ and $Q$ are two compatible modules, then the \textit{composition} $P \mid Q$ is the module whose atoms are the union of the atoms from $P$ and $Q$. The interface variables of $P \mid Q$ are the interface variables of $P$ and $Q$, and the private variables of $P \mid Q$ are the private variables of $P$ and $Q$. The external variables of $P \mid Q$ consist of the external variables of $P$ that are not interface variables of $Q$, and the external variables of $Q$ that are not interface variables of $P$.

\textbf{Example 3.3} The module ISA from Figure 3.3 can be seen as the parallel composition of three modules. The module ISASTall has the interface variable \textit{stall}; the module ISARegFile has the external variables \textit{op}, \textit{inp}, \textit{src1}, \textit{src2}, \textit{dest}, and \textit{stall}, and the interface variable \textit{isaRegFile}; the module ISAOut has the external variables \textit{op}, \textit{dest}, \textit{stall}, and \textit{isaRegFile}, and the interface variable \textit{out}. The operation \textbf{hide} makes the interface variable \textit{isaRegFile} private:

\begin{align*}
\text{ISA} &= \textbf{hide} \text{isaRegFile in } ISASTall \mid ISARegFile \mid ISAOut
\end{align*}


module $Opr1$

interface opr1

external stall, pipe1.op, pipe2.op, pipe1.inp, wbReg, regFile, src1

atom $Opr1$ controls opr1

update

[] \lnot\lnot stall' \rightarrow opr1' :=

if src1' = pipe1.dest \land pipe1.op \neq NOP \land pipe1.op \neq STORE

then if pipe1.op = LOAD then pipe1.inp else aluOut'

else if src1' = pipe2.dest \land pipe2.op \neq NOP \land pipe2.op \neq STORE

then wbReg else regFile[src1']

module $Opr2$

interface opr2

external stall, pipe1.op, pipe2.op, pipe1.inp, wbReg, regFile, src2

atom $Opr2$ controls opr2

update

[] \lnot\lnot stall' \rightarrow opr2 :=

if ((src2' = pipe1.dest) \land \lnot(pipe1.op = NOP) \land \lnot(pipe1.op = STORE)) then

if (pipe1.op = LOAD) then pipe1.inp else aluOut' fi

else if ((src2' = pipe2.dest) \land \lnot(pipe2.op = NOP) \land \lnot(pipe2.op = STORE)) then wbReg

else regFile[src2'] fi fi

module $Pipe1$

interface pipe1.op, pipe1.inp, pipe1.dest

external stall, inp, op, dest

atom $Pipe1$ controls pipe1.op, pipe1.dest, pipe1.inp

init

[] \text{true} \rightarrow pipe1.op' := NOP

update

[] \text{true} \rightarrow pipe1.op' := \text{if stall'} \text{then NOP else op'};

pipe1.dest' := dest'; pipe1.inp' := inp'

Figure 3.4: Pipeline stage 1
module Pipe2

interface pipe2.op, pipe2.dest, wbReg, aluOut

external pipe1.op, pipe1.inp, pipe1.dest, opr1, opr2

atom ALU controls aluOut

update
\[\text{pipe1.op} = \text{AND} \rightarrow \text{aluOut}' := \text{opr1} \land \text{opr2}\]
\[\text{pipe1.op} = \text{OR} \rightarrow \text{aluOut}' := \text{opr1} \lor \text{opr2}\]

atom Pipe2 controls pipe2.op, pipe2.dest

init
\[\text{true} \rightarrow \text{pipe2.op}' := \text{NOP}\]

update
\[\text{true} \rightarrow \text{pipe2.op}' := \text{pipe1.op}, \text{pipe2.dest}' := \text{pipe1.dest}\]

atom WbReg controls wbReg

update
\[\text{pipe1.op} = \text{AND} \lor \text{pipe1.op} = \text{OR} \rightarrow \text{wbReg}' := \text{aluOut}'\]
\[\text{pipe1.op} = \text{LOAD} \rightarrow \text{wbReg}' := \text{pipe1.inp}\]

module RegFile

interface regFile

external pipe2.op, pipe2.dest, wbReg, aluOut

atom RegFile controls regFile

init
\[\text{true} \rightarrow \text{forall i do regFile}'[i] := 0\]

update
\[\text{pipe2.op} = \text{AND} \lor \text{pipe2.op} = \text{OR} \lor \text{pipe2.op} = \text{LOAD} \rightarrow\]
\[\text{forall i do regFile}'[i] := \text{if pipe2.dest = i then wbReg else regFile}[i]\]

Figure 3.5: Pipeline stages 2 and 3
module PipeOut

interface out

external op, regFile, dest

atom Out controls out

update

\[ \neg \text{stall}' \land op' = \text{STORE} \rightarrow \text{out}' := \text{regFile}[\text{dest}'] \]

module Stall

interface stall

external op, dest, pipe1.\text{op}, pipe1.\text{dest}, pipe2.\text{op}, pipe2.\text{dest}

atom Stall controls stall

update

\[ \[ \text{op}' = \text{STORE} \land \text{pipe1.\text{op}} \neq \text{NOP} \land \text{pipe1.\text{op}} \neq \text{STORE} \land \text{dest}' = \text{pipe1.\text{dest}} \rightarrow \\
\text{stall}' := \text{true} \]

\[ \[ \text{op}' = \text{STORE} \land \text{pipe2.\text{op}} \neq \text{NOP} \land \text{pipe2.\text{op}} \neq \text{STORE} \land \text{dest}' = \text{pipe2.\text{dest}} \rightarrow \\
\text{stall}' := \text{true} \]

\[ \text{default} \rightarrow \text{stall}' := \text{false} \]

Figure 3.6: Pipeline output and stall

Given two compatible modules $P$ and $Q$, it is seen that one gets the same traces by (1) viewing $P$ and $Q$ as transition constraints and interpreting $P\|Q$ as composition of transition constraints, and (2) first viewing $P\|Q$ as composition of modules and then interpreting the result as a transition constraint.

**Fact 3.3** For any two compatible modules $P$ and $Q$, the constraints $C(P)$ and $C(Q)$ are compatible and $C(P\|Q) = C(P)\|C(Q)$.

**Example 3.4** Consider the three-stage pipeline defined by the reactive module $\text{PIPELINE}$ shown in Figures 3.4, 3.5 and 3.6. In the first stage of the pipeline, the operands are fetched; in the second stage, the operations are performed; in the third stage, the result is written into the register file. The $\text{PIPELINE}$ module is the parallel composition of seven modules. The first stage consists of the modules $\text{Pipe1}$, $\text{Opr1}$, and $\text{Opr2}$. Forwarding logic in $\text{Opr1}$ and $\text{Opr2}$ ensures that correct values are given to the second stage, even if the value in question has not yet been written into the register file. The second stage consists of the module
Pipe2, which has an ALU atom that processes arithmetic operations using the operands from the first stage and writes the results into a write-back register called \( \text{wbReg} \). The third stage is consists of the module \( \text{RegFile} \), which copies \( \text{wbReg} \) into the appropriate register. The \( \text{PipeOut} \) module outputs a register value in response to a \texttt{STORE} instruction. The \texttt{Stall} module controls the \texttt{stall} signal, which is set to \textit{true} whenever a \texttt{STORE} instruction cannot be accepted due to data dependencies.

Our goal is to show that \texttt{PIPELINE} is a correct implementation of the instruction set architecture \texttt{ISA}. This is the case if every sequence of instructions given to \texttt{PIPELINE} produces a sequence of outputs (and stalls) that is permitted by \texttt{ISA}. The module \texttt{ISA} is refinable by \texttt{PIPELINE}, so it remains to be shown that every trace of \texttt{PIPELINE} is a trace of \texttt{ISA}.

**Theorem 3.4** Let \( P \) and \( Q \) be two modules such that \( Q \) is refinable by \( P \). Let \( W \) be a module such that (1) \( W \) is compatible with \( P \), and (2) the interface variables of \( W \) include the private variables of \( Q \), and are disjoint from the external variables of \( P \). Then \( Q^u \) is projection refinable by \( P\parallel W \), and if \( P\parallel W \preceq Q^u \) then \( P \preceq Q \).

**Proof:** Since \( Q \) is refinable by \( P \) and the interface variables of \( W \) include the private variables of \( Q \), we get that every interface variable of \( Q^u \) is an interface variable of \( P\parallel W \) and every external variable of \( Q^u \) is an observable variable of \( P\parallel W \). Since \( Q^u \) does not have any private variables, we have that \( Q^u \) is projection refinable by \( P\parallel W \). Consider the transition constraints \( C(P) \), \( C(Q) \) and \( C(W) \).

1. \( C(W) \) is compatible with \( C(P) \).

2. \( \text{Priv}(C(Q)) \subseteq \text{Obs}(C(W)) \).

3. Since \( \text{Obs}(C(W)) \cap \text{Obs}(C(P)) \) is contained in the set of external variables of \( W \), we have that \( W \) is non-blocking on \( \text{Obs}(C(W)) \cap \text{Obs}(C(P)) \).

Suppose \( P\parallel W \preceq Q^u \). Then \( C(P)\parallel C(W) \preceq C(Q)^u \). From Theorem 3.4, we get that \( C(P) \preceq C(W) \preceq C(Q) \). Therefore \( P \preceq Q \).

The state space of a module may be exponential in the size of the module description. Consequently, even checking projection refinement may not be feasible. However, typically both the implementation \( P \) and the specification \( Q \) consist of the parallel composition of several modules, in which case it may be possible to reduce the problem of checking
CHAPTER 3. ASSUME-GUARANTEE REASONING

if \( P \preceq Q \) to several subproblems that involve smaller state spaces. The assume-guarantee rule for reactive modules allows us to conclude \( P \preceq Q \) as long as each component of the specification \( Q \) is refined by the corresponding components of the implementation \( P \) within a suitable environment. In the following we assume that the refinement problem has been reduced to a projection refinement problem by supplying a suitable witness module and therefore the specification does not have any private variables. We prove a general version of the assume-guarantee rule for reactive modules [AH96].

**Lemma 3.5** Let \( P \) and \( Q \) be modules such that \( Q \) has no private variables and every interface variable of \( Q \) is an interface variable of \( P \). Let \( \mathcal{R} \) be a set of modules. Suppose the composition of \( P \) and all the modules in \( \mathcal{R} \) exists and is denoted by \( S \), and \( S \preceq Q \). Let \( Z = \{ R \in \mathcal{R} \mid \exists y \in \text{Inf}(Q). \exists z \in \text{Inf}(R). y \succ_{S} z \lor y \succ_{Q} z \} \). Let \( C \) be the composition of all the modules in the set \( Z \) and let \( D \) be the composition of all the modules in the set \( \mathcal{R} \setminus Z \). Then for all \( \tau \), we have that \( C(P)^{\tau} \parallel C(C)^{\tau} \parallel C(D)^{\tau-1} \preceq C(Q)^{\tau} \).

**Proof:** We have that \( S = P \parallel C \parallel D \). Let \( Y \) be the least subset of \( \text{Var}(S) \) such that  
(1) \( \text{Inf}(Q) \cup \text{Ext}(S) \subseteq Y \),  
(2) \( Y \) is awaits-closed in \( Q \), and  
(3) \( Y \) is awaits-closed in \( S \). From the definition of \( C \), we have that \( Y \subseteq \text{Var}(P \parallel C) \cup \text{Ext}(S) \). We prove the theorem by induction on \( \tau \).

**Base step.** We have to show that \( C(P)^{0} \parallel C(C)^{0} \parallel C(D)^{-1} \preceq C(Q)^{0} \). Since every trace is a trace of \( C(D)^{-1} \), we need to show \( C(P)^{0} \parallel C(C)^{0} \preceq C(Q)^{0} \). Consider a trace \( \sigma = s_{0} \) of length one of \( C(P) \parallel C(C) \). Since \( s_{0} \) is an initial state of \( C(P) \parallel C(C) \), we have that \( s_{0} \) is an initial state of \( P \parallel C \). Let \( s'_{0} \) be the restriction of \( s_{0} \) to \( Y \). Since \( Y \subseteq \text{Var}(P \parallel C) \), the valuation \( s'_{0} \) is obtained by executing a subset of atoms of \( P \parallel C \). From Fact 3.1, we can extend \( s'_{0} \) to \( t_{0} \) such that \( t_{0} \) is an initial state of \( P \parallel C \parallel D \). Since \( P \parallel C \parallel D \preceq Q \), we get that \( [t_{0}]_{Q} \) is an initial state of \( Q \). But \( [t_{0}]_{Q} = [s]_{Q} \) because \( \text{Var}(Q) \subseteq Y \). Therefore \( [s]_{Q} \) is an initial state of \( Q \).

**Inductive step.** Suppose \( \tau > 0 \) and \( C(P)^{\tau} \parallel C(C)^{\tau} \parallel C(D)^{\tau-1} \preceq C(Q)^{\tau} \). We will show that \( C(P)^{\tau} \parallel C(C)^{\tau} \parallel C(D)^{\tau-1} \preceq C(Q)^{\tau} \). Consider a trajectory \( \sigma = \sigma . s \) of length \( \tau + 1 \) of \( C(P)^{\tau} \parallel C(C)^{\tau} \parallel C(D)^{\tau-1} \). Then \( \sigma \) is a trajectory of \( C(P) \) and \( C(C) \), and \( \sigma' \) is a trajectory of \( C(D) \). Therefore \( \sigma \) is a trajectory of \( P \) and \( C \), and \( \sigma' \) is a trajectory of \( D \). Let \( s' \) be the restriction of \( s \) to \( Y \). Since \( Y \subseteq \text{Var}(P \parallel C) \cup \text{Ext}(S) \), the valuation \( s' \) is obtained by updating all external variables of \( S \) and executing a subset of atoms of \( S \). From Fact 3.1, we can extend \( s' \) to a state \( t \) of \( S = P \parallel C \parallel D \). Therefore, we get that \( \sigma'. t \) is
a trajectory of $P \parallel C \parallel D$. Since $P \parallel C \parallel D \preceq Q$, we get that $[\sigma^I \ell]_Q$ is a trace of $Q$ and hence of $C(Q)$. But $[\ell]_Y = [s]_Y$. Since $Y$ is an awaits-closed in $Q$ and $\text{Inf}(Q) \subseteq Y$, we have that $[\sigma^I, s]_Q$ is also a trace of $C(Q)$.

**Theorem 3.6 (Assume-guarantee rule for modules) Let $P = P_1 \parallel \cdots \parallel P_n$ and $Q = Q_1 \parallel \cdots \parallel Q_m$ be reactive modules such that $Q$ is projection refinable by $P$. For all $1 \leq i \leq m$, let $\Gamma_i$ be the composition of arbitrary compatible components from $P$ and $Q$ with the exception of $Q_i$. Suppose (1) $\Gamma_i \preceq Q_i$ for all $1 \leq i \leq m$, and (2) the relation $\succ_P \cup \succ_Q$ is acyclic. Then $P \preceq Q$.

**Proof:** Consider the set of atoms in $Q$. For each atom we can construct a module without private variables containing that single atom. The interface variables of this module are the controlled variables of the atom and the external variables are those read and awaited variables that are not controlled by the atom. Clearly, we can express $Q$ as the composition $A_1 \parallel A_2 \parallel \cdots \parallel A_l$ of such modules. For all $1 \leq j \leq l$, let $\Gamma_j'$ be equal to $\Gamma_i$ if $A_j$ was built from an atom in $Q_i$. For all $1 \leq j \leq l$ and $1 \leq i \leq n$, if $A_j$ was built from an atom in $P_i$ then $\Gamma_j' = \Gamma_i \preceq A_j$. Let $\tilde{\supseteq}$ denote the transitive closure of $\succ_P \cup \succ_Q$. Since $\succ_P \cup \succ_Q$ is acyclic, the relation $\tilde{\supseteq}$ is a partial order. Let $\prec$ be the relation on the set $\{A_1, A_2, \ldots, A_l\}$ such that $A_k \prec A_j$ iff there is a $u \in \text{Inf}(A_k)$ and a $v \in \text{Inf}(A_j)$ such that $v \prec u$. Since $\tilde{\supseteq}$ is a partial order, we have that $\prec$ is a partial order. For each $A_j$ let $Z(A_j) = \{A_k | A_k \prec A_j\}$ and let $Z^C(A_j) = \{A_i | A_i \not\prec A_j\}$. Consider the proof obligation $\Gamma_j' \preceq A_j$. We denote the components of $\Gamma_j'$ by $B_j$. We partition $B_j$ into three parts in the following way. Let $C_j$ be the components from the set $\{P_1, \ldots, P_n\}$, that is, the set $C_j = B_j \cap \{P_1, \ldots, P_n\}$. Let $C_j$ be the composition of the modules in $C_j$. Let $D_j = \{A_k \in B_j | \exists y \in \text{Inf}(A_j), \exists x \in \text{Inf}(A_k), y \succ_{\Gamma_j} x \lor y \succ_{\Gamma_j} x\}$. Let $E_j = B_j \setminus D_j$. Let $D_j$ and $E_j$ be the composition of modules from $C_j$ and $D_j$ respectively. Thus we have that $\Gamma_j' = C_j \parallel D_j \parallel E_j \preceq A_j$. We know that $A_j$ does not have any private variables and every interface variable of $A_j$ is also an interface variable of $C_j$. Therefore, we can apply Lemma 3.5 to conclude that $C(C_j)^\tau \parallel C(D_j)^\tau \parallel C(E_j)^\tau^{-1} \preceq C(A_j)^\tau$ for all $\tau$. Moreover, for all $j$ we have that $\succ_{\tau_j} \subseteq \tilde{\supseteq}$ and $\succ_{\tau_j} \subseteq \tilde{\supseteq}$. From the definition of $\prec$ we get that $D_j \subseteq Z(A_j)$. Therefore, from Theorem 3.3 we get that $C(P_1) \parallel C(P_2) \parallel \cdots \parallel C(P_n) \preceq C(A_1) \parallel C(A_2) \parallel C(A_3) \parallel C(A_4)$, or $C(P_1) \parallel \cdots \parallel C(P_n) \preceq C(A_1) \parallel \cdots \parallel C(A_l)$. Therefore $P_1 \parallel \cdots \parallel P_n \preceq A_1 \parallel \cdots \parallel A_l = Q_1 \parallel \cdots \parallel Q_m$. 

**Example 3.5** We illustrate our assume-guarantee rule through the same communication protocol that was discussed in Example 3.1. We now write the specification and implement-
module SndxSpec

external ack: bool

interface snd: bool

private wait: bool

atom controls wait reads ack, wait

init
\[ \text{true} \rightarrow \text{wait}' := \text{false} \]

update
\[ \neg \text{wait} \rightarrow \text{wait}' := \text{true} \]
\[ \text{wait} \land \neg \text{ack} \rightarrow \text{wait}' := \text{false} \]
\[ \neg \text{wait} \rightarrow \]

endatom

atom controls snd reads wait awaits wait

init
\[ \text{true} \rightarrow \text{snd}' := \text{false} \]

update
\[ \neg \text{wait} \land \neg \text{wait}' \rightarrow \text{snd}' := \text{true} \]
\[ \text{default} \rightarrow \text{snd}' := \text{false} \]

endatom

endmodule

Figure 3.7: Specification of Sender
module $\texttt{RecSpec}$

interface $\texttt{ack: bool}$

external $\texttt{snd: bool}$

private $\texttt{pending: bool}$

atom controls $\texttt{pending}$ reads $\texttt{pending, snd}$

init
\[
\[\text{true} \rightarrow \text{pending}' := \text{false}
\]

update
\[
\[\neg \text{pending} \land \text{snd} \rightarrow \text{pending}' := \text{true} \\
\[\text{pending} \rightarrow \text{pending}' := \text{false} \\
\[\text{pending} \rightarrow
\]

endatom

atom controls $\texttt{ack}$ reads $\texttt{pending}$ awaits $\texttt{pending}$

init
\[
\[\text{true} \rightarrow \text{ack}' := \text{false}
\]

update
\[
\[\text{pending} \land \neg \text{pending}' \rightarrow \text{ack}' := \text{true} \\
\[\text{default} \rightarrow \text{ack}' := \text{false}
\]

endatom

endmodule

Figure 3.8: Specification of Receiver
module SndImpl

external ack: bool

interface wait: bool

private wait, error: bool

atom controls wait, error reads ack, wait, error

init
[] true \rightarrow wait' := false; error' := false

update
[] \neg wait \rightarrow wait' := true
[] wait \land ack \rightarrow wait' := false
[] \neg wait \rightarrow
[] \neg wait \land ack \rightarrow error' := true
[] error \rightarrow wait' := nondet

dematom

atom controls snd reads wait awaits wait

init
[] true \rightarrow snd' := false

update
[] \neg wait \land wait' \rightarrow snd' := true
[] default \rightarrow snd' := false

dematom

demodule

Figure 3.9: Implementation of the sender
module RcvImpl

interface ack:bool

external snd:bool

private pending, error:bool

atom controls pending, error reads pending, snd

init
\[ true \rightarrow \text{pending'} := \text{false}; \text{error'} := \text{false} \]

update
\[ -\text{pending} \land \text{snd} \rightarrow \text{pending'} := \text{true} \]
\[ \text{pending} \rightarrow \text{pending'} := \text{false} \]
\[ \text{pending} \land \text{snd} \rightarrow \text{error'} := \text{true} \]
\[ \text{pending} \rightarrow \]

endatom

atom controls ack reads pending, error awaits pending

init
\[ true \rightarrow \text{ack'} := \text{false} \]

update
\[ \text{pending} \land -\text{pending'} \rightarrow \text{ack'} := \text{true} \]
\[ \text{error'} \rightarrow \text{ack'} := \text{true} \]
\[ \text{default} \rightarrow \text{ack'} := \text{false} \]

endatom

endmodule

Figure 3.10: Implementation of Receiver
CHAPTER 3. ASSUME-GUARANTEE REASONING

...tation in a more structured way using reactive modules. The specification of the protocol is given by $\text{SndSpec} \parallel \text{RcvSpec}$, where \text{SndSpec} and \text{RcvSpec} are modules shown in Figure 3.7 and Figure 3.8 respectively. The \text{SndSpec} module partitions the observable variables of the constraint \text{SndSpec} in Figure 3.1 into an external variable \text{ack} and an interface variable \text{snd}. Similarly, the \text{RcvSpec} module partitions the observable variables of the constraint \text{RcvSpec} in Figure 3.1 into an external variable \text{snd} and an interface variable \text{ack}. The implementation of the protocol is given by $\text{SndImpl} \parallel \text{RcvImpl}$, where \text{SndImpl} and \text{RcvImpl} are modules shown in Figure 3.9 and Figure 3.10 respectively. Again, we have that $\text{SndImpl} \not\leq \text{SndSpec}$ and $\text{RcvImpl} \not\leq \text{RcvSpec}$. However, we can show that $\text{SndImpl} \parallel \text{RcvSpec} \leq \text{SndSpec}$ and $\text{RcvImpl} \parallel \text{SndSpec} \leq \text{RcvSpec}$, and we can conclude from the assume-guarantee rule in Theorem 3.6 that $\text{SndImpl} \parallel \text{RcvImpl} \leq \text{SndSpec} \parallel \text{RcvSpec}$.

We make use of Theorem 3.6 as follows. First we decompose the specification $Q$ into its components $Q_1 \parallel \cdots \parallel Q_n$. Then we find for each component $Q_i$ of the specification a suitable module $\Gamma_i$ (called an obligation module) and check that $\Gamma_i \preceq Q_i$. This is beneficial if the state space of $\Gamma_i$ is smaller than the state space of $P$. The module $\Gamma_i$ is the parallel composition of two kinds of modules—essential modules and constraining modules. The essential modules are chosen from the implementation $P$ so that every interface variable of $Q_i$ is an interface variable of some essential module. There may, however, be some external variables of $Q_i$ that are not observable for the essential modules. In this case, to ensure that $Q_i$ is refinable by $\Gamma_i$, we need to choose constraining modules from either from the implementation $P$ or from the specification $Q$ (other than $Q_i$). Once $Q_i$ is refinable by $\Gamma_i$, if the refinement check $\Gamma_i \preceq Q_i$ goes through, then we are done. Typically, however, the external variables of $\Gamma_i$ need to be constrained in order for the refinement check to go through. Until this is achieved, we must add further constraining modules to $\Gamma_i$.

It is preferable to choose constraining modules from the specification, which is less detailed than the implementation and therefore gives rise to smaller state spaces (in the undesirable limit, if we choose $\Gamma_i = P$, then the proof obligation $\Gamma_i \preceq Q_i$ involves the state space of $P$ and is no simpler than the original proof obligation $P \preceq Q$). Unfortunately, due to lack of detail, the specification often does not supply a suitable choice of constraining modules. According to the following simple property of the refinement relation, however, we can arbitrarily “enrich” the specification by composing it with new modules.

**Theorem 3.7 (Abstraction modules)** For all modules $P$, $Q$, and $A$, if $P \preceq Q \parallel A$ and...
CHAPTER 3. ASSUME-GUARANTEE REASONING

Q is refifiable by P, then \( P \preceq Q \).

So, before applying the assume-guarantee rule, we may add modules to the specification and prove \( P \preceq Q \| A_1 \| \cdots A_k \) instead of \( P \preceq Q \). The new modules \( A_1, \ldots, A_k \) are called abstraction modules, as they usually give high-level descriptions for some implementation components, in order to provide a sufficient supply of constraining modules. In summary, the creativity required from the human verification expert is the construction of suitable abstraction modules, which on one hand, need to be as detailed as required to serve as constraining modules in assume-guarantee reasoning, and on the other hand, should be as abstract as possible to minimize their state spaces.

3.3 Verification of three-stage pipeline

We prove that \( PIPELINE \preceq ISA \) using Theorems 3.1, 3.4, 3.6, and 3.7. We note that \( ISA \) is refifiable by \( PIPELINE \), but not projection refifiable. This is because \( isaRegFile \) in \( ISA \) is a private variable. We claim that the module \( ISARegFile \) is a witness module for \( isaRegFile \). We then use Theorem 3.4 to reduce the proof obligation \( PIPELINE \preceq ISA \) to \( ISARegFile \| PIPELINE \preceq ISA^\prime \). This proof obligation can be expanded in terms of component modules to

\[
ISARegFile \| RegFile \| Opr1 \| Opr2 \| \text{Pipel} \| \text{Pipe2} \| \text{PipeOut} \| \text{Stall} \preceq ISARegFile \| ISAOut \| ISAStall.
\]

Let us start by identifying \( ISAOut \) with \( Q_1 \). We need to find an obligation module \( \Gamma_1 \), such that \( \Gamma_1 \preceq ISAOut \). There is only one interface variable for \( ISAOut \), namely \textit{out}. The component of \( PIPELINE \) that generates \textit{out} is \textit{PipeOut}. Thus \textit{PipeOut} is the only essential module for \( \Gamma_1 \). However, the proof obligation

\[
\Gamma_1 = \text{PipeOut} \preceq Q_1 = ISAOut
\]

fails trivially, because \( ISAOut \) is not refifiable by \textit{PipeOut}. The module \( ISAOut \) has an external variable \( isaRegFile \) that is not present in \textit{PipeOut}. To achieve refinability, we add \( ISARegFile \), the module controlling \( isaRegFile \), to \( \Gamma_1 \) and try to prove

\[
\Gamma_1 = ISARegFile \| PipeOut \preceq Q_1 = ISAOut.
\]
CHAPTER 3. ASSUME-GUARANTEE REASONING

This fails because the input regFile to PipeOut is not constrained. We add RegFile to constrain regFile, but in vain, because the check

$$\Gamma_1 = ISARegFile \parallel RegFile \parallel PipeOut \preceq Q_1 = ISAOut$$

also fails. The reason now is that the inputs to RegFile are not constrained. We add Pipe2 for this purpose, and then Pipe1, Opr1, Opr2, and Stall to constrain the inputs to Pipe2. At last, we are able to prove the proof obligation

$$\Gamma_1 = ISARegFile \parallel RegFile \parallel Pipe1 \parallel Pipe2 \parallel Opr1 \parallel Opr2 \parallel Stall \parallel PipeOut \preceq Q_1 = ISAOut.$$ 

Now, according to Theorem 3.6, the assume-guarantee proof looks as follows:

$$\begin{align*}
ISARegFile \parallel RegFile \parallel Pipe1 \parallel Pipe2 \parallel Opr1 \parallel Opr2 \parallel Stall \parallel PipeOut & \preceq ISAOut \\
ISARegFile & \preceq ISARegFile \\
Stall & \preceq ISASTall \\
ISARegFile \parallel RegFile \parallel Pipe1 \parallel Pipe2 \parallel Opr1 \parallel Opr2 \parallel Stall \parallel PipeOut & \preceq ISAOut \parallel ISARegFile \parallel ISASTall
\end{align*}$$

However, notice that the biggest module on the left side above the line is exactly the same as the module on the left side below the line. Hence, the compositional approach did not yield much advantage.

So let us return to the PIPELINE module with the intent of adding abstraction modules. We will add three abstraction modules—AbsOpr1, AbsOpr2, and AbsRegFile, corresponding to Opr1, Opr2, and RegFile. Notice that whenever the required operand specified by src1 is currently being produced by ALU or is in wbReg, module Opr1 looks ahead and finds it. Otherwise, it gets the operand from the register file in PIPELINE. It is observed that the specification variable isaRegFile[ src1 ] contains the same value that will be produced by the forwarding logic. This observation can be used to write the following abstraction module for Opr1.

```plaintext
module AbsOpr1
  external isaRegFile, src1, stall
  interface opr1
  atom AbsOpr1 controls opr1
  update
    ![−stall' \rightarrow opr1' := isaRegFile[ src1 ]]$
```
CHAPTER 3. ASSUME-GUARANTEE REASONING

Note that the abstraction module leaves the value of opr1 unspecified if stall is true. The implementation module Opr1, on the other hand, specifies a value for opr1 in every round. Such incomplete specification is an essential characteristic of abstraction modules. A similar abstraction module AbsOpr2 can be written for Opr2.

To write an abstraction module for the implementation register file, regFile, observe that the value of regFile in every round must be equal to the value of isaRegFile from two rounds earlier. Thus, the abstraction module for RegFile can be written as AbsRegFile||ISARegFile, where AbsRegFile and ISARegFile are given below.

\[
\text{module ISARegFile}_d
\]
\[
\text{atom} \ ISARegFile_d \text{ controls } isaRegFile_d \\
\text{init} \\
\qquad \left[ \text{true} \rightarrow \text{forall } i \text{ do } isaRegFile_d[i] := 0 \right] \\
\text{update} \\
\qquad \left[ \text{true} \rightarrow \text{forall } i \text{ do } isaRegFile_d'[i] := isaRegFile[i] \right]
\]

\[
\text{module AbsRegFile}
\]
\[
\text{atom} \ AbsRegFile \text{ controls } regFile \\
\text{init} \\
\qquad \left[ \text{true} \rightarrow \text{forall } i \text{ do } regFile'[i] := 0 \right] \\
\text{update} \\
\qquad \left[ \text{true} \rightarrow \text{forall } i \text{ do } regFile'[i] := isaRegFile_d[i] \right]
\]

On composing AbsRegFile and ISARegFile with ISA, we find that the new specification is not projection refinable by ISARegFile||PIPELINE, because of the new specification variable isaRegFile. To regain projection refinability, a witness module needs to be written for the abstraction module isaRegFile, and composed with PIPELINE. A suitable witness is simply the module ISARegFile. After adding the abstraction modules, according to
Theorem 3.6, we obtain the following assume-guarantee proof:

\[
\begin{align*}
\text{PipeOut} & \parallel \text{Pipe1} \parallel \text{Pipe2} \parallel \text{Stall} \parallel \\
\text{AbsRegFile} & \parallel \text{AbsRegFile}_d \parallel \text{ISA RegFile} \\
\text{Opr1} & \parallel \text{AbsOpr2} \parallel \text{Pipe1} \parallel \text{Pipe2} \\
\text{AbsRegFile} & \parallel \text{AbsRegFile}_d \parallel \text{ISA RegFile} \\
\text{Opr2} & \parallel \text{AbsOpr1} \parallel \text{Pipe1} \parallel \text{Pipe2} \\
\text{AbsRegFile} & \parallel \text{AbsRegFile}_d \parallel \text{ISA RegFile} \\
\text{RegFile} & \parallel \text{AbsRegFile}_d \parallel \text{ISA RegFile} \parallel \text{Stall} \\
\text{Stall} & \parallel \text{ISA Stall} \\
\text{ISA RegFile} & \parallel \text{ISA RegFile} \\
\end{align*}
\]

\[
\begin{align*}
\text{ISA RegFile} & \parallel \text{AbsOpr1} \parallel \text{AbsOpr2} \parallel \text{AbsRegFile} \\
\text{Pipe1} & \parallel \text{Pipe2} \parallel \text{Opr1} \parallel \text{Opr2} \parallel \text{PipeOut} \parallel \text{Stall} \\
\text{ISA RegFile}_d & \parallel \text{ISA Stall} \\
\end{align*}
\]

All proof obligations above the line satisfy projection refinability, and involve smaller state spaces than the conclusion of the proof. Following Theorem 3.1, they can be discharged by a transition-invariant check. Let us now focus on the modules below the line. Notice that the composite module on the left side is \( \text{PIPELINE} \parallel \text{ISA RegFile} \parallel \text{ISA RegFile}_d \), and the composite module on the right side is \( \text{ISA''} \parallel \text{ISA RegFile}_d \parallel \text{AbsOpr1} \parallel \text{AbsOpr2} \parallel \text{AbsRegFile} \). By Theorem 3.7, we can remove \( \text{ISA RegFile}_d \parallel \text{AbsOpr1} \parallel \text{AbsOpr2} \parallel \text{AbsRegFile} \) from the right side to obtain the refinement \( \text{PIPELINE} \parallel \text{ISA RegFile} \parallel \text{ISA RegFile}_d \leq \text{ISA''} \). The module \( \text{ISA RegFile} \parallel \text{ISA RegFile}_d \) is a witness for the refinement \( \text{PIPELINE} \leq \text{ISA} \). Hence, by Theorem 3.4, we conclude that \( \text{PIPELINE} \leq \text{ISA} \).

**Related work.** Most approaches to pipeline verification work off an abstraction of the circuit in which the datapath bitvector variables are modeled as integers and the datapath functions modeled as uninterpreted functions over integers. The approach in [HIKB98, IHB98] uses finite instantiations of integer variables and a reachability algorithm that can handle uninterpreted functions to perform a partial exploration of the control states of the pipeline. The approach in [HIK98] extracts a control token net from a pipeline and performs state exploration on an abstract interpretation of the net. Several researchers have used theorem proving to verify that a pipeline refines an ISA specification. These approaches express the transition relation of the pipeline and the ISA in quantifier free first-order logic.
with uninterpreted functions. They construct an abstraction function that relates a state of the pipeline to a state of the ISA. A decision procedure is then used to verify that the abstraction function is indeed correct. For simple pipelines, the abstraction function can be constructed automatically by flushing the pipeline [BD94], that is, fresh instructions are not allowed to enter the pipeline and the instructions already in are completed. For more complicated pipelines, simple flushing yields huge terms that cannot be handled by term rewriting decision procedures. Therefore, researchers have tried to decompose the proof into smaller lemmas. The problem has been decomposed into two sub-problems by manually constructing a design intermediate in abstraction between the pipeline and the ISA [SH97, SH98a, SH98b], and verifying that the pipeline refines the design which in turn refines the ISA. The “completion functions” approach [HSG98, HSG99] decomposes the problem by treating the effect of each unfinished instruction in the pipeline separately.

### 3.4 Verification of Tomasulo’s algorithm

The Tomasulo algorithm allows processors to execute instructions in data flow order. There could be multiple functional units, each of which could be pipelined independently. Out of order execution of an instruction is allowed, if operands are available. The specification module is the simple Instruction Set Architecture (ISA) module that we saw in Figure 3.3. Recently, there have been a few efforts to verify this algorithm [DP97, McM98]. We present below our use of witness and abstraction modules to verify the algorithm in an assume-guarantee fashion.

The implementation module *TOMASULO* has five main components: Implementation registers, reservation stations, bus, schedulers and a stall generator. Each register in the implementation has three fields: *valid*, *value* and *tag*. If the *valid* bit is true then the *value* field contains the current value of the register, otherwise the register is waiting for its value from the reservation station pointed to by *tag*. The register file is modeled as three arrays — *impRegs.valid*, *impRegs.value* and *impRegs.tag*. Each reservation station has four fields: *valid*, *aVal*, *bVal* and *op*. The *valid* bit indicates if the station is currently being used. The operation to be performed is stored in *op* and the operands are stored in *aVal* and *bVal*. The operand values need not be available when an instruction is allocated to reservation station. Consequently each of *aVal* and *bVal* have three fields, similar to a register file: *valid*, *value* and *tag*. A reservation station is said to be enabled if its *valid* bit
is true and both its operand values are available. The set of reservation stations is modeled as eight arrays — \( \text{st.valid, st.Op, st.aVal.valid, st.aVal.value, st.aVal.tag, st.bVal.valid, st.bVal.value} \) and \( \text{st.bVal.tag} \). Each reservation station can have pipelines of arbitrary depth. We do not model these pipelines. We have a bus that chooses an enabled reservation station, performs its operation and broadcasts its result and tag. The registers and reservation stations snoop this bus and update their values if their tag matches the tag on the bus. There are two schedulers: the first \( \text{allocSt} \) schedules an invalid reservation station (one whose \text{valid} bit is \text{false}) for an incoming instruction. The second \( \text{opSt} \) schedules an enabled reservation station to access the bus. The variable \( \text{stall} \) is set to \text{true} whenever the \text{TOMASULO} model is unable to accept an incoming instruction due to data dependencies.

The module for the \( i \)-th implementation register is given below. The environment for the register consists of instruction inputs, bus values, and the variables \( \text{stall} \) and \( \text{allocSt} \). One such module is created for each of the implementation registers.

```
module IMPREG\[i\]
    external bus.value, bus.tag, bus.valid, op, inp, dest, allocSt, stall
    interface impRegs.valid, impRegs.value, impRegs.tag
    atom IMPREG\[i\] controls impRegs.valid\[i\], impRegs.value\[i\], impRegs.tag\[i\]
        init
            []true \rightarrow impRegs.valid'\[i\] := true; impRegs.value'\[i\] := 0
        update
            []\neg stall' \land (op' = STORE) \land (dest' = i) \rightarrow
                impRegs.valid'\[i\] := true; impRegs.value'\[i\] := inp'
            []\neg stall' \land ((op' = AND) \lor (op' = OR)) \land (dest' = i) \rightarrow
                impRegs.valid'\[i\] := false; impRegs.tag'\[i\] := allocSt'
            []\neg impRegs.valid'\[i\] \land bus.valid \land (impRegs.tag'\[i\] = bus.tag) \rightarrow
                impRegs.valid'\[i\] := true; impRegs.value'\[i\] := bus.value
```

Each reservation station consists of four modules — one that controls the valid bit \( \text{valid} \), one that controls opcode \( \text{op} \), and one each for the two operands \( \text{aVal} \) and \( \text{bVal} \). Their descriptions are given below.

```
module STATION VALID\[s\]
    interface st.valid\[s\]
    external stall, allocSt, bus.valid, bus.tag, op
    atom STATION VALID\[s\] controls st.valid\[s\]
```
init
\[ \square \text{true} \rightarrow \text{st.valid}'[s] := \text{false} \]

update
\[ \square \neg \text{stall}' \land (\text{allocSt}' = s) \land ((\text{op}' = \text{AND}) \lor (\text{op}' = \text{OR})) \rightarrow \text{st.valid}'[s] := \text{true} \]
\[ \square \text{bus.valid}' \land \text{st.valid}[s] \land (\text{bus.tag}' = s) \rightarrow \text{st.valid}'[s] := \text{false} \]

module \textit{STATION\_OP}[s]

interface \textit{st.Op}[s]

external stall, op, allocSt

atom \textit{STATION\_OP}[s] controls \textit{st.Op}[s]

update
\[ \square \neg \text{stall}' \land (\text{allocSt}' = s) \land ((\text{op}' = \text{AND}) \lor (\text{op}' = \text{OR})) \rightarrow \text{st.Op}'[s] := \text{op}' \]

module \textit{STATION\_AVAL}[s]

interface \textit{st.a Val.valid}[s], \textit{st.a Val.value}[s], \textit{st.a Val.tag}[s]

external \textit{impRegs.value}, \textit{impRegs.tag}, \textit{impRegs.valid}, \textit{bus.valid}, \textit{bus.tag}, \textit{bus.value}, \textit{st.valid}[s], \textit{st.all}, \textit{src1 op}, \textit{allocSt}

atom \textit{STATION\_AVAL}[s] controls \textit{st.a Val.valid}[s], \textit{st.a Val.value}[s], \textit{st.a Val.tag}[s]

init
\[ \square \text{true} \rightarrow \text{st.a Val.valid}'[s] := \text{false} \]

update
\[ \square \neg \text{stall}' \land (\text{allocSt}' = s) \land ((\text{op}' = \text{AND}) \lor (\text{op}' = \text{OR})) \rightarrow \]
\[ \text{st.a Val.valid}'[s] := \text{impRegs.valid}'[s]; \text{st.a Val.value}'[s] := \text{impRegs.value}'[s]; \]
\[ \text{st.a Val.tag}'[s] := \text{impRegs.tag}'[s] \]
\[ \square \text{bus.valid} \land \text{st.valid}[s] \land \neg \text{st.a Val.valid}'[s] \land (\text{bus.tag} = \text{st.a Val.tag}[s]) \rightarrow \]
\[ \text{st.a Val.valid}'[s] := \text{true}; \text{st.a Val.value}'[s] := \text{bus.value} \]

module \textit{STATION\_BVAL}[s]

interface \textit{st.b Val.valid}[s], \textit{st.b Val.value}[s], \textit{st.b Val.tag}[s]

external \textit{impRegs.value}, \textit{impRegs.tag}, \textit{impRegs.valid}, \textit{bus.valid}, \textit{bus.tag}, \textit{bus.value}, \textit{st.valid}[s], \textit{st.all}, \textit{src2}, \textit{op}, \textit{allocSt}

“Same as \textit{STATION\_AVAL}[s] with \textit{src1} replaced by \textit{src2}”

\[ \textit{STATION}[s] := \textit{STATION\_VALID}[s] || \textit{STATION\_OP}[s] || \textit{STATION\_AVAL}[s] || \textit{STATION\_BVAL}[s] \]

The bus module computes the result of the operation that is scheduled by \textit{opSt} and “broad-
casts” the result and the tag of the reservation station available to the registers and other reservation stations.

**module BUS**

**interface** bus.valid, bus.value, bus.tag

**external** stall, allocSt, st valid, st a Val valid, st a Val value,

st b Val valid, st Op, st b Val value, st Op, opSt

**atom** BUS controls bus valid, bus value, bus tag

**init**

[] true → bus valid' := false

**update**

[] st valid[opSt'] ∧ st a Val valid[opSt'] ∧ st b Val valid[opSt'] ∧ st Op[opSt'] = AND →

bus valid' := true; bus value' := st a Val value[opSt'] ∧ st b Val value[opSt'];

bus tag' := opSt'

[] st valid[opSt'] ∧ st a Val valid[opSt'] ∧ st b Val valid[opSt'] ∧ st Op[opSt'] = OR →

bus valid' := true; bus value' := st a Val value[opSt'] ∨ st b Val value[opSt'];

bus tag' := opSt'

[] default → bus valid' := false

The two scheduler variables opSt and allocSt are set non-deterministically. If we want to be more precise, we could set allocSt to an available reservation station (if one is available) and opSt to a valid reservation station with correct operands (if one exists).

**module ALLOC ST**

**interface** allocSt

**atom** ALLOC ST controls allocSt

**init** update

[] true → allocSt' := nondet

**module OP ST**

**interface** opSt

**atom** OP ST controls opSt

**init** update

[] true → opSt' := nondet

The OUT module processes the LOAD instruction.
module OUT
  interface out
    external impRegs.value, op, dest
    atom OUT controls out
    init
      [] true → out' := 0
    update
      [] (op' = LOAD) → out' := impRegs.value[dest']

The stall module sets stall to true when the execution cannot proceed due to data dependencies or non-availability of reservation stations.

module STALL
  interface stall
    external impRegs.valid, impRegs.tag, st.valid, bus.valid, bus.tag allocSt, op, dest, src1, src2
    atom STALL controls stall
    init
      [] true → stall' := false
    update
      [] st.valid[allocSt'] ∧ (op' = AND) ∨ (op' = OR) → stall' := true
      [] (op' = LOAD) ∧ (¬impRegs.valid[dest']) → stall' := true
      [] ((op' = AND) ∧ (op' = OR)) ∧
        ((¬impRegs.valid[src1'] ∧ bus.valid ∧ (impRegs.tag[src1'] = bus.tag)) ∨
         (¬impRegs.valid[src2'] ∧ bus.valid ∧ (impRegs.tag[src2'] = bus.tag)) ∨
         (¬impRegs.valid[dest'] ∧ bus.valid ∧ (impRegs.tag[dest'] = bus.tag)) →
          stall' := true
      [] (op' = STORE) ∧ ¬impRegs.valid[dest'] ∧ bus.valid ∧ (impRegs.tag[dest'] = bus.tag) →
          stall' := true
      [] default → stall' := false

A Tomasulo module with 4 registers and 4 reservation stations can now be constructed as follows:

TOMASULO := IMPREGS || STATIONS || BUS || STALL || ALLOC_ST || OP_ST || OUT

Our aim is to show that TOMASULO ≤ ISA. To ensure projection comparability, we start by writing a witness module for the ISA registers, and composing the witness
with the \textit{TOMASULO} module. Just as in the proof of the 3-stage pipeline, we will use \textit{ISARegFile} to witness the register file of the specification. From Theorem 3.4, it suffices to show that \textit{TOMASULO}||\textit{ISARegFile} \preceq \textit{ISA}”. An abstraction module can be written for the implementation registers, using the following observation: whenever an implementation register is valid its value must coincide with its counterpart in the ISA.

\textbf{module} $\text{ABS\_REG}[i]$

\textbf{external} bus.value, bus.tag, bus.valid, op, imp, dest, allocSt, stall

\textbf{interface} impRegs.valid, impRegs.value, impRegs.tag

\textbf{atom} $\text{ABS\_REG\_VALID\_TAG}[i]$ \textbf{controls} impRegs.valid[i], impRegs.tag[i]

\textbf{init}
\[ \text{true} \rightarrow \text{impRegs.valid}'[i] := \text{true} \]

\textbf{update}
\[ \neg \text{stall}' \land (\text{op}' = \text{STORE}) \land (\text{dest}' = i) \rightarrow \]
\[ \text{impRegs.valid}'[i] := \text{true} \]
\[ \neg \text{stall}' \land ((\text{op}' = \text{AND}) \lor (\text{op}' = \text{OR})) \land (\text{dest}' = i) \rightarrow \]
\[ \text{impRegs.valid}'[i] := \text{false}; \text{impRegs.tag}'[i] := \text{allocSt}' \]
\[ \neg \text{impRegs.valid}[i] \land \text{bus.valid} \land (\text{impRegs.tag}[i] = \text{bus.tag}) \rightarrow \]
\[ \text{impRegs.valid}'[i] := \text{true} \]

\textbf{atom} $\text{ABS\_REG\_VAL}[i]$ \textbf{controls} impRegs.value[i]

\textbf{init}
\[ \text{true} \rightarrow \text{impRegs.value}'[i] := 0 \]

\textbf{update}
\[ \neg \text{impRegs.valid}[i] \rightarrow \text{impRegs.value}'[i] := \text{nondet} \]
\[ \text{impRegs.valid}[i] \rightarrow \text{impRegs.value}'[i] := \text{isaRegFile}'[i] \]

$\text{ABS\_REGS} := \text{ABS\_REG}[0] || \text{ABS\_REG}[1] || \text{ABS\_REG}[2] || \text{ABS\_REG}[3]$

We also want to add abstraction modules for the reservation stations. To make it easier to write abstraction modules, we define an auxiliary variable called history station for each reservation station. The intent is that a history station contains the correct values of the operands of, and the result eventually produced by, the corresponding reservation station. It can get these values from the register file of \textit{ISA}, since instructions are executed atomically in the \textit{ISA}.

\textbf{module} $\text{HIST\_STN\_AVAL}[s]$

\textbf{interface} histSt.aVal[s]
CHAPTER 3. ASSUME-GUARANTEE REASONING

external isaRegFile, op, src1, allocSt, stall
atom HIST_STN_AVAL[i] controls histSt.aVal'[s]
  init
  [] true → histSt.aVal'[s] := 0
update
  [] ¬stall' ∧ (allocSt' = s) ∧ ((op' = AND) ∨ (op' = OR)) →
    histSt.aVal'[s] := isaRegFile[src1']

module HIST_STN_BVAL[s]
  interface histSt.bVal[s]
    “same as HIST_STN_AVAL[s] with src1 replaced by src2”

module HIST_STN_RESULT[s]
  interface histSt.result[s]
  external isaRegFile, op, dest, allocSt, stall
atom HIST_STN_RESULT[s] controls histSt.result[s]
  init
  [] true → histSt.result'[s] := 0
update
  [] ¬stall' ∧ (allocSt' = s) ∧ ((op' = AND) ∨ (op' = OR)) →
    histSt.result'[s] := isaRegFile[dest']

HIST_STN[s] := HIST_STN_AVAL[s]∥ HIST_STN_BVAL[s]∥ HIST_STN_RESULT[s]

The abstraction module for the reservation stations and the bus module can now be written with the help of the corresponding history stations.

module ABS_STN_AVAL[s]
  interface st.aVal.value[s]
  external st.aVal.valid[s], histSt.aVal[s]
atom ABS_STN_AVAL[s] controls st.aVal.value[s]
  init update
  [] st.valid'[s] ∧ st.aVal.valid'[s] → st.aVal.value'[s] := histSt.aVal'[s]
  [] default → st.aVal.value'[s] := nondet

module ABS_STN_BVAL[s]
interface  st.b.Val.value[s]
external  st.b.Val.valid[s], histSt.b.Val[s]

" same as ABS_STN_AVAL[s] with histSt.a.Val replaced by histSt.b.Val"

ABS_STN[s] := ABS_STN_AVAL[s] || ABS_STN_BVAL[s] || STATION_OP || STATION_VALID[s]

module ABS_BUS_VAL
interface bus.value
external opSt, bus.valid, histSt.result
atom ABS_BUS_VAL controls bus.value

update
[] bus.valid' := histSt.result[opSt']
[] !bus.valid' := nondet

Because of the introduction of history stations in the specification, TOMASULO is no longer projection comparable with ISA. We make them projection comparable by adding the history station modules to the implementation also. The various lemmas into which the proof is decomposed are given below.

\[ ISARegFile \parallel ABS_REGS \parallel STALL \preceq ISAOut \]
\[ HIST_STS \parallel ISARegFile \parallel ABS_STS \preceq ABS_BUS_VAL \]
\[ HIST_STS \parallel ISARegFile \parallel ABS_STS \preceq ABS_REG[0] \]
\[ ABS_BUS_VAL \parallel IMPREG[0] \parallel STALL \preceq ABS_STS \parallel STALL \preceq ABS_STS \parallel STATION[0] \]
\[ HIST_STS \parallel ISARegFile \parallel STALL \preceq ABS_STS \parallel STATION[1] \]
\[ STALL \preceq ISAStall \]
\[ IMPREGS \parallel STATIONS \parallel BUS \preceq ISARegFile \parallel ISAOut \parallel ABS_REGS \]
\[ STALL \parallel ALLOC_ST \parallel OP_ST \preceq ISAStall \parallel HIST_STS \]
\[ ISARegFile \parallel HIST_STS \preceq ABS_STS \parallel ABS_BUS_VAL \]

We found that the decomposed lemmas were much easier to deal with in terms of memory use, than the original proof. All lemmas in the compositional proof were discharged in a few minutes. In Table 3.1, we list the number of boolean history dependent variables
### Table 3.1: Size of lemmas in the proof of TOMASULO

(latches) in the obligation modules for each of the proof obligations into which the original proof is decomposed. A superscript of * means that the corresponding model checking run ran out of memory. The monolithic refinement check ran out of memory, when we used 4 registers and 4 reservation stations. The compositional check was able to complete all the lemmas in this case. When we raised the number of registers and reservation stations to 8 each, we were able to complete all but one lemma of the compositional proof.

**Related work.** Damm and Pnueli [DP97] use theorem proving to show that the Tomásulo’s algorithm refines a nondeterministic generalization of the ISA that executes instructions preserving only the dataflow dependencies among them. Our proof of Tomásulo’s algorithm is inspired by that of McMillan [McM98] that also uses compositional model checking with proof decomposition being done by an assume-guarantee rule.
Chapter 4

Assume-Guarantee Reasoning with Sample

Specifications are typically less detailed than the implementation. For example, the specification of an adder might simply state that the output is the sum of the two inputs, whereas the implementation might be a gate-level adder circuit, which operates at the detail of individual bits. Nonetheless, common notions of correctness require specifications to operate in “lock-step” with the implementation: every possible computation step of the implementation must be matched by an admissible computation step of the specification. If the natural time scale of the specification is less detailed than that of the implementation—for example, if the gate-level adder requires several clock cycles to compute a sum—then the specification often is “slowed down” by stuttering, even for perfectly synchronous designs. A prominent example of this occurs in pipeline verification, where the Instruction Set Architecture (ISA) specification usually is slowed down by introducing a nondeterministic stall signal to stretch its time scale to match that of the pipeline [HQR98, McM98]. Instead of slowing down the specification, we pursue the alternative of “speeding up” the implementation. For this purpose, we use an operator called Sample, which samples the behavior of the implementation at appropriately defined sampling instants. ¹

Our motivation for sampling arose specifically from the attempt of verifying a 64-processor V(ideo) G(raphics) I(mage) chip designed by the Infopad project at the University

¹The Sample operator is similar, but not identical, to the Next operator of Reactive Modules [AH96]: while Next changes the time scale of a module and its environment, Sample does not constrain the environment, which therefore may offer multiple inputs between sampling instances.
of California, Berkeley [STUR98]. There, the specification consists of ISAs for the individual processors and FIFO buffers that abstract the point-to-point communication protocols, which interact subtly with the processor pipelines. Since the implementation contains level-sensitive latches and different parts of the circuit are active at high vs. low phases of the clock, sampling must be used to match the implementation time scale with the specification time scale. While the computational advantages of sampling in state-space exploration have been demonstrated in [AHR98], the VGI is still far beyond the scope of exhaustive search. Hence, we needed to generalize a compositional verification methodology to accommodate the Sample operator.

In Chapter 3, we showed how to make refinement checking scalable by making use of the compositional structure of both implementation and specification, and dividing the verification task at hand into simpler subtasks. A refinement-checking problem of the form $P_1 \parallel P_2 \preceq Q_1 \parallel Q_2$ is decomposed into the two proof obligations $P_1 \parallel Q_2 \preceq Q_1$ and $Q_1 \parallel P_2 \preceq Q_2$ (the apparent circularity in such proofs is resolved by an induction over time). In this chapter, we generalize the assume-guarantee method to accommodate the sampling operator. If implementation and specification operate at the same time scale, witness modules generate values for hidden specification variables at each step. However, if a single macro-step of the specification corresponds to several micro-steps of the implementation, it is necessary to provide witness modules that operate at the micro-step level. The purpose of such a witness is to generate the correct value for the specification signal to be witnessed and to maintain that value until the next sampling instance. Dually, if implementation and specification operate at the same time scale, refinement constraints provide abstract definitions for implementation variables at each step. If one specification step corresponds to several implementation steps, then it no longer suffices for the refinement constraints to supply values for the implementation variables at the rate of the specification —at sampling instances— but additional constraints need to be provided between sampling instances. Providing different refinement constraints at the macro and micro levels enables a separation of concerns: while macro-level constraints (at sampling instances) tend to describe the functional behavior of an implementation variable, micro-level constraints (between sampling instances) tend to describe its timing behavior. This separation of functionality and timing is particularly natural for collections of synchronous blocks that communicate asynchronously.

We develop the theory and methodology to carry out assume-guarantee reasoning when specifications are abstract in both space (fewer variables/components) and time (fewer
observation points). The crux of the theory lies in the ability to distribute the Sample operator over the parallel composition of implementation components using micro-level refinement constraints. The resulting assume-guarantee proof rule produces refinement obligations both at the macro level and at the micro level, which are then discharged by our model checker Mocha [AHM+98]. Working with specifications at an abstract level of temporal granularity is not new. While a processor pipeline takes several steps to execute an instruction, its ISA specification executes an instruction atomically in a single step, and the pipeline state can be related to the ISA state by an abstraction function that uses the “pipeline flushing” operation [BD94]. Clock abstraction on dynamic switch-level circuits [BJ95, KSL95] generates gate-level circuits without clocks to make their verification easier. Temporal abstraction hierarchies [AHR98] have been used for efficient state space exploration. However, we are not aware of any compositional refinement checks between implementations and specifications that operate at different time scales.

We have used this methodology successfully in the verification of VGI, a digital signal processing chip developed by the Infopad group at the University of California at Berkeley. VGI is a very large design with 64 compute processors each containing approximately 800 latches and 30,000 gates. Verification of such a large design is clearly beyond the scope of existing model checkers. In order to handle the proof obligations that are generated by our new assume-guarantee rule, we extended the model checker Mocha with the capability for dealing with the sampling operator in refinement checks. We are not aware of any other model checker that currently offers such a capability. Using the enhanced version of Mocha we discovered several bugs in the VGI design and fixed them. In this process, we found it extremely useful to employ Mocha as a debugging tool that supports the concurrent activities of (re)design and formal (re)verification: design insights would suggest the definition of abstraction modules for model checking, and Mocha would produce error traces that suggest corrections to the design. In this way, design and formal verification become a single activity (“formal design”) that involves similar mental processes, rather than two decoupled activities, one followed by the other with little interaction.

**Motivating example.** We consider a design that computes the Greatest Common Divisor (GCD) of two numbers. We will start with a synchronous specification shown in Figure 4.1(a). Given two inputs $a$ and $b$, the module $GCDsl$ computes the $GCD$ of $a$ and $b$ and places the result in the output $r$. The boolean input $validin$ asserts that the inputs are valid in the current round and the boolean output $validout$ asserts that the output
(a) High level specification

(b) Intermediate specification

Figure 4.1: GCD Specification

(a) Implementation

(b) Implementation with witness modules

Figure 4.2: GCD Implementation
CHAPTER 4. ASSUME-GUARANTEE REASONING WITH SAMPLE

is valid in the current round. Module $GCDsp1$ operates synchronously, with a delay of one round. If inputs $a$ and $b$ are given in the current round, then the output is available at $r$ in the next round.

We refine our specification and add more spatial and temporal detail on how the $GCD$ is computed. We use Euclid’s algorithm to compute the $GCD$:

$$GCD(a, b)$$

\{Given positive non-zero integers $a$ and $b$, compute $GCD(a, b)$ \}

1. if $(a = b)$ return $(a)$;
2. if $((a = 1) \text{ or } (b = 1))$ return $(1)$;
3. $small := \min(a, b)$;
4. $big := \max(a, b)$;
5. return $(GCD(small, \text{big} - small))$;

The resulting refinement $GCDsp2$ shown in Figure 4.1(b) has three modules: $IntS$, $DoneS$, and $CompS$. Given two numbers, the $DoneS$ module decides if the $GCD$ is computed trivially (if the numbers are equal, or one of the numbers is 1). If so, it sends the result, otherwise, it resends the numbers in increasing order. Suppose $small$ and $big$ are sent by the $DoneS$ module. The $CompS$ module responds by sending $small$ and $big - small$. The $IntS$ module takes data inputs from both $CompS$ and the environment and feeds the data to the $DoneS$ module. The modules $IntS$, $DoneS$, and $CompS$ communicate with each other using point-to-point communication links. Valid bits ($valid_1$, $valid_2$ and $valid_3$) are used to validate the presence of meaningful data on these links. For instance, if $IntS$ wants to send two numbers to $DoneS$, it places the numbers in $aout$ and $bout$, and sets $valid_1$ to $true$. Each of these communications is assumed to complete in one round.

While $GCDsp1$ requires only one round to compute the $GCD$, module $GCDsp2$ requires multiple rounds depending on the data inputs. We add an additional variable $inprogress$ in module $GCDsp2$ and set it to $true$ whenever a $GCD$ computation is in progress. Using this variable and the $Sample$ operator in Section 4.1, we will formally state how $GCDsp2$ refines $GCDsp1$.

Our final level of refinement uses a single physical broadcast channel for communication between the modules. Time-division multiplexed access (TDMA) is used to share the channel. Communication in the channel is conducted in units called frames. A frame
is divided, in time, into several time-slots. Each module is allocated one or more time-slots to send data. There is a beacon module $Bc$, that signals the beginning of a frame. Each module has its local counter that is synchronized on the $Bc$ module's signal. Once the frame starts, each module sends data in its allocated time-slots. A valid bit sent on the channel indicates if the data being sent in the current time-slot is valid. The allocation of time-slots to modules is done statically at configuration time, and stays fixed thereafter. Thus, every module knows the identity of the sender in each time-slot. Figure 4.2(a) shows the block diagram of the implementation $Impl$. In our example, a frame is divided into 6 slots. The figure also shows the allocation of time-slots within the frame to individual modules—the first two time-slots are given to the $Intf$ module, the next two to the $Done$ module, and the last two to the $Comp$ module. The $Intf$, $Done$, and $Comp$ modules are intended to have the same functionality as the specification modules $IntfS$, $DoneS$, and $CompS$ from $GCDSp2$. However, while the communication between modules in $GCDSp2$ happens in a single round through point-to-point links, the communication between modules in $Impl$ is through a shared channel, and takes several rounds. Let $sync$ be a variable of the module $Bc$ that is set to $true$ whenever the $Bc$ module sends the synchronizing signal. We will use $sync$ and the $Sample$ operator to relate $Impl$ to $GCDSp2$ in Section 4.1.

4.1 The sample operator

Let $A$ be a transition constraint and $\varphi$ be a predicate on the primed and unprimed observable variables of $A$. We define $B = (Sample \ A \at \varphi)$ to be a transition constraint. The private and observable variables of $B$ are the private and observable variables of $A$. The initial predicate of $B$ is equal to the initial predicate of $A$. The update predicate of $B$ is $true$ at the pair of states $(s, t)$ iff there is a sequence of states $s_0, s_1, \ldots, s_n$, such that (1) $s = s_0$, (2) $t = s_n$, (3) $s_i \rightarrow_A s_{i+1}$ for $i = 0, 1, \ldots, n - 1$, and (4) $\varphi(s_i, s_{i+1})$ is $false$ for $i = 0, 1, \ldots, n - 2$ and $true$ for $i = n - 1$. Informally, $B$ updates from state $s$ to $t$ if there exists a sequence of rounds of $A$ starting at $s$ and ending at $t$, such that the final round satisfies $\varphi$, and none of the intermediate rounds satisfy $\varphi$. Given a trajectory of $A$, we use the term sampling instants to refer to the instants in the trajectory where $\varphi$ is true.

We note that the $Sample$ operator differs from the $Next$ operator of [AH96] in two ways: (1) $Sample$ operates on transition constraints, whereas $Next$ operates on modules, and (2) $Sample$ does not constrain the environment between sampling instants, whereas
Next constrains the environment to not change between sampling instants. In the rest of this paper, whenever we write $B = (\text{Sample } A \text{ at } \varphi)$, we assume that $\varphi$ is a predicate on the primed and unprimed observable variables of $A$.

Example. The module GCDSp1 in the GCD computation example is the high-level specification. The intermediate-level specification and implementation modules are composed as follows:

$$
\begin{align*}
\text{GCDSp2} & = \text{IntfS} \parallel \text{DoneS} \parallel \text{CompS} \\
\text{Impl} & = \text{Bc} \parallel \text{Ch} \parallel \text{Intf} \parallel \text{Done} \parallel \text{Comp}
\end{align*}
$$

We wish to relate the intermediate specification GCDSp2 to the high level specification GCDSp1. Recall that GCDSp1 computes GCD in one round, whereas GCDSp2 takes multiple rounds. Also recall that inprogress is a variable of GCDSp2 that is set to true when GCDSp2 is doing the GCD computation. If we sample the behaviors of GCDSp2 during the instances where inprogress is false, the sampling should conform to the behaviors allowed by GCDSp1. Using the Sample operator, we can express this requirement as:

$$(\text{Sample GCDSp2 at } (\neg \text{inprogress}')) \preceq \text{GCDSp1}$$

We also wish to relate GCDSp2 to the final implementation Impl. Recall that every communication in GCDSp2 happens in a single round, whereas every communication in Impl takes several rounds (as many rounds it takes to transmit a frame) to complete. Recall that the Bc module has a variable sync that is set to true when it sends the synchronization signal. Though Impl and GCDSp2 operate at different time scales, if we consider any trace of Impl and sample only the instants where sync is true, the resulting subsequence should be a trace of GCDSp2. Using the Sample operator, we can express this requirement as:

$$(\text{Sample Impl at sync'}) \preceq \text{GCDSp2}$$

Properties of Sample. The Sample operator has several appealing properties. First, the refinement relation between two transition constraints is maintained by application of Sample, as given by the following theorem.

**Theorem 4.1 (Sampled refinement)** Consider two transition constraints $A$ and $B$ such that $A \preceq B$. If $\varphi$ is a predicate on the observable variables of $B$, then $(\text{Sample } A \text{ at } \varphi) \preceq (\text{Sample } B \text{ at } \varphi)$. 
CHAPTER 4. ASSUME-GUARANTEE REASONING WITH SAMPLE 82

Proof: Consider any trace $\gamma$ of Sample $A$ at $\varphi$, with $\sigma = s_0, s_1, \ldots, s_n$ as the witnessing run. Then there must exist run $\hat{\sigma} = t_0, t_1, \ldots, t_n$ of $A$ such that $\sigma$ is equal to the subsequence of states in $\hat{\sigma}$ obtained by including $t_i$ if $i = 0$ or $\varphi(t_{i-1}, t_i)$. Since $A \preceq B$, there must exist a run $\hat{\sigma}' = t'_0, t'_1, \ldots, t'_n$ of $B$, such that $[\hat{\sigma}]_B = [\hat{\sigma}']_B$. Finally, corresponding to $\hat{\sigma}'$, there exists a run $\sigma'$ of (Sample $B$ at $\varphi$) such that $[\gamma]_B = [\sigma]_B = [\sigma']_B$.

The converse of the above theorem does not hold. It is easy to create modules $A$ and $B$ such that $A \not\preceq B$, but (Sample $A$ at $\varphi$) $\preceq$ (Sample $B$ at $\varphi$). The next theorem asserts the distributivity of Sample with respect to the parallel-composition operator. The following theorem asserts the distributivity of Sample with respect to the parallel-composition operator.

**Theorem 4.2 (Distributivity of sample)** Let $A$ and $B$ be transition constraints and $\varphi$ be a predicate on the observable variables common to $A$ and $B$. Then (Sample $(A \parallel B)$ at $\varphi$) $\preceq$ (Sample $A$ at $\varphi$) $\parallel$ (Sample $B$ at $\varphi$).

Proof: Let $C = (\text{Sample } (A \parallel B) \text{ at } \varphi)$, $D = (\text{Sample } A \text{ at } \varphi)$, and $E = (\text{Sample } B \text{ at } \varphi)$. By definition of Sample and $\parallel$ operators, every initial state of $C$ is an initial state of $D \parallel E$. Suppose $s \rightarrow_C t$. Then we know that there is a sequence of states $s_0, s_1, \ldots, s_n$, such that (1) $s = s_0$, (2) $t = s_n$, (3) $s_i \rightarrow_{A \parallel B} s_{i+1}$ for $i = 0, 1, \ldots, n - 1$, and (4) $\varphi(s_i, s_{i+1})$ is false for $i = 0, 1, \ldots, n - 2$ and true for $i = n - 1$. Since $s_i \rightarrow_{A \parallel B} s_{i+1}$ implies $s_i \rightarrow_A s_{i+1}$ and $s_i \rightarrow_B s_{i+1}$, we have $s \rightarrow_D t$ and $s \rightarrow_E t$ as well. Thus $s \rightarrow_{D \parallel E} t$.

In practice, the traces of (Sample $A$ at $\varphi$) $\parallel$ (Sample $B$ at $\varphi$) form a large superset of the traces of (Sample $(A \parallel B)$ at $\varphi$). It is desirable to constrain the observable variables of $A$ and $B$ while distributing the Sample operator over parallel composition. We can strengthen the above theorem in the presence of suitable transition constraints $T_A$ and $T_B$ on the observable variables of $A$ and $B$, respectively. The resulting theorem, given below, will be used in the next section to carry out assume-guarantee reasoning between different time scales.

**Theorem 4.3 (Constrained distributivity of sample)** Let $A$, $B$, $T_A$ and $T_B$ be transition constraints such that every variable of $T_A$ is an observable variable of $A$ and every variable of $T_B$ is an observable variable of $B$. If $A \parallel B \preceq T_A \parallel T_B$ and $\varphi$ is a predicate on the observable variables common to $A$ and $B$, then (Sample $(A \parallel B)$ at $\varphi$) $\preceq$ (Sample $(A \parallel T_A)$ at $\varphi$) $\parallel$ (Sample $(B \parallel T_B)$ at $\varphi$).
CHAPTER 4. ASSUME-GUARANTEE REASONING WITH SAMPLE

Proof: The proof is similar in spirit to the proof of Theorem 4.2. Let $C = (\text{Sample } (A\parallel B) \text{ at } \varphi)$, $D = (\text{Sample } (A\parallel T_A) \text{ at } \varphi)$, and $E = (\text{Sample } (B\parallel T_B) \text{ at } \varphi)$. By definition of $\text{Sample}$ and $\parallel$ operators, every initial state of $C$ is an initial state of $D\parallel E$. Suppose $s \rightarrow_C t$. Then we know that there is a sequence of states $s_0, s_1, \ldots, s_n$, such that (1) $s = s_0$, (2) $t = s_n$, (3) $s_i \rightarrow_{A\parallel B} s_{i+1}$ for $i = 0, 1, \ldots, n - 1$, and (4) $\varphi(s_i, s_{i+1})$ is false for $i = 0, 1, \ldots, n - 2$ and true for $i = n - 1$. Since every variable of $T_A$ is an observable variable of $A$, and every variable of $T_B$ is an observable variable of $B$, and $A\parallel B \preceq T_A\parallel T_B$, we have $[s_i]_{T_A\parallel T_B} \rightarrow_{T_A\parallel T_B} [s_{i+1}]_{T_A\parallel T_B}$ for $i = 0, 1, \ldots, n - 1$. Since $s_i \rightarrow_{A\parallel B} s_{i+1}$ and $[s_i]_{T_A\parallel T_B} \rightarrow_{T_A\parallel T_B} [s_{i+1}]_{T_A\parallel T_B}$ imply $s_i \rightarrow_{A\parallel T_A} s_{i+1}$ and $s_i \rightarrow_{B\parallel T_B} s_{i+1}$, we have $s \rightarrow_{D\parallel E} t$ as well. Thus $s \rightarrow_{D\parallel E} t$.

4.2 Refinement checking with sample

We generalize the methodology for assume-guarantee style refinement checking given in Chapter 3 to accommodate the $\text{Sample}$ operator.

Sampled witness constraints. Let $A$ and $B$ be transition constraints. Recall that $B^u$ is the transition constraint obtained by making every private variable of $B$ an observable variable. We have seen in Chapter 3 that the refinement check can be reduced to a projection refinement check in the following way. We compose $A$ with a witness constraint $W$ such that (1) the observable variables of $W$ include the private variables of $B$, and (2) $W$ is non-blocking on $\text{Obs}(W) \cap \text{Obs}(A)$. Then $B^u$ is projection refinable by the composition $A\parallel W$ and if $A\parallel W \preceq B^u$ then $A \preceq B$. If the sample operator needs to be applied to the implementation to relate it to the specification, the witness could be composed either before or after applying the $\text{Sample}$ operator.

Theorem 4.4 (Sampled witnesses) Consider two transition constraints $A$ and $B$, and a predicate $\varphi$ on the variables of $A$ such that $B$ is refinable by $A$. Let $W$ be a transition constraint such that (1) $W$ is compatible with $A$, (2) $\text{Priv}(B) \subseteq \text{Obs}(W)$, and (3) $W$ is non-blocking on $\text{Obs}(W) \cap \text{Obs}(A)$. Then (1) $B^u$ is projection refinable by $(\text{Sample } (A\parallel W) \text{ at } \varphi)$, and (2) $(\text{Sample } (A\parallel W) \text{ at } \varphi) \preceq B^u$ implies $(\text{Sample } A \text{ at } \varphi) \preceq B$.

Proof: The proof is very similar to the proof of Theorem 3.2.

Theorem 4.4 is a generalization of Theorem 3.2. The latter can be obtained by setting $\varphi$ to $\text{true}$ in Theorem 4.4. The witness $W$ could be a module also from the following
Corollary 4.5 Consider two transition constraints $A$ and $B$, and a predicate $\varphi$ on the variables of $A$ such that $B$ is refinable by $A$. Let $W$ be a module such that the interface variables of $W$ include the private variables of $B$, and are disjoint from the observable variables of $A$. Then (1) $B^u$ is projection refinable by $(\text{Sample } A\|\mathcal{C}(W))$ at $\varphi$, and (2) $(\text{Sample } A\|\mathcal{C}(W))$ at $\varphi$ $\preceq B^u$ implies $(\text{Sample } A$ at $\varphi$) $\preceq B$.

Assume-guarantee reasoning. Suppose the left side of a refinement relation is of the form $\text{Sample}(A\|B)$ at $\varphi$. It is not directly possible to apply the assume-guarantee rule from Theorem 3.3 in such cases. However, we can distribute the $\text{Sample}$ operator with respect to parallel composition using Theorem 4.2. In practice, Theorem 4.2 tends to provide abstractions that are too coarse to be useful. To see why, imagine $A$ and $B$ as modules, each constraining the other's inputs. By distributing the $\text{Sample}$ inside the parallel composition, $B$ is allowed to constrain the inputs to $A$ only at the sampling instants. The inputs to $A$ are essentially unconstrained between sampling instants. Symmetrically, the inputs to $B$ are constrained at sampling instants by $A$ and left unconstrained between sampling instants. In several common situations, the interactions between $A$ and $B$ can be orthogonalized into (1) functionality, and (2) timing of the communication protocol used for the interaction. The functionality determines values at the sampling instants, whereas the timing determines how these values propagate between sampling instants.

Suppose $T_A$ and $T_B$ are transition constraints that specify how the inputs to $A$ and $B$ behave between sampling instants. Then, we can use Theorem 4.3 to distribute the $\text{Sample}$ operator, while constraining the inputs to $A$ by $T_A$ and the inputs to $B$ by $T_B$. Thus, we get the following generalization of the assume-guarantee rule, with the $\text{Sample}$ operator.

Theorem 4.6 (Assume-guarantee with sample) Let $A = A_1\|A_2\|\ldots\|A_n$ and $B = B_1\|B_2\|\ldots\|B_m$ be transition constraints. Let $T_i$ be a transition constraint on the observable variables of $A_i$, for $1 \leq i \leq n$. Let $<$ be a well-founded order on the components of $B$, and let $T$, $Z$ and $Z^C$ be defined as follows: $T = \{T_1, \ldots, T_n\}$, $Z(B_k) = \{B_j \mid B_j \prec B_k\}$, and $Z^C(B_k) = \{B_j \mid B_j \notin Z(B_k)\}$. For each $B_k$, let $C_i$ be some composition of transition constraints from $A$ and $U_i$ be some composition of transition constraints from $T$ such that if $C_i = \{A_{i1}, A_{i2}, \ldots, A_{ik}\}$ then $U_i = \{T_{i1}, T_{i2}, \ldots, T_{ik}\}$. Also for each $B_k$,
CHAPTER 4. ASSUME-GUARANTEE REASONING WITH SAMPLE

Let $D_i$ be some composition of transition constraints from $Z(B_i)$ and $E_i$ be some composition of transition constraints from $Z^C(B_i)$. If $A \preceq T_1 \cdots \parallel T_n$, and $\psi$ is any predicate such that $(\text{Sample } (C_i \parallel U_i) \text{ at } \psi)^\gamma \parallel D_i^\gamma \parallel E_i^{\gamma-1} \preceq B_i^\gamma$ for all $1 \leq i \leq m$ and $\tau \in \mathbb{N}$, then $(\text{Sample } A \text{ at } \psi) \preceq B$.

**Proof:** We know that $A = A_1 \parallel A_2 \cdots \parallel A_n \preceq C_1 \parallel C_2 \cdots \parallel C_m$. Therefore we have that $\text{Sample } A \text{ at } \psi \preceq \text{Sample } (C_1 \parallel C_2 \cdots \parallel C_m) \text{ at } \psi$. We apply Theorem 3.3 to get

$$\text{Sample } (C_1 \parallel U_1) \text{ at } \psi \cdots \parallel \text{Sample } (C_m \parallel U_m) \text{ at } \psi \preceq B_1 \cdots \parallel B_m = B.$$ 

From the constrained distributivity of sample (Theorem 4.2), we get that

$$\text{Sample } (C_1 \parallel \cdots \parallel C_m) \text{ at } \psi \preceq \text{Sample } (C_1 \parallel U_1) \text{ at } \psi \cdots \parallel \text{Sample } (C_m \parallel U_m) \text{ at } \psi.$$ 

Combining the three statements above, we get that $\text{Sample } A \text{ at } \psi \preceq B$. 

In the above theorem, note that the antecedent $A \preceq T_1 \cdots \parallel T_n$ can itself be discharged by traditional assume-guarantee rule presented in Theorem 3.3.

**Verification of GCD algorithm.** Recall the high-level specification $GCDSp1$, intermediate specification $GCDSp2$, and implementation $Impl$ from the previous sections. As stated in Section 4.1, the refinements we would like to verify are:

$$(\text{Sample } GCDSp2 \text{ at } (\neg \text{inprogress'})) \preceq GCDSp1$$

$$(\text{Sample } Impl \text{ at } sync') \preceq GCDSp2$$

In this section, we will focus on how to carry out the second refinement, which relates the intermediate specification $GCDSp2$ to the implementation $Impl$. We first observe that $GCDSp2$ is not projection refinable by $Impl$, due to the presence of private variables in $GCDSp2$ that represent point-to-point communication channels. The module $Impl$ has a single channel that is shared by all modules using TDMA. The specification, while more abstract in time, provides individual point-to-point channels for communication. Each round of $GCDSp2$ corresponds to multiple rounds of $Impl$, during which a frame is transmitted. It is possible to relate the values that appear on the shared implementation channel, at particular time-slots during the communication of a frame, to values that appear on particular point-to-point channels in the specification. For instance, the values of the specification variables $aout$ and $bou$ at the end of each round are expected to be equal to the values occurring in time-slots 1 and 2 of the frame. We can write a witness module $IntfW$ that
looks at the implementation channel during the transmission of the frame, collects the values at time-slots 1 and 2, and assigns them to \textit{aout} and \textit{bout}, respectively. If the values in time-slots 1 and 2 are valid, then \textit{valid}$_1$ is set to \textit{true}. Further, the values assigned to \textit{aout}, \textit{bout}, and \textit{valid}$_1$ are retained till the end of the frame. Similar witness modules \textit{DoneW} and \textit{CompW} can be written. All these witnesses take inputs from the channel as shown in Figure 4.2(b). Let \textit{ImplW} be the module given by

$$\text{ImplW} = \text{Impl} \vDash \text{IntfW} \vDash \text{DoneW} \vDash \text{CompW}.$$  

Note that the witnesses merely observe the values on the channel without interfering with it. Due to Theorem 4.4, it suffices to check that \((\text{Sample ImplW at sync'}) \preceq \text{GCDSp2}\). Recall that \text{GCDSp2} = \text{IntfS} \vDash \text{DoneS} \vDash \text{CompS}. We apply Theorem 4.6 with the order \text{DoneS} \prec \text{CompS} \prec \text{IntfS}. Let us consider the component \text{CompS} of \text{GCDSp2}. The component of \text{Impl} that is intended to implement the functionality of \text{CompS} is \text{Comp}. We wish to check if:

$$\text{DoneS}^\tau \preceq \text{Comp}^\tau.$$  

This check fails because the outputs of module \text{CompS}, namely, \text{ain}, \text{bin} and \text{valid}$_3$ are not present in module \text{Comp}. Adding the witness module and appropriately constraining it, we obtain the obligation:

$$\text{DoneS}^\tau \preceq \text{Comp}^\tau.$$  

This still fails, because we have not constrained the inputs of \text{Comp}. In this obligation, the specification \text{CompS} looks at the inputs \text{small}, \text{big}, and \text{valid}$_2$ in every round and produces corresponding outputs \text{ain}, \text{bin} and \text{valid}$_3$ in the next round. The implementation \text{Comp} anticipates two values at time-slots 3 and 4 (which correspond to \text{small} and \text{big}, respectively) of every frame. If these inputs are valid, then \text{Comp} generates values in time-slots 5 and 6 (which correspond to \text{ain} and \text{bin} respectively) of the next frame. The module \text{Comp} makes the following assumptions: (1) the inputs are available at time-slots 3 and 4, (2) either both inputs are available at a given frame, or none of the inputs are available, and (3) if both inputs are available, the first input (from time-slot 3) is smaller than the second input (from time-slot 4). In our refinement obligation, the inputs to \text{Comp} have to be constrained both at the sampling instants and between sampling instants, in order to satisfy the above assumptions. The specification component \text{DoneS} supplies the
constraint at sampling instants, which ensures that assumptions 2 and 3 are satisfied. The
timing assumption \( \text{DoneW} \) constrains the inputs between sampling instants and ensures
that assumption 1 is satisfied. Thus we get the proof obligation

\[
(Sample (Comp \parallel Comp W \parallel Ch \parallel \text{DoneW}) \text{ at } sync')^\top \parallel \text{DoneS}^\top \leq \text{CompS}^\top.
\]

Similarly, we can verify the correctness of modules \( \text{Done} \) and \( \text{Intf} \) separately. The
complete refinement proof, which is a direct application of Theorem 4.6, uses the ordering
\( \text{DoneS} \prec \text{CompS} \prec \text{IntfS} \):

\[
\begin{align*}
(Sample (Done || Done W || Ch || Bc || Intf W) \text{ at } sync')^\top & \leq \text{DoneS}^\top \\
(Sample (Comp || Comp W || Ch || Bc || Done W) \text{ at } sync')^\top \parallel \text{DoneS}^\top & \leq \text{CompS}^\top \\
(Sample (Intf || Intf W || Ch || Bc || Comp W) \text{ at } sync')^\top \parallel \text{CompS}^\top \parallel \text{DoneS}^\top & \leq \text{IntfS}^\top
\end{align*}
\]

\[
(Sample (ImplW) \text{ at } sync') \leq \text{GCDSp}^2
\]

Each of the obligations above the line involves a single implementation component,
possibly along with specification components, witnesses and abstract constraints to
constrain the inputs. They can be automatically discharged by Mocha. We thus conclude
that \( (Sample \text{ Impl at } sync') \leq \text{GCDSp}^2 \).

### 4.3 Verification of VGI

The VGI chip [STUR98] is an array of DSP processors designed to be part of
a system for web-based image processing [SR97]. The VGI chip contains a total of 96
processors and has approximately 6M transistors. Of the 96 processors, 64 are identical 3-
stage pipelined compute processors. Each compute processor has about 30,000 logic gates.
Data is communicated between the processors by means of FIFO queues. No assumption is
made about the relative speeds at which data is produced and consumed in the processors.
Hence, to transfer data reliably an elaborate handshake mechanism is used between the
sender and the receiver. In addition, the interaction between the control of the pipeline
and the control of the communication unit is quite complex. We give a brief summary of
our verification of the VGI chip here; full details can be found in Sriram Rajamani’s thesis
[Raj99].

We focus on the verification of the 64 compute processors and the communication between them. A single processor is described partly in VHDL and partly in circuit
schematics. We translated the description into reactive modules which is the input language to our model checker MoCha. After a number of discussions with the designers, we produced a formal specification of the design which embodies the programmer’s view of the system, also in Reactive Modules. The sheer size of the design together with the well-known state explosion problem precluded the direct use of model checking techniques to verify the implementation against the specification. Existing techniques that flatten the design hierarchy and use BDD-based state exploration [BHSV+96] can verify designs with at most 50–100 latches reliably. Clearly, the VGI design, which has about 800 latches per compute processor, is well beyond the scope of such tools. We demonstrate how model checking can be scaled up using assume-guarantee reasoning to handle the VGI design. To the best of our knowledge, the largest design that has been ever verified using model checking has been reported by Errikkson [Eir98]. Compositional techniques used in that effort for decomposing the verification task did not readily apply to the VGI, because the implementation and specification operate on different time scales (several consecutive implementation steps realize a single specification step). We used the techniques developed in this chapter to obtain proof obligations that were small enough to be discharged automatically by MoCha. In the process, we found several subtle bugs that were unknown to the designers. Three of these bugs will be explained in the discussion in Section 4.3.3.

A significant part of the verification effort was invested in producing a correct specification. Only an informal specification of the design existed in the form of English description and elaborate timing diagrams. The fact that no behavioral description of the design was available (the datapath was designed directly in schematic) made the task of producing the specification even more difficult.

A number of features are desirable in the specification for the VGI chip. First, the specification should be at a level of abstraction such that a high degree of confidence in its correctness can be established by informal means such as code review. Specifically, the specification should embody the view that the programmer/compiler has of the VGI chip, which is that of a dataflow architecture with a set of processing elements connected through queues. For this high-level view, every processing element behaves as if each instruction is executed atomically in one step, and the communication between the processors behaves like FIFO queues. The behavior of a program written with this high-level view should not depend on the delay in transferring a data token from one processor to another. Such FIFO queues can be modeled using nondeterministic delay. This makes necessary the availability
of nondeterminism in the specification language.

Second, the specification should have an operational as well as a mathematical semantics. Operational semantics permits the execution of specifications; mathematical semantics permits their formal verification. Executability is especially desirable in the case of the VGI processor because the design is part of a bigger system. If all essential features of the design that are necessary for correct interaction with the environment have been captured by the specification, it can be used in place of the actual design for simulating the system.

Third, the design itself (the "implementation") should be describable in the same language as the specification, and a refinement operator should be available for relating the implementation and the specification. In our case, the refinement operator must relate two different time scales. The implementation has a clock signal $clk$ with activity on both the HIGH and LOW phases in different parts of the design. For instance, in the execute phase of the pipeline a bus carries an operand when $clk$ is HIGH and the result when $clk$ is LOW. But the specification does not mention $clk$ at all. In fact, the whole computation happens in just one step. Thus, one round in the specification is equal to two rounds in the implementation, one with $clk = \text{HIGH}$ and one with $clk = \text{LOW}$. Therefore, our formal notion of refinement samples the implementation whenever $clk$ is LOW and checks if the sampled behavior is present in the specification.

Reactive Modules, our modeling language for both specification and implementation, has all the desirable features mentioned above—mathematical semantics, executability, and support for nondeterminism and sampling.

4.3.1 The Problem

A compute processor in the VGI chip has an instruction memory, a register file containing three register pairs, a 3-stage pipelined datapath, a control unit, and three data output buses and one control output bus for sending tokens to other processors. Each register pair can be configured either as a queue or as general purpose registers. Each output bus may or may not be connected to another processor. A processor $P$ can send data to another processor $Q$ if a data output bus of $P$ is connected to a register pair of $Q$ that has been configured as a queue. A handshake protocol is used between $P$ and $Q$ for transferring data reliably. There is a programmable interconnection network that allows
any processor to be connected to any other processor. In a typical dataflow computation, programs are loaded into the instruction memory of some subset of the set of processors, and the appropriate data connections between the processors are made by programming the network. Each processor with its own program acts as an “actor” in a data flow network, consuming tokens from its input and producing tokens at its output. In any network of compute processors, each processor is in a certain configuration depending on the register pairs configured as queues, and the output buses connected to downstream processors. Let $C_{VGI}$ denote this set of $2^3 \times 2^4 = 128$ configurations. Figure 4.3 shows a processor configuration where the register pair R2–R3 is configured as a queue, and a data and a control output queue are configured to send out tokens.

Our specification for the processor configuration shown in Figure 4.3 consists of modules ISA, DataQueue and ControlQueue. The module ISA contains other modules such as program memory, register file, control unit and ALU inside it and is a specification of the pipelined datapath of processor P1. Every instruction gets executed atomically in one round in the ISA. The specification for the data output bus of P1 together with the queue of processor P2 is a 4-place FIFO buffer DataQueue. ControlQueue is identical to DataQueue except for the data width and is the specification for the control output bus of P1 together with the queue of processor P3. Performing verification against the composition

\footnote{The dotted rectangle in the lower portion of Figure 4.4 shows abstraction modules. We defer their description to Section 4.3.2.}
of ISA, DataQueue and ControlQueue will ensure that instructions are executed correctly and data is transferred reliably from P1 to P2 and P3. We can similarly write specifications for processors P2 and P3. Then the specification for the network of processors P1, P2 and P3 can be obtained by composing the specifications of the individual processors. In Figure 4.4, note that the register pair R2-R3 is missing. Since they have been configured as an input queue, they are part of the distributed output queue of an upstream processor, and will be specified in that processor. Our verification methodology, described in the next section, will let us prove that an arbitrary network of compute processors satisfies its specification.

4.3.2 The Proof

Each compute processor in VGI starts a computation in the positive phase of the clock and finishes it in the negative phase of the clock. We decided to sample at the end of each computation. Hence, the sampling predicate $\varphi$ is $\text{clk} = \text{LOW}$. In the rest of this section, we use $\varphi$ to refer to $\text{clk} = \text{LOW}$. In Section 4.3.1, we showed how to obtain a specification for an arbitrary network of processors. Our goal is to verify that an arbitrary network of processors implements its corresponding specification, using refinement checking. Let $P_1, P_2, \ldots, P_n$ be the compute processors in an arbitrary network, and let $Q_1, Q_2, \ldots, Q_n$ be their respective specifications. For the correct functioning of a processor it is essential that all input signals change only when $\text{clk}$ is $\text{HIGH}$. Let $T_i$ be a module that says that all external signals of $P_i$ change only when $\text{clk}$ is $\text{HIGH}$. 

Figure 4.4: Specification module for refinement check
CHAPTER 4. ASSUME-GUARANTEE REASONING WITH SAMPLE

The verification problem is to check

$$\text{Sample } (P_1 \parallel P_2 \parallel \cdots \parallel P_n) \text{ at } \varphi \preceq Q_1 \parallel Q_2 \parallel \cdots \parallel Q_n$$

We can apply our new assume-guarantee rule as follows:

$$\begin{align*}
\text{Sample } (P_1 \parallel T_1) \text{ at } \varphi & \preceq Q_i \text{ for all } 1 \leq i \leq n \\
P_1 \parallel P_2 \parallel \cdots \parallel P_n & \preceq T_1 \parallel T_2 \parallel \cdots \parallel T_n \\
\text{Sample } (P_1 \parallel P_2 \parallel \cdots \parallel P_n) \text{ at } \varphi & \preceq Q_1 \parallel Q_2 \parallel \cdots \parallel Q_n
\end{align*}$$

The second antecedent says that the inputs of any processor in the network change only when clk is HIGH. Since any input to a processor has to be the output of some other processor, this antecedent can be discharged easily by proving that for all $1 \leq i \leq n$, the outputs of $P_i$ change only when clk is HIGH. This is an easy proof local to each processor and computationally trivial. In the first antecedent, there are $n$ symmetric proof obligations, one for each $P_i$. For $X \in \mathcal{C}_{VG},$ let $Y$ be its specification and $T_X$ be the environment constraint that says that all inputs change only when clk is HIGH. If we prove $\text{Sample } (X \parallel T_X) \text{ at } \varphi \preceq Y$ for each $X \in \mathcal{C}_{VG}$, then we have proved that $\text{Sample } (P_1 \parallel T_i) \text{ at } \varphi \preceq Q_i$ for all $1 \leq i \leq n$. Thus, we have decomposed the proof of an arbitrary network of compute processors to $|\mathcal{C}_{VG}|$ proofs about individual processor configurations that have 800 latches each. This is still beyond the scope of monolithic model checking. We show how to discharge this proof for a single processor configuration, with further applications of the generalized assume-guarantee rule described earlier. We implemented support for the $\text{Sample}$ operator in MOCHA, in order to carry out this refinement check.

We describe the compositional proof for the configuration in Figure 4.3 whose specification is given in Figure 4.4. We describe a compute processor in more detail. The processor has a 3-stage pipeline — the fetch stage IF, the execute stage EX, and the communicate stage COM, with pipelat latches between IF and EX, and lout latches between EX and COM. There is feedback from the EX stage to the IF stage. The IF stage is controlled by mir1reg and fetches data from the input queues, the register file, or the feedback. The signal stallempty is asserted if an instruction wants to read from an input queue that is empty. The EX stage contains the ALU and is controlled by mir2reg, a delayed version of mir1reg. The output of the ALU abus_r can be written back to the register file or sent out on one or more queues. For receiving data/control tokens, the downstream processor should have a register pair configured as a 2-place queue. Every data or control token
CHAPTER 4. ASSUME-GUARANTEE REASONING WITH SAMPLE

that is computed is latched into lout. If the first send fails, then the COM stage keeps on sending the data in lout until the send succeeds. Signals send and sendack are used for handshake between the sender and the receiver. In the meantime, other instructions might be executing in the EX stage of the pipeline. The pipeline is stalled and a signal stallpipe asserted when the COM stage is trying to send a token and the instruction in the EX stage also wants to send out a token. The invariant that synchronizes the operation of the ISA and the pipeline is that the instruction being executed by the ISA is the instruction in the IF stage of the implementation.

To decompose the proof, we wrote abstraction modules for send, sendack, abus_r, stallpipe, and pipelat_a.s as shown in the dotted rectangle in Figure 4.4. In order to write abstraction modules for send and stallpipe, we had to add auxiliary history variables exsend, num, and sendackp. The variable exsend is true whenever the the current instruction in the EX phase wants to send. The variable num keeps track of the number of items in the receiver’s 2-place input queue. The variable sendackp predicts the implementation’s sendack. The abstraction module for abus_r is written in terms of the two stall signals and the output of the ALU in the specification. Using these abstraction modules, the proof can be decomposed nicely in the reverse direction of the flow of data in the processor. The following lemmas were verified.

1. The output queue is verified using the abstraction modules for the variables abus_r, send, and stallpipe. Intuitively, this means that data written into the queue is not lost, no data is written twice, and correct behavior is preserved going into and coming out of stalls (either stallempty or stallpipe).

2. The abstraction module for send is verified using the abstraction module for sendack.

3. The abstraction module for sendack is verified using the abstraction modules for stallpipe and send.

4. The abstraction module for stallpipe is verified using abstraction modules for send and sendack of both the control and data queues.

5. The abstraction module for the variable abus_r is verified using the abstraction module for the pipelat_a.s signals, which are inputs to the EX stage. Since the bus is generated by the data path of the implementation, this proof amounts to verifying
the correctness of the data path. We have not been able to complete this proof. We believe that this is essentially a combinational verification problem that is amenable to existing techniques geared for combinational verification.

6. The abstraction module for pipelat.a.s is verified using the abstraction module for abus.r. This lemma amounts to verifying the correctness of feedback from the EX stage to the register file and the pipelat.a.s registers.

In each lemma described above, the part of the implementation under investigation was sampled at clk equal to LOW under some timing assumptions on the inputs between sampling instants. For example, in Lemma 1, it was assumed that the send signal does not change value when clk changes from LOW to HIGH, and all signals at the receiver end (such as read and save_d) change values only when clk is HIGH. All such assumptions were discharged separately. Notice the circular dependencies between Lemmas 1, 2, 3, and 4, and also Lemmas 5 and 6. For Lemmas 2, 3, 4, 5, and 6, we also wrote supporting abstraction modules for mir1reg and mir2reg. These supporting refinements were verified separately. In total, about 35 lemmas needed to be proved. In every lemma except Lemma 5, we used symmetry arguments [McM98] to reduce the datapath width to just 1 bit. In Lemma 5, the symmetry is broken because of arithmetic operations and hence the full datapath width of 16 bits needs to be considered. Thus, assume-guarantee reasoning provides a clean separation between the verification of the datapath and control of the processor. It is clear in the overall proof that the datapath width is irrelevant in verifying the control that is moving data around. This also suggests that compositional reasoning provides a formal framework under which combinational verification of the datapath and FSM verification of the control can coexist. None of the individual lemmas took more than a few minutes on a 625 MHz DEC Alpha 21164.

4.3.3 Discussion

In this section, we describe the bugs we found in the design. We fixed all the bugs and verified our fixes with Mocha.

1. If the sending processor writes two successive values into the queue and the receiving processor waits for one cycle and then does two successive reads, the second read returns an incorrect value.
2. Suppose \texttt{stallempty} is asserted in cycle \( n \) but released in cycle \( n + 1 \). Also, suppose send to an output queue fails in cycle \( n + 1 \). Then although \texttt{stallpipe} should be asserted in cycle \( n + 2 \), it is not and as a result the instruction in \texttt{EX} stage gets overwritten.

3. A particular sequence of events involving 4 sends and 4 reads interleaved in a specific way, with a stall at a precise moment destroys the data in the \texttt{lout} register. This results in the loss of an output token. The error trace that led to the discovery of this bug had ten steps.

We now describe the process by which we found these bugs and the insights we gained about the interaction between design and verification. We found all these bugs while doing the proof of Lemma 1, the lemma stating the correctness of the data transfer between the sender and the receiver. Recall that we needed abstraction modules for the environment signals \texttt{busr}, \texttt{send} and \texttt{stallpipe}. Initially, we tried to write the abstraction modules based on the definitions of these signals in the implementation. Since the definition of these signals in the implementation was buggy, our abstract modules were not strong enough to satisfy the assumptions of the output queue on its environment. As a result we got error traces. We then decided to ignore the implementation and defined these signals abstractly so that we could discharge Lemma 1. At this point, the abstract definitions of the signals \texttt{busr}, \texttt{send}, and \texttt{stallpipe} were strong enough to satisfy the environment assumptions of the output queue. We then reimplemented these signals according to the specification prescribed by these abstract definitions, and verified that our implementation was correct. These design fixes were quite complicated and we actually had to do some logic design ourselves. In this way, \texttt{MOCHA} can be used as a debugging tool which tests a proposed design fix by looking at all possible sequences of events. If an error trace is generated then it can be examined to further refine the fix. Thus, the distinction between verifying and designing gets blurred and actually both activities proceed in parallel. We believe that design and verification are symbiotic activities in the sense that the designer’s intuition embodied in abstraction modules aids verification and the model checker aids the designer by testing that a proposed solution is correct under all possible situations. We believe that the mental processes involved in doing verification exist when the design is being created and therefore, given the right interface to a verification tool, it is not a big burden to do “formal design.”
We have shown by our verification of the VGI chip that compositional model checking under the assume-guarantee paradigm can scale to “real” designs. We also believe that it is a general technique not just limited to DSP chips. In our proof, the first step that decomposes the proof obligation on a network of processors to one on a single processor relies on the symmetry inherent in VGI. But the second step involving proof decomposition with the aid of abstraction modules is quite general and applicable to a variety of large and complex designs [Eír98, HQR98, McM98].
Chapter 5

Verifying Parameterized Shared-Memory Systems

Shared-memory multiprocessors are an important class of supercomputing systems. In recent years a number of such systems have been designed in both academia and industry. The design of a correct and efficient shared memory is one of the most difficult tasks in the design of such systems. The shared-memory interface is a contract between the designer and the programmer of the multiprocessor. In general, there is a tradeoff between the ease of programming and the flexibility of shared-memory semantics necessary for an efficient implementation. Not surprisingly, a number of abstract shared-memory models have been developed.

All abstract memory models can be understood in terms of the fundamental serial-memory model. A serial memory behaves as if there is a centralized memory that services read and write requests atomically such that a read to a location returns the latest value written to that location. Coherence\(^1\) requires that the global temporal order of events (reads and writes) at different processors be a trace of serial memory. Sequential consistency [Lam79] ignores the global temporal order and requires only that some interleaving of the local temporal orders of events at different processors be a trace of serial memory. Although sequential consistency is a strictly weaker property than coherence, the absence of a synchronizing global clock between the different processors in a multiprocessor makes a sequentially consistent memory indistinguishable from a serial memory. Compared to co-

\(^1\)Implementors of cache-based shared-memory systems have used the notion of cache coherence for a long time but the definition of coherence as stated here was first given in [ABM93].
CHAPTER 5. VERIFYING PARAMETERIZED SHARED-MEMORY SYSTEMS

herence, sequential consistency clearly offers more flexibility for an efficient implementation; yet, most real systems that claim to be sequentially consistent actually end up implementing coherence. In an effort to get more flexibility for implementation, memory models that relax local temporal order of events at each processor have been developed in recent years. These models are not sequentially consistent and result in a more complicated programmer's interface. These memory models, such as weak ordering, partial store ordering, total store ordering, and release consistency [AG96], relax the processor order of events in different ways and provide fence or synchronization operations across which sequentially consistent behavior is guaranteed.

We focus on the verification of sequential consistency for two reasons. First, the interface provided by sequential consistency is clear, easy to understand, and widely believed to be the correct tradeoff between implementation flexibility and complexity of the programmer's view of shared memory. In fact, there is a trend of thought [Hil98] that considers the performance gains achieved by relaxed semantics not worth the added complexity of the programmer's interface and advocates sequential consistency as the shared-memory interface for future multiprocessors. Second, even relaxed memory models have fence operations across which sequentially consistent behavior should be observed. Hence, the techniques developed in this paper will be useful for their verification also.

High-level descriptions of shared-memory systems are typically parameterized by the number \( n \) of processors, the number \( m \) of memory locations, and the number \( v \) of data values that can be written in a memory location. A parameterized memory system consists of a central-control part \( C \) and a processor part \( P \). Both \( C \) and \( P \) are functions that take values for \( m \) and \( v \) and return a finite-state process. An instantiation of the system containing \( n \) processors, \( m \) memory locations, and \( v \) data values is constructed by composing \( C(m, v) \) with \( n \) copies of \( P(m, v) \). We would like to verify sequential consistency for all values of the parameters. However, sequential consistency is not a local property; correctness for \( m \) processors (locations, values) cannot be deduced by reasoning about individual processors (locations, values). The following observations about real shared-memory systems, which we assume in our modeling, are crucial for our results. We assume that the memory system is monotonic and symmetric with respect to both the set of locations, and the set of data values. Monotonicity in locations (data values) means that a sequence is a run of the system with some set of possible locations (data values) if and only if it is a run of the system with a larger set of locations (data values). Symmetry in locations means that, if \( \sigma \) is a run of
CHAPTER 5. VERIFYING PARAMETERIZED SHARED-MEMORY SYSTEMS

the memory system, and $\lambda_i$ is a permutation on the set of locations, then $\lambda_i(\sigma)$ is also a run of the memory system. Finally, symmetry in data values means that, if $\sigma$ is a run of the memory system, and $\lambda_v$ is any function from data values to data values, then $\lambda_v(\sigma)$ is also a run of the memory system. We also assume that the memory system is projectible with respect to locations. This means that every run of the system projected onto a subset of locations is a run of the system with just that subset of locations.

Even for fixed values of the parameters, checking if a memory system is sequentially consistent is undecidable [AMP96]. The main reason for the problem being undecidable is that the specification of sequential consistency allows a processor to read the value at a location after an unbounded number of succeeding writes to that location by other processors. In real systems, finite resources such as buffers and queues bound the number of writes that can be pending. It is sufficient to construct a witness that observes the reads and writes occurring in the system (without interfering with it) and reorders them while preserving the order of events in each processor such that a trace of serial memory is obtained. We call such a witness a serializer. If a finite-state serializer exists, then it can be composed with a fixed-parameter instantiation of the memory system and the problem of deciding sequential consistency is reduced to a language-containment check between two finite-state automata which can be discharged by model checking. In the concrete examples we have looked at (see below), we have indeed seen that a finite-state serializer exists for fixed values of the parameters.

However, our goal is to verify sequential consistency for arbitrary values of the parameters. Towards this end, we develop two novel proof frameworks. The first framework lets us prove sequential consistency for a fixed number of processors but arbitrary number of locations and data values. A serializer $\Omega$ is location symmetric if for any run $\sigma$ and any permutation $\lambda_i$ on the set of memory locations $\lambda_i(\Omega(\sigma)) = \Omega(\lambda_i(\sigma))$. A serializer $\Omega$ is data symmetric if for any run $\sigma$ and any function $\lambda_v$ from data values to data values $\lambda_v(\Omega(\sigma)) = \Omega(\lambda_v(\sigma))$. A serializer is local if its output on any run $\sigma$ is an interleaving of its outputs on $\sigma$ projected onto each memory location. We show that existence of a local, location symmetric and data symmetric serializer for a memory system with $n$ processors and $n$ memory locations and two data values implies that the memory system with $n$ processors is sequentially consistent for any number of memory locations and data values.

The second framework is based on a novel induction scheme and lets us prove sequential consistency for arbitrary number of processors, locations and data values. In-
ductive proofs on parameterized systems [KM89] use an implementation preorder and show
the existence of a process invariant such that the composition of the invariant with an addi-
tional process is smaller than the process invariant in the preorder. The preorders typically
used—for instance, trace containment and simulation—preserve the temporal sequence of
events. Since we check a sufficient condition for sequential consistency by the mechanism
of an serializer that reorders the read/write events of the processors in the system, pre-
orders that preserve the temporal sequence of events do not suffice for our purpose. Our
inductive proof strategy first determines a process invariant $I_1$ of the memory system with
respect to the trace-containment preorder to get a finite-state abstraction that can generate
all sequences of observable actions for any number of processors. We then find a merge
invariant $I_2$ such that (1) the single-processor memory system containing $I_2$ is sequen-
tially consistent, and (2) there is an witness that maps every run $\sigma$ of $I_2 \parallel P$ that can be produced
in an environment of $I_1$ to a run $\sigma'$ of $I_2$, such that the read/write events in $\sigma'$ are an inter-
leaving of the read/write events of $I_2$ and $P$ in $\sigma$, and the traces obtained from $\sigma$ and $\sigma'$ are
identical. Given a run $\gamma$ of the memory system with $n > 1$ processors, we use the witness to
create a run $\gamma'$ of the memory system with $n - 1$ processors, such that $\gamma$ and $\gamma'$ are identical
when projected to the events of the first $n - 2$ processors, and the read/write events of the
$(n - 1)$-st processor in $\gamma'$ are an interleaving of the read/write events of the $(n - 1)$-st and
$n$-th processors in $\gamma$. By doing this $n$ times, we generate a run of the memory system with
a single processor, which is sequentially consistent by the base case of the induction.

The induction demonstrates sequential consistency for any number of processors,
but given $m$ and $v$. We would like sufficient conditions under which using fixed values for
$m$ and $v$ lets us conclude sequential consistency for all $m$ and $v$. To that end, we impose a
few requirements on the process and merge invariants. The requirements of symmetry and
monotonicity on memory locations and data values and projectibility on memory locations
are identical to the corresponding assumptions on the memory system. There are two
other requirements called location independence and data reducibility. A process is location
independent if it has the property that a sequence of events is a run of the process with
$m$ locations if and only if the $m$ sequences obtained by projecting onto individual memory
locations are runs of the process with a single location. A process is data reducible if
whenever a sequence $\sigma$ is not a run of the process, there is a function $\lambda_v$ mapping data
values to just two data values such that $\lambda_v(\sigma)$ is also not a run of the process. We show
that if the two invariants satisfy location symmetry, location monotonicity, and location
independence, and the witness is local, location symmetric and data symmetric, then it suffices to do the induction for three memory locations and two data values. As a result, the correctness of the memory system can be proved by discharging two finite-state lemmas using a model checker—one that proves the correctness of the process invariant, and another that proves the correctness of the merge invariant. We would like to point out that although the rewards of the inductive framework are greater, so is the effort involved.

Our proof framework can be applied to a variety of protocols; in particular, all cache-coherence protocols described in [AB86] fall into its domain. We demonstrate the method by verifying a snoopy cache coherence protocol [HP96a]. The correctness of the snoopy cache coherence protocol is argued informally in [HP96a]. We show that a finite-state serializer exists for this example. We reduce the proof of the parameterized protocol to finite-state lemmas as described above, and discharge them by our model checker MOCHA [AHM+98]. Manual effort is required to construct the process and merge invariants, and the serializer, and to verify that the assumptions on the memory system and the requirements on the invariants and serializer are indeed satisfied.

Related work. The notion of a serializer is related to work on representative interleaving sequences [KP92] and timestamping [Lam78, PSCH98]. In a memory system with n processors, the output of a serializer is an interleaving of the n processor orders of memory events. It is a convenient or representative interleaving in the sense that it is a trace of serial memory which can be verified easily. The process whereby a particular implementation of a serializer arrives at this interleaving is by assigning logical timestamps ordered by causality (rather than global time) to memory events.

Abstract memory models of parameterized shared-memory systems have been verified using automatic techniques, mechanical theorem proving and systematic manual proof techniques. Symbolic methods [MS91, CGH+93, EM95] and symmetry reduction [ID96] have been used to alleviate the state explosion problem in model checking finite instances of cache coherence protocols. These papers do not deal with sequential consistency—they verify coherence—and perform verification for fixed values of the various parameters. The method in [PD95] uses model checking to prove coherence for an arbitrary number of processors but cannot deal with sequential consistency. The “test model checking” approach of [NGMG98] offers a necessary condition for sequential consistency that can be checked by test automata monitoring the events at the different processors. A problem with their approach is that they might miss violations of sequential consistency. They also perform the verifi-
cation for fixed values of the parameters. Sufficient conditions for sequential consistency have been formulated in branching temporal logic [Gra94] and as an abstract transition system [GMG91]. There is no experimental data to assess if these sufficient conditions hold on practical shared-memory protocols. Mechanical theorem proving [LD92, PD96] and systematic manual proof methods [LLOR99, PSCH98] have been used to verify shared memory systems for arbitrary values of parameters. These approaches require a lot of human effort. In our approach, a large part of the proof is performed automatically with a model checker.

5.1 I/O-processes

We use I/O-processes that synchronize on observable actions to model memory systems. Formally, an I/O-process $A$ is a 5-tuple $\langle \text{Priv}(A), \text{Obs}(A), S(A), S_I(A), T(A) \rangle$ with the following components:

- A set $\text{Priv}(A)$ of private actions and a set $\text{Obs}(A)$ of observable actions, such that $\text{Priv}(A) \cap \text{Obs}(A) = \emptyset$. The set $\text{Act}(A)$ is the union of $\text{Priv}(A)$ and $\text{Obs}(A)$. Private actions are outputs, whereas observable actions can be both inputs and outputs. The set of extended actions $\Pi(A)$ is given by $\text{Priv}(A) \times \{\text{out}\} \cup \text{Obs}(A) \times \{\text{in, out}\}$.

- A finite set $S(A)$ of states.

- A set $S_I(A) \subseteq S(A)$ of initial states.

- A transition relation $T(A) \subseteq S(A) \times \Pi(A) \times S(A)$.

For all $\pi \in \Pi(A)$, the first component is denoted by $\text{First}(\pi)$ and the second component by $\text{Second}(\pi)$. If $\text{Second}(\pi) = \text{in}$ then $\pi$ is called an input action. If $\text{Second}(\pi) = \text{out}$ then $\pi$ is called an output action. The length of a sequence of actions or extended actions $\sigma = \pi_0, \pi_2, \ldots, \pi_{k-1}$ is $k$ and is denoted by $|\sigma|$. For all $i < |\sigma|$, the $i$th element $\pi_i$ is denoted by $\sigma(i)$. Let $\epsilon$ denote the empty sequence. A sequence $\sigma$ of extended actions of $A$ is a run if either $\sigma = \epsilon$ or $\sigma = \pi_0, \pi_2, \ldots, \pi_{k-1}$ and there exist states $s_0, s_1, s_2, \ldots, s_k$ such that $s_0 \in S_I(A)$ and $\langle s_i, \pi_i, s_{i+1} \rangle \in T(A)$ for all $0 \leq i < k$. We say that $\sigma$ is a run of $A$ leading to the state $s_k$. The projection operators $\text{First}$ and $\text{Second}$ are extended to runs in the natural way. The set of all runs of the I/O-process $A$ is denoted by $\Sigma(A)$. A run is closed if $\text{Second}(\pi_i) = \text{out}$ for all actions $\pi_i$ in the run. A set of runs is closed if all runs in the set are closed. For any set $\beta \subseteq \text{Act}(A)$, the restriction of the run $\sigma$ to $\beta$ is the subsequence
obtained by considering the elements from $\beta \times \{\text{in}, \text{out}\}$ in $\sigma$, and is denoted by $[\sigma]_\beta$. For any run $\sigma$ of I/O-process $A$, the restriction of $\sigma$ to $\text{Obs}(A)$ is called a trace and is denoted by $\text{tr}_A(\sigma)$. The set of all traces of the I/O-process $A$ is denoted by $\Gamma(A)$.

Let $A_1$ and $A_2$ be two I/O-processes. We say that $A_1$ refines $A_2$, denoted by $A_1 \preceq A_2$, if (1) $\text{Obs}(A_1) \subseteq \text{Obs}(A_2)$, and (2) every trace of $A_1$ is a trace of $A_2$. The I/O-processes $A_1$ and $A_2$ are compatible if (1) $\text{Priv}(A_1) \cap \text{Act}(A_2) = \emptyset$ and (2) $\text{Priv}(A_2) \cap \text{Act}(A_1) = \emptyset$.

We observe that if $A_1$ and $A_2$ are compatible then $\text{Act}(A_1) \cup \text{Act}(A_2)$ can be partitioned into three subsets --- $\text{Obs}(A_1) \cap \text{Obs}(A_2)$, $\text{Act}(A_1) \setminus \text{Act}(A_2)$ and $\text{Act}(A_2) \setminus \text{Act}(A_1)$. The composition $A = A_1 || A_2$ of two compatible I/O-processes $A_1$ and $A_2$ is the I/O-process $A$ such that

- $\text{Priv}(A) = \text{Priv}(A_1) \cup \text{Priv}(A_2)$, and $\text{Obs}(A) = \text{Obs}(A_1) \cup \text{Obs}(A_2)$.
- $S(A) = S(A_1) \times S(A_2)$, and $S_I(A) = S_I(A_1) \times S_I(A_2)$.
- $(s_1, s_2), (a, x), (t_1, t_2) \in T(A)$ iff one of the following three conditions holds:
  
  - $a \in \text{Obs}(A_1) \cap \text{Obs}(A_2)$ and either
    - (1) $x = \text{in}$ and $(s_1, (a, \text{in}), t_1) \in T(A_1)$ and $(s_2, (a, \text{in}), t_2) \in T(A_2)$, or
    - (2) $x = \text{out}$ and $(s_1, (a, \text{out}), t_1) \in T(A_1)$ and $(s_2, (a, \text{in}), t_2) \in T(A_2)$, or
    - (3) $x = \text{out}$ and $(s_2, (a, \text{out}), t_2) \in T(A_2)$ and $(s_1, (a, \text{in}), t_1) \in T(A_1)$.

- $a \in \text{Act}(A_1) \setminus \text{Act}(A_2)$ and
  
  $(s_1, (a, x), t_1) \in T(A_1)$ and $s_2 = t_2$.

- $a \in \text{Act}(A_2) \setminus \text{Act}(A_1)$ and
  
  $(s_2, (a, x), t_2) \in T(A_2)$ and $s_1 = t_1$.

Suppose that $A_1$ and $A_2$ are compatible I/O-processes. Let $\sigma_1$ be a sequence of actions in $\Pi(A_1)$ and $\sigma_2$ be a sequence of actions in $\Pi(A_2)$. We use the convention that for any set of actions $X$ and an extended action $\pi$, we say that $\pi \in X$ if $\text{First}(\pi) \in X$. We now define inductively when a sequence $\sigma$ is a join of sequences $\sigma_1$ and $\sigma_2$. We say that $\sigma$ is a join of sequences $\sigma_1$ and $\sigma_2$ iff one of the following hold:

1. $\sigma = \epsilon$ and $\sigma_1 = \sigma_2 = \epsilon$.

2. $\sigma = \sigma', (a, x)$ and one of the following are true.
• \( a \in \text{Act}(A_1) \setminus \text{Act}(A_2) \),
  \( \sigma_1 = \sigma'_1.a(x) \), and \( \sigma' \) is a join of \( \sigma'_1 \) and \( \sigma_2 \).

• \( a \in \text{Act}(A_2) \setminus \text{Act}(A_1) \),
  \( \sigma_2 = \sigma'_2.a(x) \), and \( \sigma' \) is a join of \( \sigma_1 \) and \( \sigma'_2 \).

• \( a \in \text{Obs}(A_1) \cap \text{Obs}(A_2) \),
  \( \sigma_1 = \sigma'_1.\pi_1 \) and \( \sigma_2 = \sigma'_2.\pi_2 \) such that \( \sigma \) is a join of \( \sigma'_1 \) and \( \sigma'_2 \) and either (1) \( x = \text{in} \)
  and \( \pi_1 = \langle a, \text{in} \rangle \) and \( \pi_2 = \langle a, \text{in} \rangle \), or (2) \( x = \text{out} \) and \( \pi_1 = \langle a, \text{out} \rangle \) and \( \pi_2 = \langle a, \text{out} \rangle \).

A run \( \sigma_1 \) of \( A_1 \) can be closed by \( A_2 \) if there is a run \( \sigma_2 \) of \( A_2 \) such that a join of \( \sigma_1 \) and \( \sigma_2 \) is closed.

**Theorem 5.1** Let \( P \) and \( Q \) be two compatible I/O-processes. Then \( \sigma \) is a run of \( P \| Q \)
leading to \( \langle p, q \rangle \in S(P) \times S(Q) \) iff there is a run \( \sigma_1 \) of \( P \) leading to \( p \) and a run \( \sigma_2 \) of \( Q \)
leading to \( q \) such that \( \sigma \) is a join of \( \sigma_1 \) and \( \sigma_2 \).

**Proof:** \((\Rightarrow)\) We do induction on the prefix partial order \( \prec \) over the set of runs of \( P \| Q \).

**Base step:** If \( \sigma = \epsilon \) is a run of \( P \| Q \) leading to initial state \( \langle p_0, q_0 \rangle \), then \( \sigma_1 = \epsilon \)
is a run of \( P \) leading to \( p_0 \) and \( \sigma_2 = \epsilon \) is a run of \( Q \) leading to \( q_0 \).

**Inductive step:** Let \( \sigma = \sigma'.\pi \) be a run of \( P \| Q \) leading to \( \langle p, q \rangle \). Then there is a state \( \langle p', q' \rangle \) of \( P \| Q \) such that \( \sigma' \) is a run leading to \( \langle p', q' \rangle \) and \( \langle \langle p', q' \rangle, \pi, \langle p, q \rangle \rangle \in T(P \| Q) \).
Let \( \pi = \langle a, x \rangle \). By inductive hypothesis, there is a run \( \sigma'_1 \) of \( P \) leading to \( p' \) and a run \( \sigma'_2 \)
 of \( Q \) leading to \( q \) such that \( \sigma' \) is a join of \( \sigma'_1 \) and \( \sigma'_2 \).

• \( a \in \text{Obs}(P) \cap \text{Obs}(Q) \)

**Case 1.** \( x = \text{in} \) and \( \langle p', \pi, p \rangle \in T(P) \) and \( \langle q', \pi, q \rangle \in T(Q) \)
Let \( \sigma_1 = \sigma'_1.\pi \) and \( \sigma_2 = \sigma'_2.\pi \).

**Case 2.** \( x = \text{out} \) and \( \langle p', \pi, p \rangle \in T(P) \) and \( \langle q', \langle a, \text{in} \rangle, q \rangle \in T(Q) \).
Let \( \sigma_1 = \sigma'_1.\pi \) and \( \sigma_2 = \sigma'_2.\langle a, \text{in} \rangle \).

**Case 3.** \( x = \text{out} \) and \( \langle q', \pi, q \rangle \in T(Q) \) and \( \langle p', \langle a, \text{in} \rangle, p \rangle \in T(P) \).
Let \( \sigma_1 = \sigma'_1.\langle a, \text{in} \rangle \) and \( \sigma_2 = \sigma'_2.\pi \).
\section*{Chapter 5. Verifying Parameterized Shared-Memory Systems}

\begin{itemize}
\item $a \in \text{Act}(P) \setminus \text{Act}(Q)$
  
  We have that $\langle p, \pi, p' \rangle \in T(P)$ and $q = q'$. Let $\sigma_1 = \sigma'_1, \pi$ and $\sigma_2 = \sigma'_2$.

\item $\pi \in \text{Act}(Q) \setminus \text{Act}(P)$
  
  We have that $p = p'$ and $\langle q, \pi, q' \rangle \in T(Q)$. Let $\sigma_1 = \sigma'_1$ and $\sigma_2 = \sigma'_2, \pi$.
\end{itemize}

In all cases, we get that $\sigma_1$ is a run of $P$ and $\sigma_2$ a run of $Q$ such that $\sigma$ is a join of $\sigma_1$ and $\sigma_2$.

$(\Leftarrow)$ We do induction on the prefix partial order $\prec$ over the set of joins of runs of $P$ and $Q$. In the following, let $\sigma$ be a join of a run $\sigma_1$ of $P$ leading to state $p$ and a run $\sigma_2$ of $Q$ leading to state $q$.

\textbf{Base step:} $\sigma = \epsilon$. Then $\sigma_1 = \sigma_2 = \epsilon$, and $p$ is an initial state of $P$ and $q$ is an initial state of $Q$. Clearly $\epsilon$ is a run of $P||Q$ leading to $\langle p, q \rangle$.

\textbf{Inductive step:} Let $\sigma = \sigma'_1, \pi$ and $\pi = \langle a, x \rangle$. We use the fact that $\sigma$ is a join of $\sigma_1$ and $\sigma_2$ in the arguments below.

\textbf{Case 1.} $a \in \text{Obs}(P) \cap \text{Obs}(Q)$.

Then $\sigma_1 = \sigma'_1, \pi_1$ and $\sigma_2 = \sigma'_2, \pi_2$. There is a state $p'$ of $P$ such that $\sigma'_1$ is a run of $P$ leading to $p'$ and $\langle p', \pi_1, p \rangle \in T(P)$, and there is a state $q'$ of $Q$ such that $\sigma'_2$ is a run of $Q$ leading to $q'$ and $\langle q', \pi_2, q \rangle \in T(Q)$. Since $\sigma'$ is a join of $\sigma'_1$ and $\sigma'_2$, we have from the induction hypothesis that $\sigma$ is a run of $P||Q$ leading to $\langle p', q' \rangle$. Moreover $\langle \langle p', q' \rangle, \langle a, x \rangle, \langle p, q \rangle \rangle \in T(P||Q)$. Therefore $\sigma$ is a run of $P||Q$ leading to $\langle p, q \rangle$.

\textbf{Case 2.} $a \in \text{Act}(P) \setminus \text{Act}(Q)$.

Then $\sigma_1 = \sigma'_1, \langle a, x \rangle$. There is a state $p'$ of $P$ such that $\sigma'_1$ is a run of $P$ leading to $p'$ and $\langle p', \langle a, x \rangle, p \rangle \in T(P)$. Since $\sigma'$ is a join of $\sigma'_1$ and $\sigma_2$, we have from the induction hypothesis that $\sigma$ is a run of $P||Q$ leading to $\langle p', q \rangle$. Moreover $\langle \langle p', q \rangle, \langle a, x \rangle, \langle p, q \rangle \rangle \in T(P||Q)$. Therefore $\sigma$ is a run of $P||Q$ leading to $\langle p, q \rangle$.

\textbf{Case 3.} $a \in \text{Act}(Q) \setminus \text{Act}(P)$.

Similar to Case 2.

\textbf{Corollary 5.2 Let $P$ and $Q$ be two compatible I/O-processes. Then $\sigma$ is a run of $P||Q$ iff there is a run $\sigma_1$ of $P$ and a run $\sigma_2$ of $Q$ such that $\sigma$ is a join of $\sigma_1$ and $\sigma_2$.}
Proof: The proof follows trivially from an application of Theorem 5.1.

**Theorem 5.3** Let $P$ and $Q$ be two compatible I/O-processes. Suppose $\sigma_1$ is a run of $P$ and $\sigma_2$ be a run of $Q$. Let $\tau_1 = tr_P(\sigma_1)$ and $\tau_2 = tr_Q(\sigma_2)$. Then the following statements are true.

- For all joins $\sigma$ of $\sigma_1$ and $\sigma_2$ there is a join $\tau$ of $\tau_1$ and $\tau_2$ such that $\tau = tr_P(\|Q(\sigma))$.
- For all joins $\tau$ of $\tau_1$ and $\tau_2$ there is a join $\sigma$ of $\sigma_1$ and $\sigma_2$ such that $\tau = tr_P(\|Q(\sigma))$.

Proof:

- We do induction on the prefix partial order $\prec$ over the set of joins of runs of $P$ and $Q$. Let $\sigma$ be a join of a run $\sigma_1$ of $P$ and a run $\sigma_2$ of $Q$. Let $\tau = tr_P(\|Q(\sigma))$, $\tau_1 = tr_P(\sigma_1)$ and $\tau_2 = tr_Q(\sigma_2)$.

**Base step:** $\sigma = \epsilon$. Then $\sigma_1 = \sigma_2 = \epsilon$. Therefore $\tau_1 = \tau_2 = \epsilon$ and $\tau = \epsilon$ is a join of $\tau_1$ and $\tau_2$.

**Inductive step:** Let $\sigma = \sigma'.\pi$ and $\pi = \langle a, x \rangle$. Let $\tau' = tr_P(\|Q(\sigma'))$. We use the fact that $\sigma$ is a join of $\sigma_1$ and $\sigma_2$ in the arguments below.

**Case 1.** $a \in \text{Obs}(P) \cap \text{Obs}(Q)$.

Then $\sigma_1 = \sigma'_1.\pi_1$ and $\sigma_2 = \sigma'_2.\pi_2$. Let $\tau'_1 = tr_P(\sigma'_1)$ and $\tau'_2 = tr_Q(\sigma'_2)$. We have that $\tau = \tau'.\pi$, $\tau_1 = \tau'_1.\pi_1$ and $\tau_2 = \tau'_2.\pi_2$. Since $\sigma'$ is a join of $\sigma'_1$ and $\sigma'_2$, we have from the induction hypothesis that $\tau'$ is a join of $\tau'_1$ and $\tau'_2$. Therefore $\tau$ is a join of $\tau_1$ and $\tau_2$.

**Case 2.** $a \in \text{Act}(P) \setminus \text{Act}(Q)$.

Then $\sigma_1 = \sigma'_1.\langle a, x \rangle$. Let $\tau'_1 = tr_P(\sigma'_1)$. Since $\sigma'$ is a join of $\sigma'_1$ and $\sigma_2$, we have from the induction hypothesis that $\tau'$ is a join of $\tau'_1$ and $\tau_2$. If $a \not\in \text{Obs}(P \| Q)$ we get that $\tau = \tau'$ and $\tau_1 = \tau'_1$. Therefore $\tau$ is a join of $\tau_1$ and $\tau_2$. If $a \in \text{Obs}(P \| Q)$ we get that $\tau = \tau'.\pi$ and $\tau_1 = \tau'_1.\pi$. Again we get that $\tau$ is a join of $\tau_1$ and $\tau_2$.

**Case 3.** $a \in \text{Act}(Q) \setminus \text{Act}(P)$.

Same as Case 2.

- We do induction on the prefix partial order $\prec$ over the set of joins of traces of $P$ and $Q$. Let $\tau$ be a join of a trace $\tau_1 = tr_{\sigma_1}(P)$ of $P$ and a trace $\tau_2 = tr_{\sigma_2}(Q)$ of $Q$. 


**Base step:** $\tau = \epsilon$. Then $\tau_1 = \tau_2 = \epsilon$ and consequently $\sigma_1 = \sigma_2 = \epsilon$. Therefore $\sigma = \epsilon$ is a join of $\sigma_1$ and $\sigma_2$ such that $\tau = tr_\sigma(P\parallel Q)$.

**Inductive step:** Let $\tau = \tau', \pi$ and $\pi = \langle a, x \rangle$. We use the fact that $\tau$ is a join of $\tau_1$ and $\tau_2$ in the arguments below.

**Case 1.** $a \in Obs(P) \cap Obs(Q)$.

Then $\tau_1 = \tau'_1, \pi_1$ and $\tau_2 = \tau'_2, \pi_2$. Let $\sigma'_1$ be the greatest prefix of $\sigma_1$ such that $\tau'_1 = tr_{\sigma'_1}(P)$. Then $\sigma_1 = \sigma'_1, \pi_1, \gamma_1$ where $\gamma_1$ is a sequence of extended private actions of $P$. Let $\sigma'_2$ be the greatest prefix of $\sigma_2$ such that $\tau'_2 = tr_{\sigma'_2}(P)$. Then $\sigma_2 = \sigma'_2, \pi_2, \gamma_2$ where $\gamma_2$ is a sequence of extended private actions of $Q$. Since $\tau'$ is a join of $\tau'_1$ and $\tau'_2$, we have from the induction hypothesis that there is a join $\sigma'$ of $\sigma'_1$ and $\sigma'_2$ such that $\tau' = tr_{\sigma'}(P\parallel Q)$. Let $\sigma = \sigma', \pi, \gamma_1, \gamma_2$. Then $\sigma$ is a join of $\sigma_1$ and $\sigma_2$ such that $\tau = tr_\sigma(P\parallel Q)$.

**Case 2.** $a \in Obs(P) \setminus Obs(Q)$.

Then $\tau_1 = \tau'_1, \pi$. Let $\sigma'_1$ be the greatest prefix of $\sigma_1$ such that $\tau'_1 = tr_{\sigma'_1}(P)$. Then $\sigma_1 = \sigma'_1, \pi_1, \gamma_1$ where $\gamma_1$ is a sequence of extended private actions of $P$. Since $\tau'$ is a join of $\tau'_1$ and $\tau_2$, we have from the induction hypothesis that there is a join $\sigma'$ of $\sigma'_1$ and $\sigma_2$ such that $\tau' = tr_{\sigma'}(P\parallel Q)$. Let $\sigma = \sigma', \pi, \gamma_1$. Then $\sigma$ is a join of $\sigma_1$ and $\sigma_2$ such that $\tau = tr_\sigma(P\parallel Q)$.

**Case 3.** $a \in Obs(Q) \setminus Obs(P)$.

Same as Case 2.

---

**Corollary 5.4** Let $P$ and $Q$ be two compatible I/O-processes. Then $\tau$ is a trace of $P\parallel Q$ iff there is a trace $\tau_1$ of $P$ and a trace $\tau_2$ of $Q$ such that $\tau$ is a join of $\tau_1$ and $\tau_2$.

**Proof:** Suppose $\tau$ is a trace of $P\parallel Q$. Then there is a run $\sigma$ of $P\parallel Q$ such that $\tau = tr_{P\parallel Q}(\sigma)$. From Theorem 5.1, we have that there are runs $\sigma_1$ of $P$ and $\sigma_2$ of $Q$ such that $\tau$ is a join of $\sigma_1$ and $\sigma_2$. Let $\tau_1 = tr_P(\sigma_1)$ and $\tau_2 = tr_Q(\sigma_2)$. From Theorem 5.3, we get that $\tau = tr_{P\parallel Q}(\sigma)$ is a join of a trace $\tau_1$ of $P$ and a trace $\tau_2$ of $Q$.

Suppose that $\tau$ is a join of a trace $\tau_1$ of $P$ and a trace $\tau_2$ of $Q$. Let $\tau_1 = tr_P(\sigma_1)$ and $\tau_2 = tr_Q(\sigma_2)$. From Theorem 5.3, there is a join $\sigma$ of $\sigma_1$ and $\sigma_2$ such that $\tau = tr_{P\parallel Q}(\sigma)$. Therefore $\tau$ is a trace of $P\parallel Q$. 

$\blacksquare$
CHAPTER 5. VERIFYING PARAMETERIZED SHARED-MEMORY SYSTEMS

Theorem 5.5 Suppose \( P \) and \( Q \) are I/O-processes and \( R \) is a I/O-process compatible with both \( P \) and \( Q \). If \( P \preceq Q \) then \( P \| R \preceq Q \| R \).

Proof: Since \( \text{Obs}(P) \subseteq \text{Obs}(Q) \), we have that \( \text{Obs}(P \| R) = \text{Obs}(P) \cup \text{Obs}(R) \subseteq \text{Obs}(Q) \cup \text{Obs}(R) = \text{Obs}(Q \| R) \). Suppose \( \tau \) is a trace of \( P \| R \). From Corollary 5.4 there is a trace \( \tau_1 \) of \( P \) and a trace \( \tau_2 \) of \( Q \) such that \( \tau \) is a join of \( \tau_1 \) and \( \tau_2 \). Since \( P \preceq Q \), we have that \( \tau_1 \) is a trace of \( Q \) as well. Therefore, from Corollary 5.4 we have that \( \tau \) is a trace of \( Q \| R \). \( \blacksquare \)

5.2 Parameterized memory systems

A parameterized memory system \( M \) has three parameters—the number \( n \) of processors, the number \( m \) of memory locations, and the number \( v \) of data values. The parameterized memory system \( M \) is built from two parameterized I/O-processes \( C \) and \( P \) which have two parameters—the number \( m \) of memory locations, and the number \( v \) of data values. Intuitively, the I/O-process \( P \) represents a single processor in the system and \( C \) represents a central controller. The I/O-process \( M(n, m, v) \) is built from the I/O-processes \( C(m, v) \) and \( P(m, v) \) by composing \( C(m, v) \) and \( n \) copies of \( P(m, v) \). Given \( n > 0, m > 0, \) and \( v > 0 \), the memory system \( M(n, m, v) \) is an I/O-process that has processors numbered from \( 0 \ldots n - 1 \), memory locations numbered from \( 0 \ldots m - 1 \), and data values numbered from \( 0 \ldots v - 1 \).

We now formally define a parameterized memory system. Let \( \mathbb{N} \) be the set of all non-negative integers. Let \( \mathbb{N}^+ \) be the set of all positive integers. For any \( k > 0 \), let \( \mathbb{N}_k \) denote the set of all non-negative integers less than \( k \). A parameterized I/O-process is a tuple \( \langle \text{PrivNames}_A, \text{ObsNames}_A, A \rangle \) such that

1. \( \text{PrivNames}_A \) and \( \text{ObsNames}_A \) are sets disjoint from each other, and

2. \( A \) is a function that maps \( \mathbb{N}^+ \times \mathbb{N}^+ \) to I/O-processes such that for all \( m > 0 \) and \( v > 0 \), we have that \( \text{Priv}(A(m, v)) = \text{PrivNames}_A \times \mathbb{N}_m \times (\mathbb{N}_v \cup \{ \bot \}) \) and \( \text{Obs}(A(m, v)) = \text{ObsNames}_A \times \mathbb{N}_m \times (\mathbb{N}_v \cup \{ \bot \}) \).

A parameterized memory system is a pair \( \langle C, P \rangle \) of parameterized I/O-processes satisfying the following properties.

1. \( \text{PrivNames}_C \cap ((\text{ObsNames}_P \cup \text{PrivNames}_P) \times \mathbb{N}^+) = \emptyset \), and

\( \text{ObsNames}_C \cap (\text{PrivNames}_P \times \mathbb{N}^+) = \emptyset \).
2. $R \in \text{PrivNames}_P$ and $W \in \text{PrivNames}_P$.

The functions $\text{name}$, $\text{loc}$, and $\text{val}$ are defined on $\text{Act}(C(m,v)) \cup \text{Act}(P(m,v))$, and extract respectively the first, second, and third components of the actions. Given some $m$ and $v$, let $\text{RdWr}(m,v)$ be the union of the set of read actions $\{\langle R,j,k \rangle | j < m \text{ and } k < v \}$ and the set of write actions $\{\langle W,j,k \rangle | j < m \text{ and } k < v \}$.

For all $m, v > 0$, and for all $k \geq 0$, let $P_k(m,v)$ denote the I/O-process that is obtained from $P(m,v)$ by renaming every private action $a$ to action $a'$, such that (1) $\text{name}(a')$ is the pair $\langle \text{name}(a), k \rangle$, (2) $\text{loc}(a') = \text{loc}(a)$, and (3) $\text{val}(a') = \text{val}(a)$. A parameterized memory system defines a function that maps $\mathbb{N} \times \mathbb{N}^+ \times \mathbb{N}^+$ to I/O-processes as follows:

$$M(0, m, v) = C(m, v)$$
$$M(n + 1, m, v) = M(n, m, v) || P_n(m, v)$$

For particular $n, m, v$, we say that $M(n, m, v)$ is a memory system. Note that $M(n, m, v)$ is compatible with $P_n(m, v)$, due to the renaming of private actions in $P_n(m, v)$, and the conditions on the names of private and observable actions of $C$ and $P$ described above. The observable actions of $M(n, m, v)$ are the same for all $n > 0$ and given by $\text{Obs}(C(m,v)) \cup \text{Obs}(P(m,v))$. We use $\text{Obs}_M$ to denote the set $\bigcup_{m,v} \text{Obs}(C(m,v)) \cup \text{Obs}(P(m,v))$. We define a function $\text{proc}$ on the set of actions $\bigcup_{n,m,v} \text{Priv}(P_k(m,v))$ such that if $a \in \text{Priv}(P_k(m,v))$, then $\text{proc}(a) = k$.

In order to reduce the proof of sequential consistency of the parameterized memory system to finite state model checking obligations, we make some assumptions about memory systems. We first state a few additional definitions. Let $\sigma$ be a run of the memory system $M(n, m, v)$. Let $R \subseteq \mathbb{N}_m$. We denote by $\sigma|_R$ the run $\sigma$ restricted to the memory locations in $R$. Formally, we have $\sigma|_R = [\sigma]_\beta$, where $\beta = \{a \mid a \in \text{Act}(M(n,m,v)) \text{ and } \text{loc}(a) \in R\}$. For $j < m$, we write $\sigma|_j$ for $\sigma|_{\{j\}}$. A function $\lambda : \mathbb{N}_k \rightarrow \mathbb{N}_k$ is called a $k$-map. A one-one $k$-map is called a $k$-permutation. A function $\lambda : \mathbb{N}_k \cup \{\bot\} \rightarrow \mathbb{N}_k \cup \{\bot\}$ such that $\lambda$ restricted to $\mathbb{N}_k$ is a $k$-map and $\lambda(\bot) = \bot$ is called a $\bot$-extended $k$-map. Let $M(n, m, v)$ be a memory system. If $\lambda$ is a $m$-permutation then $\text{loc}_\lambda$ is defined to be a function from $\text{Act}(M(n, m, v))$ to $\text{Act}(M(n, m, v))$ such that if $a = \langle \text{name}, j, k \rangle$ then $\text{loc}_\lambda(a) = \langle \text{name}, \lambda(j), k \rangle$. If $\lambda$ is a $\bot$-extended $\nu$-map then $\text{val}_\lambda$ is defined to be a function from $\text{Act}(M(n, m, v))$ to $\text{Act}(M(n, m, v))$ such that if $a = \langle \text{name}, j, k \rangle$ then $\text{val}_\lambda(a) = \langle \text{name}, j, \lambda(k) \rangle$. $\text{loc}_\lambda$ and $\text{val}_\lambda$ can be extended to action sequences, extended actions and extended action sequences in the natural way.
Assumption 5.1 (Location projectibility)

1. If $\sigma \in \Sigma(C(m,v))$, then for all $j \leq m$, we have $\sigma|_{\mathbb{N}_j} \in \Sigma(C(j,v))$.

2. If $\sigma \in \Sigma(P(m,v))$, then for all $j \leq m$, we have $\sigma|_{\mathbb{N}_j} \in \Sigma(P(j,v))$.

Assumption 5.2 (Location symmetry)
Let $\lambda$ be an $m$-permutation. Then

1. for all $\sigma \in \Sigma(C(m,v))$, we have that $\text{loc}_\lambda(\sigma) \in \Sigma(C(m,v))$, and

2. for all $\sigma \in \Sigma(P(m,v))$, we have that $\text{loc}_\lambda(\sigma) \in \Sigma(P(m,v))$.

Assumption 5.3 (Location monotonicity)
For all $n, v, m_1, m_2$, if $m_1 \leq m_2$, then

1. for all $\sigma \in \text{Act}(C(m_1,v))^*$, we have $\sigma \in \Sigma(C(m_1,v))$ iff $\sigma \in \Sigma(C(m_2,v))$, and

2. for all $\sigma \in \text{Act}(P(m_1,v))^*$, we have $\sigma \in \Sigma(P(m_1,v))$ iff $\sigma \in \Sigma(P(m_2,v))$.

Assumption 5.4 (Data symmetry)
Let $\lambda$ be a $\perp$-extended $v$-map. Then

1. for all $\sigma \in \Sigma(C(m,v))$, we have that $\text{val}_\lambda(\sigma) \in \Sigma(C(m,v))$, and

2. for all $\sigma \in \Sigma(P(m,v))$, we have that $\text{val}_\lambda(\sigma) \in \Sigma(P(m,v))$.

Assumption 5.5 (Data monotonicity)
For all $n, v_1, v_2$, if $v_1 \leq v_2$, then

1. for all $\sigma \in \text{Act}(C(m,v_1))^*$, we have $\sigma \in \Sigma(C(m,v_1))$ iff $\sigma \in \Sigma(C(m,v_2))$, and

2. for all $\sigma \in \text{Act}(P(m,v_1))^*$, we have $\sigma \in \Sigma(P(m,v_1))$ iff $\sigma \in \Sigma(P(m,v_2))$.

5.3 Sequential consistency

Let $K$ be a set. Let $G = \langle K, E \rangle$ be a directed graph over $K$ and $P \subseteq K$. The $P$-restriction of $G$ is the directed graph $\langle P, E \cap P \times P \rangle$ and is denoted by $G|_P$. Let $G = \langle K, E \rangle$ be a graph over $K$ and $G' = \langle K', E' \rangle$ be a graph over $K'$. We say that $G$ is isomorphic to $G'$ if there is a one-one onto function $f : K \rightarrow K'$ such that for all $x, y \in K$, we have that $\langle x, y \rangle \in E$ iff $\langle f(x), f(y) \rangle \in E'$. We write $G \cong G'$ if $G$ is isomorphic to $G'$. If $K = K'$, we define the union of $G$ and $G'$ to be $\langle K, E \cup E' \rangle$. We write $G \cup G'$ for the union of $G$ and $G'$. Let $G_1$ and $G_2$ be graphs over $K$ and let $P \subseteq K$. Then we have that $(G_1 \cup G_2)|_P = G_1|_P \cup G_2|_P$. Suppose $G_1'$ and $G_2'$ are graphs over $K'$, and $G_1$ and $G_2$
CHAPTER 5. VERIFYING PARAMETERIZED SHARED-MEMORY SYSTEMS

are isomorphic to $G_1'$ and $G_2'$ respectively. Then we have that $G_1 \cup G_2$ is isomorphic to $G_1' \cup G_2'$. Suppose $G = \langle K, E \rangle$ is an acyclic directed graph over $K$. Then we can obtain total orderings of the vertices in $G$ that respect the dependencies specified by its edges. Since the edges form a partial order, several such total orders may exist. Formally, a one-one function $f : K \to \mathbb{N}_{|K|}$ is a good order of $G$ if for all $x, y \in K$, if $\langle x, y \rangle \in E$ then $f(x) < f(y)$.

A function $F$ from $\mathbb{N}_r$ to $\mathcal{P}(K)$ is an $r$-partition of $K$ if (1) $\bigcup_{i < r} F(i) = K$, and (2) for all $i, j < r$, we have that $F(i)$ and $F(j)$ are disjoint. Let $F$ be an $r$-partition of $K$ and let $R \subseteq \mathbb{N}_r$. Then $A$ is the function such that $A(F, R) = \bigcup_{i \in R} F(i)$. Let $T$ be a function from $\mathbb{N}_r$ to $\mathcal{P}(K \times K)$. The tuple $\langle F, T \rangle$ is called an ordered $r$-partition of $K$ if (1) $F$ is an $r$-partition of $K$, and (2) for all $i < r$, we have that $T(i) \subseteq F(i) \times F(i)$ and $T(i)$ is a total order of $F(i)$. For any $R \subseteq \mathbb{N}_r$, let $A(\langle F, T \rangle, R) = A(F, R)$. If $\langle F, T \rangle$ is an ordered $r$-partition of $K$ then $G(\langle F, T \rangle)$ is defined to be the directed graph $\langle K, \bigcup_{i < r} T(i) \rangle$.

Let $Memop(n, m, v)$ be the union of the sets $\{\langle R, i \rangle, j, k \rangle | i < n \text{ and } j < m \text{ and } k < v\}$ and $\{\langle W, i \rangle, j, k \rangle | i < n \text{ and } j < m \text{ and } k < v\}$. Thus $Memop(n, m, v)$ denotes the set of read and write operations of $M(n, m, v)$. The functions name, loc, and val, which were originally defined on actions of $P(m, v)$ and $C(m, v)$, can be defined analogously on actions of $M(n, m, v)$. Thus, the four functions name, loc, val, and proc are defined on all members of $Memop(n, m, v)$. We use Memop to denote the set $\bigcup_{n, m, v} Memop(n, m, v)$.

Let $\sigma = \pi_0, \pi_1, \ldots, \pi_{k-1}$ be a sequence in $Memop(n, m, v)^*$, the set of finite sequences with elements from $Memop(n, m, v)$. Define $h'$ to be the function where $h'(\sigma)$ is an ordered $n$-partition $\langle F, T \rangle$ of $\mathbb{N}_k$ such that for all $i < n$, (1) $F(i) = \{x < k | \text{proc}(\pi_x) = i\}$, and (2) for all $x, y \in F(i)$, we have that $\langle x, y \rangle \in T(i)$ iff $x < y$. We extend $h'$ to operate on arbitrary sequences $\sigma$ by first restricting it to actions in $Memop$. Formally, for any $\sigma$, we have that $h'(\sigma) = h'(\[\sigma]\_\text{Memop})$. We extend $h'$ to operate on sequences of extended actions by operating it on the first component of each extended action. Formally, if $\sigma$ is a sequence of extended actions, then $h'(\sigma) = h'(\text{First}(\sigma))$. We observe that for all runs $\sigma$ of $M(n, m, v)$, the directed graph $G(h'(\sigma))$ is acyclic.

We are interested in defining which sequences from $\text{Memop}^*$ are serial. Intuitively, a sequence from $\text{Memop}^*$ is serial if it can be produced by serial memory where each read from a location returns the value written by the last write to that location. We state this formally below. Let $\sigma = \pi_0, \pi_2, \ldots, \pi_{k-1}$ be a sequence in $Memop^*$. We define $lw_\sigma$ as a function that associates with each position $i$ in $\sigma$, the position $j$ in $\sigma$ where the most recent write to the location $\text{loc}(\pi_i)$ was done. Let $WS_\sigma(j)$ be the set of numbers
\textbf{CHAPTER 5. VERIFYING PARAMETERIZED SHARED-MEMORY SYSTEMS} 112

\textit{i} such that \( \pi_i \) is a write event to the location of the event \( \pi_j \). Formally, we have that \( W S_\sigma(j) = \{ i \leq j \mid \text{loc}(\pi_i) = \text{loc}(\pi_j) \text{ and } \exists n_1. \text{name}(\pi_i) = (W, n_1) \} \). Then \( lw_\sigma \) is a mapping from the set \( \{0, 1, \ldots, k-1\} \) to \( \{0, 1, \ldots, k-1\} \cup \{\bot\} \) defined as follows.

\[
lw_\sigma(j) = \begin{cases} 
\max(W S_\sigma(j)), & \text{if } W S_\sigma(j) \neq \emptyset \\
\bot, & \text{otherwise.}
\end{cases}
\]

The sequence \( \sigma \) is \textit{serial} if the following conditions are satisfied.

1. For all \( i < k \) and \( j < k \), if \( lw_\sigma(i) = lw_\sigma(j) = \bot \), then \( \text{val}(\pi_i) = \text{val}(\pi_j) \).

2. For all \( i < k \), if \( lw_\sigma(i) \neq \bot \), then \( \text{val}(\pi_i) = \text{val}(\pi_{lw_\sigma(i)}) \).

Let \( \sigma = \pi_0, \pi_1, \ldots, \pi_{k-1} \) be a sequence in \( \text{Memop}^* \). If \( f \) is a \( k \)-permutation then \( \pi_{f(0)}, \pi_{f(1)}, \ldots, \pi_{f(k-1)} \) is called a \textit{permutation of} \( \sigma \). If \( f \) is a good order of \( \mathcal{G}(h'(\sigma)) \) then the sequence \( \pi_{f^{-1}(0)}, \pi_{f^{-1}(1)}, \ldots, \pi_{f^{-1}(k-1)} \) is called a \textit{linearization of} \( \sigma \). Let \( M \) be a parameterized memory system and define \( \Sigma_M \) to be \( \bigcup_{n,m,v} \Sigma(M(n,m,v)) \). A function \( \Omega \) from \( \Sigma_M \) to \( \text{Memop}^* \) is called an \textit{observer} for \( M \) if \( \Omega(\sigma) \) is a permutation of \( [\sigma]_{\text{Memop}} \) for all \( \sigma \in \Sigma_M \).

\textbf{Definition 5.1 (Serializer)} Let \( M \) be a parameterized memory system and let \( \Omega \) be an observer for \( M \). Then \( \Omega \) is a \textit{serializer} for \( M(n,m,v) \) if for every run \( \sigma \in \Sigma(M(n,m,v)) \), the sequence \( \Omega(\sigma) \) is both serial and a linearization of \( \sigma \).

\textbf{Definition 5.2 (Sequential consistency [Lam79])} Let \( M \) be a parameterized memory system. The memory system \( M(n,m,v) \) is \textit{sequentially consistent} if it has a serializer. The parameterized memory system \( M \) is \textit{sequentially consistent} if \( M(n,m,v) \) is sequentially consistent for all \( n > 0, m > 0, \text{ and } v > 0 \).

We define below three restrictions on an observer \( \Omega \).

- \( \Omega \) is \textit{local} on \( M(n,m,v) \) if for all \( \sigma \in \Sigma(M(n,m,v)) \) and for all \( j < m \), we have that \( \Omega(\sigma_{\mid j}) = \Omega(\sigma)_{\mid j} \). We say that \( \Omega \) is local if \( \Omega \) is local on \( M(n,m,v) \) for all \( n, m \) and \( v \).

- \( \Omega \) is \textit{location symmetric} on \( M(n,m,v) \) if for all \( \sigma \in \Sigma(M(n,m,v)) \) and for every \( m \)-permutation \( \lambda \), we have that \( \Omega(\text{loc}_\lambda(\sigma)) = \text{loc}_\lambda(\Omega(\sigma)) \). We say that \( \Omega \) is location symmetric if \( \Omega \) is location symmetric on \( M(n,m,v) \) for all \( n, m \) and \( v \).
• \( \Omega \) is data symmetric on \( M(n, m, v) \) if for all \( \sigma \in \Sigma(M(n, m, v)) \) and for every \(-\) extended \( r \)-map \( \lambda \), we have that \( \Omega(\operatorname{val}_\lambda(\sigma)) = \operatorname{val}_{\lambda}(\Omega(\sigma)) \). We say that \( \Omega \) is data symmetric if \( \Omega \) is data symmetric on \( M(n, m, v) \) for all \( n, m \) and \( v \).

We now prove a theorem about directed graphs that is essential for the results in Sections 5.4 and 5.5.

**Theorem 5.6** Let \( K \) be a set. Let \( \langle F, T \rangle \) be an ordered \( r \)-partition and \( \langle F', T' \rangle \) be an ordered \( s \)-partition of \( K \) for some \( r \) and \( s \). Suppose for all \( R \subseteq N_r \) such that \( |R| \leq s \) the \( \mathcal{A}(F, R) \)-restriction of \( \mathcal{G}(\langle F, T \rangle) \cup \mathcal{G}(\langle F', T' \rangle) \) is acyclic. Then \( \mathcal{G}(\langle F, T \rangle) \cup \mathcal{G}(\langle F', T' \rangle) \) is acyclic.

**Proof:** Let \( G \) be the graph \( \mathcal{G}(\langle F, T \rangle) \cup \mathcal{G}(\langle F', T' \rangle) \). We will prove that \( G \) is acyclic.

Suppose \( r \leq s \). Since \( N_r \subseteq N_r \) and \( |N_r| = r \leq s \), we get that the \( \mathcal{A}(F, N_r) \)-restriction of \( G \) is acyclic. But \( \mathcal{A}(F, N_r) = K \) and the \( K \)-restriction of \( G \) is the same as \( G \). Therefore \( G \) is acyclic.

Suppose \( r > s \). Let \( R = \{ i_0, i_1, \ldots, i_{k-1} \} \subseteq N_r \). Let \( \oplus \) and \( \ominus \) denote addition modulo-\( k \) and subtraction modulo-\( k \) respectively. We define the notion of an \( R \)-cycle of the graph \( G \). The sequence \( x_0, y_0, x_1, y_1, \ldots, x_{k-1}, y_{k-1} \) of vertices in the graph \( G \) is said to be an \( R \)-cycle of \( G \) if for all \( j < k \) we have that (1) \( \langle x_j, y_j \rangle \in T(i_j) \), and (2) \( \langle y_j, x_{j+1} \rangle \in T'(p) \) for some \( p < s \). It is not too hard to see that if \( G \) has a cycle then it has an \( R \)-cycle for some \( R \subseteq N_r \). First, we show that \( G \) does not have an \( R \)-cycle for any \( |R| \leq s \). If it does then the \( \mathcal{A}(F, R) \)-restriction of \( G \) has a cycle too, which is a contradiction. Second, we show that if \( G \) has an \( R \)-cycle then it has an \( R' \)-cycle for some \( R' \subseteq R \). Let \( R = \{ i_0, i_1, \ldots, i_{k-1} \} \) be such that \( |R| = k > s \), and suppose \( G \) has an \( R \)-cycle \( x_0, y_0, x_1, y_1, \ldots, x_{k-1}, y_{k-1} \). Since \( k > s \) there is some \( p < s \) and \( a, b < k \) such that (1) \( x_a \in F'(p) \), (2) \( y_b \in F'(p) \), and (3) \( a < b < a + s \). We have that \( \langle y_b, x_a \rangle \notin T'(p) \) otherwise \( G \) has an \( R \)-cycle where \( |R| \leq s \), which is a contradiction. Since \( T'(p) \) is a total order on \( F'(p) \), we get that \( \langle x_a, y_b \rangle \notin T'(p) \).

Moreover \( \langle y_{a+1}, x_a \rangle \in T'(p) \) and \( \langle y_b, x_{b+1} \rangle \in T'(p) \). Since \( T'(p) \) is transitive we have that \( \langle y_{a+1}, x_{b+1} \rangle \in T'(p) \). Let \( R' = R \setminus \{ i_a, \ldots, i_b \} \). We can now short-circuit the \( R \)-cycle to get the \( R' \)-cycle \( \langle x_0, y_0 \rangle, \ldots, \langle x_{a+1}, y_{a+1} \rangle, \langle x_{b+1}, y_{b+1} \rangle, \ldots, \langle x_{k-1}, y_{k-1} \rangle \). A simple induction will show that \( G \) does not have an \( R \)-cycle for any \( R \subseteq N_r \). Therefore \( G \) is acyclic. \( \square \)
5.4 Parameterized memory systems with fixed number of processors

In this section, we are concerned with parameterized shared memory systems in which the number of processors is fixed, for example at \( n \). We will show that demonstrating sequential consistency by means of a local, location-symmetric and data-symmetric observer for \( n \) memory locations and 2 data values is sufficient to prove sequential consistency for any number of locations and values.

Let \( \Omega \) be an observer for the parameterized memory system \( M \). Let \( \sigma \) be a run of \( M(n, m, v) \) and let \( [\sigma]_{\text{Memop}(n,m,v)} = \pi_0, \ldots, \pi_{k-1} \). Let \( \Omega(\sigma) = \pi_{f^{-1}(0)}, \ldots, \pi_{f^{-1}(k-1)} \) for some \( k \)-permutation \( f \). Let \( h^\Omega \) be the function where \( h^\Omega(\sigma) \) is an ordered \( m \)-partition \( (F', T') \) of \( N_k \) such that for all \( j < m \), (1) \( F'(j) = \{ x \in K | \text{loc}(\pi_x) = j \} \), and (2) for all \( x, y \in F(i) \), we have that \( \langle x, y \rangle \in T'(i) \) iff \( f(x) < f(y) \). Let \( \Lambda^\Omega \) be the function on \( \Sigma(M(n, m, v)) \) such that \( \Lambda^\Omega(\sigma) = G(h^\sigma) \cup G(h^\Omega(\sigma)) \).

**Lemma 5.7** Let \( \Omega \) be a local observer for the parameterized memory system \( M \). Let \( \sigma \) be a run of \( M(n, m, v) \) and \( R \) be a subset of \( \mathbb{N}_m \). Then \( \Lambda^\Omega(\sigma|R) \) is isomorphic to \( \Lambda^\Omega(\sigma)\rceil_{\mathcal{A}(h^\Omega(\sigma)|R)} \).

**Proof:** Let \( [\sigma]_{\text{Memop}(n,m,v)} = \pi_0, \pi_1, \ldots, \pi_{a-1} \). Then \( G(h^\sigma) = \langle \mathbb{N}_a, E_1 \rangle \) and \( G(h^\Omega(\sigma)) = \langle \mathbb{N}_a, E_2 \rangle \) for some \( E_1, E_2 \subseteq \mathbb{N}_a \times \mathbb{N}_a \). We have that \( [\sigma|R]_{\text{Memop}(n,m,v)} = ([\sigma]_{\text{Memop}(n,m,v)})|_R \). Let \( i_0 < i_1 < \ldots < i_{b-1} < a \) be such that 
\[
[\sigma|R]_{\text{Memop}(n,m,v)} = ([\sigma]_{\text{Memop}(n,m,v)})|_R = \pi_{i_0}, \pi_{i_1}, \ldots, \pi_{i_{b-1}}.
\]

Then \( G(h'(\sigma|R)) = \langle \mathbb{N}_b, F_1 \rangle \) and \( G(h^\Omega(\sigma|R)) = \langle \mathbb{N}_b, F_2 \rangle \) for some \( F_1, F_2 \subseteq \mathbb{N}_b \times \mathbb{N}_b \). Also \( \mathcal{A}(h^\Omega(\sigma), R) = \{ i_0, i_1, \ldots, i_{b-1} \} \). Therefore, we have that 
\[
\begin{align*}
G(h'(\sigma)|_{\mathcal{A}(h^\Omega(\sigma), R)}) &= \langle \mathcal{A}(h^\Omega(\sigma), R), E_1 \cap \mathcal{A}(h^\Omega(\sigma), R) \times \mathcal{A}(h^\Omega(\sigma), R) \rangle, \text{ and} \\
G(h^\Omega(\sigma)|_{\mathcal{A}(h^\Omega(\sigma), R)}) &= \langle \mathcal{A}(h^\Omega(\sigma), R), E_2 \cap \mathcal{A}(h^\Omega(\sigma), R) \times \mathcal{A}(h^\Omega(\sigma), R) \rangle.
\end{align*}
\]

Let \( \alpha : \mathbb{N}_b \rightarrow \mathcal{A}(h^\Omega(\sigma), R) \) be the one-one strictly monotonic function such that \( \alpha(j) = i_j \) for all \( j < b \). Then \( [\sigma|R]_{\text{Memop}(n,m,v)} = \pi_{\alpha(0)}, \pi_{\alpha(1)}, \ldots, \pi_{\alpha(b-1)} \).

Let \( \Omega(\sigma) = \pi_{f^{-1}(0)}, \pi_{f^{-1}(1)}, \ldots, \pi_{f^{-1}(a-1)} \) for some \( a \)-permutation \( f \). Let \( \Omega(\sigma|R) = \pi_{\alpha(g^{-1}(0))}, \pi_{\alpha(g^{-1}(1))}, \ldots, \pi_{\alpha(g^{-1}(b-1))} \) for some \( b \)-permutation \( g \). Let \( p \) be a memory location in \( R \). Since \( \Omega \) is local, we have that \( \Omega(\sigma)|_p = \Omega(\sigma|_p) = \Omega(\sigma|R)|_p \). Suppose
\[ \Omega(\sigma|R)|_p = \pi_{\alpha(g^{-1}(x_0))}, \pi_{\alpha(g^{-1}(x_1))}, \ldots, \pi_{\alpha(g^{-1}(x_{k-1}))}, \text{ where } x_0 < x_1 < \ldots < x_{k-1} < b. \]

Suppose \( \Omega(\sigma)|_p = \pi_{f^{-1}(y_0)}, \pi_{f^{-1}(y_1)}, \ldots, \pi_{f^{-1}(y_{k-1})} \), where \( y_0 < y_1 < \ldots < y_{k-1} < a \). Then \( \alpha(g^{-1}(x_i)) = f^{-1}(y_i) \) for all \( i < k \). In other words, we have that \( y_i = f(\alpha(g^{-1}(x_i))) \) for all \( i < k \). Moreover, for all \( r,s < k \) we have that \( x_r < x_s \) iff \( y_r < y_s \). Let \( x, y < b \) be such that \( \text{loc}(x) = \text{loc}(y) = p \). We show that \( g(x) < g(y) \) iff \( f(\alpha(x)) < f(\alpha(y)) \). Suppose \( x = g^{-1}(x_r) \) and \( y = g^{-1}(x_s) \) for some \( x_r, x_s < b \). Then \( g(x) < g(y) \) iff \( x_r < x_s \) iff \( y_r < y_s \) iff \( f(\alpha(g^{-1}(x_r))) < f(\alpha(g^{-1}(x_s))) \) iff \( f(\alpha(x)) < f(\alpha(y)) \).

For all \( x \in \mathbb{N}_n \), we have that \( x \in \mathbb{N}_n \) iff \( \alpha(x) \in \mathcal{A}(h^\Omega(\sigma), R) \). We have that \( \langle x, y \rangle \in F_1 \) iff \( \text{proc}(x) = \text{proc}(y) \) and \( g(x) < g(y) \) iff \( \text{proc}(x) = \text{proc}(y) \) and \( f(\alpha(x)) < f(\alpha(y)) \) iff \( \langle x, y \rangle \in E_1 \cap \mathcal{A}(h^\Omega(\sigma), R) \times \mathcal{A}(h^\Omega(\sigma), R) \). Therefore \( \alpha \) is an isomorphism from \( G(h'(\sigma)|_R) \) to \( G(h'(\sigma))|_{\mathcal{A}(h^\Omega(\sigma), R)} \). We have that \( \langle x, y \rangle \in F_2 \) iff \( \text{loc}(x) = \text{loc}(y) \) and \( g(x) < g(y) \) iff \( \text{loc}(x) = \text{loc}(y) \) and \( f(\alpha(x)) < f(\alpha(y)) \) iff \( \langle x, y \rangle \in E_2 \cap \mathcal{A}(h^\Omega(\sigma), R) \times \mathcal{A}(h^\Omega(\sigma), R) \). Therefore \( \alpha \) is an isomorphism from \( G(h(\sigma)|_R) \) to \( G(h(\sigma))|_{\mathcal{A}(h^\Omega(\sigma), R)} \). We get that

\[
\begin{align*}
\Lambda^\Omega(\sigma)|_{\mathcal{A}(h^\Omega(\sigma), R)} & = (G(h'(\sigma)) \cup G(h^\Omega(\sigma)))|_{\mathcal{A}(h^\Omega(\sigma), R)} \\
& = G(h'(\sigma))|_{\mathcal{A}(h^\Omega(\sigma), R)} \cup G(h^\Omega(\sigma))|_{\mathcal{A}(h^\Omega(\sigma), R)} \\
& \approx G(h'(\sigma)|_R) \cup G(h(\Omega(\sigma)|_R))) \\
& = \Lambda^\Omega(\sigma)|_R.
\end{align*}
\]

Thus \( \Lambda^\Omega(\sigma)|_{\mathcal{A}(h^\Omega(\sigma), R)} \) is isomorphic to \( \Lambda^\Omega(\sigma)|_R \).

\[\text{Lemma 5.8} \quad \text{Let } \Omega \text{ be a location symmetric observer for the parameterized memory system } M. \text{ Let } \sigma \text{ be a run of } M(n, m, v), R \text{ be a subset of } \mathbb{N}_m \text{ and } \lambda \text{ be an } m\text{-permutation. Then } \Lambda^\Omega(\text{loc}_\lambda(\sigma))|_{\mathcal{A}(h^\Omega(\lambda(\sigma)), \lambda(R))} \text{ is isomorphic to } \Lambda^\Omega(\sigma)|_{\mathcal{A}(h^\Omega(\sigma), R)}. \]

\[\text{Proof:} \quad \text{Let } [\sigma]_{\text{Memop}(n, m, v)} = \pi_0, \pi_1, \ldots, \pi_{a-1}. \text{ Then } G(h'(\sigma)) = \langle N_n, E_1 \rangle \text{ and } G(h^\Omega(\sigma)) = \langle N_n, E_2 \rangle \text{ for some } E_1, E_2 \subseteq N_n \times N_n. \text{ Let } \pi'_i = \text{loc}_\lambda(\pi_i). \text{ Therefore } [\text{loc}_\lambda(\sigma)]_{\text{Memop}(n, m, v)} = \pi'_0, \pi'_1, \ldots, \pi'_{a-1}. \text{ Then } G(h'(\text{loc}_\lambda(\sigma))) = \langle N_n, F_1 \rangle \text{ and } G(h^\Omega(\text{loc}_\lambda(\sigma))) = \langle N_n, F_2 \rangle \text{ for some } F_1, F_2 \subseteq N_n \times N_n. \text{ Then } x \in \mathcal{A}(h(\Omega(\sigma), R) \text{ iff } \text{loc}(x) \in R \text{ iff } \text{loc}(\pi_x) = \text{loc}(\pi'_x) \in \lambda(R) \text{ iff } x \in \mathcal{A}(h(\Omega(\text{loc}_\lambda(\sigma)))) \lambda(R)). \text{ Therefore } \mathcal{A}(h(\Omega(\sigma)), R) = \mathcal{A}(h(\Omega(\text{loc}_\lambda(\sigma)))) \lambda(R)). \text{ We} \]
denote the set $\mathcal{A}(h(\Omega(\sigma)), R)$ by $K$. Therefore, we have that
\[
\mathcal{G}(h'(\sigma))|_{\mathcal{A}(h(\Omega(\sigma)), R)} = (K, E_1 \cap K \times K),
\]
\[
\mathcal{G}(h^\Omega(\sigma))|_{\mathcal{A}(h(\Omega(\sigma)), R)} = (K, E_2 \cap K \times K),
\]
\[
\mathcal{G}(h'(\text{loc}_\lambda(\sigma)))|_{\mathcal{A}(h(\Omega(\lambda^s(\sigma), \lambda(R))), \lambda(R))} = (K, F_1 \cap K \times K),
\]
\[
\mathcal{G}(h^\Omega(\text{loc}_\lambda(\sigma)))|_{\mathcal{A}(h(\Omega(\lambda^s(\sigma), \lambda(R))), \lambda(R))} = (K, F_2 \cap K \times K).
\]

Let $\Omega(\sigma) = \pi_f^{-1}(0), \ldots, \pi_f^{-1}(a-1)$ for some $a$-permutation $f$. Then $\text{loc}_\lambda(\Omega(\sigma)) = \pi_g^{-1}(0), \ldots, \pi_g^{-1}(a-1)$ for some $a$-permutation $g$. Since $\Omega$ is location symmetric $\Omega(\text{loc}_\lambda(\sigma)) = \text{loc}_\lambda(\Omega(\sigma))$. We get that $f^{-1}(i) = g^{-1}(i)$ for all $i < a$. Since $f$ and $g$ are one-one functions, we get that $g(i) = f(i)$ for all $i < a$.

Let $x, y \in K$. We have that $\langle x, y \rangle \in E_1 \cap K \times K$ iff $\text{proc}(\pi_x) = \text{proc}(\pi_y)$ and $g(x) < g(y)$ iff $\text{proc}(\pi_x) = \text{proc}(\pi_y)$ and $f(x) < f(y)$ iff $\langle x, y \rangle \in E_1 \cap K \times K$. Therefore the identity function from $K$ to $K$ is an isomorphism from $\mathcal{G}(h'(\text{loc}_\lambda(\sigma)))|_{\mathcal{A}(h(\Omega(\lambda^s(\sigma)), \lambda(R)))}$ to $\mathcal{G}(h'^(\sigma))|_{\mathcal{A}(h(\Omega(\sigma)), R)}$. We also have that $\langle x, y \rangle \in E_2$ iff $\text{loc}(\pi_x) = \text{loc}(\pi_y)$ and $g(x) < g(y)$ iff $\text{loc}(\pi_x) = \text{loc}(\pi_y)$ and $f(x) < f(y)$ iff $\langle x, y \rangle \in E_2 \cap K \times K$. Therefore the identity function from $K$ to $K$ is an isomorphism from $\mathcal{G}(h^\Omega(\text{loc}_\lambda(\sigma)))|_{\mathcal{A}(h(\Omega(\lambda^s(\sigma)), \lambda(R)))}$ to $\mathcal{G}(h^\Omega(\sigma))|_{\mathcal{A}(h(\Omega(\sigma)), R)}$. We get that
\[
\Lambda^\Omega(\text{loc}_\lambda(\sigma))|_{\mathcal{A}(h(\Omega(\lambda^s(\sigma)), \lambda(R)))} = (\mathcal{G}(h'(\text{loc}_\lambda(\sigma))) \cup \mathcal{G}(h^\Omega(\text{loc}_\lambda(\sigma))))|_{\mathcal{A}(h(\Omega(\lambda^s(\sigma)), \lambda(R)))}
\]
\[
\simeq (\mathcal{G}(h'(\text{loc}_\lambda(\sigma)))|_{\mathcal{A}(h(\Omega(\lambda^s(\sigma)), \lambda(R)))}) \cup (\mathcal{G}(h^\Omega(\text{loc}_\lambda(\sigma)))|_{\mathcal{A}(h(\Omega(\lambda^s(\sigma)), \lambda(R)))})
\]
\[
\simeq \mathcal{G}(h'(\sigma))|_{\mathcal{A}(h(\Omega(\sigma)), R)} \cup \mathcal{G}(h^\Omega(\sigma))|_{\mathcal{A}(h(\Omega(\sigma)), R)}
\]
\[
\simeq \Lambda^\Omega(\sigma)|_{\mathcal{A}(h(\Omega(\sigma)), R)}.
\]

Thus $\Lambda^\Omega(\text{loc}_\lambda(\sigma))|_{\mathcal{A}(h(\Omega(\lambda^s(\sigma)), \lambda(R)))}$ is isomorphic to $\Lambda^\Omega(\sigma)|_{\mathcal{A}(h(\Omega(\sigma)), R)}$. 

\textbf{Lemma 5.9} Suppose the parameterized memory system $M = \langle C, P \rangle$ satisfies Assumptions 5.1, 5.2 and 5.3. For all $n > 0$ and $v > 0$, if there is a local and location symmetric serializer for $M(n, n, v)$, then for all $m > 0$, there is a serializer for $M(n, m, v)$.

\textbf{Proof}: Fix some $n > 0$ and $v > 0$. Suppose $\Omega$ is a local and location symmetric serializer for $M(n, n, v)$. Fix some $m > 0$. We now show how to define a serializer $\Omega'$ for $M(n, m, v)$. Let $\sigma \in \Sigma(M(n, m, v))$.

Fix $p < m$ and let $\lambda$ be the $m$-permutation such that $\lambda(p) = 0$, $\lambda(0) = p$, and $\lambda(j) = j$ if $j$ is neither 0 nor $p$. We show that $\Omega(\sigma)|_p$ is serial. We have that $\text{loc}_\lambda(\sigma)|_p$ is a
run of $M(n, 1, v)$ and therefore of $M(n, n, v)$. Since $\Omega$ is a serializer for $M(n, n, v)$, we have
that $\Omega(\text{loc}_\lambda(\sigma)|_p)$ is serial.

$$\Omega(\text{loc}_\lambda(\sigma)|_0) = \Omega(\text{loc}_\lambda(\sigma)|_0) \quad \text{since } \Omega \text{ is local},$$
$$\text{loc}_\lambda(\Omega(\sigma)|_0) \quad \text{since } \Omega \text{ is location symmetric},$$
$$\text{loc}_\lambda(\sigma)|_p = \text{loc}_\lambda(\gamma)|_p \quad \text{for all } \gamma.$$

Since $\Omega(\text{loc}_\lambda(\sigma)|_0)$ is serial, we have that $\text{loc}_\lambda(\Omega(\text{loc}_\lambda(\sigma)|_0))$ is serial. Thus we get that
$$\text{loc}_\lambda(\Omega(\text{loc}_\lambda(\sigma)|_0)) = \text{loc}_\lambda(\text{loc}_\lambda(\Omega(\sigma)|_p)) = \Omega(\sigma)|_0 \quad \text{is serial}.$$

Suppose $\sigma$ is a run of $M(n, n, v)$. Since $\Omega$ is a serializer for $M(n, n, v)$ we have
that $\Omega(\sigma)$ is a linearization of $\Lambda^\Omega(\sigma)$. Therefore $\Lambda^\Omega(\sigma)$ is acyclic.

We show that $\Lambda^\Omega(\sigma)$ is acyclic. Let $R \subseteq \mathbb{N}_m$ such that $|R| \leq n$. We argue that
the graph $\Lambda^\Omega(\sigma)|_{\mathcal{A}(h(\Omega(\sigma)), R)}$ is acyclic. Let $\lambda$ be an $m$-permutation that maps $R$ one-one
onto $\mathbb{N}_{|R|}$ and $\mathbb{N}_m \setminus R$ one-one onto $\mathbb{N}_m \setminus \mathbb{N}_{|R|}$. Since $\sigma$ is a run of $M(n, m, v)$, we have
that $\text{loc}_\lambda(\sigma)|_{\mathbb{N}_{|R|}}$ is a run of $M(n, n, v)$ using Assumptions 5.1, 5.2 and 5.3. Therefore
$\Lambda^\Omega(\text{loc}_\lambda(\sigma)|_{\mathbb{N}_{|R|}})$ is acyclic.

$$\Lambda^\Omega(\text{loc}_\lambda(\sigma)|_{\mathbb{N}_{|R|}}) = \Lambda^\Omega(\text{loc}_\lambda(\sigma))|_{\mathcal{A}(h(\Omega(\text{loc}_\lambda(\sigma)), \mathbb{N}_{|R|})} \quad \text{from Lemma 5.7},$$
$$= \Lambda^\Omega(\text{loc}_\lambda(\sigma))|_{\mathcal{A}(h(\Omega(\sigma)), \lambda(R))} \quad \text{since } \lambda(R) = \mathbb{N}_{|R|},$$
$$= \Lambda^\Omega(\sigma)|_{\mathcal{A}(h(\Omega(\sigma)), R)} \quad \text{from Lemma 5.8}.$$

Thus we get that the $\mathcal{A}(h(\Omega(\sigma)), R)$-restriction of $\Lambda^\Omega(\sigma)$ is acyclic. We use Theorem 5.6
to conclude that $\Lambda^\Omega(\sigma)$ is acyclic. Let $\Omega'(\sigma)$ be a linearization of $\Lambda^\Omega(\sigma)$. Notice that
$\Omega'(\sigma)|_p = \Omega(\sigma)|_p$ for any $p < m$. Therefore $\Omega'(\sigma)$ is serial. Moreover $\Omega'(\sigma)$ is a linearization
of $\sigma$.

**Lemma 5.10** Suppose the parameterized memory system $M$ satisfies Assumptions 5.4 and 5.5. Let $\Omega$ be a data symmetric observer for $M$. For all $n > 0$ and $m > 0$, if $\Omega$ is a serializer for $M(n, m, 2)$ then for all $v > 0$, we have that $\Omega$ is a serializer for $M(n, m, v)$.

**Proof:** Consider any $v > 0$. Let $\sigma \in \Sigma(M(n, m, v))$. Since $\Omega$ is an observer for $M$, we have
that $\Omega(\sigma)$ is a linearization of $\sigma$. Suppose $\Omega(\sigma) = \pi_1, \pi_2, \ldots, \pi_k$ is not serial.

**Case 1.** There are $i \leq k$ and $j \leq k$ such that $lw_r(i) = lw_r(j) = \perp$ and $\text{val}(\pi_i) \neq \text{val}(\pi_j)$.

**Case 2.** For some $i \leq k$, there is a $j < i$ such that $lw_r(i) = j$ and $\text{val}(\pi_i) \neq \text{val}(\pi_j)$. 


In both cases, we can define a function \( \perp \)-extended \( \nu \)-map \( \lambda \) such that

\[
\lambda(x) = \begin{cases} 
\perp, & \text{if } x = \perp \\
1, & \text{if } x = \nu a_l(\pi_i) \\
0, & \text{otherwise}
\end{cases}
\]

Then \( \nu a_l(\sigma) \) is a run of \( M(n, m, 2) \). It is easy to see that \( \lambda(\Omega(\sigma)) \) is not serial. Consequently \( \Omega(\lambda(\sigma)) = \lambda(\Omega(\sigma)) \) is not serial which is a contradiction. Therefore we get that \( \Omega(\sigma) \) is serial.

**Proposition 5.11** Suppose the parameterized memory system \( M \) satisfies Assumptions 1–5. For all \( n > 0 \), if there is a local, location symmetric and data symmetric serializer for \( M(n, n, 2) \) then for all \( m > 0 \) and \( v > 0 \), there is a serializer for \( M(n, m, v) \).

**Proof:** Let \( \Omega : \Sigma_M \rightarrow Memop^* \) be a local, location symmetric and data symmetric serializer for \( M(n, n, 2) \). Consider any \( v > 0 \). From Lemma 5.10, we have that \( \Omega \) is a serializer for \( M(n, n, v) \). Now consider any \( m > 0 \). From Lemma 5.9, we have that there is a serializer for \( M(n, m, v) \).

### 5.5 Parameterized memory systems with arbitrary number of processors

In Section 5.4, we showed how to prove sequential consistency for fixed number of processors but arbitrary number of memory locations and data values. Since our objective is to prove sequential consistency for an arbitrary number of processors, we now give a method based on induction over the number of processors.

#### 5.5.1 Induction on the set of processors

We show how to check sequential consistency of \( M(n, m, v) \) for all \( n > 0 \) by induction over the number of processors. We do not need any of the Assumptions 1–5 for the results in this section. We would like to analyze a processor in an environment consisting of an arbitrary number of processors. Hence, we would like an upper bound on the trace set \( \Gamma(M(n, m, v)) \) for all \( n \). A sufficient condition for this upper bound is captured by process invariants [KM89].
A parameterized I/O-process \(<\text{PrivNames}_I, \text{ObsNames}_I, I_1>\) is a possible process invariant for the parameterized memory system \(M = \langle C, P \rangle\) if

1. \(\text{ObsNames}_I = \text{ObsNames}_C \cup \text{ObsNames}_P\), and

2. \(\text{PrivNames}_I \cap \text{PrivNames}_P = \emptyset\).

**Definition 5.3 (Process invariant)** Suppose that \(<\text{privNames}_I, \text{ObsNames}_I, I_1>\) is a possible process invariant for the parameterized memory system \(M\). Suppose the following condition is true for all \(m\) and \(v\):

\[
\langle A_I(m, v) \rangle \quad 1. \quad C(m, v) \preceq I_1(m, v)
\]
\[
2. \quad I_1(m, v) \| P(m, v) \preceq I_1(m, v)
\]

Then \(<\text{privNames}_I, \text{ObsNames}_I, I_1>\) is a process invariant of \(M\).

**Proposition 5.12** Suppose \(<\text{PrivNames}_I, \text{ObsNames}_I, I_1>\) is a process invariant of the parameterized memory system \(M\). Then, for all \(n > 0\), \(m > 0\), and \(v > 0\), we have that \(M(n, m, v) \preceq I_1(m, v)\).

**Proof:** We prove this by induction over \(n\).

**Base Case:** Fix \(m\) and \(v\). From the definition of \(M\), we have that \(M(0, m, v) = C(m, v)\). \(C(m, v) \preceq I_1(m, v)\) from the first property of \(I_1\). Therefore \(M(0, m, v) \preceq I_1(m, v)\).

**Inductive Case:** Suppose \(M(i, m, v) \preceq I_1(m, v)\) for all \(m\) and \(v\). Fix some \(m\) and \(v\). Since \(M(i, m, v) \preceq I_1(m, v)\), we have from Theorem 5.5 that \(M(i, m, v) \| P(m, v) \preceq I_1(m, v) \| P(m, v)\). From the definition of \(M\), we get \(M(i + 1, m, v) = M(i, m, v) \| P(m, v)\). Therefore \(M(i + 1, m, v) \preceq I_1(m, v) \| P(m, v)\). But \(I_1(m, v) \| P(m, v) \preceq I_1(m, v)\) from the second property of \(I_1\). Therefore \(M(i + 1, m, v) \preceq I_1(m, v)\).

If the parameterized memory system \(M\) is sequentially consistent, then by our definition, there exists an observer \(\Omega\) for \(M\) such that for every sequence \(\sigma\) of memory operations of \(M(n, m, v)\), the function \(\Omega\) produces a rearranged sequence \(\sigma'\) of memory operations such that (1) \(\sigma'\) is serial, and (2) \(\sigma\) and \(\sigma'\) agree on the ordering of the memory operations of each individual processor. We wish to provide an inductive construction that produces such an observer for arbitrary \(n\). The construction uses the notion of a generalized processor called a merge invariant, and a witness function that works like an observer for a two-processor system consisting of the merge invariant and \(P(m, v)\).
Consider the memory system \( I_2(m,v) \| P(m,v) \) for some value of \( m \) and \( v \). Notice that \( \text{Obs}(I_2(m,v) \| P(m,v)) = \text{Obs}(I_2(m,v)) = \text{Obs}(P(m,v)) \). Let \( \Upsilon(m,v) \) denote \( \text{RdWr}(m,v) \cup \text{Obs}(I_2(m,v)) \), the set of memory and observable actions of \( I_2(m,v) \). Let \( \sigma = \pi_0, \pi_1, \ldots, \pi_{k-1} \) be a sequence in \( \text{RdWr}(m,v) \cup \Upsilon(m,v)^* \). Define \( h' \) to be the function where \( h'(\sigma) \) is an ordered 3-partition \( \langle F,T \rangle \) of \( \mathbb{N}_k \) such that for all \( i < n \), (1) \( F(0) = \{ x < k | \pi_x \in \text{RdWr}(m,v) \} \), (2) \( F(1) = \{ x < k | \pi_x \in \text{RdWr}(m,v) \} \), (3) \( F(2) = \{ x < k | \pi_x \in \text{Obs}(I_2(m,v)) \} \), and (4) for all \( x,y \in F(i) \), we have that \( (x,y) \in T(i) \) iff \( x < y \). We extend \( h' \) to operate on arbitrary sequences \( \sigma \) by first restricting it to actions in \( \text{RdWr}(m,v) \cup \Upsilon(m,v) \). Formally, for any \( \sigma \), we have that \( h'(\sigma) = h'(\pi_0, \pi_1, \ldots, \pi_{k-1}) \). We extend \( h' \) to operate on sequences of extended actions by operating it on the first component of each extended action. Formally, if \( \sigma \) is a sequence of extended actions, then \( h'(\sigma) = h'(\text{First}(\sigma)) \). We observe that for all runs \( \sigma \) of \( P(m,v) \| I_2(m,v) \), the directed graph \( G(h'(\sigma)) \) is acyclic.

Let \( \pi_0, \pi_1, \ldots, \pi_{k-1} \) be a sequence in \( \text{RdWr}(m,v) \cup \Upsilon(m,v)^* \) and \( f \) be a \( k \)-permutation. Then \( \text{prime}(\pi_0, \pi_1, \ldots, \pi_{k-1}) \) is called a permutation of \( \sigma \). If \( f \) is a good order of \( G(h'(\sigma)) \), then the sequence \( \text{prime}(\pi_{f(0)}, \pi_{f(1)}, \ldots, \pi_{f(k-1)}) \) is called a linearization of \( \sigma \). Let \( \sigma \) be a run of \( I_2(m,v) \| P(m,v) \). A sequence \( \beta \in \Upsilon(m,v)^* \) is a permutation of \( \sigma \) if \( \beta \) is a permutation of \( \beta \| \text{RdWr}(m,v) \cup \Upsilon(m,v) \). A sequence \( \beta \in \Upsilon(m,v)^* \) is a linearization of \( \sigma \) if \( \beta \) is a linearization of \( \beta \| \text{RdWr}(m,v) \cup \Upsilon(m,v) \). A function \( \Theta \) from \( \Sigma_{I_2} \) to \( \Upsilon(m,v) \) is a merging function if for all \( m \) and \( v \) and for all \( \sigma \in \Sigma(I_2(m,v) \| P(m,v)) \), we have that \( \Theta(\sigma) \) is
a permutation of \( \sigma \). For any run \( \sigma \) of \( I_2(m, v) \), we define \( \tau \)'(\( \sigma \)) as the restriction of \( \sigma \) to \( \Upsilon(m, v) \). Let \( \Gamma(I_2(m, v)) \) be the set \( \{ \tau \)'(\( \sigma \)) | \( \sigma \in \Sigma(I_2(m, v)) \} \).

**Definition 5.4 (Merge invariant)** Let \( \langle \text{PrivNames}_1, \text{ObsNames}_1, I_1 \rangle \) be a process invariant and let \( \langle \text{PrivNames}_2, \text{ObsNames}_2, I_2 \rangle \) be a possible merge invariant for the parameterized memory system \( M \). Then \( \langle \text{PrivNames}_2, \text{ObsNames}_2, I_2 \rangle \) is a merge invariant of \( M \) with respect to \( \langle \text{PrivNames}_1, \text{ObsNames}_1, I_1 \rangle \) if there exists a merging function \( \Theta \) such that the following two conditions are true for all \( m \) and \( v \):

\[ [B_{01}, I_2(m, v)] \] For every closed trace \( a_1, a_2, \ldots \) of \( I_1(m, v) \), we have that the sequence \( a_1, a_2, \ldots \) is a run of \( I_2(m, v) \).

\[ [B_{11}, I_2(m, v)] \] For every closed run \( \sigma \) of \( I_2(m, v) \| C(m, v) \), the sequence
unprime(\( \sigma \) | \( \text{RdW}'(m, v) \)) is serial.

\[ [B_{21}, I_2(m, v)] \] For every run \( \sigma \) of \( I_2(m, v) \| P(m, v) \) that can be closed by \( I_1(m, v) \), we have that \( \Theta(\sigma) \in \Sigma(I_2(m, v)) \) and \( \Theta(\sigma) \) is a linearization of \( \sigma \).

We define the following restrictions on the merging function \( \Theta \).

- We say that \( \Theta \) is **local** on \( I_2(m, v) \| P(m, v) \) if for all \( \sigma \in \Sigma(I_2(m, v) \| P(m, v)) \) and \( j < m \), we have that \( \Theta(\sigma)_j = \Theta(\sigma||_j) \). We say that \( \Theta \) is local if \( \Theta \) is local on \( I_2(m, v) \| P(m, v) \) for all \( m \) and \( v \).

- We say that \( \Theta \) is **location symmetric** on \( I_2(m, v) \| P(m, v) \) if for every \( m \)-permutation \( \lambda \) and for all \( \sigma \in \Sigma(I_2(m, v) \| P(m, v)) \), we have that \( \Theta(\text{loc}_\lambda(\sigma)) = \text{loc}_\lambda(\Theta(\sigma)) \). We say that \( \Theta \) is location symmetric if \( \Theta \) is location symmetric on \( I_2(m, v) \| P(m, v) \) for all \( m \) and \( v \).

- We say that \( \Theta \) is **data symmetric** on \( I_2(m, v) \| P(m, v) \) if for every \( \bot \)-extended \( v \)-map \( \lambda \) and for all \( \sigma \in \Sigma(I_2(m, v) \| P(m, v)) \), we have that \( \Theta(\text{val}_\lambda(\sigma)) = \text{val}_\lambda(\Theta(\sigma)) \). We say that \( \Theta \) is data symmetric if \( \Theta \) is data symmetric on \( I_2(m, v) \| P(m, v) \) for all \( m \) and \( v \).

**Theorem 5.13** Let \( \langle \text{PrivNames}_1, \text{ObsNames}_1, I_1 \rangle \) be a process invariant for a parameterized memory system \( M \). Let \( \langle \text{PrivNames}_2, \text{ObsNames}_2, I_2 \rangle \) be a merge invariant of \( M \) with respect to \( \langle \text{PrivNames}_1, \text{ObsNames}_1, I_1 \rangle \). Then \( M \) is sequentially consistent.
CHAPTER 5. VERIFYING PARAMETERIZED SHARED-MEMORY SYSTEMS

Proof: Since $I_2$ is a merge invariant of $M$, there is a merging function $\Theta$ such that $B1_{I_2}(m, v)$ and $B2_{I_2, i_1, \Theta}(m, v)$ are true for some $m$ and $v$. We prove by induction on $n$ that the memory system $M(n, m, v) \| I_2(m, v)$ is sequentially consistent for all $n, m$ and $v$. Since $M(n, m, v) \preceq I_1(m, v)$, we have from $B0_{I_1, I_2}(m, v)$ that every closed trace of $M(n, m, v)$ is a run of $I_2(m, v)$. Therefore, every run of $M(n, m, v)$ is a run of $M(n, m, v) \| I_2(m, v)$. Therefore $M(n, m, v)$ is sequentially consistent for all $n, m$ and $v$. In the following, fix $m$ and $v$.

Base Case: From the definition of $M$, we have that $M(0, m, v) = C(m, v)$. Therefore $M(0, m, v) \| I_2(m, v)$ is sequentially consistent from $B1_{I_2}(m, v)$.

Inductive Case: Suppose $M(i, m, v) \| I_2(m, v)$ is sequentially consistent. Consider a closed run $\sigma$ of $M(i + 1, m, v) \| I_2(m, v) = M(i, m, v) \| P_{i+1}(m, v) \| I_2(m, v)$. From Theorem 5.1, there is a run $\sigma_1$ of $M(i, m, v)$ and a run $\sigma_2$ of $P_{i+1}(m, v) \| I_2(m, v)$ such that $\sigma$ is a join of $\sigma_1$ and $\sigma_2$. We get that $\sigma_2$ can be closed by $M(i, m, v)$. Since $I_1$ is a process invariant of $M$, we have that $M(i, m, v) \preceq I_1(m, v)$. Therefore $\sigma_2$ can be closed by $I_1(m, v)$ also. From $B2_{I_1, I_2, \Theta}(m, v)$, we get that $\Theta(\sigma_2) \in \Sigma(I_2(m, v))$ rearranges $\sigma_2$. Let $\tau_1 = tr_{M(i, m, v)}(\sigma_1)$ and $\tau_2 = tr_{P_{i+1}(m, v)}(I_2(m, v))(\sigma_2)$. From the first part of Theorem 5.3, we have that $\tau_1$ and $\tau_2$ are joinable. We have that $\tau_2 = tr_{P_{i+1}(m, v)}(I_2(m, v))(\sigma_2) = tr_{I_2(m, v)}(\Theta(\sigma_2))$ from the property of $\Theta$. Using the second part of Theorem 5.3, we have that there is a join $\sigma'$ of $\sigma_1$ and $\Theta(\sigma_2)$. From Theorem 5.1, we get that $\sigma'$ is a run of $M(i, m, v) \| I_2(m, v)$. Moreover, it is easy to see that $\sigma'$ is closed. Hence $\sigma'$ is sequentially consistent from the induction hypothesis. Therefore, there is a linearization of $\sigma'$ that is serial. Since any linearization of $\sigma'$ is also a linearization of $\sigma$, we have a linearization of $\sigma$ that is serial. Hence $\sigma$ is sequentially consistent.

Suppose that we manage to come up with possible invariants $I_1$ and $I_2$, and a merging function $\Theta$. How do we verify for all $m$ and $v$ that $A_{I_1}(m, v)$, $B1_{I_2}(m, v)$, and $B2_{I_2, i_1, \Theta}(m, v)$ hold? In the following section, we describe sufficient conditions whereby proving these obligations for fixed values of $m$ and $v$ will let us conclude that they hold for all $m$ and $v$.

5.5.2 Reduction to a fixed number of memory locations and data values

In this section, we use Assumptions 5.1, 5.2 and 5.3 on the parameterized memory system $M$. Further, we impose requirements on the process and merge invariants and the
merging function that will reduce the verification problem to one on a fixed number of memory locations.

**Requirement 5.1 (Location projectibility)** We require for the possible process invariant $I_1$ and the possible merge invariant $I_2$ that

1. if $\sigma \in \Sigma(I_1(m, v))$ then for all $j \leq m$, we have $\sigma|_{v_j} \in \Sigma(I_1(j, v))$, and

2. if $\sigma \in \Sigma(I_2(m, v))$ then for all $j \leq m$, we have $\sigma|_{v_j} \in \Sigma(I_2(j, v))$.

**Requirement 5.2 (Location symmetry)** Let $\lambda$ be an $m$-permutation. We require for the possible process invariant $I_1$ and the possible merge invariant $I_2$ that

1. for all $\sigma \in \Sigma(I_1(m, v))$, we have that $\text{loc}_\lambda(\sigma) \in \Sigma(I_1(m, v))$, and

2. for all $\sigma \in \Sigma(I_2(m, v))$, we have that $\text{loc}_\lambda(\sigma) \in \Sigma(I_2(m, v))$.

**Requirement 5.3 (Location monotonicity)** For all $n, v, m_1, m_2$, if $m_1 \leq m_2$, then

1. for all $\sigma \in \text{Act}(I_1(m_1, v))^*$, we have $\sigma \in \Sigma(I_1(m_1, v))$ iff $\sigma \in \Sigma(I_1(m_2, v))$, and

2. for all $\sigma \in \text{Act}(I_2(m_1, v))^*$, we have $\sigma \in \Sigma(I_2(m_1, v))$ iff $\sigma \in \Sigma(I_2(m_2, v))$.

**Requirement 5.4 (Location independence)** We require for the possible process invariant $I_1$ and the possible merge invariant $I_2$ that

1. $\sigma \in \Gamma(I_1(m, v))$ if $\sigma \in \text{Act}(I_1(m, v))^*$, and for all $j < m$, we have $\sigma|_j \in \Gamma(I_1(m, v))$, and

2. $\sigma \in \Gamma(I_2(m, v))$ if $\sigma \in \text{Act}(I_2(m, v))^*$, and for all $j < m$, we have $\sigma|_j \in \Gamma(I_2(m, v))$.

**Requirement 5.5 (Data symmetry)** Let $\lambda$ be an $\bot$-extendable $v$-map. We require for the possible process invariant $I_1$ and the possible merge invariant $I_2$ that

1. for all $\sigma \in \Sigma(I_1(m, v))$, we have that $\text{val}_\lambda(\sigma) \in \Sigma(I_1(m, v))$, and

2. for all $\sigma \in \Sigma(I_2(m, v))$, we have that $\text{val}_\lambda(\sigma) \in \Sigma(I_2(m, v))$.

**Requirement 5.6 (Data monotonicity)** For all $n, m, v_1, v_2$, if $v_1 \leq v_2$, then

1. for all $\sigma \in \text{Act}(I_1(m, v_1))^*$, we have $\sigma \in \Sigma(I_1(m, v_1))$ iff $\sigma \in \Sigma(I_1(m, v_2))$, and
2. for all $\sigma \in \text{Act}(I_2(m, v_1))$, we have $\sigma \in \Sigma(I_2(m, v_1))$ iff $\sigma \in \Sigma(I_2(m, v_2))$.

**Requirement 5.7 (Data reducibility)** We require for the possible process invariant $I_1$ and the possible merge invariant $I_2$ that

1. if $\sigma \notin \Gamma(I_1(m, v))$ then there is a $\bot$-extended $\nu$-map with range $\{0, 1, \bot\}$ such that $\text{val}_\lambda(\sigma) \notin \Gamma(I_1(m, v))$.

2. if $\sigma \notin \Gamma'(I_2(m, v))$ then there is a $\bot$-extended $\nu$-map with range $\{0, 1, \bot\}$ such that $\text{val}_\lambda(\sigma) \notin \Gamma'(I_2(m, v))$.

**Lemma 5.14** Let $M$ be a parameterized memory system satisfying Assumptions 5.1, 5.2 and 5.3. Let $I_1$ be a possible process invariant satisfying Requirements 5.1, 5.2, 5.3 and 5.4. Then for all $m > 0$ and $v > 0$, if $A_{I_1}(1, 2)$ is true then $A_{I_1}(m, v)$ is true.

**Proof:** Fix $m > 0$ and $v > 0$.

1. Given $C(1, 2) \preceq I_1(1, 2)$, we show that $C(m, v) \preceq I_1(m, v)$. Let $\sigma$ be a trace of $C(m, v)$. Fix $p < m$ and let $\lambda$ be the $m$-permutation such that $\lambda(p) = 0$, $\lambda(0) = p$, and $\lambda(j) = j$ if $j$ is neither 0 nor $p$. Then $\sigma|_p = \text{loc}_{\lambda^{-1}}(\text{loc}_\lambda(\sigma)|_0)$. We have that $\text{loc}_\lambda(\sigma)|_1$ is a trace of $C(1, v)$ from Assumptions 5.1, 5.2 and 5.3. If $\text{loc}_\lambda(\sigma)|_0$ is not a trace of $I_1(1, v)$ then from Requirement 5.7 there is a $\bot$-extended $\nu$-map $\lambda$ with range $\{0, 1, \bot\}$ such that $\text{val}_\lambda(\text{loc}_\lambda(\sigma)|_0)$ is not a trace of $I_1(1, v)$. Consequently, it is not a trace of $I_1(1, 2)$ from Requirement 5.6. But this is a contradiction since $\text{val}_\lambda(\text{loc}_\lambda(\sigma)|_0)$ is a trace of $C(1, 2)$. Therefore $\text{loc}_\lambda(\sigma)|_0$ is a trace of $I_1(1, v)$. Thus we get that $\sigma|_p = \text{loc}_{\lambda^{-1}}(\text{loc}_\lambda(\sigma)|_0)$ is a trace of $I_1(m, v)$ by Requirements 5.3 and 5.2. From Requirement 5.4, we get that $\sigma$ is a trace of $I_1(m, v)$.

2. Given $I_1(1, 2)\parallel P(1, 2) \preceq I_1(1, 2)$, we show that $I_1(m, v)\parallel P(m, v) \preceq I_1(m, v)$. Let $\sigma$ be a trace of $I_1(m, v)\parallel P(m, v)$. Fix $p < m$ and let $\lambda$ be the $m$-permutation such that $\lambda(p) = 0$, $\lambda(0) = p$, and $\lambda(j) = j$ if $j$ is neither 0 nor $p$. Then $\sigma|_p = \text{loc}_{\lambda^{-1}}(\text{loc}_\lambda(\sigma)|_0)$. We have that $\text{loc}_\lambda(\sigma)|_0$ is a trace of $I_1(1, v)\parallel P(1, v)$ from Assumptions 5.1, 5.2 and 5.3, and Requirements 5.1, 5.2 and 5.3. If $\text{loc}_\lambda(\sigma)|_0$ is not a trace of $I_1(1, v)$ then from Requirement 5.7 there is a $\bot$-extended $\nu$-map $\lambda$ with range $\{0, 1, \bot\}$ such that $\text{val}_\lambda(\text{loc}_\lambda(\sigma)|_0)$ is not a trace of $I_1(1, v)$. Consequently $\text{val}_\lambda(\text{loc}_\lambda(\sigma)|_0)$ is not a trace of $I_1(1, 2)$ from Requirement 5.6. But this is a contradiction since $\text{val}_\lambda(\text{loc}_\lambda(\sigma)|_0)$ is
a trace of \( I_1(1, 2)\|P(1, 2) \). Therefore \( \text{loc}_\lambda(\sigma)|_0 \) is a trace of \( I_1(1, v) \). Thus we get that \( \sigma|_0 = \text{loc}_{\lambda^{-1}}(\text{loc}_\lambda(\sigma)|_0) \) is a trace of \( I_1(m, v) \) by Requirements 5.3 and 5.2. From Requirement 5.4, we get that \( \sigma \) is a trace of \( I_1(m, v) \).

\[\medskip\]

**Lemma 5.15** Let \( M \) be a parameterized memory system satisfying Assumptions 5.1, 5.2 and 5.3. Let \( I_2 \) be a possible merge invariant satisfying Requirements 5.1, 5.2 and 5.3. Then for all \( m > 0 \) and \( v > 0 \),

1. if \( B_0 I_2(1, 2) \) is true then \( B_0 I_2(m, v) \) is true, and

2. if \( B_1 I_2(1, 2) \) is true then \( B_1 I_2(m, v) \) is true.

**Proof:** The proof runs exactly along the lines of the proof of Lemma 5.14.

Let \( \Theta \) be a merging function. Let \( \sigma \in \Sigma(I_2(m, v)\|P(m, v)) \) and \( [\sigma]_{\Sigma(m, v)} = \pi_0, \pi_1, \ldots, \pi_{k-1} \). Let \( \Theta(\sigma) = \pi f^{-1}(0), \pi f^{-1}(1), \ldots, \pi f^{-1}(k-1) \) for some \( k \)-permutation \( f \). Let \( h^{\Theta}(\sigma) \) be the function where \( h^{\Theta}(\sigma) \) is an ordered \( m \)-partition \( \langle F', T' \rangle \) of \( \mathbb{N}_k \) such that for all \( j < m \), (1) \( F'(j) = \{ x \in K \mid \text{loc}(\pi_x) = j \} \), and (2) for all \( x, y \in F(i) \), we have that \( \langle x, y \rangle \in T'(i) \) iff \( f(x) < f(y) \). Let \( \Lambda^\Theta \) be the function on \( \Sigma(I_2(m, v)\|P(m, v)) \) such that \( \Lambda^\Theta(\sigma) = G(h^\Theta(\sigma)) \cup G(h^\Theta(\sigma)) \).

**Lemma 5.16** Let \( \Theta \) be a local merging function. Let \( \sigma \) be a run of \( I_2(m, v)\|P(m, v) \) and \( R \) be a subset of \( \mathbb{N}_m \). Then \( \Lambda^\Theta(\sigma|R) \) is isomorphic to \( \Lambda^\Theta(\sigma)|_{\Lambda(h^\Theta(\sigma), R)} \).

**Lemma 5.17** Suppose we are given a location symmetric merging function \( \Theta \), an \( m \)-permutation \( \lambda \), a run \( \sigma \) of \( I_2(m, v)\|P(m, v) \), and a set \( R \subseteq \mathbb{N}_m \). Then we have that \( \Lambda^\Theta(\text{loc}_\lambda(\sigma))|_{\Lambda(h^\Theta(\text{loc}_\lambda(\sigma)), \lambda(R))} \) is isomorphic to \( \Lambda^\Theta(\sigma)|_{\Lambda(h^\Theta(\sigma), R)} \).

**Lemma 5.18** Let \( M \) be a parameterized memory system satisfying Assumptions 1, 2 and 3. Let \( I_1 \) be a possible process invariant and let \( I_2 \) be a possible merge invariant for \( M \) satisfying Requirements 1–4. Then for all \( v > 0 \), if there is a local and location symmetric merging function \( \Theta \) satisfying \( B_2 I_2, I_1, \Theta(3, v) \) then for all \( m > 0 \), there is a merging function \( \Theta' \) satisfying \( B_2 I_2, I_1, \Theta'(m, v) \).

**Proof:** Fix some \( v > 0 \). Suppose \( \Theta \) is a local and location symmetric merging function satisfying \( B_2 I_2, I_1, \Theta(3, v) \). Fix some \( m > 0 \). We now show how to define a merging function \( \Theta' \) satisfying \( B_2 I_2, I_1, \Theta'(m, v) \). Let \( \sigma \in \Sigma(I_2(m, v)\|P(m, v)) \).
CHAPTER 5. VERIFYING PARAMETERIZED SHARED-MEMORY SYSTEMS 126

Fix \( p < m \) and let \( \lambda \) be the \( m \)-permutation such that \( \lambda(p) = 0, \lambda(0) = p, \) and
\( \lambda(j) = j \) if \( j \) is neither \( 0 \) nor \( p. \) We show that \( \Theta(\sigma)|_{p} \in \Gamma'(I_2(m,v)). \) We have that \( \text{loc}_\lambda(\sigma)|_{0} \)
is a run of \( I_2(1,v)\|P(1,v) \) and therefore of \( I_2(3,v)\|P(3,v) \). Since \( \Theta \) is a merging function
for \( I_2(3,v)\|P(3,v) \), we have that \( \Theta(\text{loc}_\lambda(\sigma)|_{0}) \in \Gamma'(I_2(3,v)) \).

\[
\Theta(\text{loc}_\lambda(\sigma)|_{0}) = \Theta(\text{loc}_\lambda(\sigma)|_{0}) \quad \text{since } \Theta \text{ is local,}
\]
\[
= \text{loc}_\lambda(\theta(\sigma)|_{0}) \quad \text{since } \Theta \text{ is location symmetric,}
\]
\[
= \text{loc}_\lambda(\theta(\sigma)|_{p}) \quad \text{since } \text{loc}_\lambda(\gamma)|_{0} = \text{loc}_\lambda(\gamma)|_{p} \text{ for all } \gamma.
\]

Since \( \Theta(\text{loc}_\lambda(\sigma)|_{0}) \in \Gamma'(I_2(3,v)) \), we have that \( \text{loc}_\lambda(\theta(\sigma)|_{0}) \in \Gamma'(I_2(3,v)) \). Thus
we get that \( \text{loc}_\lambda(\theta(\sigma)|_{0}) = \text{loc}_\lambda(\theta(\sigma)|_{p}) = \Theta(\sigma)|_{p} \in \Gamma'(I_2(3,v)) \). Therefore
\( \Theta(\text{loc}_\lambda(\sigma)|_{0}) \in \Gamma'(I_2(m,v)) \).

Suppose \( \sigma \) is a run of \( I_2(3,v)\|P(3,v) \). Since \( \Theta \) is a serializer for \( I_2(3,v)\|P(3,v) \),
we have that \( \Theta(\sigma) \) is a linearization of \( \Lambda^\Theta(\sigma) \). Therefore \( \Lambda^\Theta(\sigma) \) is acyclic.

We show that \( \Lambda^\Theta(\sigma) \) is acyclic. Let \( R \subseteq \mathbb{N}_m \) such that \( |R| \leq 3 \). We argue that
the graph \( \Lambda^\Theta(\sigma)|_{\mathcal{A}(h^\Theta(\sigma),R)} \) is acyclic. Let \( \lambda \) be an \( m \)-permutation that maps \( R \) one-one
onto \( \mathbb{N}_{|R|} \) and \( \mathbb{N}_m \setminus R \) one-one onto \( \mathbb{N}_m \setminus \mathbb{N}_{|R|} \). Since \( \sigma \) is a run of \( I_2(3,v)\|P(3,v) \), we have
that \( \text{loc}_\lambda(\sigma)|_{|R|} \) is a run of \( I_2(3,v)\|P(3,v) \) using Assumptions 5.1, 5.2 and 5.3. Therefore
\( \Lambda^\Theta(\text{loc}_\lambda(\sigma)|_{|R|}) \) is acyclic.

\[
\Lambda^\Theta(\text{loc}_\lambda(\sigma)|_{|R|}) = \Lambda^\Theta(\text{loc}_\lambda(\sigma))|_{\mathcal{A}(h^\Theta(\text{loc}_\lambda(\sigma)),|R|)} \quad \text{from Lemma 5.16,}
\]
\[
= \Lambda^\Theta(\text{loc}_\lambda(\sigma))|_{\mathcal{A}(h^\Theta(\text{loc}_\lambda(\sigma)),\lambda(R))} \quad \text{since } \lambda(R) = \mathbb{N}_{|R|},
\]
\[
= \Lambda^\Theta(\sigma)|_{\mathcal{A}(h^\Theta(\sigma),R)} \quad \text{from Lemma 5.17.}
\]

Thus we get that the \( \mathcal{A}(h^\Theta(\sigma),R) \)-restriction of \( \Lambda^\Theta(\sigma) \) is acyclic. We use Theorem 5.6
to conclude that \( \Lambda^\Theta(\sigma) \) is acyclic. Let \( \Theta'(\sigma) \) be a linearization of \( \Lambda^\Theta(\sigma) \). Notice that
\( \Theta'(\sigma)|_{p} = \Theta(\sigma)|_{p} \) for any \( p < m \). Therefore \( \Theta'(\sigma) \in \Gamma'(I_2(m,v)) \) from Requirement 5.4.
Moreover \( \Theta'(\sigma) \) is a linearization of \( \sigma \).

\textbf{Lemma 5.19} Let \( M \) be a parameterized memory system satisfying Assumptions 5.4 and 5.5. Let \( I_1 \) be a possible process invariant and let \( I_2 \) be a possible merge invariant for \( M \) satisfying Requirements 1–4. Then for all \( m > 0 \), if \( \Theta \) is a data symmetric merging function satisfying \( B_{2I_2,I_1,\Theta}(m,2) \) then for all \( v > 0 \), \( \Theta \) is a merging function satisfying \( B_{2I_2,I_1,\Theta}(m,v) \).

\textbf{Proof:} Consider any \( v > 0 \). Let \( \sigma \in \Sigma(I_2(m,v)\|P(m,v)) \). Suppose \( \Theta(\sigma) \) does not satisfy \( B_{2I_2,I_1,\Theta}(m,v) \).
CHAPTER 5. VERIFYING PARAMETERIZED SHARED-MEMORY SYSTEMS

Case 1. \( \Theta(\sigma) \) is not a linearization of \( \sigma \). Let \( \lambda \) be a \( \perp \)-extended \( v \)-map such that \( \lambda(k) = 0 \) for all \( k < v \). Then \( \text{val}_\lambda(\sigma) \) is a run of \( \Sigma(I_2(m,1) \parallel P(m,1)) \) from Assumptions 5.4 and 5.5, and Requirements 5.5 and 5.6. But \( \Theta(\text{val}_\lambda(\sigma)) = \text{val}_\lambda(\Theta(\sigma)) \) because \( \Theta \) is value symmetric. Therefore \( \Theta(\text{val}_\lambda(\sigma)) \) is also not a linearization of \( \text{val}_\lambda(\sigma) \) which is a contradiction.

Case 2. \( \Theta(\sigma) \not\in \Sigma'(I_2(m,v)) \). From Requirement 5.7, we have that there is a \( \perp \)-extended \( v \)-map with range \{0, 1, \perp\} such that \( \text{val}_\lambda(\Theta(\sigma)) \not\in \Sigma'(I_2(m,v)) \). But \( \text{val}_\lambda(\Theta(\sigma)) = \Theta(\text{val}_\lambda(\sigma)) \). because \( \Theta \) is value symmetric. Moreover \( \text{val}_\lambda(\sigma) \in \Sigma(I_2(m,1) \parallel P(m,1)) \) from Assumptions 5.4 and 5.5, and Requirements 5.5 and 5.6. Therefore we get that \( \Theta(\text{val}_\lambda(\sigma)) \not\in \Sigma'(I_2(m,2)) \) which is a contradiction.

\[ \blacksquare \]

**Theorem 5.20** Let \( M \) be a parameterized memory system satisfying Assumptions 1–5. Let \( I_1 \) be a possible process invariant and let \( I_2 \) be a possible merge invariant for \( M \) satisfying Requirements 1–4. Let \( \Theta \) be a local, location symmetric and data symmetric merging function. Suppose \( A_{I_1}(1,2) \), \( B_{0_{I_1},I_2}(1,2) \) and \( B_{1_{I_1},I_2}(1,2) \) are true, and \( \Theta \) is a witness for \( B_{2_{I_2},I_1,\Theta}(3,2) \). Then \( M(n,m,v) \) is sequentially consistent for all \( n > 0 \), \( m > 0 \), and \( v > 0 \); that is, \( M \) is sequentially consistent.

**Proof:** Fix \( m > 0 \) and \( v > 0 \). From Lemma 5.14, we have that \( A_{I_1}(m,v) \) is true. From Lemma 5.15, we have that \( B_{0_{I_2},I_2}(m,v) \) and \( B_{1_{I_2},I_2}(m,v) \) are true. From Lemma 5.19, we have that \( \Theta \) is a witness for \( B_{2_{I_2},I_1,\Theta}(3,v) \). From Lemma 5.18, we have that there is a witness \( \Theta' \) for \( B_{2_{I_2},I_1,\Theta'}(m,v) \). From Theorem 5.13, we have that \( M(n,m,v) \) is sequentially consistent for all \( n \).

\[ \blacksquare \]

5.6 Verification of a snoopy cache coherence protocol

We show how the theory developed in the previous section can be used to verify sequential consistency of memory systems with an arbitrary number of processors, locations and data values using a model checker. We consider a simplified snoopy cache coherence protocol from [HP96a]. Each processor has a cache for fast access to the main memory. There is a single bus for communication between the cache controllers. In each cache
CHAPTER 5. VERIFYING PARAMETERIZED SHARED-MEMORY SYSTEMS

controller, corresponding to each location there is a I/O-process with five states. The I/O-process for the controller is obtained by composing the I/O-processes for the locations. Figure 5.1 shows the I/O-process for location $j$. A location could be in one of three modes —invalid, shared, or exclusive. If a location is in the shared mode, then the processor can read its value but cannot modify it. If a location is in exclusive mode, then the processor can both read and write it. If a location is in the invalid mode, the processor can neither read nor write it. These modes are denoted by $INV$ for invalid, $RS$ for shared and $WX$ for exclusive. If the processor wants to access the location, it synchronizes over the bus with other caches that have the latest value stored in the location, and makes a transition to either the shared or the exclusive mode. For simplicity we assume that all the caches in various processors are of the same size as the main memory, and every location is initialized to the same value in the shared mode. Consequently, we do not model the main memory.

The protocol is described by the parameterized memory system $(C, P)$, where $P(m, v)$ models one processor with a local cache and $C(m, v)$ models the memory controller. Since we are not modeling the memory controller in this example for simplicity, we can think of $C(m, v)$ as an I/O-process with a single state with self-loops on all possible input actions. For the parameterized I/O-process $P$, we have that $PrivNames_P = \{R, W\}$ and $ObsNames_P = \{RB, WB\}$. The actions R and W stand for read and write respec-
Figure 5.2: Snoopy cache coherence invariant

tively, while $RB$ and $WB$ stand for read synchronize and write synchronize (over the bus) respectively. Figure 5.1 shows the cache controller machine for a single location $j$ in which two data values can be stored. It is easy to see how it can be extended to $v$ data values for an arbitrary $v$. In each state the actions that can be taken label the outgoing arcs.

Each action of the I/O-process for the $j$th location with two data values is a 4-tuple from the set $\{R, W, RB, WB\} \times \{j\} \times \{0,1\} \times \{in, out\}$. The last component denotes whether the action is an input or an output action. Recall that private actions can be only output actions, while observable actions can be either input or output. A $-$ in the action name is a wild card and means several actions that can be obtained by substituting values for $-$. The names of the states are self-explanatory. For example, the state labeled $\langle RS, 0 \rangle$ means that the location is in shared mode and the current value for the location in the cache is 0. The I/O-process $P(m, 2)$ is the parallel composition of $m$ of these I/O-processes, one for each location.

We will now give an inductive proof using merge and process invariants. For this example, the process invariant $I_1$ and the merge invariant $I_2$ are identical. We refer to the two together as $I$. The invariant $I(m, v)$ is the parallel composition of $m$ I/O-processes where the $j$th I/O-process is given in Figure 5.2. We will not concern ourselves with proving formally that Assumptions 5.1–5.5 are satisfied by the protocol. A complete proof would use
a theorem prover to discharge these assumptions. We also convince ourselves informally that $I$ satisfies Requirements 5.1–5.7.

We give some intuition behind our construction of the invariant. The invariant $I$ is such that for all $m$ and $v$, we have that $I(m, v)$ is a generalization of the processor $P(m, v)$. The processor is generalized so that it can make a read request on the bus for a location even if it already has the location in shared mode. In addition, it can make a read or a write request on the bus even if the location is in exclusive mode. Thus, the I/O-process for the invariant in Figure 5.2 has more actions than the I/O-process in Figure 5.1. For example, the additional actions that can be taken are $\langle RB, j, 0, out \rangle$ from states $\langle RS, 0 \rangle$ and $\langle WX, 0 \rangle$ and $\langle WB, j, 0, out \rangle$ from the state $\langle WX, 0 \rangle$. The merging function $\Theta$ is the identity function. It is easy to see that the merging function is local, location symmetric and data symmetric. We used Mocha to verify the proof obligations $A_I(1, 2)$, $B0_I(1, 2)$, $B1_I(1, 2)$ and $B2_I(3, 2)$. Mocha required a few minutes to check these obligations.

\footnote{It is not too hard to convince ourselves informally that the protocol satisfies Assumptions 5.1–5.5. Indeed, we encourage the reader to go through this exercise.}

\footnote{This simple merging works because the snoopy protocol implements coherence.}
Chapter 6

Conclusions

The fundamental obstacle to the use of model checking for formal verification of reactive systems designed in the industry is the state explosion problem. In this dissertation, we have presented several techniques for combating this problem. In Chapters 2, 3 and 4, we presented novel techniques to make model checking scale to larger designs. In Chapter 5, we have extended model checking to a domain where it could not be directly applied before—the domain of parameterized shared-memory multiprocessor protocols. We summarize our technical contributions below and point out interesting direction for future research.

In Chapter 2, we showed that all linear properties expressible by Büchi automata, which includes linear temporal logic (LTL) properties, can be translated to query post-$\mu$ calculus. The translated formula is linear in the size of the automaton and has an alternation depth of two. Since the translation of LTL formulas to Büchi automata is exponential in the worst case, our translation for LTL formula to post-$\mu$ queries has exponential worst case complexity. The complexity and alternation depth of the translation match corresponding translation to pre-$\mu$ calculus queries. We also gave a translation to post-$\mu$ queries of linear size and alternation depth one for co-Büchi automata. Again, the translation matched the corresponding translation to pre-$\mu$ calculus queries. Finally, we translated safety automata to post-$\mu$ queries which yields a symbolic algorithm for detecting violations of all safety properties as soon as possible. We compared forward reasoning to backward reasoning on one example—the sliding window protocol, and obtained very encouraging results.

The exact characterization of the intersection of the two logics, query pre-$\mu$ calculus and query post-$\mu$ calculus, remains open. The notion of equi-linear properties has been defined [GK94] to be those properties that cannot distinguish between two Kripke structures...
with the same language. We conjecture that a pre-$\mu$ query $\varphi$ is expressible as a post-$\mu$ query iff $\varphi$ is equi-linear. There is a need for further investigation into the efficacy of forward reasoning as a heuristic for efficient model checking. We would like to develop intuition about the type of systems for which forward reasoning works better so that we can provide guidelines for verifiers.

In Chapter 3, we present a compositional verification methodology based on an assume-guarantee proof decomposition rule. We focused on refinement checking and showed that if the specification is suitably enriched to provide specifications of signals present in the environment of implementation components, the verification problem can be decomposed into a set of smaller verification problems. Each of these sub-problems involves an implementation that typically has a much smaller state space than the original implementation. We explained this methodology by using the assume-guarantee rule to prove that a simple three-stage pipeline refines its Instruction Set Architecture (ISA). We then demonstrated the effectiveness of the proof framework by carrying out a proof of an implementation of the Tomasulo's algorithm for out-of-order execution in a microprocessor. In Chapter 4, we generalized the assume-guarantee rule of Chapter 3 to deal with the notion of refinement where several steps of the implementation correspond to a single step of the specification. We used the generalized assume-guarantee rule to verify VGI, a chip for digital signal processing designed by the Infopad group at the University of California at Berkeley. The part of chip that we verified contained 64 compute processors. Each processor had approximately 30,000 logic gates and 800 latches.

Our experience with verifying designs such as the VGI has shown us that the assume-guarantee rule is typically not enough for proof decomposition. We also need supporting rules for dealing with data structures such as bitvectors and arrays, that abound in hardware designs. Moreover, it is sometimes easier to perform proof decomposition along non-structural boundaries. For example, in processor verification it might be easier to do a case analysis on the type of the input instruction. Each instruction type might result in a different path being taken through the processor and consequently, for each case a particular path will be analyzed rather than the whole design. We would like to investigate other proof decomposition techniques and come up with design idioms that are general enough so that realistic designs can be contrived out of combinations of them and structured enough so that a library of proof decomposition techniques can be developed for them.

In Chapter 5, we showed how to use model checking to verify sequential consistency
on parameterized shared-memory multiprocessor systems for any number of processors, memory locations and data values. We introduced the notion of a serializer that reorders the memory events occurring in the system, without interfering with the system, to produce a trace of serial memory. We presented two model checking proof methods based on certain reasonable assumptions on the memory system such as symmetry and monotonicity of locations and data values, and projectibility of locations. The first method lets us prove the correctness of a system with fixed number of processors for any number of memory locations and data values. We showed that for all \( n > 0 \), if a serializer that is local, location symmetric and data symmetric exists for a memory system with \( n \) processors, \( n \) memory locations and two data values, then the memory system with \( n \) processors is sequentially consistent for any number of memory locations and data values. Moreover if the serializer is finite-state, then it can be composed with the memory system and the composition can be modeled checked against the serial memory specification. The second method lets us prove sequential consistency for any number of processors, memory locations and data values. It is based on a novel induction scheme involving two invariant processes—process invariant and merge invariant, and a serializer-like witness for the merge invariant. The verifier has to construct these invariants and the witness manually. We show that if the invariants satisfy a set of requirements including the assumptions on the memory system as well location independence and data reducibility, and the witness is local, location symmetric and data symmetric then it suffices to perform the inductive proof for three memory locations and two data values. We used the inductive method to verify a snoopy cache coherence protocol.

There are several interesting directions to pursue in this area. Intuitively, it would seem that if a shared memory protocol is sequentially consistent for some constant number of memory locations and data values, then it should be sequentially consistent for any number of memory locations and data values. We would like to characterize such protocols and state precisely the assumptions underlying such a result. We have found the inductive proof method based on process and merge invariants quite difficult to apply in practice. We tried to apply it to the lazy caching protocol [ABM93], but couldn’t find a merge invariant satisfying all the requirements mentioned in Section 5.5.2. The first framework described above which can prove sequential consistency for a fixed number of processors is easier to use in practice. But, even with that method there are two obstacles to achieving the holy grail of fully automatic verification—the state explosion problem as the number of processors increases and the manual construction of the serializer. To solve the first
problem, we would like to investigate other compositional ways of decomposing the proof obligation. We believe that the second problem cannot be solved completely. The verifier has to provide some input to describe the serializer. But we would like to make it easier to describe it. One interesting direction is to construct and model check the serializer for a single location against the serial memory specification. As a second step, it can be checked that such identically constructed serializers for the various locations can work together, that is, their outputs can be interleaved together preserving the processor order of events. Thus, not only has the construction of the serializer been simplified, the model checking problem has been broken down into two sub-problems. For each of these problems, we would like to do a further proof decomposition to avoid the state explosion problem due to increase in the number of processors.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


Appendix A

Sliding window protocol

In this appendix, we give the reactive modules description of the sliding window protocol referred to in Section 2.4.3.

#define WINDOW 1
#define MSGSIZE 2
#define SEQSIZ E 2

type msgType: (0..MSGSIZE-1)
type seqType: (0..SEQSIZE-1)
type winType: (0..WINDOW-1)
type windowType: (0..WINDOW)
type msgWinArray: array winType of msgType
type boolWinArray: array winType of bool

module SlidingWindowProtocol
  interface seqX, seq, index, ackX, ack: seqType; msgP, msgX, msgC: msgType;
  msgValid, ackValid: bool; sndStore, recStore: msgWinArray;
  busy, recvd: boolWinArray; window: windowType

lazy atom controls msgP, seq reads msgP, seq, window
  init
    []true → seq' := 0
update
\[ \text{window} < \text{WINDOW} \rightarrow \text{msgP}' := \text{nondet}; \text{seq}' := \text{seq} + 1 \]

endatom

atom \texttt{controls} \texttt{sndStore \ reads \ seq, sndStore \ awaits \ seq, msgP}
\texttt{init}
\[ \text{true} \rightarrow \text{forall} \ i \ \text{sndStore}'[i] := 0 \]
\texttt{update}
\[ \text{seq}' = \text{seq} + 1 \rightarrow \]
\[ \text{forall} \ i \ \text{sndStore}'[i] := \text{if} \ (\text{seq} \mod \text{WINDOW} = i) \ \text{then} \ \text{msgP}' \ \text{else} \ \text{sndStore}[i] \ \text{fi} \]

endatom

atom
\texttt{controls} \texttt{msgX, seqX, msgValid}
\texttt{reads} \texttt{index, seq, msgX, seqX, msgValid, seq}
\texttt{awaits} \texttt{sndStore, seq, busy, index}
\texttt{init}
\[ \text{true} \rightarrow \text{msgValid}' := \text{false} \]
\texttt{update}
\[ \text{seq}' = \text{seq} + 1 \rightarrow \text{msgX}' := \text{sndStore}'[\text{seq} \mod \text{WINDOW}]; \text{msgValid}' := \text{true}; \text{seqX}' := \text{seq} \]
\[ \text{busy}'[\text{index} \mod \text{WINDOW}] \rightarrow \]
\[ \text{msgX}' := \text{sndStore}'[\text{index} \mod \text{WINDOW}]; \text{msgValid}' := \text{true}; \text{seqX}' := \text{index} \]
\[ \text{true} \rightarrow \text{msgValid}' := \text{false}; \text{msgX}' := \text{nondet}; \text{seqX}' := \text{nondet} \]

endatom

atom \texttt{controls} \texttt{index \ reads \ index, window \ awaits \ busy}
\texttt{init}
\[ \text{true} \rightarrow \text{index}' := 0 \]
\texttt{update}
\[ \text{busy}'[\text{index} \mod \text{WINDOW}] \land \text{window} > 0 \rightarrow \text{index}' := \text{index} + 1 \]

endatom
atom controls window reads window, index, seq awaits index, seq
init
[] true → window' := 0
update
[] (seq' = seq + 1) ∧ (index' = index + 1) → window' := window + 1
[] (seq' = seq + 1) ∧ (index' = index + 1) → window' := window - 1
endatom

#foreach j = (0..WINDOW-1)
atom controls busy [\$j] reads seq, ackX, ackValid, busy [\$j], seq, ackX awaits seq
init
[] true → busy'[%j] := false
update
[] seq' = seq + 1 ∧ (seq%WINDOW = %j) → busy'[%j] := true
[] (seq' = seq + 1) ∧ ackValid ∧ (ackX%WINDOW = %j) → busy'[%j] := false
endatom
#endforeach

lazy atom controls ack, msgC reads ack, recvd, msgC, reqStore
init
[] true → ack' := 0
update
[] recvd[ack%WINDOW] → msgC' := reqStore[ack%WINDOW]; ack' := ack + 1
endatom

#foreach j = (0..WINDOW-1)
atom controls recvd [\$j] reads recvd [\$j], ack awaits seqX, msgValid, ack
init
[] true → recvd'[%j] := false
update
[] msgValid' ∧ (seqX' − ack' < WINDOW) ∧ (seqX%WINDOW = %j) → recvd'[%j] := true
[] ack' = ack + 1 ∧ (ack%WINDOW = %j) → recvd'[%j] := false
atom controls recStore reads recd, recStore awaits msgX, msgValid, seqX, ack
init
[] true → forall recStore'[i] := 0
update
[] msgValid' ∧ (seqX' - ack' < WINDOW) →
forall i recStore'[i] := if (seqX' % WINDOW = i) then msgX' else recStore[i] fi
endatom

atom controls ackX, ackValid reads ack, ackX, ackValid awaits ack, seqX, msgValid
init
[] true → ackValid' := false
update
[] ack' = ack + 1 → ackX' := ack; ackValid' := true
[] msgValid' ∧ (ack' - seqX' > 0) ∧ (ack' - seqX' <= WINDOW) →
ackX' := seqX'; ackValid' := true
[] true → ackValid' := false; ackX' := nondet
endatom

endmodule