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, such as binary decision diagrams, to represent the state sets encountered during state space exploration. Symbolic model checking has proved to be effective for verifying hardware designs. Traditionally, symbolic checking of temporal logic specifications is performed by backward fixpoint reasoning with the operator *pre*. Backward reasoning can be wasteful since unreachable states are explored. We suggest the use of forward fixpoint reasoning based on the operator *post*. We show how all linear temporal logic specifications can be model checked symbolically 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 combat 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, which can be significantly 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, which 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. 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 and use the notion of a serializer to reduce the problem of verifying sequential consistency to that of checking language inclusion between finite state machines.

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 . 123
5.6 Verification of a snoopy cache coherence protocol 128

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 . . . 130

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, technically and otherwise, 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 for numerous technical discussions.

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. 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. 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, 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.

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 has made 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, Dil96] and symbolic [McM93, BHSV+96]. 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 ∼800 state variables and ∼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 *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. *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 *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 *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 [Hoa85, Mil89]. 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, BHSV+96]. They compute expressions that represent state sets using, in addition to positive boolean operations, the functions *pre* and $\widetilde{pre}$, which map a set of states to a subset of its *predecessor* states. Formally, given a set U of states, the set $pre(U)$ contains the states for which there exists a successor state in U, and the set $\widetilde{pre}(U)$ 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 $pre(U)$, until no more states can be added. Formally, the model checker calculates the least fixpoint of the expression $U = p \vee pre(U)$. 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 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 $\widetilde{pre}$ by the functions $post$ and $\widetilde{post}$, respectively, which map a set of states to a subset of its *successor* states. Formally, given a set U of states, the set $post(U)$ contains the states for which there exists a predecessor state in U, and the set $\widetilde{post}(U)$ 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., $\Box(p \to \Diamond q)$), 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 ω-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*-μ, a fixpoint calculus that is based on post operations in the same way in which the traditional μ-calculus, here called *pre*-μ, is based on pre operations [Koz83]. While *pre*-μ expressions refer to the future of a given state in a model, *post*-μ expressions refer to its past. Therefore, in stark contrast to the fact that every LTL and CTL specification has an equivalent expression in *pre*-μ, almost no LTL or CTL specification, including response, has an equivalent expression in *post*-μ. 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* $\mathcal{E}$. For a specification ϕ, which is true in some states of a model and false in others, the query $\mathcal{E}(\phi)$ is true in a model iff ϕ is false in all states of the model. The query logic *post*-$\mu_\emptyset$ contains all queries of the form $\mathcal{E}(\phi)$ and $\neg\mathcal{E}(\phi)$, for *post*-μ expressions ϕ. On the positive side, we prove that every ω-regular (Büchi) specification has an equivalent query in *post*-$\mu_\emptyset$. As with the translation from Büchi automata to *pre*-μ expressions [EL86, BC96], the translation from Büchi automata to *post*-$\mu_\emptyset$ 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*-$\mu_\emptyset$, 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*-$\mu_\emptyset$ 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.*

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*-μ and *post*-μ, and the query logics they induce. In Section 2.2, we translate Büchi automata into equivalent *post*-$\mu_\emptyset$ queries of alternation depth two, and co-Büchi automata into equivalent alternation-free *post*-$\mu_\emptyset$ 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 μ-calculi

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

The formulas of *pre*-μ 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, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\exists \quad \varphi$, or $\forall \quad \varphi$, for propositions $p \in P$, variables $X \in V$, and modal expressions φ and ψ. 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 φ_i is a modal expression, and the variables X_i are pairwise distinct. If $\lambda = \mu$, then B is a μ-*block*; otherwise B is a ν-*block*. Let the function $flag$ map each block to its flag. For the equational block B, let $vars(B) = \{X_i \mid i \in I\}$ be the set of variables on the left-hand sides of the equations of B. A *block tuple* $\mathcal{B} = \langle B_1, \ldots, B_n \rangle$ is a finite list of equational blocks such that the variable sets $vars(B_j)$, for $1 \leq j \leq n$, are pairwise disjoint. For the block tuple $\mathcal{B}$, let $vars(\mathcal{B}) = \bigcup_{1 \leq j \leq n} vars(B_j)$. For every variable $X \in vars(\mathcal{B})$, let $expand_{\mathcal{B}}(X)$ be the modal expression on the right-hand side of the unique equation in $\mathcal{B}$ whose left-hand

side is X. A *pre-μ formula* $\phi = \langle \mathcal{B}, X_0 \rangle$ is a pair that consists of a block tuple $\mathcal{B}$ and a variable $X_0 \in \mathit{vars}(\mathcal{B})$. The variable X_0 is called the *root variable* of ϕ. The formula ϕ is a *pre-μ sentence* if every variable that occurs in some modal expression of $\mathcal{B}$ is a member of $\mathit{vars}(\mathcal{B})$.

The semantics of a *pre-μ* formula is defined with respect to a Kripke structure and a valuation for the variables. A *Kripke structure* is a tuple $K = \langle P, W, R, L \rangle$ 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 *init*; a state $w \in W$ is *initial* if $\mathit{init} \in L(w)$. We define four functions *pre*, $\widetilde{\mathit{pre}}$, *post* and $\widetilde{\mathit{post}}$ from 2^W to 2^W as follows. For any set $U \subseteq W$ of states, let

$$
\begin{aligned}
\mathit{pre}(U) &= \{w \in W \mid \text{ there exists a state } w' \in U \text{ with } R(w, w')\}, \\
\widetilde{\mathit{pre}}(U) &= \{w \in W \mid \text{ for all states } w' \text{ with } R(w, w'), \text{ we have } w' \in U\}, \\
\mathit{post}(U) &= \{w \in W \mid \text{ there exists a state } w' \in U \text{ with } R(w', w)\}, \\
\widetilde{\mathit{post}}(U) &= \{w \in W \mid \text{ for all states } w' \text{ with } R(w', w), \text{ we have } w' \in U\}.
\end{aligned}
$$

The functions *pre* and *post* can be defined on states $u \in W$ as $\mathit{pre}(u) = \mathit{pre}(\{u\})$ and $\mathit{post}(u) = \mathit{post}(\{u\})$ 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 Γ and Γ' 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 $X \in V \backslash V'$. 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 $\sqsubseteq$ over $\mathcal{P}_{V,K}$ as $\Gamma \sqsubseteq \Gamma'$ iff for all $X \in V$, we have that $\Gamma(X) \subseteq \Gamma'(X)$. The tuple $\langle \mathcal{P}_{V,K}, \sqsubseteq \rangle$ is a complete lattice. For any subset $Y \subseteq \mathcal{P}_{V,K}$, let $\mathit{glb}(Y)$ and $\mathit{lub}(Y)$ 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 $\Gamma \sqsubseteq \Gamma'$ then $f(\Gamma) \sqsubseteq f(\Gamma')$. The function f is said to be continuous if for all chains $\Gamma_0 \sqsubseteq \Gamma_1 \sqsubseteq \Gamma_2 \sqsubseteq \ldots$, we have that $f(\mathit{lub}(\{\Gamma_i | i \geq 0\})) = \mathit{lub}(\{f(\Gamma_i | i \geq 0\}))$. 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 $lub(\{\Gamma | \Gamma \sqsubseteq f(\Gamma)\})$ and the least fixpoint of f is $glb(\{\Gamma | f(\Gamma) \sqsubseteq \Gamma\})$.*

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 $\perp$ 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 $\mu f = \bigcup_{i \geq 0} f^i(\perp)$ 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 Γ for a set V of variables, every modal expression φ over the propositions P and the variables V defines a set $\varphi^K(\Gamma) \subseteq W$ of states that satisfy φ. The definition proceeds inductively as follows.

- $p^K(\Gamma) = \{w \in W \mid p \in L(w)\}$.
- $(\neg p)^K(\Gamma) = \{w \in W \mid p \notin L(w)\}$
- $X^K(\Gamma) = \Gamma(X)$.
- $(\varphi \vee \psi)^K(\Gamma) = \varphi^K(\Gamma) \cup \psi^K(\Gamma)$.
- $(\varphi \wedge \psi)^K(\Gamma) = \varphi^K(\Gamma) \cap \psi^K(\Gamma)$.
- $(\exists \quad \varphi)^K(\Gamma) = pre(\varphi^K(\Gamma))$.
- $(\forall \quad \varphi)^K(\Gamma) = \widetilde{pre}(\varphi^K(\Gamma))$.

Given K, every block tuple $\mathcal{B} = \langle B_1, \ldots, B_n \rangle$ over P and V defines a function $\mathcal{B}^K$ from $\mathcal{P}_{V,K}$ to $\mathcal{P}_{V,K}$: inductively, if $n = 0$, then $\mathcal{B}^K(\Gamma) = \Gamma$; if B_1 is a μ-block, then $\mathcal{B}^K(\Gamma)$ is the

least fixpoint of the function $F^K_{\mathcal{B},\Gamma}$; if B_1 is a ν-block, then $\mathcal{B}^K(\Gamma)$ is the greatest fixpoint of $F^K_{\mathcal{B},\Gamma}$. The monotonic function $F^K_{\mathcal{B},\Gamma}$ from $\mathcal{P}_{V,K}$ to $\mathcal{P}_{V,K}$ is defined by

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

Note that for a *pre-*μ sentence $\phi = \langle \mathcal{B}, X_0\rangle$, the function $\mathcal{B}^K$ is a constant function. Given K, the sentence ϕ defines the set $\phi^K = \mathcal{B}^K(\Gamma)(X_0)$ of states (for any choice of Γ). For a state $w \in W$ and a *pre-*μ sentence ϕ, 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* ϕ, if there is an initial state w of K such that $w \models_K \phi$.[1] The *model-checking problem for pre-*μ is to decide, given a Kripke structure K and a *pre-*μ sentence ϕ, whether $K \models \phi$.

We now define the notion of alternation depth of a block tuple $\mathcal{B}$ as in [EL86, CKS92]. Given a block tuple $\mathcal{B} = \langle B_1,\ldots,B_n\rangle$, 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 $\mathit{vars}(B_j)$. 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 *label* from the set of blocks to $\mathbb{N}$ by induction as follows.

$$\begin{aligned} \mathit{label}(B_n) &= 1. \\ \mathit{label}(B_i) &= \max(Y_i \cup Z_i). \end{aligned}$$

where
$Y_i = \{1\} \cup \{\mathit{label}(B_j) | j > i \wedge \mathit{flag}(B_j) = \mathit{flag}(B_i) \wedge B_j \to^* B_i \to^* B_j\}$, and
$Z_i = \{1\} \cup \{1 + \mathit{label}(B_j) | j > i \wedge \mathit{flag}(B_j) \neq \mathit{flag}(B_i) \wedge B_j \to^* B_i \to^* B_j\}$.

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

Example 2.1 Consider the *pre-*μ formula $\varphi = \langle B_1, B_2\rangle$, where B_1 is a ν-block containing the single equation $X_1 = \exists\ \ X_2$ and B_2 is a μ-block containing the single equation $X_2 = (p \wedge X_1) \vee \exists\ \ X_2$. This formula is equivalent to the linear temporal logic formula $\Box\Diamond p$. Then $\mathit{label}(B_2) = 1$. Also $B_1 \to B_2$ and $B_2 \to B_1$. Therefore $Y_1 = \{1\}$ and $Z_1 = \{1, 2\}$, and $\mathit{label}(B_1) = \max(\{1, 2\}) = 2$. Thus we get that the alternation depth of φ is two. ■

The logic *post-*μ is obtained from the logic *pre-*μ by replacing the future modal operators $\exists$ and $\forall$ by the past modal operators $\exists$- and $\forall$- , with the interpretations

[1] Note that we work, for convenience, with the dual of the usual requirement that *all* initial states satisfy a *pre-*μ sentence.

$(\exists\text{-}\ \varphi)^K(\Gamma) = post(\varphi^K(\Gamma))$ and $(\forall\text{-}\ \varphi)^K(\Gamma) = \widetilde{post}(\varphi^K(\Gamma))$. The semantics of *post-μ* 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 *post-μ* sentence ϕ, define ϕ^{-1} to be the *pre-μ* sentence obtained from ϕ by replacing each occurrence of $\exists\text{-}$ and $\forall\text{-}$ by $\exists$ and $\forall$, respectively. Then, for every state w of K, we have $w \models_K \phi$ iff $w \models_{K^{-1}} \phi^{-1}$.

2.1.2 Query logics

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

- $\mathcal{E}(\phi)$, where ϕ is a sentence of $\mathcal{L}$,
- $\neg\theta_1$ and $\theta_1 \vee \theta_2$, where θ_1 and θ_2 are queries of $\mathcal{L}_\mathcal{E}$.

We sometimes refer to queries in $\mathcal{L}_\emptyset$ and $\mathcal{L}_\mathcal{E}$ as *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
- $K \models \theta_1 \vee \theta_2$ iff $K \models \theta_1$ or $K \models \theta_2$.

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 *bad* of error states, we would like to check whether there is an initial state that has a path to a state in *bad*. The property of having a path to a state in *bad* is expressed by the *pre-μ* sentence $\varphi = \langle B\rangle$, where B is a μ-block with a single equation $X = bad \vee \exists\ \ X$. This sentence is equivalent to the CTL property $\exists\Diamond bad$. Clearly φ talks about an event in the future whereas any

sentence in *post-*μ, which uses only past operators, talks only about events in the past. Therefore no sentence of *post-*μ is equivalent to φ. The invariant checking problem can also be expressed by the *pre-*μ query $\mathcal{E}(init \wedge \varphi)$. Consider the *post-*$\mu$ sentence $\varphi' = \langle B' \rangle$, where B' is a μ-block containing the single equation $X' = init \vee \exists\text{-}\ X'$. Then the invariant checking problem is also expressed by the *post-*μ query $\mathcal{E}(\varphi' \wedge bad)$. ∎

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 ϕ^K has been computed (either explicitly or implicitly, using BDDs), the evaluation of the query $\mathcal{E}(\phi)$ requires constant time. Therefore, checking a query in *pre-*$\mu_\mathcal{E}$ or *post-*$\mu_\mathcal{E}$ is not harder than model checking *pre-*μ or *post-*μ 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 β_1 and β_2 are pre-bisimilarity relations, so is $\beta_1 \cup \beta_2$. The pre-bisimilarity relation β 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 β 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 β is a pre-bisimulation between K^{-1} and K'^{-1}. The relation $\beta \subseteq W \times W'$ is a *init-post-bisimulation* between K and K' if β is a 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 β between K and K' such that $\beta(w, w')$.*

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

The following are easy extensions of the above result for *pre-μ*.

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 ϕ of pre-μ, we have $K \models \phi$ iff $K' \models \phi$.*

- *The following three statements are equivalent:*

  (1) *There is a pre-bisimulation between K and K'.*

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

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

Proof:

- ($\Longrightarrow$) Let β be an init-pre-bisimulation between K and K'. Consider a sentence ϕ of *pre-μ*. Suppose $K \models \phi$. Then there is an initial state w of K such that $w \models_K \phi$. Since β be 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 $K \models \phi$.

  ($\Longleftarrow$) Suppose for all sentences ϕ of *pre-μ*, 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-μ* sentence φ_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-μ* sentence can distinguish between w and w'. From Proposition 2.2, there is a pre-bisimilarity relation β_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 β_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) $\Longrightarrow$ (2)) Suppose β is a pre-bisimulation between K and K'. Let ϕ be a *pre-μ* 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 β is a pre-bisimulation, there is a state

$w \in W$ such that $\beta(w, w')$. Since $w \not\models \phi$ we have from Proposition 2.2 that $w' \not\models \phi$. Therefore, we have that $K' \models \mathcal{E}(\phi)$. Similarly, we can show that if $K' \models \mathcal{E}(\phi)$ then $K \models \mathcal{E}(\phi)$. It easily follows that $K \models \neg\mathcal{E}(\phi)$ iff $K' \models \neg\mathcal{E}(\phi)$.

((2) $\Longrightarrow$ (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 θ_1 and θ_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) $K \models \theta_1 \vee \theta_2$ iff $K \models \theta_1$ or $K \models \theta_2$ iff $K' \models \theta_1$ or $K' \models \theta_2$ iff $K' \models \theta_1 \vee \theta_2$.

((3) $\Longrightarrow$ (1)) Suppose for all queries θ of *pre*-$\mu_{\mathcal{E}}$, we have $K \models \theta$ iff $K' \models \theta$. In particular, this holds for all queries θ of *pre*-$\mu_{\emptyset}$. Consider a state w of K. Suppose for all states v of K' there is a *pre*-μ sentence φ_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*-μ sentence can distinguish between w and w'. From Proposition 2.2, there is a pre-bisimilarity relation β between K and K' such that $\beta(w, w')$. Thus, for each state $w \in W$ we can get pre-bisimilarity relation β_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'.

■

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 ϕ of post-μ, 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 θ of post-$\mu_{\emptyset}$, we have $K \models \theta$ iff $K' \models \theta$.*

  (3) *For all queries θ 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. ■

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.[2] Many useful linear properties, namely, the ω-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 $\ell : S \to 2^P$ that assigns to each state a set of propositions. [3] 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 $R(u_{i+1}, u_i)$ (resp. $r(u_{i+1}, u_i)$). The path π is *initialized* if u_0 is an initial state. We denote the concatination of two paths π_1 and π_2 by $\pi_1 \cdot \pi_2$. For a finite path or reverse-path π, let $last(\pi)$ 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 $Inf(\pi)$ we denote the set of states that appear in π infinitely often. The labeling functions L and ℓ 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

[2] 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.

[3] 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) lables the states. This notational difference between the two approaches is technical.

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 π_1 of K and an *accepting* initialized path π_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*-$\mu_\emptyset$, by translating automata into equivalent *post*-$\mu_\emptyset$ queries. The *post*-$\mu_\emptyset$ query θ is *equivalent* to the automaton A if for every Kripke structure K, we have $K \models \exists A$ iff $K \models \theta$.

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 $V = \{X_s | s \in S\} \cup \{X'_s | s \in S\} \cup \{X, X'\}$. For each state s of A, let γ_s be a variable-free and modality-free expression that characterizes the state s locally, namely, $\gamma_s = \bigwedge_{p \in \ell(s)} p \wedge \bigwedge_{p \notin \ell(s)} \neg p$. Now, let B_A be the following μ-block, which consists of $|S| + 1$ equations, with $vars(B_A) = \{X_s \mid s \in S\} \cup \{X\}$:

$$
\begin{aligned}
X_s &= \begin{cases} \gamma_s \wedge (init \vee \bigvee_{t \in pre(s)} \exists\text{-}\, X_t) & \text{if } s \in S_0, \\ \gamma_s \wedge \bigvee_{t \in pre(s)} \exists\text{-}\, X_t & \text{if } s \notin S_0, \end{cases} \\
X &= \bigvee_{f \in S_F} X_f.
\end{aligned}
$$

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 π of A is *accepting* if π 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*-$\mu_\emptyset$ query $\theta_A = \neg\mathcal{E}(\langle\langle B_A\rangle, X\rangle)$.

If a finite error trace exists, we would like a model-checking algorithm to detect it as soon as possible. By evaluating the query θ_A as follows (in the standard way), this can indeed be guaranteed. The evaluation of the μ-block B_A over a Kripke structure K proceeds in iterations. Let $X^K(i) \subseteq W$ denote the value of variable $X \in \mathit{vars}(B_A)$ after the i-th iteration, and let $\Gamma^K(i)$ denote the K-valuation that assigns to each variable in $X \in \mathit{vars}(B_A)$ the value $X^K(i)$. Initially, $X^K(0) = \emptyset$ for all $X \in \mathit{vars}(B_A)$. In all subsequent iterations, the value of each variable $X \in \mathit{vars}(B_A)$ is updated according to the equation $X^K(i+1) = \mathit{expand}(X)^K(\Gamma^K(i))$. Since the modal expressions in B_A are monotonic, we have that $\Gamma^K(0) \sqsubseteq \Gamma^K(1) \sqsubseteq \Gamma^K(2) \sqsubseteq \ldots$. 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 $w \in X_s^K(i)$ iff there is an initialized path π_1 of K ending in w and an initialized path π_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 $X_s^K(0) = \emptyset$ 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 π_1 and π_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

$$X_s^K(i+1) = \begin{cases} \gamma_s \wedge (\mathit{init} \vee \bigvee_{t \in \mathit{pre}(s)} \mathit{post}(X_t^K(i))) & \text{if } s \in S_0, \\ \gamma_s \wedge \bigvee_{t \in \mathit{pre}(s)} \mathit{post}(X_t^K(i)) & \text{if } s \notin S_0. \end{cases}$$

(Case 1. $s \notin S_0$) Suppose $v \in X_s^K(i+1)$. From the definition of $X_s^K(i+1)$, we have that $L(v) = \ell(s)$, and there is $t \in \mathit{pre}(s)$ and $u \in X_t^K(i)$ such that $R(u, v)$. From the induction hypothesis, there is an initialized path π_1 in K ending in u and an initialized path π_2 in A ending in t such that $L(\pi_1) = \ell(\pi_2)$ and $|\pi_1| = |\pi_2| \leq i$. Extend π_1 by v and π_2 by s to get the desired initialized paths. Suppose there are initialized paths π_1 of K ending in v and π_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 π_1 and t be the predecessor of s in π_2. 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 $u \in X^K_t(i)$. Using the definition of $X^K_s(i+1)$, we conclude that $v \in X^K_s(i+1)$.

(Case 2. $s \in S_0$**)** Suppose $v \in X^K_s(i+1)$. If $v \in (\gamma_s \wedge \mathit{init})^K$ 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 $v \notin (\gamma_s \wedge \mathit{init})^K$, we have that $L(v) = \ell(s)$, and there is $t \in \mathit{pre}(s)$ and $u \in X^K_t(i)$ such that $R(u,v)$. We follow the same reasoning as in Case 1. Suppose there are initialized paths π_1 of K ending in v and π_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 $v \in (\gamma_s \wedge \mathit{init})^K$ and hence $v \in X^K_s(i+1)$. If $|\pi_1| > 1$, we follow the same reasoning as in Case 1.

■

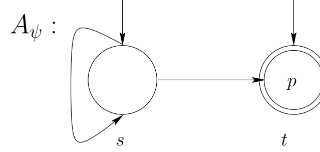
Corollary 2.6 *Let $A = \langle P, S, S_0, S_F, r, \ell\rangle$ be a safety automaton and $K = \langle P, W, R, L\rangle$ be a Kripke structure. Let $\Gamma \in \mathcal{P}_{V,K}$. For all $s \in S$ and $w \in W$, we have that $w \in \langle B_A\rangle^K(\Gamma)(X_s)$ iff there is an initialized path π_1 of K ending in w and an initialized path π_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-$\mu_\emptyset$ query $\theta_A = \neg\mathcal{E}(\langle\langle B_A\rangle, X\rangle)$ 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 θ_A.*

Proof: Let K be a Kripke structure and $\Gamma \in \mathcal{P}_{V,K}$. 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 π_1 of K and an initialized accepting path π_2 of A such that $L(\pi_1) = \ell(\pi_2)$ and $|\pi_1| = |\pi_2| = m$. Let the path π_1 end in state w and the path π_2 end in state s. From Lemma 2.5, we have that $w \in X^K_s(m)$ and therefore $w \in X^K(m+1)$. From the monotonicity property, we get that $\langle B_A\rangle^K(\Gamma)(X) \neq \emptyset$. Therefore $K \models \theta_A$. Suppose $K \models \theta_A$. Then $\langle B_A\rangle^K(\Gamma)(X) \neq \emptyset$. This means that $\langle B_A\rangle^K(\Gamma)(X_s) \neq \emptyset$ for some $s \in S_F$. Therefore, we have that $w \in X^K_s(i)$ 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 π satisfies ψ iff p always holds along π. The safety automaton A_ψ in the figure accepts exactly all the computations that satisfy ψ.



The μ-block B_A contains the equations

$$
\begin{aligned}
X_s &= \neg p \wedge (\mathit{init} \vee \exists\text{-}\ X_s). \\
X_t &= p \wedge (\mathit{init} \vee \exists\text{-}\ X_s). \\
X &= X_t.
\end{aligned}
$$

■

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 π of A is *accepting* if $\mathit{Inf}(\pi) \cap S_F \neq \emptyset$; that is, some accepting state of A occurs infinitely often in π. It is well-known [EL86, Dam94, BC96] that for every Büchi automaton A, there exists a *pre*-$\mu_\emptyset$ query ϑ_A such that for every Kripke structure K, we have $K \models \exists A$ iff $K \models \vartheta_A$. We now show that there exists also a *post*-$\mu_\emptyset$ query θ_A with the same property, thereby proving that the model-checking problem for Büchi automata lies in the intersection of *pre*-$\mu_\emptyset$ and *post*-$\mu_\emptyset$. We define two equational blocks: a ν-block B_1 and a μ-block B_2. The block B_1 contains the following $|S_F| + 1$ equations, with $\mathit{vars}(B_1) = \{X' \mid f \in S_F\} \cup \{X'\}$:

$$
\begin{aligned}
X'_f &= X_f \wedge \textstyle\bigvee_{t \in \mathit{pre}(f)} \exists\text{-}\ X'_t, \\
X' &= \textstyle\bigvee_{f \in S_F} X'_f.
\end{aligned}
$$

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

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

Then $\theta_A = \neg\mathcal{E}(\langle\langle B_1, B_2, B_A\rangle, X'\rangle)$. 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 θ_A is two. Since Büchi automata are expressively equivalent to the ω-regular languages, the query logic *post*-$\mu_\emptyset$ can specify all ω-regular properties.

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

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

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

$$X'_f \quad = \quad \gamma_f \wedge \bigvee_{t \in post(f)} \exists \quad X'_t,$$

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

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

Then $\theta_A = \neg\mathcal{E}(\langle\langle C_1, C_2, C_3\rangle, X'\rangle)$. 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*-$\mu_\emptyset$ or *pre*-$\mu_\emptyset$ 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 $w \in \langle B_2\rangle^K(\Gamma)(X'_s)$ iff there is $f \in S_F$, $u \in \Gamma(X'_f)$, a path π_1 in K from u to w, and a path π_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 π_1 in K from u to w, and a path π_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 $F^K_{\langle B_2\rangle,\Gamma}$ defined below.

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

where $E(s,\Gamma') = \gamma_s \wedge (\bigvee_{t \in pre(s) \cap S_F} post(\Gamma(X'_t)) \vee \bigvee_{t \in pre(s) \cap S \setminus S_F} post(\Gamma'(X'_t)))$. Consider an evaluation Γ' such that $F^K_{\langle B_2\rangle,\Gamma}(\Gamma') \subseteq \Gamma'$. Define a new valuation Δ as follows.

$$\Delta(X) = \begin{cases} \Gamma'(X'_s) \cap reach(s) & \text{if } X = X'_s \text{ forsome } s \in S \setminus S_F, \\ \Gamma'(X'_s) & \text{otherwise.} \end{cases}$$

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

Case 1. There is $t \in \mathit{pre}(s) \cap S_F$ and $v \in \Gamma(X'_t)$ such that $R(v, w)$. Because of the paths $\pi'_1 = v \cdot w$ and $\pi'_2 = t \cdot s$, we have that $w \in \mathit{reach}(s)$.

Case 2. There is some $t \in \mathit{pre}(s) \cap S \setminus S_F$ and $v \in \Delta(X'_t)$ such that $R(v, w)$. Since $v \in \Delta(X'_t)$ we have that $v \in \mathit{reach}(t)$ also. Hence, there is $f \in S_F$, $u \in \Gamma(X'_f)$, a path π_1 in K from u to v, and a path π_2 in A from f to t such that $L(\pi_1) = \ell(\pi_2)$. Let π'_1 be obtained from π_1 by appending w and π'_2 be obtained from π_2 by appending s. This gives us that $w \in \mathit{reach}(s)$.

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

($\Longleftarrow$) Suppose there is $f \in S_F$, $u \in \Gamma(X'_f)$, a path π_1 in K from u to w, and a path π_2 in A from f to s such that $L(\pi_1) = \ell(\pi_2)$. Suppose that $w \notin \langle B_2\rangle^K(\Gamma)(X'_s)$. Consider the smallest n such that $\pi_2(n) \in S \setminus S_F$ and $\pi_1(n) \notin \langle B_2\rangle^K(\Gamma)(X'_t)$ 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 $\pi_1(n-1) \in \langle B_2\rangle^K(\Gamma)(X'_{t'})$. In either case, since $\langle B_2\rangle^K(\Gamma)$ is a fixpoint we get that $\pi_1(n) \in \langle B_2\rangle^K(\Gamma)(X'_t)$ which is a contradiction. Thus we get that $w \in \langle B_2\rangle^K(\Gamma)(X'_s)$. ∎

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

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

($\Longrightarrow$) Suppose $K \models \exists A$. Then there is an initialized path π_1 in K and an accepting initialized path π_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 $0 \le m < n$, $\pi_1(m) = \pi_1(n)$, 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 *before* on $[m, n)$ as follows.

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

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

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

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

$$F^K_{\mathcal{B},\Gamma}(\Gamma')(X) = \begin{cases} (X_f \wedge (\bigvee_{t \in pre(f)} \exists\text{-}\ (X'_t)))^K(\langle B_2, B_A \rangle^K(\Gamma')) & \text{if} X \in \mathit{vars}(B_1), \\ \langle B_2, B_A \rangle^K(\Gamma')(X) & \text{otherwise.} \end{cases}$$

Notice that Γ 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.

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

where

$$\begin{aligned} E(f, \Gamma') \quad = \quad \langle B_A \rangle^K(\Gamma')(X_f) \quad \wedge \quad & (\bigvee_{t \in pre(f) \cap S_F} \mathit{post}(\Gamma'(X'_t)) \vee \\ & \bigvee_{t \in pre(f) \cap S \setminus S_F} \mathit{post}(\langle B_2 \rangle^K(\Gamma')(X'_t))). \end{aligned}$$

We now show that $\Delta \sqsubseteq F^K_{\mathcal{B},\Gamma}(\Delta)$ for any valuation Γ. All we need to show is that if $f \in S_F$ and $u \in \mathit{add}(f)$ then $u \in F^K_{\mathcal{B},\Gamma}(\Delta)(X'_f)$. Suppose $u \in \mathit{add}(f)$ 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 σ_1 in K to u and σ_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 $s = \pi_2(\mathit{before}(k))$. If $s \in S_F$ then $v = \pi_1(\mathit{before}(k)) \in \Delta(X'_s)$. Then there is $s \in \mathit{pre}(f) \cap S_F$ such that $u \in \mathit{post}(\Delta(X'_s))$. Therefore $u \in F^K_{\mathcal{B},\Gamma}(\Delta)(X'_f)$. If $s \in S \setminus S_F$ then there is a path $\sigma_1 = \pi_1(k), \pi_1(k+1), \ldots, \pi_1(n-1), \pi_1(m), \ldots, \pi_1(\mathit{before}(k))$ in K and a path $\sigma_2 = \pi_2(k), \pi_2(k+1), \ldots, \pi_2(n-1), \pi_2(m), \ldots, \pi_2(\mathit{before}(k))$ in A such that $L(\sigma_1) = \ell(\sigma_2)$. Therefore, from Lemma 2.8 we have that $v = \pi_1(\mathit{before}(k)) \in \langle B_2 \rangle^K(\Delta)(X'_s)$. Then there is $s \in \mathit{pre}(f) \cap S \setminus S_F$ such that $u \in \mathit{post}(\langle B_2 \rangle^K(\Delta)(X'_s))$. Therefore $u \in F^K_{\mathcal{B},\Gamma}(\Delta)(X'_f)$. Thus, we get that $\Delta \sqsubseteq F^K_{\mathcal{B},\Gamma}(\Delta)$. From Theorem 2.1, we get that the greatest fixpoint $\mathcal{B}^K(\Gamma)$ of $F^K_{\mathcal{B},\Gamma}$ is equal to $\mathit{lub}(\{\Gamma' | \Gamma' \sqsubseteq F^K_{\mathcal{B},\Gamma}(\Gamma')\})$. Therefore we have that $\Delta \sqsubseteq \mathcal{B}^K(\Gamma)$. This means that there is some $f \in S_F$ such that $\mathcal{B}^K(\Gamma)(X'_f) \neq \emptyset$. Therefore $\mathcal{B}^K(\Gamma)(X') \neq \emptyset$ and $K \models \theta_A$.

($\Longleftarrow$) Suppose $K \models \neg\mathcal{E}(\langle \mathcal{B}, X' \rangle)$. We incrementally construct infinite reverse-paths π_1 in K and π_2 in A maintaining the invariants that (1) $L(\pi_1) = \ell(\pi_2)$, and (2) if $\mathit{last}(\pi_1) = w$ and $\mathit{last}(\pi_2) = s$ then $s \in S_F$ and $w \in \mathcal{B}^K(\Gamma)(X'_s)$. Since $\mathcal{B}^K(X') \neq \emptyset$, there is $w \in \mathcal{B}^K(\Gamma)(X'_f)$ for some $f \in S_F$. Also $w \in \mathcal{B}^K(\Gamma)(X'_f)$ means that $w \in \mathcal{B}^K(\Gamma)(X'_f)$. Therefore $L(w) = \ell(f)$. Let $\pi_1 = w$ and $\pi_2 = f$ initially. Since $w \in \mathcal{B}^K(\Gamma)(X'_f)$, there is a $t \in \mathit{pre}(f)$ and $v \in \mathcal{B}^K(\Gamma)(X'_t)$ such that $R(v, w)$.

Case 1. $t \in S_F$ Augment π_1 by appending v and π_2 by appending t.

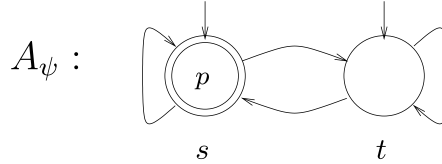
Case 2. $t \in S \setminus S_F$ We have that $\mathcal{B}^K(\Gamma) = \langle B_2 \rangle^K(\mathcal{B}^K(\Gamma))$. Therefore we get that $v \in \langle B_2 \rangle^K(\mathcal{B}^K(\Gamma))(X'_t)$. From Lemma 2.8, we get that there are $u \in W$, $f' \in S_F$, a path π'_1 in K from u to v, and a path π'_2 in A from f' to t such that $u \in \mathcal{B}^K(\Gamma)(X'_{f'})$ and $L(\pi'_1) = \ell(\pi'_2)$. Augment π_1 by appending π'_1 after reversing it and π_2 by appending π'_2 after reversing it.

The above process can be repeated ad infinitum to get infinite reverse-paths π_1 in K and π_2 in A such that $L(\pi_1) = \ell(\pi_2)$ and there are infinitely many occurrences of states in S_F on π_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 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 $\pi_1(m) \in \mathcal{B}^K(\Gamma)(X'_f)$ and therefore $\pi_1(m) \in \mathcal{B}^K(\Gamma)(X_f)$. From Theorem 2.7, we get that there are initialized paths σ_1 in

K to $\pi_1(m)$ and σ_2 in A to $\pi_2(m)$ such that $L(\sigma_1) = \ell(\sigma_2)$. We get initialized paths $\pi_1' = \sigma_1(0), \sigma_1(1), \ldots, \sigma_1(last(\sigma_1) - 1) \cdot (\pi_1(m), \pi_1(n-1), \pi_1(n-2), \ldots, \pi_1(m+1))^\omega$ in K and $\pi_2' = \sigma_2(0), \sigma_2(1), \ldots, \sigma_2(last(\sigma_2) - 1) \cdot (\pi_2(m), \pi_2(n-1), \pi_2(n-2), \ldots, \pi_2(m+1))^\omega$ in A such that $L(\pi_1') = \ell(\pi_2')$. ■

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 $pre\text{-}\mu_\emptyset \cap post\text{-}\mu_\emptyset$. 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 $post\text{-}\mu_\emptyset$ is also exponential.

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



The ν-block B_1 contains the equations

$$X_s' = X_s \wedge (\exists\text{-}\ X_s' \vee \exists\text{-}\ X_t').$$
$$X' = X_s'.$$

The μ-block B_2 contains the single equation $X_t' = \neg p \wedge (\exists\text{-}\ X_s' \vee \exists\text{-}\ X_t')$. The μ-block B_A contains the equations

$$X_s = p \wedge (init \vee \exists\text{-}\ X_s \vee \exists\text{-}\ X_t).$$
$$X_t = \neg p \wedge (init \vee \exists\text{-}\ X_s \vee \exists\text{-}\ X_t).$$

■

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-

Büchi automaton can be translated into an *alternation-free pre-*$\mu_\emptyset$ query.[4] 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-*$\mu_\emptyset$ query θ_A, thereby proving that the model-checking problem for co-Büchi automata lies in the intersection of alternation-free *pre-*$\mu_\emptyset$ and alternation-free *post-*$\mu_\emptyset$. A *co-Büchi automaton* A is a finite automaton such that a path π of A is *accepting* if $\mathit{Inf}(\pi) \subseteq S_F$; that is, all the non-accepting states of A occur in π only finitely often. We define an equational ν-block B_3 that contains the following $|S_F| + 1$ equations, with $\mathit{vars}(B_3) = \{X'_f \mid f \in S_F\} \cup \{X'\}$:

$$\begin{array}{rcl} X'_f & = & X_f \wedge \bigvee_{t \in pre(f) \cap S_F} \exists\text{-}\ X'_t, \\ X' & = & \bigvee_{f \in S_F} X'_f. \end{array}$$

Then, $\theta_A = \neg\mathcal{E}(\langle\langle B_3, B_A\rangle, X'\rangle)$. Note that θ_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-$\mu_\emptyset$ query $\theta_A = \neg\mathcal{E}(\langle\langle B_3, B_A\rangle, X'\rangle)$ is equivalent to A. The query θ_A can be constructed in linear time and its alternation depth is one.*

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

($\Longrightarrow$) Suppose $K \models \exists A$. Then there is an initialized path π_1 in K and an accepting initialized path π_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 *before* on $[m, n)$ as follows (exactly as in the proof of Theorem 2.9).

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

[4] The results in [KV98a] refer to sentences of the form $\forall A$, for deterministic Büchi automata A. Since an ω-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 $add(s) = \{\pi_1(k) | m \leq k < n \text{ and } \pi_2'(k) = s\}$. Define the valuation Δ as follows.

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

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

$$F^K_{\mathcal{B},\Gamma}(\Gamma')(X) = \begin{cases} (X_f \wedge (\bigvee_{t \in pre(f) \cup S_F} \exists\text{-}\ (X_t')))^K(\langle B_A \rangle^K(\Gamma')) & \text{if} X \in vars(B_3), \\ \langle B_A \rangle^K(\Gamma')(X) & \text{otherwise.} \end{cases}$$

Notice that Γ 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.

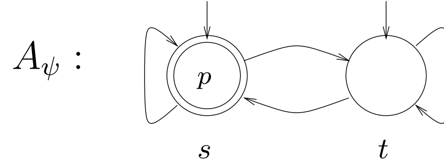
$$F^K_{\mathcal{B},\Gamma}(\Gamma')(X) = \begin{cases} \bigvee_{f \in S_F} \Gamma'(X_f') & \text{if } X = X', \\ \langle B_A \rangle^K(\Gamma')(X_f) \wedge & \text{if } X = X_f' \text{ for some } f \in S_F, \\ \bigvee_{t \in pre(f) \cap S_F} post(\Gamma'(X_t')) & \\ \langle B_A \rangle^K(\Gamma')(X) & \text{otherwise.} \end{cases}$$

We now show that $\Delta \sqsubseteq F^K_{\mathcal{B},\Gamma}(\Delta)$ for any valuation Γ. All we need to show is that if $f \in S_F$ and $u \in add(f)$ then $u \in F^K_{\mathcal{B},\Gamma}(\Delta)(X_f')$. Suppose $u \in add(f)$ 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 σ_1 in K to u and σ_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 $s = \pi_2(before(k))$. Since $s \in S_F$ then $v = \pi_1(before(k)) \in \Delta(X_s')$. Then there is $s \in pre(f) \cap S_F$ such that $u \in post(\Delta(X_s'))$. Therefore $u \in F^K_{\mathcal{B},\Gamma}(\Delta)(X_f')$. Thus, we get that $\Delta \sqsubseteq F^K_{\mathcal{B},\Gamma}(\Delta)$. From Theorem 2.1, we get that the greatest fixpoint $\mathcal{B}^K(\Gamma)$ of $F^K_{\mathcal{B},\Gamma}$ is equal to $lub(\{\Gamma' | \Gamma' \sqsubseteq F^K_{\mathcal{B},\Gamma}(\Gamma')\})$. Therefore we have that $\Delta \sqsubseteq \mathcal{B}^K(\Gamma)$. This means that there is some $f \in S_F$ such that $\mathcal{B}^K(\Gamma)(X_f') \neq \emptyset$. Therefore $\mathcal{B}^K(\Gamma)(X') \neq \emptyset$ and $K \models \theta_A$.

($\Longleftarrow$) Suppose $K \models \neg\mathcal{E}(\langle \mathcal{B}, X' \rangle)$. We incrementally construct infinite reverse-paths π_1 in K and π_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 $w = last(\pi_1)$ and $s = last(\pi_1)$ then $w \in \mathcal{B}^K(\Gamma)(X_s')$. There are f and w such that $f \in S_F$ and $w \in \mathcal{B}^K(\Gamma)(X_f')$. Since $w \in \mathcal{B}^K(\Gamma)(X_f')$, we have that $w \in \mathcal{B}(\Gamma)(X_f)$. Therefore $L(w) = \ell(f)$. Let $\pi_1 = w$ and

$\pi_2 = f$ initially. Since $w \in \mathcal{B}^K(\Gamma)(X'_f)$, there is a $t \in pre(f) \cup S_F$ and $v \in \mathcal{B}^K(\Gamma)(X'_t)$ such that $R(v, w)$. Augment π_1 by appending v and π_2 by appending t. The above process can be repeated ad infinitum to get infinite reverse-paths π_1 in K and π_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 $0 \leq m < n$ 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 $\pi_1(m) \in \mathcal{B}^K(\Gamma)(X'_f)$ and therefore $\pi_1(m) \in \mathcal{B}^K(\Gamma)(X_f)$. From Theorem 2.7, we get that there are initialized paths σ_1 in K to $\pi_1(m)$ and σ_2 in A to $\pi_2(m)$ such that $L(\sigma_1) = \ell(\sigma_2)$. We get initialized paths $\pi'_1 = \sigma_1(0), \sigma_1(1), \ldots, \sigma_1(last(\sigma_1)-1) \cdot (\pi_1(m), \pi_1(n-1), \pi_1(n-2), \ldots, \pi_1(m+1))^\omega$ in K and $\pi'_2 = \sigma_2(0), \sigma_2(1), \ldots, \sigma_2(last(\sigma_2)-1) \cdot (\pi_2(m), \pi_2(n-1), \pi_2(n-2), \ldots, \pi_2(m+1))^\omega$ in A such that $L(\pi'_1) = \ell(\pi'_2)$. ∎

Example 2.5 Consider the LTL formula $\psi = \Diamond\Box p$. A computation π satisfies ψ iff p always holds after a certain point along π. The co-Büchi automaton A_ψ in the figure accepts exactly all the computations that satisfy ψ.



The ν-block B_3 contains the equations

$$
\begin{aligned}
X'_s &= X_s \wedge \exists\text{-}\ X_s. \\
X &= X'_s.
\end{aligned}
$$

∎

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 θ is *equivalent* to a *pre*-μ sentence ϕ if for every Kripke structure K, we have $K \models \phi$ iff $K \models \theta$.

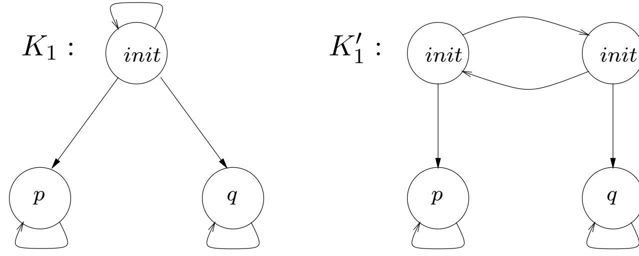


Figure 2.1: K_1 and K'_1 are post-bisimilar

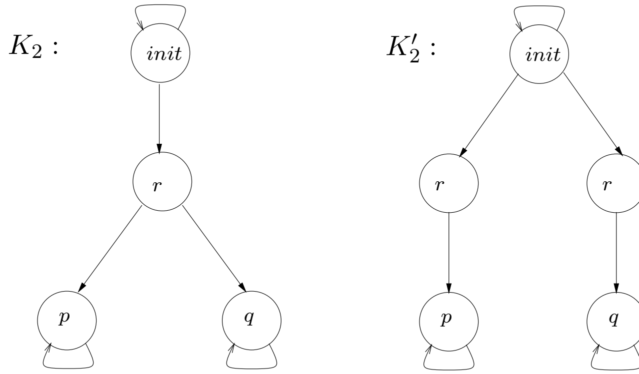


Figure 2.2: K_2 and K'_2 are post-bisimilar

Proposition 2.11 *There exist pre-μ sentences (in fact, CTL sentences) that have no equivalent post-$\mu_{\mathcal{E}}$ queries.*

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

1. Consider the formula $\phi_1 = \langle\langle\langle\mu, \{X = \exists \quad p \wedge \exists \quad q\}\rangle\rangle, X\rangle$, which is equivalent to the CTL sentence $\exists \quad p \wedge \exists \quad q$. It distinguishes between the post-bisimilar Kripke structures K_1 and K'_1 in Figure 2.1. Indeed, K_1 satisfies ϕ_1 but K'_1 does not satisfy ϕ_1.

2. Consider the formula $\phi_2 = \langle\langle\langle\mu, \{X_1 = (r \wedge X_2 \wedge X_3) \vee \exists \quad X_1\}\rangle, \langle\mu, \{X_2 = p \vee \exists \quad X_2\}\rangle, \langle\mu, \{X_3 = q \vee \exists \quad X_3\}\rangle\rangle, X_1\rangle$, which is equivalent to the CTL sentence $\exists\Diamond(r \wedge$

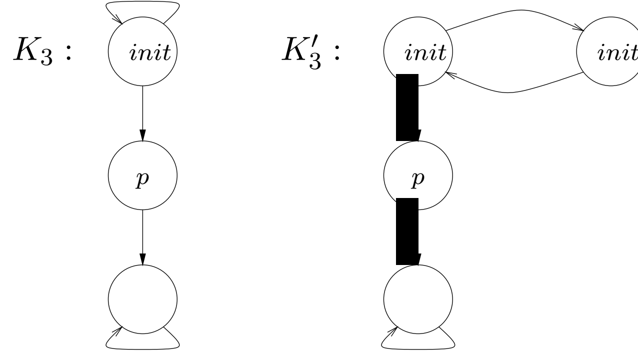


Figure 2.3: K_3 and K_3' are post-bisimilar but not pre-bisimilar

$\exists\Diamond p \wedge \exists\Diamond q)$. It distinguishes between the post-bisimilar Kripke structures K_2 and K_2' in Figure 2.2. Indeed, K_2 satisfies ϕ_2 but K_2' does not satisfy ϕ_2.

3. Consider the formula $\phi_3 = \langle\langle\langle \nu, \{X = \exists \quad p \wedge \exists \quad X\}\rangle\rangle, X\rangle$, which is equivalent to the CTL sentence $\exists\Box\exists \quad p$. It distinguishes between the post-bisimilar Kripke structures K_3 and K_3' in Figure 2.3. Indeed, K_3 satisfies ϕ_3 but K_3' does not satisfy ϕ_3.

■

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

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 ϕ_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 $\mathcal{L}_1$ 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 ϕ_1 of $\mathcal{L}_1$ such that $K \models \phi_1$ but $K' \not\models \phi_1$, then there is a sentence ϕ_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 $\mathcal{L}_2$ is as distinguishing as $\mathcal{L}_1$ but $\mathcal{L}_1$ is not as distinguishing as $\mathcal{L}_2$. In this section, we study the distinguishing and the expressive powers of *pre-*μ and *post-*μ and the query logics they induce.

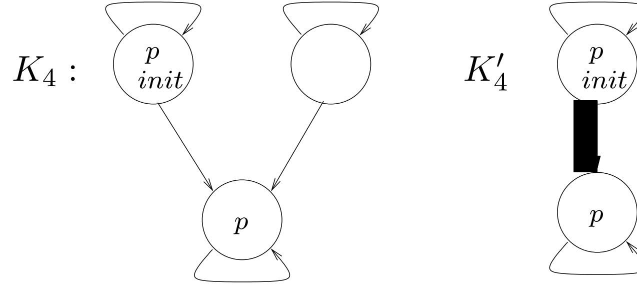
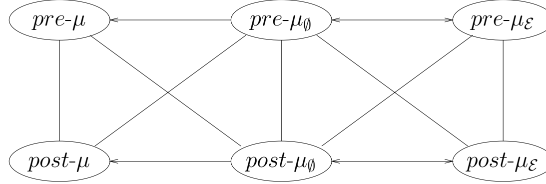


Figure 2.4: K_4 and K'_4 are init-post-bisimilar but not post-bisimilar

Proposition 2.12 *The distinguishing 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 distinguishing as* $\mathcal{L}_2$*. A line without arrow indicates incomparability.*



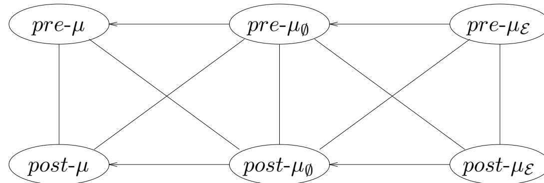
Proof: Proposition 2.3 implies that the distinguishing powers of *pre-*$\mu_\emptyset$ and *pre-*$\mu_\mathcal{E}$ coincide. Similarly, proposition 2.4 implies that the distinguishing powers of *post-*$\mu_\emptyset$ and *post-*$\mu_\mathcal{E}$ coincide. In order to prove the incomparability results, 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 *init*, then the hierarchy of pre logics collapses. Formally, suppose that $K \models \mathcal{E}(\neg\exists\diamondminus\, init)$. Then for every formula ϕ of *pre-μ*, we have that $K \models \mathcal{E}(\phi)$ iff $K \not\models \exists\Diamond\phi$.
- If we restrict attention to Kripke structures in which all the states can reach some state satisfying *init*, then the hierarchy of post logics collapses. Formally, suppose that $K \models \mathcal{E}(\neg\exists\Diamond init)$. Then for every formula ϕ of *post-μ*, we have that $K \models \mathcal{E}(\phi)$ iff $K \not\models \exists\diamondminus\, \phi$.

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 ϕ iff K satisfies the query $\neg\mathcal{E}(init \wedge \phi)$, the query logics *pre-$\mu_\emptyset$* and *post-$\mu_\emptyset$* are more expressive than *pre-μ* and *post-μ*, respectively. In order to prove the advantage of the full query logics *pre-$\mu_\mathcal{E}$* and *post-$\mu_\mathcal{E}$* over its subsets *pre-$\mu_\emptyset$* and *post-$\mu_\emptyset$*, it is easy to see that no query of the query logics *pre-$\mu_\emptyset$* and *post-$\mu_\emptyset$* is equivalent to the query $\mathcal{E}(p) \vee \mathcal{E}(q)$.
■

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 ω-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 ω-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*-$\mu_\emptyset$ 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*-$\mu_\emptyset$, 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*-$\mu_\emptyset$ query of alternation depth two, and when A is a co-Buchi automaton, the required formula is of the form $\Diamond\Box p$, which is equivalent to a *post*-$\mu_\emptyset$ query of alternation depth one.

We showed that every query that can be specified in both *pre*-$\mu_\emptyset$ and *post*-$\mu_\emptyset$ cannot distinguish between structures that are both pre-bisimilar and post-bisimilar. The exact characterization of the intersection *pre*-$\mu_\emptyset \cap$ *post*-$\mu_\emptyset$ remains open. In [GK94], the authors identified a set of temporal-logic sentences called equi-linear. In particular, a *pre*-μ 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 $\exists\Box\exists \quad p$ is not equi-linear, the CTL sentence $\exists\Box\exists\Diamond p$ 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*-μ sentence is equi-linear iff there is an equivalent *post*-$\mu_\emptyset$ query.

2.4.2 Union of pre and post logics

In this chapter, we primarily think of $post$-$\mu_{\mathcal{E}}$ 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 $post$-$\mu_{\emptyset}$ query. In addition, $post$-μ also permits the easy evaluation of branching past temporal operators that cannot be evaluated using pre-μ. For example, the sentence $\forall\Box(grant \rightarrow (\neg init)\forall\widetilde{\mathcal{W}}req)$, where $\widetilde{\mathcal{W}}$ 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 $post$-$\mu_{\emptyset}$ query, but no equivalent pre-$\mu_{\mathcal{E}}$ query; that is, it can be model checked by symbolic forward state traversal but not by symbolic backward state traversal.

While the intersection pre-$\mu_{\mathcal{E}} \cap post$-$\mu_{\mathcal{E}}$ identifies the queries that can be model checked by both forward and backward symbolic state traversal, it is the "union" $(pre$-$\mu \cup post$-$\mu)_{\mathcal{E}}$[5] that identifies the queries that can be model checked at all symbolically, by mixed forward and backward state traversal.[6] The logics pre-$\mu_{\mathcal{E}}$ and $post$-$\mu_{\mathcal{E}}$ are strict subsets of $(pre$-$\mu \cup post$-$\mu)_{\mathcal{E}}$. For example, the pre-$\mu \cup post$-μ sentence $init \wedge \exists\ \ p_1 \wedge \exists\ \ p_2 \wedge \exists\text{-}\ q_1 \wedge \exists\text{-}\ q_2$ is not expressible in either pre-$\mu_{\mathcal{E}}$ or $post$-$\mu_{\mathcal{E}}$. Furthermore, it is the alternation-free fragment of $(pre$-$\mu \cup post$-$\mu)_{\mathcal{E}}$ 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 $(pre$-$\mu \cup post$-$\mu)_{\emptyset}$.

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

[5] By the *union* pre-$\mu \cup post$-μ we refer to the logic with all four modal operators $\exists\ $, $\forall\ $, $\exists\text{-}$, and $\forall\text{-}$. It has, of course, strictly more sentences than the union of the sets of pre-μ and $post$-μ sentences.

[6] In fact, not only can model checking algorithms be extended from pre-μ to $(pre$-$\mu \cup post$-$\mu)_{\mathcal{E}}$ without extra cost, the satisfiability problem for the union is also not harder than the satisfiability problem for either pre-μ or $post$-μ [Var98].

WINDOW	*Forward*	*Backward*	*Reach-optimized backward*
2 (30)	18	222	91
3 (45)	300	4584	-
4 (50)	5231	-	-

Table 2.1: Experimental results for sliding window protocol

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 ϕ, 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 ϕ is given by the LTL sentence

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

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 ϕ 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.

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 $P_1 \| P_2 \preceq Q$ (where $\preceq$ is the trace-containment relation) by introducing abstraction and witness modules. Suppose that the state space of the implementation $P_1 \| P_2$ is too large to be handled by exhaustive search algorithms. A naive compositional approach would attempt to prove both $P_1 \preceq Q$ and $P_2 \preceq Q$, and then conclude $P_1 \| P_2 \preceq Q$. Though sound, the naive 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

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 $P_1 \| P_2 \preceq Q$ from the two proof obligations $P_1 \| A_2 \preceq A_1 \| Q$ and $A_1 \| P_2 \preceq Q \| A_2$.

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 $P' = P_1 \| A_2 \preceq A_1 \| Q = Q'$, 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 $P' \| W$, and it suffices to prove $P' \| W \preceq Q'$ 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

[MC81, Sta85, AL95, AH96, McM97]. 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, BBLS92, 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 a $'$ 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 $f(b) = \mathit{false}$ is a valuation for V. A *transition constraint* A is a 4-tuple $\langle \mathit{Priv}(A), \mathit{Obs}(A), \mathcal{I}(A), \mathcal{U}(A) \rangle$ with the following components:

- A finite set $\mathit{Priv}(A)$ of *private* variables and a finite set $\mathit{Obs}(A)$ of *observable* variables, such that $\mathit{Priv}(A) \cap \mathit{Obs}(A) = \emptyset$. The set $\mathit{Var}(A)$ is the union of $\mathit{Priv}(A)$ and $\mathit{Obs}(A)$.
- A *initial predicate* $\mathcal{I}(A)$ over $\mathit{Var}(A)'$ and an *update predicate* $\mathcal{U}(A)$ over $\mathit{Var}(A) \cup \mathit{Var}(A)'$.

A state of a transition constraint A is a valuation for $\mathit{Var}(A)$. 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 $\mathit{Obs}(A)$. If s is a valuation to a set of variables and $P \subseteq \mathit{Var}(A)$, we use $[s]_P$ to denote the valuation restricted to the variables in P. We also use $[s]_A$ to denote $[s]_{\mathit{Obs}(A)}$. 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 $Y \subseteq \mathit{Var}(A)$ for some constraint A. Then A is said to be *non-blocking* for Y if (1) for all valuations Γ of Y there is an initial state t of A such that

$[t]_Y = \Gamma$, and (2) for all states s of A and valuations Γ of Y' there is a state t of A such that $s \rightarrow_A t$ 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* $A\|B$ is the transition constraint $\langle Priv(A) \cup Priv(B), Obs(A) \cup Obs(B), \mathcal{I}(A) \wedge \mathcal{I}(B), \mathcal{U}(A) \wedge \mathcal{U}(B)\rangle$.

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 $\overline{s}$ of A, the projection $[\overline{s}]_B$ 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 $\overline{s}$ of A, the projection $[\overline{s}]_B$ 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 $s \rightarrow_A t$, then $[s]_B \rightarrow_B [t]_B$.*

Proof: ($\Longrightarrow$) Since B is projection refinable by A, we have that for every trajectory $\overline{s}$ of A the projection $[\overline{s}]_B$ 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 $s_0, s_1, \ldots, s$ of A ending in s. Since $s \rightarrow_A t$ 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 $[s]_B \rightarrow_B [t]_B$.

($\Longleftarrow$) We show that for every trajectory $\overline{s}$ of A, the sequence $[\overline{s}]_B$ is a trajectory of B by induction on the length of $\overline{s}$.

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

Inductive step. Suppose $\overline{s} = s_0, s_1, \ldots, s_n$ is a trajectory of A for some $n > 0$. Then $\overline{s}' = s_0, s_1, \ldots, s_{n-1}$ is a trajectory of A, state s_{n-1} is a 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^u 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^u 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 $A \| W \preceq B^u$ 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) $Priv(B) \subseteq Obs(W)$, and (3) W is non-blocking on $Obs(W) \cap Obs(A)$. Then B^u is projection refinable by $A \| W$, and if $A \| W \preceq B^u$ then $A \preceq B$.*

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

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

Base step. Let $\overline{s} = s_0$. 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 $[t_0]_{Var(A)} = s_0$.

Inductive step. Let $\overline{s} = s_0, s_1, \ldots, s_n$ 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 $s = [s_{n-1}]_{Var(W)}$. 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 $[t]_{Var(A)} = s$. Thus we get a trajectory $\overline{t} = t_0, t_1, \ldots, t_{n-1}, t$ of $A\|W$ such that $[\overline{s}]_A = [\overline{t}]_A$. ∎

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^τ to denote the transition constraint with the same variables as A, which restricts their valuations only up to τ update rounds. Formally, a state sequence $\overline{s} = s_0, \ldots, s_n$ is a trajectory of A^τ if (1) $n \leq \tau$ and $\overline{s}$ 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^τ.

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 $Z(B_i) = \{B_j | B_j \prec B_i\}$, and let $Z^C(B_i) = \{B_j | B_j \notin Z(B_i)\}$. For each B_i, let C_i be some composition of transition constraints from A, let D_i be some composition of transition constraints from $Z(B_i)$, and let E_i be some composition of transition constraints from $Z^C(B_i)$. If $C_i^\tau \| D_i^\tau \| E_i^{\tau-1} \preceq B_i^\tau$ 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 $\langle B_j, \tau' \rangle \prec \langle B_k, \tau \rangle$ iff $B_j \prec B_k$ or $B_j = B_k$ and $\tau' < \tau$. It is not too difficult to see that $\prec$ 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 $\prec$ 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 $\langle B_j, \tau' \rangle \prec \langle B_k, \tau \rangle$. Then we show that $A \models \langle B_k, \tau \rangle$. Consider a trace σ of A of length $\tau + 1$.

1. Since σ is a trace of A, we have that σ is a trace of A_i and hence a trace of A_i^τ for all $1 \leq i \leq n$.

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

3. **Case 1.** If $\tau = 0$ then σ 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 σ' 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 σ' is a trace of B_j and hence of $B_j^{\tau-1}$ for all $1 \leq j \leq m$. Therefore σ 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 σ is a trace of C_k^τ, D_k^τ and $E_k^{\tau-1}$, we get that σ is a trace of B_k^τ 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 *SndSpec*$\|$*RcvSpec*, where *SndSpec* and *RcvSpec* are transition constraints shown in Figure 3.1. The *SndSpec* constraint has observable variables *ack* and *snd*, and a private variable *wait*. When a message is produced during some round, the variable *snd* is set to *true*, and reset to *false* in the next round. Further, the variable *wait* is set to *true*, to denote that *SndSpec* is waiting for an *ack*. When *wait* is *true* and *SndSpec* receives an *ack*, *wait* is reset to *false*. The *RcvSpec* constraint has observable variables *snd* and *ack*, and a private variable *pending*. When *RcvSpec* receives a message (indicated by *snd* being *true*), it records the event by setting *pending* to *true*. It then waits for an arbitrary amount of time, and then resets *wait* to *false*, simultaneously sending an *ack*. The implementation of the protocol is given by *SndImpl*$\|$*RcvImpl*, where *SndImpl* and *RcvImpl* are transition constraints shown in Figure 3.2. The constraint *SndImpl* differs from *SndSpec* in its response to an incorrect *ack*. If *SndImpl* receives an *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, *SndSpec* just ignores the offending *ack*. Thus *SndImpl* $\not\preceq$ *SndSpec*. Similarly, *RcvImpl* differs from *RcvSpec*, if it receives a message when *pending* is *true*. Consequently *RcvImpl* $\not\preceq$ *RcvSpec*. However, we can apply the assume-guarantee rule in Theorem 3.3 in

constraint *SndSpec*
  observable *snd*,*ack*:*bool*
  private *wait*:*bool*
  init
    $\neg wait' \wedge \neg snd'$
  update
    $(\neg wait \vee (wait \wedge ack \wedge \neg wait')) \wedge (\neg wait \wedge wait') = snd'$
endconstraint
constraint *RcvSpec*
  observable *ack*,*snd*:*bool*
  private *pending*:*bool*
  init
    $\neg pending' \wedge \neg ack'$
  update
    $((\neg pending \wedge snd \wedge pending') \vee pending) \wedge (pending \wedge \neg pending') = ack'$
endconstraint

Figure 3.1: Specification of sender/receiver

constraint *SndImpl*

observable *snd*,*ack*:*bool*

private *wait*,*error*:*bool*

init

$\neg wait' \wedge \neg error' \wedge \neg snd'$

update

$((\neg wait \wedge error' = error)\vee$

$(wait \wedge ack \wedge \neg wait' \wedge error' = error)\vee$

$(\neg wait \wedge ack \wedge error')\vee$

$(error \wedge error' = error))$

$\wedge$

$(\neg wait \wedge wait') = snd'$

endconstraint

constraint *RcvImpl*

observable *ack*,*snd*:*bool*

private *pending*,*error*:*bool*

init

$\neg pending' \wedge error' \wedge \neg ack'$

update

$((\neg pending \wedge snd \wedge pending' \wedge error' = error)\vee$

$(pending \wedge error' = error)\vee$

$(pending \wedge snd \wedge error'))$

$\wedge \quad ((pending \wedge \neg pending' = ack') \vee error)$

endconstraint

Figure 3.2: Implementation of sender/receiver

the following way. Let $\prec$ be the empty partial order on the set *SndSpec*, *RcvSpec*. Then, we can show that $\mathit{SndImpl}^{\tau} \| \mathit{RcvSpec}^{\tau-1} \preceq \mathit{SndSpec}^{\tau}$ and $\mathit{RcvImpl}^{\tau} \| \mathit{SndSpec}^{\tau-1} \preceq \mathit{RcvSpec}^{\tau}$ for all τ. Thus we conclude that $\mathit{SndImpl} \| \mathit{RcvImpl} \preceq \mathit{SndSpec} \| \mathit{RcvSpec}$. ■

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 $\mathit{Intf}(P)$ and $\mathit{Extl}(P)$ 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 *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 *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 *initial command*, which specifies the possible initial values for the variables of the atom, and an *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.

Example 3.2 Consider the simple instruction set architecture defined by the reactive module *ISA* of Figure 3.3. The module *ISA* has five external variables (inputs) —the operation *op*, the immediate operand *inp*, the source registers *src1* and *src2*, and the destination register *dest*. There are two interface variables (outputs) —the value *out* of a *STORE* instruction, and a boolean variable *stall*, which indicates if the current inputs have been accepted. If the value of *stall* is *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 *isaRegFile*.

A round of the module *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 *ISAStall* decides nondeterministically if the current inputs are processed, by setting *stall* to *true* or *false*. The third subround belongs to the atom *ISARegFile*. If the updated value of *stall* is *false*, then the current instruction is processed appropriately. If the operation is *AND* or *OR*, it is performed on the source registers and the result is placed into the destination register. If the operation is *LOAD*, the immediate operand is assigned to the destination register. The fourth subround belongs to the atom *ISAOut*. If the updated value of *stall*

module *ISA*
  external $op, inp, src1, src2, dest$
  interface $out, stall$
  private *isaRegFile*
  atom *ISAStall* **controls** *stall*
    init update
      $[] true \rightarrow stall' := nondet$
  atom *ISARegFile* **controls** *isaRegFile*
    init
      $[] true \rightarrow$ **forall** i **do** $isaRegFile'[i] := 0$
    update
      $[] \neg stall' \wedge op' = LOAD \rightarrow isaRegFile'[dest'] := inp'$
      $[] \neg stall' \wedge op' = AND \rightarrow isaRegFile'[dest'] := isaRegFile[src1'] \wedge isaRegFile[src2']$
      $[] \neg stall' \wedge op' = OR \rightarrow isaRegFile'[dest'] := isaRegFile[src1'] \vee isaRegFile[src2']$
  atom *ISAOut* **controls** *out*
    init update
      $[] \neg stall' \wedge op' = STORE \rightarrow out' := isaRegFile[dest']$

Figure 3.3: Instruction set architecture

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

Let P be a module. It can be interpreted as a transition constraint $\mathcal{C}(P)$, which has the same set of traces as P. The observable variables of $\mathcal{C}(P)$ are the observable variables of P, and the private variables of $\mathcal{C}(P)$ are the private variables of P. The initial and update predicates of $\mathcal{C}(P)$ can be derived from the initial and update commands of P, respectively. The initial predicate of $\mathcal{C}(P)$ is is defined to be *true* in states s such that s can result from the initial round of P. In similar spirit, the update predicate of $\mathcal{C}(P)$ 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 *awaits* relation on the variables of P is denoted by $\succ_P$ and defined

as $y \succ_P x$ 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 *awaits-closed* in P if for all $y \in Y$ and for all $x \in \mathit{Var}(P)$, we have that (1) if $y \succ_P x$ 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.

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

The module Q is *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 *refines* the module Q, written $P \preceq Q$, if (1) Q is refinable by P, and (2) for every trajectory $\overline{s}$ of P, the projection $[\overline{s}]_Q$ is a trace of Q.

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

Parallel Composition. The rules for compatibility of modules are more stringent than in the case of transition constraints. Two modules P and Q are *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 *composition* $P \| Q$ is the module whose atoms are the union of the atoms from P and Q. The interface variables of $P \| Q$ are the interface variables of P and Q, and the private variables of $P \| Q$ are the private variables of P and Q. The external variables of $P \| Q$ 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.

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 *stall*; the module *ISARegFile* has the external variables *op*, *inp*, *src1*, *src2*, *dest*, and *stall*, and the interface variable *isaRegFile*; the module *ISAOut* has the external variables *op*, *dest*, *stall*, and *isaRegFile*, and the interface variable *out*. The operation **hide** makes the interface variable *isaRegFile* private:

$$\mathit{ISA} \;=\; \textbf{hide}\ \mathit{isaRegFile}\ \textbf{in}\ \mathit{ISAStall} \| \mathit{ISARegFile} \| \mathit{ISAOut}$$

■

module *Opr1*

interface *opr1*

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

atom *Opr1* **controls** *opr1*

update

$[] \neg stall' \rightarrow opr1' :=$

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

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

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

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

$[]\neg stall' \rightarrow opr2 :=$

if $((src2' = pipe1.dest) \wedge \neg(pipe1.op = NOP) \wedge \neg(pipe1.op = STORE))$ **then**

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

else if $((src2' = pipe2.dest) \wedge \neg(pipe2.op = NOP) \wedge \neg(pipe2.op = STORE))$ **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

$[]true \rightarrow pipe1.op' := NOP$

update

$[]true \rightarrow pipe1.op$' := **if** $stall'$ **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
      $[] pipe1.op = AND \rightarrow aluOut' := opr1 \wedge opr2$
      $[] pipe1.op = OR \rightarrow aluOut' := opr1 \vee opr2$
  atom *Pipe2* **controls** $pipe2.op, pipe2.dest$
    init
      $[] true \rightarrow pipe2.op' := NOP$
    update
      $[] true \rightarrow pipe2.op' := pipe1.op;\ pipe2.dest' := pipe1.dest$
  atom *WbReg* **controls** $wbReg$
    update
      $[] pipe1.op = AND \vee pipe1.op = OR \rightarrow wbReg' := aluOut'$
      $[] pipe1.op = LOAD \rightarrow wbReg' := pipe1.inp$

module *RegFile*
  interface $regFile$
  external $pipe2.op, pipe2.dest, wbReg, aluOut$
  atom *RegFile* **controls** $regFile$
    init
      $[] true \rightarrow$ **forall** i **do** $regFile'[i] := 0$
    update
      $[]\ pipe2.op = AND \vee pipe2.op = OR \vee pipe2.op = LOAD \rightarrow$
        forall i **do** $regFile'[i] :=$ **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
      $[]\neg stall' \wedge op' = STORE \rightarrow out' := regFile[dest']$

module *Stall*
  interface *stall*
  external *op*, *dest*, *pipe1.op*, *pipe1.dest*, *pipe2.op*, *pipe2.dest*
  atom *Stall* **controls** *stall*
    update
      $[]\ op' = STORE \wedge pipe1.op \neq NOP \wedge pipe1.op \neq STORE \wedge dest' = pipe1.dest \rightarrow$
        $stall' := true$
      $[]\ op' = STORE \wedge pipe2.op \neq NOP \wedge pipe2.op \neq STORE \wedge dest' = pipe2.dest \rightarrow$
        $stall' := true$
      $[]$**default** $\rightarrow stall' := false$

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 $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ are compatible and $\mathcal{C}(P \| Q) = \mathcal{C}(P) \| \mathcal{C}(Q)$.*

Example 3.4 Consider the three-stage pipeline defined by the reactive module *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 *PIPELINE* module is the parallel composition of seven modules. The first stage consists of the modules *Pipe1*, *Opr1*, and *Opr2*. Forwarding logic in *Opr1* and *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 *wbReg*. The third stage is consists of the module *RegFile*, which copies *wbReg* into the appropriate register. The *PipeOut* module outputs a register value in response to a $STORE$ instruction. The *Stall* module controls the *stall* signal, which is set to *true* whenever a $STORE$ instruction cannot be accepted due to data dependencies.

Our goal is to show that *PIPELINE* is a correct implementation of the instruction set architecture *ISA*. This is the case if every sequence of instructions given to *PIPELINE* produces a sequence of outputs (and stalls) that is permitted by *ISA*. The module *ISA* is refinable by *PIPELINE*, so it remains to be shown that every trace of *PIPELINE* is a trace of *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 $P\|W$, and if $P\|W \preceq Q^u$ 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 $P\|W$ and every external variable of Q^u is an observable variable of $P\|W$. Since Q^u does not have any private variables, we have that Q^u is projection refinable by $P\|W$. Consider the transition constraints $\mathcal{C}(P)$, $\mathcal{C}(Q)$ and $\mathcal{C}(W)$.

1. $\mathcal{C}(W)$ is compatible with $\mathcal{C}(P)$.

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

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

Suppose $P\|W \preceq Q^u$. Then $\mathcal{C}(P)\|\mathcal{C}(W) \preceq \mathcal{C}(Q)^u$. From Theorem 3.4, we get that $\mathcal{C}(P) \preceq \mathcal{C}(Q)$. 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

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 $\mathcal{Z} = \{R \in \mathcal{R} | \exists y \in \mathit{Intf}(Q).\ \exists z \in \mathit{Intf}(R).\ y \succ_S z \vee y \succ_Q z\}$. Let C be the composition of all the modules in the set $\mathcal{Z}$ and let D be the composition of all the modules in the set $\mathcal{R} \setminus \mathcal{Z}$. Then for all τ, we have that $\mathcal{C}(P)^\tau \| \mathcal{C}(C)^\tau \| \mathcal{C}(D)^{\tau-1} \preceq \mathcal{C}(Q)^\tau$.*

Proof: We have that $S = P \| C \| D$. Let Y be the least subset of $\mathit{Var}(S)$ such that (1) $\mathit{Intf}(Q) \cup \mathit{Extl}(S) \subseteq Y$, (2) Y is awaits-closed in Q, and (3) Y is awaits-closed in S. From the definition of C, we have that $Y \subseteq \mathit{Var}(P \| C) \cup \mathit{Extl}(S)$. We prove the theorem by induction on τ.

Base step. We have to show that $\mathcal{C}(P)^0 \| \mathcal{C}(C)^0 \| \mathcal{C}(D)^{-1} \preceq \mathcal{C}(Q)^0$. Since every trace is a trace of $\mathcal{C}(D)^{-1}$, we need to show $\mathcal{C}(P)^0 \| \mathcal{C}(C)^0 \preceq \mathcal{C}(Q)^0$. Consider a trace $\sigma = s_0$ of length one of $\mathcal{C}(P) \| \mathcal{C}(C)$. Since s_0 is an initial state of $\mathcal{C}(P) \| \mathcal{C}(C)$, we have that s_0 is an initial state of $P \| C$. Let s_0' be the restriction of s_0 to Y. Since $Y \subseteq \mathit{Var}(P \| C)$, the valuation s_0' is obtained by executing a subset of atoms of $P \| C$. From Fact 3.1, we can extend s_0' to t_0 such that t_0 is an initial state of $P \| C \| D$. Since $P \| C \| D \preceq Q$, we get that $[t_0]_Q$ is an initial state of Q. But $[t_0]_Q = [s_0]_Q$ because $\mathit{Var}(Q) \subseteq Y$. Therefore $[s_0]_Q$ is an initial state of Q.

Inductive step. Suppose $\tau > 0$ and $\mathcal{C}(P)^{\tau'} \| \mathcal{C}(C)^{\tau'} \| \mathcal{C}(D)^{\tau'-1} \preceq \mathcal{C}(Q)^{\tau'}$ for all $\tau' < \tau$. We will show that $\mathcal{C}(P)^\tau \| \mathcal{C}(C)^\tau \| \mathcal{C}(D)^{\tau-1} \preceq \mathcal{C}(Q)^\tau$. Consider a trajectory $\sigma = \sigma'.s$ of length $\tau + 1$ of $\mathcal{C}(P)^\tau \| \mathcal{C}(C)^\tau \| \mathcal{C}(D)^{\tau-1}$. Then σ is a trajectory of $\mathcal{C}(P)$ and $\mathcal{C}(C)$, and σ' is a trajectory of $\mathcal{C}(D)$. Therefore σ is a trajectory of P and C, and σ' is a trajectory of D. Let s' be the restriction of s to Y. Since $Y \subseteq \mathit{Var}(P \| C) \cup \mathit{Extl}(S)$, 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 $S = P \| C \| D$. Therefore, we get that $\sigma'.t$ is

a trajectory of $P\|C\|D$. Since $P\|C\|D \preceq Q$, we get that $[\sigma'.t]_Q$ is a trace of Q and hence of $\mathcal{C}(Q)$. But $[t]_Y = [s]_Y$. Since Y is an awaits-closed in Q and $\mathit{Intf}(Q) \subseteq Y$, we have that $[\sigma'.s]_Q$ is also a trace of $\mathcal{C}(Q)$. ∎

Theorem 3.6 (Assume-guarantee rule for modules) *Let $P = P_1\| \cdots \|P_n$ and $Q = Q_1\| \cdots \|Q_m$ be reactive modules such that Q is projection refinable by P. For all $1 \le i \le m$, let Γ_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 $1 \le i \le m$, 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 $A_1\|A_2\|\ldots\|A_l$ of such modules. For all $1 \le j \le l$, let Γ'_j be equal to Γ_i if A_j was built from an atom in Q_i. For all $1 \le j \le l$ and $1 \le i \le n$, if A_j was built from an atom in P_i then $\Gamma'_j = \Gamma_i \preceq A_j$. Let $\widetilde{\succ}$ denote the transitive closure of $\succ_P \cup \succ_Q$. Since $\succ_P \cup \succ_Q$ is acyclic, the relation $\widetilde{\succ}$ 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 $u \in \mathit{Intf}(A_k)$ and a $v \in \mathit{Intf}(A_j)$ such that $v \widetilde{\succ} u$. Since $\widetilde{\succ}$ is a partial order, we have that $\prec$ is a partial order. For each A_j let $Z(A_j) = \{A_i | A_i \prec A_j\}$ 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 Γ'_j by $\mathcal{B}_j$. We partition $\mathcal{B}_j$ into three parts in the following way. Let $\mathcal{C}_j$ be the components from the set $\{P_1, \ldots, P_n\}$, that is, the set $\mathcal{C}_j = B_j \cap \{P_1, \ldots, P_n\}$. Let C_j be the composition of the modules in $\mathcal{C}_j$. Let $\mathcal{D}_j = \{A_k \in \mathcal{B}_j | \exists y \in \mathit{Intf}(A_j).\ \exists x \in \mathit{Intf}(A_k).\ y \succ_{\Gamma'_j} x \vee y \succ_{A_j} x\}$. Let $\mathcal{E}_j = \mathcal{B}_j \setminus \mathcal{D}_j$. Let D_j and E_j be the composition of modules from $\mathcal{C}_j$ and $\mathcal{D}_j$ respectively. Thus we have that $\Gamma'_j = C_j\|D_j\|E_j \preceq A_j$. 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 $\mathcal{C}(C_j)^\tau \| \mathcal{C}(D_j)^\tau \| \mathcal{C}(E_j)^{\tau-1} \preceq \mathcal{C}(A_j)^\tau$ for all τ. Moreover, for all j we have that $\succ_{\Gamma'_j} \subseteq \widetilde{\succ}$ and $\succ_{A_j} \subseteq \widetilde{\succ}$. From the definition of $\prec$ we get that $\mathcal{D}_j \subseteq Z(A_j)$. Therefore, from Theorem 3.3 we get that $\mathcal{C}(P_1)\|\mathcal{C}(P_2)\|\ldots\|\mathcal{C}(P_n) \preceq \mathcal{C}(A_1)\|\mathcal{C}(A_2)\|\ldots\|\mathcal{C}(A_l)$, or $\mathcal{C}(P_1\|\ldots\|P_n) \preceq \mathcal{C}(A_1\|\ldots\|A_l)$. Therefore $P_1\|\ldots\|P_n \preceq A_1\|\ldots\|A_l = Q_1\|\ldots\|Q_m$. ∎

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 implemen-

module *SndSpec*
  external *ack* : *bool*
  interface *snd* : *bool*
  private *wait* : *bool*
  atom controls *wait* **reads** *ack* **,** *wait*
    init
    $[] true \rightarrow wait' := false$
    update
    $[] \neg wait \rightarrow wait' := true$
    $[] wait \wedge ack \rightarrow wait' := false$
    $[] \neg wait \rightarrow$
  endatom
  atom controls *snd* **reads** *wait* **awaits** *wait*
    init
    $[] true \rightarrow snd' := false$
    update
    $[] \neg wait \wedge wait' \rightarrow snd' := true$
    $[]$**default** $\rightarrow snd' := false$
  endatom
endmodule

Figure 3.7: Specification of Sender

```
module RcvSpec
  interface ack : bool
  external snd : bool
  private pending : bool
  atom controls pending reads pending, snd
    init
    []true → pending' := false
    update
    []¬pending ∧ snd → pending' := true
    []pending → pending' := false
    []pending →
  endatom
  atom controls ack reads pending awaits pending
    init
    []true → ack' := false
    update
    []pending ∧ ¬pending' → ack' := true
    []default → ack' := 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 → wait' := false; error' := false
    update
    [] ¬wait → wait' := true
    [] wait ∧ ack → wait' := false
    [] ¬wait →
    [] ¬wait ∧ ack → error' := true
    [] error → wait' := nondet
  endatom
  atom controls snd reads wait awaits wait
    init
    [] true → snd' := false
    update
    [] ¬wait ∧ wait' → snd' := true
    [] default → snd' := false
  endatom
endmodule
```

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
    $[]true \rightarrow pending' := false; error' := false$
    update
    $[]\neg pending \wedge snd \rightarrow pending' := true$
    $[]pending \ \ \rightarrow pending' := false$
    $[]pending \wedge snd \rightarrow error' := true$
    $[]pending \ \ \rightarrow$
  endatom
  atom controls ack **reads** $pending\,,\ error$ **awaits** $pending$
    init
    $[]true \rightarrow ack' := false$
    update
    $[]pending \wedge \neg pending' \rightarrow ack' := true$
    $[]error' \ \ \rightarrow ack' := true$
    $[]$**default** $\rightarrow ack' := false$
  endatom
endmodule

Figure 3.10: Implementation of Receiver

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

We make use of Theorem 3.6 as follows. First we decompose the specification Q into its components $Q_1 \| \cdots \| Q_n$. Then we find for each component Q_i of the specification a suitable module Γ_i (called an *obligation module*) and check that $\Gamma_i \preceq Q_i$. This is beneficial if the state space of Γ_i is smaller than the state space of P. The module Γ_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 Γ_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 Γ_i, if the refinement check $\Gamma_i \preceq Q_i$ goes through, then we are done. Typically, however, the external variables of Γ_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 Γ_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 $P \preceq Q \| A$ and*

Q is refinable 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 refinable by *PIPELINE*, but not projection refinable. 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*^{u}. This proof obligation can be expanded in terms of component modules to

$$\begin{array}{r} \mathit{ISARegFile} \| \mathit{RegFile} \| \mathit{Opr1} \| \mathit{Opr2} \| \\ \mathit{Pipe1} \| \mathit{Pipe2} \| \mathit{PipeOut} \| \mathit{Stall} \end{array} \preceq \mathit{ISARegFile} \| \mathit{ISAOut} \| \mathit{ISAStall}.$$

Let us start by identifying *ISAOut* with Q_1. We need to find an obligation module Γ_1, such that $\Gamma_1 \preceq$ *ISAOut*. There is only one interface variable for *ISAOut*, namely *out*. The component of *PIPELINE* that generates *out* is *PipeOut*. Thus *PipeOut* is the only essential module for Γ_1. However, the proof obligation

$$\Gamma_1 = \mathit{PipeOut} \preceq Q_1 = \mathit{ISAOut}$$

fails trivially, because *ISAOut* is not refinable by *PipeOut*. The module *ISAOut* has an external variable *isaRegFile* that is not present in *PipeOut*. To achieve refinability, we add *ISARegFile*, the module controlling *isaRegFile*, to Γ_1 and try to prove

$$\Gamma_1 = \mathit{ISARegFile} \| \mathit{PipeOut} \preceq Q_1 = \mathit{ISAOut}.$$

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

$$\Gamma_1 = \mathit{ISARegFile} \| \mathit{RegFile} \| \mathit{PipeOut} \preceq Q_1 = \mathit{ISAOut}$$

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

$$\Gamma_1 = \mathit{ISARegFile} \| \mathit{RegFile} \| \mathit{Pipe1} \| \mathit{Pipe2} \| \mathit{Opr1} \| \mathit{Opr2} \| \mathit{Stall} \| \mathit{PipeOut} \preceq Q_1 = \mathit{ISAOut}.$$

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

$$\frac{\begin{array}{rl} \mathit{ISARegFile} \| \mathit{RegFile} \| \mathit{Pipe1} \| \mathit{Pipe2} \| \\ \mathit{Opr1} \| \mathit{Opr2} \| \mathit{Stall} \| \mathit{PipeOut} & \preceq \mathit{ISAOut} \\ \mathit{ISARegFile} & \preceq \mathit{ISARegFile} \\ \mathit{Stall} & \preceq \mathit{ISAStall} \end{array}}{\begin{array}{rl} \mathit{ISARegFile} \| \mathit{RegFile} \| \mathit{Pipe1} \| \mathit{Pipe2} \| \\ \mathit{Opr1} \| \mathit{Opr2} \| \mathit{Stall} \| \mathit{PipeOut} & \preceq \mathit{ISAOut} \| \mathit{ISARegFile} \| \mathit{ISAStall} \end{array}}$$

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*.

```
module AbsOpr1
  external isaRegFile, src1, stall
  interface opr1
  atom AbsOpr1 controls opr1
     update
        []¬stall′ → opr1′ := isaRegFile[src1′]
```

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 $AbsRegFile \| ISARegFile_d$, where *AbsRegFile* and $ISARegFile_d$ are given below.

module $ISARegFile_d$
  atom $ISARegFile_d$ **controls** $isaRegFile_d$
    init
      $[]\, true \rightarrow$ **forall** i **do** $isaRegFile'_d[i] := 0$
    update
      $[]\, true \rightarrow$ **forall** i **do** $isaRegFile'_d[i] := isaRegFile[i]$

module *AbsRegFile*
  atom *AbsRegFile* **controls** *regFile*
    init
      $[]\, true \rightarrow$ **forall** i **do** $regFile'[i] := 0$
    update
      $[]\, true \rightarrow$ **forall** i **do** $regFile'[i] := isaRegFile_d[i]$

On composing *AbsRegFile* and $ISARegFile_d$ with *ISA*, we find that the new specification is not projection refinable by $ISARegFile \| PIPELINE$, because of the new specification variable $isaRegFile_d$. To regain projection refinability, a witness module needs to be written for the abstraction module $isaRegFile_d$, and composed with *PIPELINE*. A suitable witness is simply the module $ISARegFile_d$. After adding the abstraction modules, according to

Theorem 3.6, we obtain the following assume-guarantee proof:

$$
\begin{array}{rcl}
\begin{array}{r} \mathit{PipeOut} \| \mathit{Pipe1} \| \mathit{Pipe2} \| \mathit{Stall} \| \\ \mathit{AbsRegFile} \| \mathit{ISARegFile}_d \| \mathit{ISARegFile} \end{array} & \preceq & \mathit{ISAOut} \\
\begin{array}{r} \mathit{Opr1} \| \mathit{AbsOpr2} \| \mathit{Pipe1} \| \mathit{Pipe2} \| \\ \mathit{AbsRegFile} \| \mathit{ISARegFile}_d \| \mathit{ISARegFile} \end{array} & \preceq & \mathit{AbsOpr1} \\
\begin{array}{r} \mathit{Opr2} \| \mathit{AbsOpr1} \| \mathit{Pipe1} \| \mathit{Pipe2} \| \\ \mathit{AbsRegFile} \| \mathit{ISARegFile}_d \| \mathit{ISARegFile} \end{array} & \preceq & \mathit{AbsOpr2} \\
\begin{array}{r} \mathit{AbsOpr1} \| \mathit{AbsOpr2} \| \mathit{Pipe1} \| \mathit{Pipe2} \| \\ \mathit{RegFile} \| \mathit{ISARegFile}_d \| \mathit{ISARegFile} \| \mathit{Stall} \end{array} & \preceq & \mathit{AbsRegFile} \| \mathit{ISARegFile}_d \\
\mathit{Stall} & \preceq & \mathit{ISAStall} \\
\mathit{ISARegFile} & \preceq & \mathit{ISARegFile} \\
\hline
\begin{array}{r} \mathit{ISARegFile} \| \mathit{ISARegFile}_d \| \mathit{RegFile} \| \\ \mathit{Pipe1} \| \mathit{Pipe2} \| \mathit{Opr1} \| \mathit{Opr2} \| \mathit{PipeOut} \| \mathit{Stall} \end{array} & \preceq & \begin{array}{l} \mathit{ISARegFile} \| \mathit{ISAOut} \| \mathit{ISAStall} \\ \mathit{AbsOpr1} \| \mathit{AbsOpr2} \| \mathit{AbsRegFile} \\ \mathit{ISARegFile}_d \end{array}
\end{array}
$$

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 $\mathit{PIPELINE} \parallel \mathit{ISARegFile} \parallel \mathit{ISARegFile}_d$, and the composite module on the right side is $\mathit{ISA}^u \| \mathit{ISARegFile}_d \| \mathit{AbsOpr1} \| \mathit{AbsOpr2} \| \mathit{AbsRegFile}$. By Theorem 3.7, we can remove $\mathit{ISARegFile}_d \| \mathit{AbsOpr1} \| \mathit{AbsOpr2} \| \mathit{AbsRegFile}$ from the right side to obtain the refinement $\mathit{PIPELINE} \| \mathit{ISARegFile} \| \mathit{ISARegFile}_d \preceq \mathit{ISA}^u$. The module $\mathit{ISARegFile} \parallel \mathit{ISARegFile}_d$ is a witness for the refinement $\mathit{PIPELINE} \preceq \mathit{ISA}$. Hence, by Theorem 3.4, we conclude that $\mathit{PIPELINE} \preceq \mathit{ISA}$.

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 is 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 — $st.valid$, $st.Op$, $st.aVal.valid$, $st.aVal.value$, $st.aVal.tag$, $st.bVal.valid$, $st.bVal.value$ and $st.bVal.tag$. 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 *allocSt* schedules an invalid reservation station (one whose *valid* bit is *false*) for an incoming instruction. The second *opSt* schedules an enabled reservation station to access the bus. The variable *stall* is set to *true* whenever the *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 *stall* and *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
      $[] \neg stall' \wedge (op' = STORE) \wedge (dest' = i) \rightarrow$
        $impRegs.valid'[i] := true; impRegs.value'[i] := inp'$
      $[] \neg stall' \wedge ((op' = AND) \vee (op' = OR)) \wedge (dest' = i) \rightarrow$
        $impRegs.valid'[i] := false; impRegs.tag'[i] := allocSt'$
      $[] \neg impRegs.valid[i] \wedge bus.valid \wedge (impRegs.tag[i] = bus.tag) \rightarrow$
        $impRegs.valid'[i] := true; impRegs.value'[i] := bus.value$

Each reservation station consists of four modules – one that controls the valid bit ($valid$), one that controls opcode (op), and one each for the two operands ($aVal$ and $bVal$). 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

$[] true \rightarrow st.valid'[s] := false$

update

$[] \neg stall' \wedge (allocSt' = s) \wedge ((op' = AND) \vee (op' = OR)) \rightarrow st.valid'[s] := true$

$[] bus.valid' \wedge st.valid[s] \wedge (bus.tag' = s) \rightarrow st.valid'[s] := false$

module $STATION_OP[s]$

interface $st.Op[s]$

external $stall, op, allocSt$

atom $STATION_OP[s]$ **controls** $st.Op[s]$

update

$[] \neg stall' \wedge (allocSt' = s) \wedge ((op' = AND) \vee (op' = OR)) \rightarrow st.Op'[s] := op'$

module $STATION_AVAL[s]$

interface $st.aVal.valid[s], st.aVal.value[s], st.aVal.tag[s]$

external $impRegs.value, impRegs.tag, impRegs.valid, bus.valid, bus.tag, bus.value,$
$st.valid[s], stall, src1\, op, allocSt$

atom $STATION_AVAL[s]$ **controls** $st.aVal.valid[s], st.aVal.value[s], st.aVal.tag[s]$

init

$[] true \rightarrow st.aVal.valid'[s] := false$

update

$[] \neg stall' \wedge (allocSt' = s) \wedge ((op' = AND) \vee (op' = OR)) \rightarrow$
$st.aVal.valid'[s] := impRegs.valid[src1']$; $st.aVal.value'[s] := impRegs.value[src1']$;
$st.aVal.tag'[s] := impRegs.tag[src1']$

$[] bus.valid \wedge st.valid[s] \wedge \neg st.aVal.valid[s] \wedge (bus.tag = st.aVal.tag[s]) \rightarrow$
$st.aVal.valid'[s] := true$; $st.aVal.value'[s] := bus.value$

module $STATION_BVAL[s]$

interface $st.bVal.valid[s], st.bVal.value[s], st.bVal.tag[s]$

external $impRegs.value, impRegs.tag, impRegs.valid, bus.valid, bus.tag, bus.value,$
$st.valid[s], stall, src2, op, allocSt$

"Same as $STATION_AVAL[s]$ with *src1* replaced by *src2*"

$STATION[s] := STATION_VALID[s] \| STATION_OP[s] \| STATION_AVAL[s] \| STATION_BVAL[s]$

The bus module computes the result of the operation that is scheduled by *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.aVal.valid*, *st.aVal.value*,
      st.bVal.valid, *st.Op*, *st.bVal.value*, *st.Op*, *opSt*
  atom *BUS* **controls** *bus.valid*, *bus.value*, *bus.tag*
    init
      $[]true \rightarrow bus.valid' := false$
    update
      $[]st.valid[opSt'] \wedge st.aVal.valid[opSt'] \wedge st.bVal.valid[opSt'] \wedge st.Op[opSt'] = AND \rightarrow$
        $bus.valid' := true; bus.value' := st.aVal.value[opSt'] \wedge st.bVal.value[opSt'];$
        $bus.tag' := opSt'$
      $[]st.valid[opSt'] \wedge st.aVal.valid[opSt'] \wedge st.bVal.valid[opSt'] \wedge st.Op[opSt'] = OR \rightarrow$
        $bus.valid' := true; bus.value' := st.aVal.value[opSt'] \vee st.bVal.value[opSt'];$
        $bus.tag' := opSt'$
      []**default** $\rightarrow bus.valid' := false$

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
      $[]true \rightarrow allocSt' := nondet$

module *OP_ST*
  interface *opSt*
  atom *OP_ST* **controls** *opSt*
    init update
      $[]true \rightarrow opSt' := nondet$

The *OUT* module processes the *LOAD* instruction.

module OUT
  interface out
  external $impRegs.value$, op, $dest$
    atom OUT **controls** out
    init
      $[] true \rightarrow out' := 0$
    update
      $[] (op' = LOAD) \rightarrow out' := impRegs.value[dest']$

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
      $[] true \rightarrow stall' := false$
    update
      $[] st.valid[allocSt'] \wedge ((op' = AND) \vee (op' = OR)) \rightarrow stall' := true$
      $[] (op' = LOAD) \wedge (\neg impRegs.valid[dest']) \rightarrow stall' := true$
      $[] ((op' = AND) | (op' = OR)) \wedge$
      $((\neg impRegs.valid[src1'] \wedge bus.valid \wedge (impRegs.tag[src1'] = bus.tag)) \vee$
      $(\neg impRegs.valid[src2'] \wedge bus.valid \wedge (impRegs.tag[src2'] = bus.tag)) \vee$
      $(\neg impRegs.valid[dest'] \wedge bus.valid \wedge (impRegs.tag[dest'] = bus.tag))) \rightarrow$
        $stall' := true$
      $[] ((op' = STORE) \wedge \neg impRegs.valid[dest'] \wedge bus.valid \wedge (impRegs.tag[dest'] = bus.tag)) \rightarrow$
        $stall' := true$
      $[]$ **default** $\rightarrow stall' := false$

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

$$\begin{aligned} IMPREGS &:= IMPREG[0] \| IMPREG[1] \| IMPREG[2] \| IMPREG[3] \\ STATIONS &:= STATION[0] \| STATION[1] \| STATION[2] \| STATION[3] \\ TOMASULO &:= IMPREGS \| STATIONS \| BUS \| STALL \| ALLOC_ST \| OP_ST \| OUT \end{aligned}$$

Our aim is to show that $TOMASULO \preceq ISA$. To ensure projection comparability, we start by writing a witness module for the *ISA* registers, and composing the witness

with the *TOMASULO* module. Just as in the proof of the 3-stage pipeline, we will use *ISARegFile* to witness the register file of the specification. From Theorem 3.4, it suffices to show that $TOMASULO \| ISARegFile \preceq ISA^u$. 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*.

module $ABS_REG[i]$

  external $bus.value$, $bus.tag$, $bus.valid$, op, inp, $dest$, $allocSt$, $stall$

  interface $impRegs.valid$, $impRegs.value$, $impRegs.tag$

  atom $ABS_REG_VALID_TAG[i]$ **controls** $impRegs.valid[i], impRegs.tag[i]$

    init

      $[]true \rightarrow impRegs.valid'[i] := true$

    update

      $[]\neg stall' \wedge (op' = STORE) \wedge (dest' = i) \rightarrow$

        $impRegs.valid'[i] := true$

      $[]\neg stall' \wedge ((op' = AND) \vee (op' = OR)) \wedge (dest' = i) \rightarrow$

        $impRegs.valid'[i] := false; impRegs.tag'[i] := allocSt'$

      $[]\neg impRegs.valid[i] \wedge bus.valid \wedge (impRegs.tag[i] = bus.tag) \rightarrow$

        $impRegs.valid'[i] := true$

  atom $ABS_REG_VAL[i]$ **controls** $impRegs.value[i]$

    init

      $[]true \rightarrow impRegs.value'[i] := 0$

    update

      $[]\neg impRegs.valid'[i] \rightarrow impRegs.value'[i] := nondet$

      $[]impRegs.valid'[i] \rightarrow impRegs.value'[i] := isaRegFile'[i]$

$ABS_REGS := ABS_REG[0] \| ABS_REG[1] \| ABS_REG[2] \| ABS_REG[3]$

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 *ISA*, since instructions are executed atomically in the *ISA*.

module $HIST_STN_AVAL[s]$

  interface $histSt.aVal[s]$

external *isaRegFile*, *op*, *src1*, *allocSt*, *stall*
atom $HIST_STN_AVAL[i]$ **controls** $histSt.aVal[s]$
init
$[] true \rightarrow histSt.aVal'[s] := 0$
update
$[] \neg stall' \wedge (allocSt' = s) \wedge ((op' = AND) \vee (op' = OR)) \rightarrow$
$histSt.aVal'[s] := isaRegFile[src1']$

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 $HIST_STN_RESULT[i]$ **controls** $histSt.result[s]$
init
$[] true \rightarrow histSt.result'[s] := 0$
update
$[] \neg stall' \wedge (allocSt' = s) \wedge ((op' = AND) \vee (op' = OR)) \rightarrow$
$histSt.result'[s] := isaRegFile[dest']$

$HIST_STN[s] := HIST_STN_AVAL[s] \| HIST_STN_BVAL[s] \| HIST_STN_RESULT[s]$
$HIST_STNS := HIST_STN[0] \| HIST_STN[1] \| HIST_STN[2] \| HIST_STN[3]$

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
$[] st.valid'[s] \wedge st.aVal.valid'[s] \rightarrow st.aVal.value'[s] := histSt.aVal'[s]$
$[]$**default** $\rightarrow st.aVal.value'[s] := nondet$

module $ABS_STN_BVAL[s]$

interface $st.bVal.value[s]$
external $st.bVal.valid[s], histSt.bVal[s]$
"same as $ABS_STN_AVAL[s]$ with $histSt.aVal$ replaced by $histSt.bVal$"

$ABS_STN[s] := ABS_STN_AVAL[s] \| ABS_STN_BVAL[s] \| STATION_OP \| STATION_VALID[s]$
$ABS_STNS := ABS_STN[0] \| ABS_STN[1] \| ABS_STN[2] \| ABS_STN[3]$

module ABS_BUS_VAL
interface $bus.value$
external $opSt, bus.valid, histSt.result$
atom ABS_BUS_VAL **controls** $bus.value$
update
$[] bus.valid' \rightarrow bus.value' := histSt.result[opSt']$
$[] \neg bus.valid' \rightarrow bus.value' := nondet$

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.

$$\frac{\begin{array}{rl} ISARegFile \| ABS_REGS \| STALL & \preceq ISAOut \\ HIST_STNS \| ISARegFile \| \; ABS_STNS & \preceq ABS_BUS_VAL \\ HIST_STNS \| ISARegFile \| \; ABS_BUS_VAL \| IMPREG[0] \| STALL & \preceq ABS_REG[0] \\ HIST_STNS \| ISARegFile \| STALL \| \; ABS_BUS_VAL \| ABS_REGS \| STATION[0] & \preceq ABS_STN_AVAL[0] \\ HIST_STNS \| ISARegFile \| STALL \| \; ABS_BUS_VAL \| ABS_REGS \| STATION[1] & \preceq ABS_STN_BVAL[0] \\ STALL & \preceq ISAStall \end{array}}{\begin{array}{rl} IMPREGS \| STATIONS \| BUS \| \; STALL \| ALLOC_ST \| OP_ST \| \; ISARegFile \| HIST_STNS & \preceq ISARegFile \| ISAOut \| ABS_REGS \| \; ISAStall \| HIST_STNS \| \; ABS_STNS \| ABS_BUS_VAL \end{array}}$$

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

Refinement Check	Number of latches	
	4 regs + 4 stns	8 regs + 8 stns
Monolithic Check	67*	155*
Data Out	12	24
Bus valid bit	0	0
Bus value	32	64
Bus tag	0	0
Register[0] valid bit	4	5
Register[0] tag	4	5
Register[0] value	20	34
Reservation Station[0] valid bit	4	5
Reservation Station[0] aVal valid bit	22	37
Reservation Station[0] aVal tag	10	12
Reservation Station[0] aVal value	35	70*

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 Tomasulo's algorithm refines a nondeterministic generalization of the ISA that executes instructions preserving only the dataflow dependencies among them. Our proof of Tomasulo'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

[1] The *Sample* operator is similar, but not identical, to the operator of Reactive Modules [AH96]: while 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.

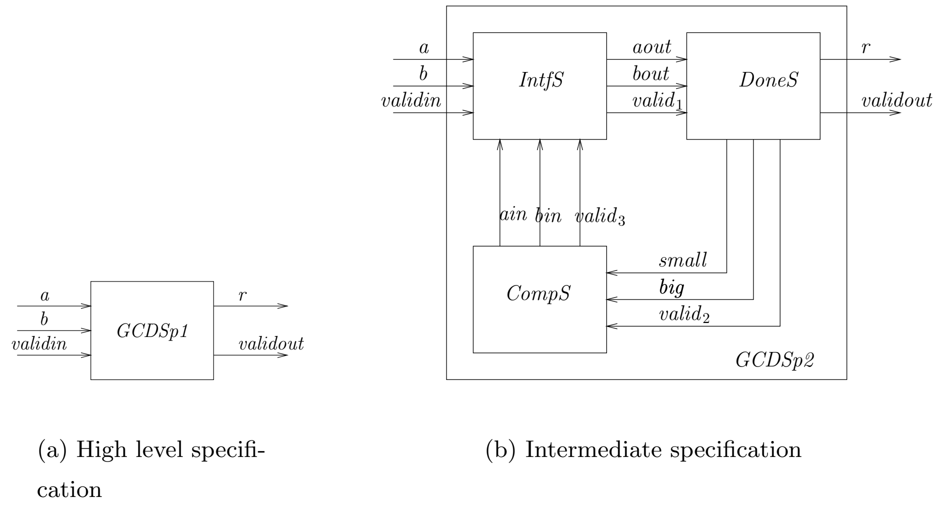
In Chapter 3, we showed how to make refinement checking scalable by making use of the compositional structure of both implementation and specification, and dividing the verification task at hand into simpler subtasks. A refinement-checking problem of the form $P_1 \| P_2 \preceq Q_1 \| Q_2$ is decomposed into the two proof obligations $P_1 \| Q_2 \preceq Q_1$ and $Q_1 \| P_2 \preceq Q_2$ (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 [JBJ95, 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.

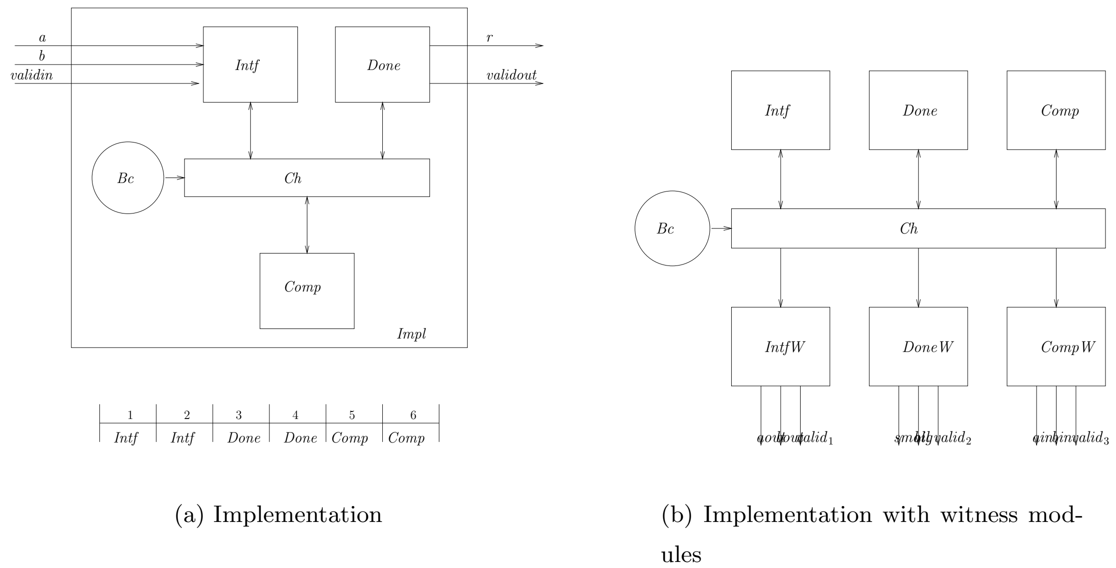
Motivating example. We consider a design that computes the Greatest Common Divisor (*GCD*) of two numbers. We will start with a synchronous specification shown in Figure 4.1(a). Given two inputs a and b, the module *GCDSp1* computes the *GCD* of a and b and places the result in the output r. The boolean 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

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** $((a = 1)$ or $(b = 1))$ **return** (1);
(3) $small :=$ **min**$(a,\ b)$;
(4) $big :=$ **max**$(a,\ b)$;
(5) **return** $(GCD(small, big - small))$;

The resulting refinement *GCDSp2* shown in Figure 4.1(b) has three modules: *IntfS*, *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 *IntfS* module takes data inputs from both *CompS* and the environment and feeds the data to the *DoneS* module. The modules *IntfS*, *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 *IntfS* 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 φ be a predicate on the primed and unprimed observable variables of A. We define $B = (\textit{Sample } A \textit{ at } \varphi)$ 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 φ, and none of the intermediate rounds satisfy φ. Given a trajectory of A, we use the term *sampling instants* to refer to the instants in the trajectory where φ is true.

We note that the *Sample* operator differs from the operator of [AH96] in two ways: (1) *Sample* operates on transition constraints, whereas operates on modules, and (2) *Sample* does not constrain the environment between sampling instants, whereas constrains the

environment to not change between sampling instants. In the rest of this paper, whenever we write $B = (\mathit{Sample}\ A\ \mathit{at}\ \varphi)$, we assume that φ 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:

$$\begin{aligned} \mathit{GCDSp2} &= \mathit{IntfS} \| \mathit{DoneS} \| \mathit{CompS} \\ \mathit{Impl} &= \mathit{Bc} \| \mathit{Ch} \| \mathit{Intf} \| \mathit{Done} \| \mathit{Comp} \end{aligned}$$

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:

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

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:

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

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 φ is a predicate on the observable variables of B, then $(\mathit{Sample}\ A\ \mathit{at}\ \varphi) \preceq (\mathit{Sample}\ B\ \mathit{at}\ \varphi)$.*

Proof: Consider any trace γ of *Sample A at* φ, with $\sigma = s_0, s_1, \dots, s_n$ as the witnessing run. Then there must exist run $\hat{\sigma} = t_0, t_1, \dots, t_n$ of A such that σ 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 $\hat{\sigma}' = t'_0, t'_1, \dots, t'_n$ of B, such that $[\hat{\sigma}]_B = [\hat{\sigma}']_B$. Finally, corresponding to $\hat{\sigma}'$, there exists a run σ' of (*Sample B at* φ) 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* φ) $\preceq$ (*Sample B at* φ). 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 φ be a predicate on the observable variables common to A and B. Then (Sample $(A \| B)$ at φ) $\preceq$ (Sample A at φ) $\|$ (Sample B at φ).*

Proof: Let C = (*Sample* $(A\|B)$ *at* φ), D = (*Sample* (A) *at* φ), and E = (*Sample* (B) *at* φ). By definition of *Sample* and $\|$ operators, every initial state of C is an initial state of $D\|E$. Suppose $s \rightarrow_C t$. Then we know that there is a sequence of states $s_0, s_1, \dots, s_n$, such that (1) $s = s_0$, (2) $t = s_n$, (3) $s_i \rightarrow_{A\|B} s_{i+1}$ for $i = 0, 1, \dots, n-1$, and (4) $\varphi(s_i, s_{i+1})$ is *false* for $i = 0, 1, \dots, n-2$ and *true* for $i = n-1$. Since $s_i \rightarrow_{A\|B} s_{i+1}$ 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 $s \rightarrow_{D\|E} t$. ∎

In practice, the traces of (*Sample A at* φ) $\|$ (*Sample B at* φ) form a large superset of the traces of (*Sample* $(A\|B)$ *at* φ). 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 $A\|B \preceq T_A\|T_B$ and φ is a predicate on the observable variables common to A and B, then (Sample $(A\|B)$ at φ) $\preceq$ (Sample $(A\|T_A)$ at φ) $\|$ (Sample $(B\|T_B)$ at φ).*

Proof: The proof is similar in spirit to the proof of Theorem 4.2. Let $C = (\mathit{Sample}\ (A \| B)\ \mathit{at}\ \varphi)$, $D = (\mathit{Sample}\ (A \| T_A)\ \mathit{at}\ \varphi)$, and $E = (\mathit{Sample}\ (B \| T_B)\ \mathit{at}\ \varphi)$. By definition of *Sample* and $\|$ operators, every initial state of C is an initial state of $D \| E$. 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) $s_i \rightarrow_{A\|B} s_{i+1}$ 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 $A \| B \preceq T_A \| T_B$, we have $[s_i]_{T_A\|T_B} \rightarrow_{T_A\|T_B} [s_{i+1}]_{T_A\|T_B}$ for $i = 0, 1, \ldots, n-1$. Since $s_i \rightarrow_{A\|B} s_{i+1}$ and $[s_i]_{T_A\|T_B} \rightarrow_{T_A\|T_B} [s_{i+1}]_{T_A\|T_B}$ imply $s_i \rightarrow_{A\|T_A} s_{i+1}$ and $s_i \rightarrow_{B\|T_B} s_{i+1}$, we have $s \rightarrow_D t$ and $s \rightarrow_E t$ as well. Thus $s \rightarrow_{D\|E} t$. ∎

4.2 Refinement checking with sample

We generalize the methodology for assume-guarantee style refinement checking given in Chapter 3 to accommodate the *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 $\mathit{Obs}(W) \cap \mathit{Obs}(A)$. Then B^u is projection refinable by the composition $A \| W$ and if $A \| W \preceq B^u$ 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 *Sample* operator.

Theorem 4.4 (Sampled witnesses) *Consider two transition constraints A and B, and a predicate φ 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) $\mathit{Priv}(B) \subseteq \mathit{Obs}(W)$, and (3) W is non-blocking on $\mathit{Obs}(W) \cap \mathit{Obs}(A)$. Then (1) B^u is projection refinable by $(\mathit{Sample}\ (A \| W)\ \mathit{at}\ \varphi)$, and (2) $(\mathit{Sample}\ (A \| W)\ \mathit{at}\ \varphi) \preceq B^u$ implies $(\mathit{Sample}\ A\ \mathit{at}\ \varphi) \preceq B$.*

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 φ to *true* in Theorem 4.4. The witness W could be a module also from the following

corollary of Theorem 4.4.

Corollary 4.5 *Consider two transition constraints A and B, and a predicate φ 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 $(Sample\ (A \| \mathcal{C}(W))\ at\ \varphi)$, and (2) $(Sample\ (A \| \mathcal{C}(W))\ at\ \varphi) \preceq B^u$ implies $(Sample\ A\ at\ \varphi) \preceq B$.*

Assume-guarantee reasoning. Suppose the left side of a refinement relation is of the form $Sample(A \| B)\ at\ \varphi$. It is not directly possible to apply the assume-guarantee rule from Theorem 3.3 in such cases. However, we can distribute the *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 *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 *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 *Sample* operator.

Theorem 4.6 (Assume-guarantee with sample) *Let $A = A_1 \| A_2 \| \dots \| A_n$ and $B = B_1 \| B_2 \| \dots \| B_m$ be transition constraints. Let T_i be a transition constraint on the observable variables of A_i, for $1 \leq i \leq n$. Let $\prec$ be a well-founded order on the components of B, and let T, Z and Z^C be defined as follows: $T = \{T_1, \dots, T_n\}$, $Z(B_i) = \{B_j | B_j \prec B_i\}$, and $Z^C(B_i) = \{B_j | B_j \notin Z(B_i)\}$. For each B_i, 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 $C_i = \{A_{i_1}, A_{i_2}, \dots, A_{i_k}\}$ then $U_i = \{T_{i_1}, T_{i_2}, \dots, T_{i_k}\}$. Also for each B_i,*

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 $A \preceq T_1 \| \ldots \| T_n$, and ψ is any predicate such that $(Sample\ (C_i \| U_i)\ at\ \psi)^\tau \| D_i^\tau \| E_i^{\tau-1} \preceq B_i^\tau$ for all $1 \leq i \leq m$ and $\tau \in \mathbb{N}$, then $(Sample\ A\ at\ \psi) \preceq B$.

Proof: We know that $A = A_1 \| A_2 \| \ldots \| A_n \preceq C_1 \| C_2 \| \ldots \| C_m$. Therefore we have that $Sample\ A\ at\ \psi \preceq Sample\ (C_1 \| C_2 \| \ldots \| C_m)\ at\ \psi$. We apply Theorem 3.3 to get

$$Sample\ (C_1 \| U_1)\ at\ \psi \| \ldots \| Sample\ (C_m \| U_m)\ at\ \psi \preceq B_1 \| \ldots \| B_m = B.$$

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

$$Sample\ (C_1 \| \ldots \| C_m)\ at\ \psi \preceq Sample\ (C_1 \| U_1)\ at\ \psi \| \ldots \| Sample\ (C_m \| U_m)\ at\ \psi.$$

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

In the above theorem, note that the antecedent $A \preceq T_1 \| \ldots \| T_n$ 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:

$$\begin{aligned}(Sample\ GCDSp2\ at\ (\neg inprogress')) &\preceq GCDSp1\\ (Sample\ Impl\ at\ sync') &\preceq GCDSp2\end{aligned}$$

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 *bout* 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 *aout* and *bout*, respectively. If the values in time-slots 1 and 2 are valid, then $valid_1$ is set to *true*. Further, the values assigned to *aout*, *bout*, and $valid_1$ are retained till the end of the frame. Similar witness modules *DoneW* and *CompW* can be written. All these witnesses take inputs from the channel as shown in Figure 4.2(b). Let *ImplW* be the module given by

$$ImplW = Impl \| IntfW \| DoneW \| CompW.$$

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 $(Sample\ ImplW\ at\ sync') \preceq GCDSp2$. Recall that $GCDSp2 = IntfS \| DoneS \| CompS$. We apply Theorem 4.6 with the order $DoneS \prec CompS \prec IntfS$. Let us consider the component *CompS* of *GCDSp2*. The component of *Impl* that is intended to implement the functionality of *CompS* is *Comp*. We wish to check if:

$$(Sample\ Comp\ at\ sync')^{\tau} \| DoneS^{\tau} \preceq CompS^{\tau}.$$

This check fails because the outputs of module *CompS*, namely, *ain*, *bin* and $valid_3$ are not present in module *Comp*. Adding the witness module and appropriately constraining it, we obtain the obligation:

$$(Sample\ (Comp \| CompW \| Ch \| Bc)\ at\ sync')^{\tau} \preceq CompS^{\tau}.$$

This still fails, because we have not constrained the inputs of *Comp*. In this obligation, the specification *CompS* looks at the inputs *small*, *big*, and $valid_2$ in every round and produces corresponding outputs *ain*, *bin* and $valid_3$ in the next round. The implementation *Comp* anticipates two values at time-slots 3 and 4 (which correspond to *small* and *big*, respectively) of every frame. If these inputs are valid, then *Comp* generates values in time-slots 5 and 6 (which correspond to *ain* and *bin* respectively) of the next frame. The module *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 *Comp* have to be constrained both at the sampling instants and between sampling instants, in order to satisfy the above assumptions. The specification component *DoneS* supplies the

constraint at sampling instants, which ensures that assumptions 2 and 3 are satisfied. The timing assumption $DoneW$ constrains the inputs between sampling instants and ensures that assumption 1 is satisfied. Thus we get the proof obligation

$$(Sample\ (Comp \| CompW \| Ch \| Bc \| DoneW)\ at\ sync')^{\tau} \| DoneS^{\tau} \preceq CompS^{\tau}.$$

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

$$\frac{\begin{array}{lcl} (Sample\ (Done \| DoneW \| Ch \| Bc \| IntfW)\ at\ sync')^{\tau} & \preceq & DoneS^{\tau} \\ (Sample\ (Comp \| CompW \| Ch \| Bc \| DoneW)\ at\ sync')^{\tau} \| DoneS^{\tau} & \preceq & CompS^{\tau} \\ (Sample\ (Intf \| IntfW \| Ch \| Bc \| CompW)\ at\ sync')^{\tau} \| CompS^{\tau} \| DoneS^{\tau} & \preceq & IntfS^{\tau} \end{array}}{(Sample\ (ImplW)\ at\ sync') \quad \preceq \quad GCDSp2}$$

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 $(Sample\ Impl\ at\ sync') \preceq GCDSp2$.

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 Eiríksson [Eír98]. 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` = `HIGH` and one with `clk` = `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

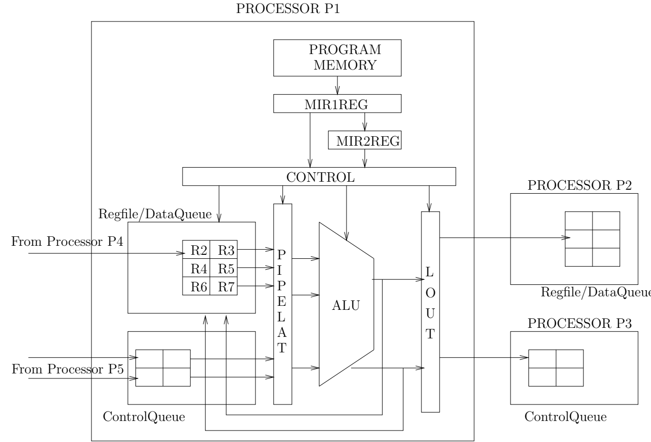


Figure 4.3: VGI processor configuration with three input and two output queues

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 $\mathcal{C}_{VGI}$ 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` [2]. 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

[2] The dotted rectangle in the lower portion of Figure 4.4 shows abstraction modules. We defer their description to Section 4.3.2.

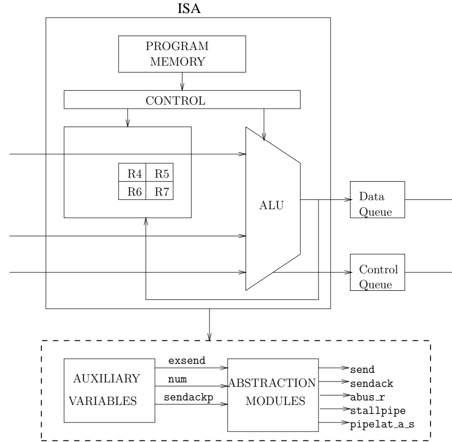


Figure 4.4: Specification module for refinement check

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 φ is $\texttt{clk} = \texttt{LOW}$. In the rest of this section, we use φ to refer to $\texttt{clk} = \texttt{LOW}$. 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 `clk` is `HIGH`. Let T_i be a module that says that all external signals of P_i change only when `clk` is `HIGH`.

The verification problem is to check

$$\mathit{Sample}\ (P_1 \| P_2 \| \ldots \| P_n)\ \mathit{at}\ \varphi \preceq Q_1 \| Q_2 \| \ldots \| Q_n$$

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

$$\frac{\begin{array}{rcl} \mathit{Sample}\ (P_i \| T_i)\ \mathit{at}\ \varphi & \preceq & Q_i \text{ forall } 1 \le i \le n \\ P_1 \| P_2 \| \ldots \| P_n & \preceq & T_1 \| T_2 \| \ldots \| T_n \end{array}}{\mathit{Sample}\ (P_1 \| P_2 \| \ldots \| P_n)\ \mathit{at}\ \varphi \ \preceq \ Q_1 \| Q_2 \| \ldots \| Q_n}$$

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 $1 \le i \le n$, 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 $X \in \mathcal{C}_{VGI}$, 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 $\mathit{Sample}\ (X \| T_X)\ \mathit{at}\ \varphi \preceq Y$ for each $X \in \mathcal{C}_{VGI}$, then we have proved that $\mathit{Sample}\ (P_i \| T_i)\ \mathit{at}\ \varphi \preceq Q_i$ for all $1 \le i \le n$. Thus, we have decomposed the proof of an arbitrary network of compute processors to $|\mathcal{C}_{VGI}|$ 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 *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

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 `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 `stallpipe` should be asserted in cycle $n+2$, it is not and as a result the instruction in `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 `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 `abus_r`, `send` and `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 `abus_r`, `send`, and `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, 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."

Our experience with verifying VGI has shown that the assume-guarantee rule needs to be augmented with rules for dealing with data structures such as bitvectors and arrays, which 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 needs to be analyzed rather than the whole design. We believe that a combination of such rules can make compositional model checking scale to "real" designs. We also believe that our methodology is general and 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].

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 systems 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 σ is a run of

the memory system, and λ_l is a permutation on the set of locations, then $\lambda_l(\sigma)$ is also a run of the memory system. Finally, symmetry in data values means that, if σ is a run of the memory system, and λ_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 Ω is *location symmetric* if for any run σ and any permutation λ_l on the set of memory locations $\lambda_l(\Omega(\sigma)) = \Omega(\lambda_l(\sigma))$. A serializer Ω is *data symmetric* if for any run σ and any function λ_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 σ is an interleaving of its outputs on σ 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 additional 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, preorders 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 sequentially consistent, and (2) there is an witness that maps every run σ of $I_2 \| P$ that can be produced in an environment of I_1 to a run σ' of I_2, such that the read/write events in σ' are an interleaving of the read/write events of I_2 and P in σ, and the traces obtained from σ and σ' are identical. Given a run γ of the memory system with $n > 1$ processors, we use the witness to create a run γ' of the memory system with $n-1$ processors, such that γ and γ' 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 γ' are an interleaving of the read/write events of the $(n-1)$-st and n-th processors in γ. 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 σ is not a run of the process, there is a function λ_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 $\langle Priv(A), Obs(A), S(A), S_I(A), T(A)\rangle$ with the following components:

- A set $Priv(A)$ of *private actions* and a set $Obs(A)$ of *observable actions*, such that $Priv(A) \cap Obs(A) = \emptyset$. The set $Act(A)$ is the union of $Priv(A)$ and $Obs(A)$. Private actions are outputs, whereas observable actions can be both inputs and outputs. The set of *extended actions* $\Pi(A)$ is given by $Priv(A) \times \{out\} \cup Obs(A) \times \{in, out\}$.
- 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 $First(\pi)$ and the second component by $Second(\pi)$. If $Second(\pi) = in$ then π is called an *input* action. If $Second(\pi) = out$ then π 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 ith element π_i is denoted by $\sigma(i)$. Let ϵ denote the empty sequence. A sequence σ 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 σ is a run of A *leading to* the state s_k. The projection operators *First* and *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 $Second(\pi_i) = out$ for all actions π_i in the run. A set of runs is closed if all runs in the set are closed. For any set $\beta \subseteq Act(A)$, the *restriction* of the run σ to β is the subsequence

obtained by considering the elements from $\beta \times \{in, out\}$ in σ, and is denoted by $[\sigma]_\beta$. For any run σ of I/O-process A, the restriction of σ to $Obs(A)$ is called a *trace* and is denoted by $tr_A(\sigma)$. 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) $Obs(A_1) \subseteq Obs(A_2)$, 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) $Priv(A_1) \cap Act(A_2) = \emptyset$ and (2) $Priv(A_2) \cap Act(A_1) = \emptyset$. We observe that if A_1 and A_2 are compatible then $Act(A_1) \cup Act(A_2)$ can be partitioned into three subsets —$Obs(A_1) \cap Obs(A_2)$, $Act(A_1) \setminus Act(A_2)$ and $Act(A_2) \setminus Act(A_1)$. 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

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

Suppose that A_1 and A_2 are compatible I/O-processes. Let σ_1 be a sequence of actions in $\Pi(A_1)$ and σ_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 π, we say that $\pi \in X$ if $First(\pi) \in X$. We now define inductively when a sequence σ is a join of sequences σ_1 and σ_2. We say that σ is a join of sequences σ_1 and σ_2 iff one of the following hold:

1. $\sigma = \epsilon$ and $\sigma_1 = \sigma_2 = \epsilon$.
2. $\sigma = \sigma'.\langle a, x\rangle$ and one of the following are true.

- $a \in Act(A_1) \setminus Act(A_2)$,
  $\sigma_1 = \sigma_1'.\langle a, x\rangle$, and σ' is a join of σ_1' and σ_2.
- $a \in Act(A_2) \setminus Act(A_1)$,
  $\sigma_2 = \sigma_2'.\langle a, x\rangle$, and σ' is a join of σ_1 and σ_2'.
- $a \in Obs(A_1) \cap Obs(A_2)$,
  $\sigma_1 = \sigma_1'.\pi_1$ and $\sigma_2 = \sigma_2'.\pi_2$ such that σ is a join of σ_1' and σ_2' and either (1) $x = in$ and $\pi_1 = \langle a, in\rangle$ and $\pi_2 = \langle a, in\rangle$, or (2) $x = out$ and $\pi_1 = \langle a, out\rangle$ and $\pi_2 = \langle a, in\rangle$, or (3) $x = out$ and $\pi_1 = \langle a, in\rangle$ and $\pi_2 = \langle a, out\rangle$.

A run σ_1 of A_1 *can be closed* by A_2 if there is a run σ_2 of A_2 such that a join of σ_1 and σ_2 is closed.

Theorem 5.1 *Let P and Q be two compatible I/O-processes. Then σ is a run of $P\|Q$ leading to $\langle p, q\rangle \in S(P) \times S(Q)$ iff there is a run σ_1 of P leading to p and a run σ_2 of Q leading to q such that σ is a join of σ_1 and σ_2.*

Proof: ($\Longrightarrow$) 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 σ' 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 σ_1' of P leading to p' and a run σ_2' of Q leading to q such that σ' is a join of σ_1' and σ_2'.

- $a \in Obs(P) \cap Obs(Q)$

Case 1. $x = in$ 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. $x = out$ and $\langle p', \pi, p\rangle \in T(P)$ and $\langle q', \langle a, in\rangle, q\rangle \in T(Q)$.
Let $\sigma_1 = \sigma_1'.\pi$ and $\sigma_2 = \sigma_2'.\langle a, in\rangle$.

Case 3. $x = out$ and $\langle q', \pi, q\rangle \in T(Q)$ and $\langle p', \langle a, in\rangle, p\rangle \in T(P)$.
Let $\sigma_1 = \sigma_1'.\langle a, in\rangle$ and $\sigma_2 = \sigma_2'.\pi$.

- $a \in Act(P) \setminus Act(Q)$
  We have that $\langle p, \pi, p' \rangle \in T(P)$ and $q = q'$. Let $\sigma_1 = \sigma_1'.\pi$ and $\sigma_2 = \sigma_2'$.

- $\pi \in Act(Q) \setminus Act(P)$
  We have that $p = p'$ and $\langle q, \pi, q' \rangle \in T(Q)$. Let $\sigma_1 = \sigma_1'$ and $\sigma_2 = \sigma_2'.\pi$.

In all cases, we get that σ_1 is a run of P and σ_2 a run of Q such that σ is a join of σ_1 and σ_2.

($\Longleftarrow$) We do induction on the prefix partial order $\prec$ over the set of joins of runs of P and Q. In the following, let σ be a join of a run σ_1 of P leading to state p and a run σ_2 of Q leading to state q.

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 ϵ is a run of $P \| Q$ leading to $\langle p, q \rangle$.

Inductive step: Let $\sigma = \sigma'.\pi$ and $\pi = \langle a, x \rangle$. We use the fact that σ is a join of σ_1 and σ_2 in the arguments below.

Case 1. $a \in Obs(P) \cap Obs(Q)$.
Then $\sigma_1 = \sigma_1'.\pi_1$ and $\sigma_2 = \sigma_2'.\pi_2$. There is a state p' of P such that σ_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 σ_2' is a run of Q leading to q' and $\langle q', \pi_2, q \rangle \in T(Q)$. Since σ' is a join of σ_1' and σ_2', we have from the induction hypothesis that σ 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 σ is a run of $P \| Q$ leading to $\langle p, q \rangle$.

Case 2. $a \in Act(P) \setminus Act(Q)$.
Then $\sigma_1 = \sigma_1'.\langle a, x \rangle$. There is a state p' of P such that σ_1' is a run of P leading to p' and $\langle p', \langle a, x \rangle, p \rangle \in T(P)$. Since σ' is a join of σ_1' and σ_2, we have from the induction hypothesis that σ 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 σ is a run of $P \| Q$ leading to $\langle p, q \rangle$.

Case 3. $a \in Act(Q) \setminus Act(P)$.
Similar to Case 2.

■

Corollary 5.2 *Let P and Q be two compatible I/O-processes. Then σ is a run of $P \| Q$ iff there is a run σ_1 of P and a run σ_2 of Q such that σ is a join of σ_1 and σ_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 σ_1 is a run of P and σ_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 σ of σ_1 and σ_2 there is a join τ of τ_1 and τ_2 such that $\tau = tr_{P\|Q}(\sigma)$.*
- *For all joins τ of τ_1 and τ_2 there is a join σ of σ_1 and σ_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 σ be a join of a run σ_1 of P and a run σ_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 τ_1 and τ_2.

  Inductive step: Let $\sigma = \sigma'.\pi$ and $\pi = \langle a, x\rangle$. Let $\tau' = tr_{P\|Q}(\sigma')$. We use the fact that σ is a join of σ_1 and σ_2 in the arguments below.

  Case 1. $a \in Obs(P) \cap Obs(Q)$.
  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 σ' is a join of σ_1' and σ_2', we have from the induction hypothesis that τ' is a join of τ_1' and τ_2'. Therefore τ is a join of τ_1 and τ_2.

  Case 2. $a \in Act(P) \setminus Act(Q)$.
  Then $\sigma_1 = \sigma_1'.\langle a, x\rangle$. Let $\tau_1' = tr_P(\sigma_1')$. Since σ' is a join of σ_1' and σ_2, we have from the induction hypothesis that τ' is a join of τ_1' and τ_2. If $a \notin Obs(P\|Q)$ we get that $\tau = \tau'$ and $\tau_1 = \tau_1'$. Therefore τ is a join of τ_1 and τ_2. If $a \in Obs(P\|Q)$ we get that $\tau = \tau'.\pi$ and $\tau_1 = \tau_1'.\pi$. Again we get that τ is a join of τ_1 and τ_2.

  Case 3. $a \in Act(Q) \setminus Act(P)$.
  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 τ 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 σ_1 and σ_2 such that $\tau = tr_\sigma(P\|Q)$.

Inductive step: Let $\tau = \tau'.\pi$ and $\pi = \langle a, x\rangle$. We use the fact that τ is a join of τ_1 and τ_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 σ_1' be the greatest prefix of σ_1 such that $\tau_1' = tr_{\sigma_1'}(P)$. Then $\sigma_1 = \sigma_1'.\pi_1.\gamma_1$ where γ_1 is a sequence of extended private actions of P. Let σ_2' be the greatest prefix of σ_2 such that $\tau_2' = tr_{\sigma_2'}(P)$. Then $\sigma_2 = \sigma_2'.\pi_2.\gamma_2$ where γ_2 is a sequence of extended private actions of Q. Since τ' is a join of τ_1' and τ_2', we have from the induction hypothesis that there is a join σ' of σ_1' and σ_2' such that $\tau' = tr_{\sigma'}(P\|Q)$. Let $\sigma = \sigma'.\pi.\gamma_1.\gamma_2$. Then σ is a join of σ_1 and σ_2 such that $\tau = tr_\sigma(P\|Q)$.

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

Then $\tau_1 = \tau_1'.\pi$. Let σ_1' be the greatest prefix of σ_1 such that $\tau_1' = tr_{\sigma_1'}(P)$. Then $\sigma_1 = \sigma_1'.\pi_1.\gamma_1$ where γ_1 is a sequence of extended private actions of P. Since τ' is a join of τ_1' and τ_2, we have from the induction hypothesis that there is a join σ' of σ_1' and σ_2 such that $\tau' = tr_{\sigma'}(P\|Q)$. Let $\sigma = \sigma'.\pi.\gamma_1$. Then σ is a join of σ_1 and σ_2 such that $\tau = tr_\sigma(P\|Q)$.

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 τ is a trace of $P\|Q$ iff there is a trace τ_1 of P and a trace τ_2 of Q such that τ is a join of τ_1 and τ_2.*

Proof: Suppose τ is a trace of $P\|Q$. Then there is a run σ of $P\|Q$ such that $\tau = tr_{P\|Q}(\sigma)$. From Theorem 5.1, we have that there are runs σ_1 of P and σ_2 of Q such that σ is a join of σ_1 and σ_2. Let $\tau_1 = tr_P(\sigma_1)$ and $\tau_2 = tr_P(\sigma_2)$. From Theorem 5.3, we get that $\tau = tr_{P\|Q}(\sigma)$ is a join of a trace τ_1 of P and a trace τ_2 of Q.

Suppose that τ is a join of a trace τ_1 of P and a trace τ_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 σ of σ_1 and σ_2 such that $\tau = tr_{P\|Q}(\sigma)$. Therefore τ is a trace of $P\|Q$. ■

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 $Obs(P) \subseteq Obs(Q)$ we have that $Obs(P\|R) = Obs(P) \cup Obs(R) \subseteq Obs(Q) \cup Obs(R) = Obs(Q\|R)$. Suppose τ is a trace of $P\|R$. From Corollary 5.4 there is a trace τ_1 of P and a trace τ_2 of Q such that τ is a join of τ_1 and τ_2. Since $P \preceq Q$, we have that τ_1 is a trace of Q as well. Therefore, from Corollary 5.4 we have that τ is a trace of $Q\|R$. ∎

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 $\langle PrivNames_A, ObsNames_A, A\rangle$ such that

1. $PrivNames_A$ and $ObsNames_A$ 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 $Priv(A(m, v)) = PrivNames_A \times \mathbb{N}_m \times (\mathbb{N}_v \cup \{\perp\})$ and $Obs(A(m, v)) = ObsNames_A \times \mathbb{N}_m \times (\mathbb{N}_v \cup \{\perp\})$.

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

1. $PrivNames_C \cap ((ObsNames_P \cup PrivNames_P) \times \mathbb{N}^+) = \emptyset$, and
   $ObsNames_C \cap (PrivNames_P \times \mathbb{N}^+) = \emptyset$.

2. $R \in PrivNames_P$ and $W \in PrivNames_P$.

The functions *name*, *loc*, and *val* are defined on $Act(C(m, v)) \cup Act(P(m, v))$, and extract respectively the first, second, and third components of the actions. Given some m and v, let $RdWr(m, v)$ 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) $name(a')$ is the pair $\langle name(a), k\rangle$, (2) $loc(a') = loc(a)$, and (3) $val(a') = val(a)$. A parameterized memory system defines a function that maps $\mathbb{N} \times \mathbb{N}^+ \times \mathbb{N}^+$ to I/O-processes as follows:

$$\begin{aligned} M(0, m, v) &= C(m, v) \\ M(n+1, m, v) &= M(n, m, v) \| P_n(m, v) \end{aligned}$$

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 $Obs(C(m, v)) \cup Obs(P(m, v))$. We use Obs_M to denote the set $\bigcup_{m,v} Obs(C(m, v)) \cup Obs(P(m, v))$. We define a function *proc* on the set of actions $\bigcup_{k,m,v} Priv(P_k(m, v))$ such that if $a \in Priv(P_k(m, v))$, then $proc(a) = k$.

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 σ be a run of the memory system $M(n, m, v)$. Let $R \subseteq \mathbb{N}_m$. We denote by $\sigma|_R$ the run σ restricted to the memory locations in R. Formally, we have $\sigma|_R = [\sigma]_\beta$, where $\beta = \{a \mid a \in Act(M(n, m, v)) \text{ and } loc(a) \in R\}$. 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 $\lambda : \mathbb{N}_k \cup \{\bot\} \longrightarrow \mathbb{N}_k \cup \{\bot\}$ such that λ 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 λ is a m-permutation then loc_λ is defined to be a function from $Act(M(n, m, v))$ to $Act(M(n, m, v))$ such that if $a = \langle name, j, k\rangle$ then $loc_\lambda(a) = \langle name, \lambda(j), k\rangle$. If λ is a $\bot$-extended v-map then val_λ is defined to be a function from $Act(M(n, m, v))$ to $Act(M(n, m, v))$ such that if $a = \langle name, j, k\rangle$ then $val_\lambda(a) = \langle name, j, \lambda(k)\rangle$. loc_λ and val_λ 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 λ be an m-permutation. Then*

1. *for all $\sigma \in \Sigma(C(m,v))$, we have that $loc_\lambda(\sigma) \in \Sigma(C(m,v))$, and*

2. *for all $\sigma \in \Sigma(P(m,v))$, we have that $loc_\lambda(\sigma) \in \Sigma(P(m,v))$.*

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

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

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

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

1. *for all $\sigma \in \Sigma(C(m,v))$, we have that $val_\lambda(\sigma) \in \Sigma(C(m,v))$, and*

2. *for all $\sigma \in \Sigma(P(m,v))$, we have that $val_\lambda(\sigma) \in \Sigma(P(m,v))$.*

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

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

2. *for all $\sigma \in Act(P(m,v_1))^*$, 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 $G \approx G'$ 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 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 $\mathcal{A}$ is the function such that $\mathcal{A}(F, R) = \bigcup_{i \in R} F(i)$. 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 $\mathcal{A}(\langle F, T \rangle, R) = \mathcal{A}(F, R)$. If $\langle F, T \rangle$ is an ordered r-partition of K then $\mathcal{G}(\langle F, T \rangle)$ 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 $\{\langle \langle R, i \rangle, j, k \rangle | i < n \; and \; j < m \; and \; k < v\}$ and $\{\langle \langle W, i \rangle, j, k \rangle | i < n \; and \; j < m \; and \; k < v\}$. 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) $F(i) = \{x < k | proc(\pi_x) = i\}$, 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 σ by first restricting it to actions in *Memop*. Formally, for any σ, we have that $h'(\sigma) = h'([\sigma]_{Memop})$. We extend h' to operate on sequences of extended actions by operating it on the first component of each extended action. Formally, if σ is a sequence of extended actions, then $h'(\sigma) = h'(First(\sigma))$. We observe that for all runs σ of $M(n, m, v)$, the directed graph $\mathcal{G}(h'(\sigma))$ is acyclic.

We are interested in defining which sequences from $Memop^*$ are serial. Intuitively, a sequence from $Memop^*$ 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_σ

as a function that associates with each position i in σ, the position j in σ where the most recent write to the location $loc(\pi_i)$ was done. Let $WS_\sigma(j)$ be the set of numbers i such that π_i is a write event to the location of the event π_j. Formally, we have that $WS_\sigma(j) = \{i \leq j | loc(\pi_i) = loc(\pi_j) \text{ and } \exists n_1.name(\pi_i) = \langle W, n_1 \rangle\}$. Then lw_σ is a mapping from the set $\{0, 1, ..., k-1\}$ to $\{0, 1, ..., k-1\} \cup \{\perp\}$ defined as follows.

$$lw_\sigma(j) = \begin{cases} \max(WS_\sigma(j)), & \text{if } WS_\sigma(j) \neq \emptyset \\ \perp, & \text{otherwise.} \end{cases}$$

The sequence σ is *serial* if the following conditions are satisfied.

1. For all $i < k$ and $j < k$, if $lw_\sigma(i) = lw_\sigma(j) = \perp$, then $val(\pi_i) = val(\pi_j)$.
2. For all $i < k$, if $lw_\sigma(i) \neq \perp$, then $val(\pi_i) = val(\pi_{lw_\sigma(i)})$.

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

Definition 5.1 (Serializer) *Let M be a parameterized memory system and let Ω be an observer for M. Then Ω is a* 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 σ.*

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

We define below three restrictions on an observer Ω.

- Ω is *local* on $M(n,m,v)$ if for all $\sigma \in \Sigma(M(n,m,v))$ and for all $j < m$, we have that $\Omega(\sigma|_j) = \Omega(\sigma)|_j$. We say that Ω is local if Ω is local on $M(n,m,v)$ for all n, m and v.
- Ω is *location symmetric* on $M(n,m,v)$ if for all $\sigma \in \Sigma(M(n,m,v))$ and for every m-permutation λ, we have that $\Omega(loc_\lambda(\sigma)) = loc_\lambda(\Omega(\sigma))$. We say that Ω is location symmetric if Ω is location symmetric on $M(n,m,v)$ for all n, m and v.

- Ω is *data symmetric* on $M(n,m,v)$ if for all $\sigma \in \Sigma(M(n,m,v))$ and for every $\perp$-extended v-map λ, we have that $\Omega(val_\lambda(\sigma)) = val_\lambda(\Omega(\sigma))$. We say that Ω is data symmetric if Ω 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 $R \subseteq \mathbb{N}_r$ 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 $\mathbb{N}_r \subseteq \mathbb{N}_r$ and $|\mathbb{N}_r| = r \leq s$, we get that the $\mathcal{A}(F,\mathbb{N}_r)$-restriction of G is acyclic. But $\mathcal{A}(F,\mathbb{N}_r) = K$ and the K-restriction of G is the same as G. Therefore G is acyclic.

Suppose $r > s$. Let $R = \{i_0, i_1, \ldots, i_{k-1}\} \subseteq \mathbb{N}_r$. 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) $\langle y_j, x_{j\oplus 1}\rangle \in T'(p)$ 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 $R \subseteq \mathbb{N}_r$. 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 $R' \subset R$. 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 $\langle x_a, y_b\rangle \in T'(p)$. Moreover $\langle y_{a\ominus 1}, x_a\rangle \in T'(p)$ and $\langle y_b, x_{b\oplus 1}\rangle \in T'(p)$. Since $T'(p)$ is transitive we have that $\langle y_{a\ominus 1}, x_{b\oplus 1}\rangle \in T'(p)$. Let $R' = R \setminus \{i_a, \ldots, i_b\}$. We can now short-circuit the R-cycle to get the R'-cycle $\langle x_0, y_0\rangle, \ldots, \langle x_{a\ominus 1}, y_{a\ominus 1}\rangle, \langle x_{b\oplus 1}, y_{b\oplus 1}\rangle, \ldots, \langle x_{k-1}, y_{k-1}\rangle$. A simple induction will show that G does not have an R-cycle for any $R \subseteq \mathbb{N}_r$. Therefore G is acyclic. ∎

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

Lemma 5.7 *Let Ω be a local observer for the parameterized memory system M. Let σ 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 $\Lambda^\Omega(\sigma)|_{\mathcal{A}(h^\Omega(\sigma),R)}$.*

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

$$[\sigma|_R]_{Memop(n,m,v)} = ([\sigma]_{Memop(n,m,v)})|_R = \pi_{i_0}, \pi_{i_1}, \ldots, \pi_{i_{b-1}}.$$

Then $\mathcal{G}(h'(\sigma|_R)) = \langle \mathbb{N}_b, F_1\rangle$ and $\mathcal{G}(h^\Omega(\sigma|_R)) = \langle \mathbb{N}_b, F_2\rangle$ 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

$$\begin{aligned}\mathcal{G}(h'(\sigma))|_{\mathcal{A}(h^\Omega(\sigma),R)} &= \langle \mathcal{A}(h^\Omega(\sigma), R), E_1 \cap \mathcal{A}(h^\Omega(\sigma), R) \times \mathcal{A}(h^\Omega(\sigma), R)\rangle, \text{ and}\\ \mathcal{G}(h^\Omega(\sigma))|_{\mathcal{A}(h^\Omega(\sigma),R)} &= \langle \mathcal{A}(h^\Omega(\sigma), R), E_2 \cap \mathcal{A}(h^\Omega(\sigma), R) \times \mathcal{A}(h^\Omega(\sigma), R)\rangle.\end{aligned}$$

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

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 Ω is local, we have that $\Omega(\sigma)|_p = \Omega(\sigma|_p) = \Omega((\sigma|_R)|_p) = \Omega(\sigma_R)|_p$. Suppose

$\Omega(\sigma|_R)|_p = \pi_{\alpha(g^{-1}(x_0))}, \pi_{\alpha(g^{-1}(x_1))}, \ldots, \pi_{\alpha(g^{-1}(x_{k-1}))}$, where $x_0 < x_1 < \ldots < x_{k-1} < b$. Suppose $\Omega(\sigma)|_p = \pi_{f^{-1}(y_0)}, \pi_{f^{-1}(y_1)}, \ldots, \pi_{f^{-1}(x_{k-1})}$, 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 $loc(\pi_{\alpha(x)}) = loc(\pi_{\alpha(y)}) = p$. 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 $x \in \mathbb{N}_a$, we have that $x \in \mathbb{N}_b$ iff $\alpha(x) \in \mathcal{A}(h^\Omega(\sigma), R)$. We have that $\langle x, y\rangle \in F_1$ iff $proc(\pi_{\alpha(x)}) = proc(\pi_{\alpha(y)})$ and $g(x) < g(y)$ iff $proc(\pi_{\alpha(x)}) = proc(\pi_{\alpha(y)})$ and $f(\alpha(x)) < f(\alpha(y))$ iff $\langle \alpha(x), \alpha(y)\rangle \in E_1 \cap \mathcal{A}(h^\Omega(\sigma), R) \times \mathcal{A}(h^\Omega(\sigma), R)$. Therefore α is an isomorphism from $\mathcal{G}(h'(\sigma|_R))$ to $\mathcal{G}(h'(\sigma))|_{\mathcal{A}(h^\Omega(\sigma),R)}$. We have that $\langle x, y\rangle \in F_2$ iff $loc(\pi_{\alpha(x)}) = loc(\pi_{\alpha(y)})$ and $g(x) < g(y)$ iff $loc(\pi_{\alpha(x)}) = loc(\pi_{\alpha(y)})$ and $f(\alpha(x)) < f(\alpha(y))$ iff $\langle \alpha(x), \alpha(y)\rangle \in E_2 \cap \mathcal{A}(h^\Omega(\sigma), R) \times \mathcal{A}(h^\Omega(\sigma), R)$. Therefore α is an isomorphism from $\mathcal{G}(h(\sigma|_R))$ to $\mathcal{G}(h(\sigma))|_{\mathcal{A}(h^\Omega(\sigma),R)}$. We get that

$$
\begin{aligned}
& \Lambda^\Omega(\sigma)|_{\mathcal{A}(h^\Omega(\sigma),R)} \\
= \; & (\mathcal{G}(h'(\sigma)) \cup \mathcal{G}(h^\Omega(\sigma)))|_{\mathcal{A}(h^\Omega(\sigma),R)} \\
= \; & \mathcal{G}(h'(\sigma))|_{\mathcal{A}(h^\Omega(\sigma),R)} \cup \mathcal{G}(h^\Omega(\sigma))|_{\mathcal{A}(h^\Omega(\sigma),R)} \\
\approx \; & \mathcal{G}(h'(\sigma|_R)) \cup \mathcal{G}(h(\Omega(\sigma|_R))) \\
= \; & \Lambda^\Omega(\sigma|_R).
\end{aligned}
$$

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

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

Proof: Let $[\sigma]_{Memop(n,m,v)} = \pi_0, \pi_1, \ldots, \pi_{a-1}$. Then $\mathcal{G}(h'(\sigma)) = \langle \mathbb{N}_a, E_1\rangle$ and $\mathcal{G}(h^\Omega(\sigma)) = \langle \mathbb{N}_a, E_2\rangle$ for some $E_1, E_2 \subseteq \mathbb{N}_a \times \mathbb{N}_a$. Let $\pi'_i = loc_\lambda(\pi_i)$. Therefore $[loc_\lambda(\sigma)]_{Memop(n,m,v)} = \pi'_0, \pi'_1, \ldots, \pi'_{a-1}$. Then $\mathcal{G}(h'(loc_\lambda(\sigma))) = \langle \mathbb{N}_a, F_1\rangle$ and $\mathcal{G}(h^\Omega(loc_\lambda(\sigma))) = \langle \mathbb{N}_a, F_2\rangle$ for some $F_1, F_2 \subseteq \mathbb{N}_a \times \mathbb{N}_a$. Then $x \in \mathcal{A}(h(\Omega(\sigma)), R)$ iff $loc(\pi_x) \in R$ iff $loc(loc_\lambda(\pi_x)) = loc(\pi'_x) \in \lambda(R)$ iff $x \in \mathcal{A}(h(\Omega(loc_\lambda(\sigma))), \lambda(R))$. Therefore $\mathcal{A}(h(\Omega(\sigma)), R) = \mathcal{A}(h(\Omega(loc_\lambda(\sigma))), \lambda(R))$. We

denote the set $\mathcal{A}(h(\Omega(\sigma)), R)$ by K. Therefore, we have that

$$\begin{aligned}
\mathcal{G}(h'(\sigma))|_{\mathcal{A}(h^{\Omega}(\sigma),R)} &= \langle K, E_1 \cap K \times K\rangle, \\
\mathcal{G}(h^{\Omega}(\sigma))|_{\mathcal{A}(h^{\Omega}(\sigma),R)} &= \langle K, E_2 \cap K \times K\rangle, \\
\mathcal{G}(h'(loc_\lambda(\sigma)))|_{\mathcal{A}(h^{\Omega}(loc_\lambda(\sigma)),\lambda(R))} &= \langle K, F_1 \cap K \times K\rangle, \text{ and} \\
\mathcal{G}(h^{\Omega}(loc_\lambda(\sigma)))|_{\mathcal{A}(h^{\Omega}(loc_\lambda(\sigma)),\lambda(R))} &= \langle K, F_2 \cap K \times K\rangle.
\end{aligned}$$

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

Let $x, y \in K$. We have that $\langle x, y\rangle \in F_1 \cap K \times K$ iff $proc(\pi_x) = proc(\pi_y)$ and $g(x) < g(y)$ iff $proc(\pi_x) = proc(\pi_y)$ 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 $\mathcal{G}(h'(loc_\lambda(\sigma)))|_{\mathcal{A}(h^{\Omega}(loc_\lambda(\sigma)),\lambda(R))}$ to $\mathcal{G}(h'(\sigma))|_{\mathcal{A}(h^{\Omega}(\sigma),R)}$. We also have that $\langle x, y\rangle \in F_2$ iff $loc(\pi_x) = loc(\pi_y)$ and $g(x) < g(y)$ iff $loc(\pi_x) = loc(\pi_y)$ 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 $\mathcal{G}(h^{\Omega}(loc_\lambda(\sigma)))|_{\mathcal{A}(h^{\Omega}(loc_\lambda(\sigma)),\lambda(R))}$ to $\mathcal{G}(h^{\Omega}(\sigma))|_{\mathcal{A}(h^{\Omega}(\sigma),R)}$. We get that

$$\begin{aligned}
& \Lambda^{\Omega}(loc_\lambda(\sigma))_{\mathcal{A}(h^{\Omega}(loc_\lambda(\sigma)),\lambda(R))} \\
= \; & (\mathcal{G}(h'(loc_\lambda(\sigma))) \cup \mathcal{G}(h^{\Omega}(loc_\lambda(\sigma))))_{\mathcal{A}(h^{\Omega}(loc_\lambda(\sigma)),\lambda(R))} \\
= \; & \mathcal{G}(h'(loc_\lambda(\sigma)))_{\mathcal{A}(h^{\Omega}(loc_\lambda(\sigma)),\lambda(R))} \cup \mathcal{G}(h^{\Omega}(loc_\lambda(\sigma)))_{\mathcal{A}(h^{\Omega}(loc_\lambda(\sigma)),\lambda(R))} \\
\approx \; & \mathcal{G}(h'(\sigma))|_{\mathcal{A}(h^{\Omega}(\sigma),R)} \cup \mathcal{G}(h^{\Omega}(\sigma))|_{\mathcal{A}(h^{\Omega}(\sigma),R)} \\
= \; & \Lambda^{\Omega}(\sigma)|_{\mathcal{A}(h^{\Omega}(\sigma),R)}.
\end{aligned}$$

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

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 $M(n, m, v)$.*

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

Fix $p < m$ and let λ 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 $loc_\lambda(\sigma)|_0$ is a

run of $M(n, 1, v)$ and therefore of $M(n, n, v)$. Since Ω is a serializer for $M(n, n, v)$, we have that $\Omega(loc_\lambda(\sigma)|_0)$ is serial.

$$\begin{aligned}
\Omega(loc_\lambda(\sigma)|_0) &= \Omega(loc_\lambda(\sigma))|_0 && \text{since } \Omega \text{ is local,} \\
&= loc_\lambda(\Omega(\sigma))|_0 && \text{since } \Omega \text{ is location symmetric,} \\
&= loc_\lambda(\Omega(\sigma)|_p) && \text{since } loc_\lambda(\gamma)|_0 = loc_\lambda(\gamma|_p) \text{ for all } \gamma.
\end{aligned}$$

Since $\Omega(loc_\lambda(\sigma)|_0)$ is serial, we have that $loc_{\lambda^{-1}}(\Omega(loc_\lambda(\sigma)|_0))$ is serial. Thus we get that $loc_{\lambda^{-1}}(\Omega(loc_\lambda(\sigma)|_0)) = loc_{\lambda^{-1}}(loc_\lambda(\Omega(\sigma)|_p)) = \Omega(\sigma)|_p$ is serial.

Suppose σ is a run of $M(n, n, v)$. Since Ω 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 λ 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 σ is a run of $M(n, m, v)$, we have that $loc_\lambda(\sigma)|_{\mathbb{N}_{|R|}}$ is a run of $M(n, n, v)$ using Assumptions 5.1, 5.2 and 5.3. Therefore $\Lambda^\Omega(loc_\lambda(\sigma)|_{\mathbb{N}_{|R|}})$ is acyclic.

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

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 σ. ∎

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

Proof: Consider any $v > 0$. Let $\sigma \in \Sigma(M(n, m, v))$. Since Ω is an observer for M, we have that $\Omega(\sigma)$ is a linearization of σ. 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 $lw_\sigma(i) = lw_\sigma(j) = \perp$ and $val(\pi_i) \neq val(\pi_j)$.

Case 2. For some $i \leq k$, there is a $j < i$ such that $lw_\sigma(i) = j$ and $val(\pi_i) \neq val(\pi_j)$.

In both cases, we can define a function $\perp$-extended v-map λ such that

$$\lambda(x) = \begin{cases} \perp, & \text{if } x = \perp \\ 1, & \text{if } x = val(\pi_i) \\ 0, & \text{otherwise} \end{cases}$$

Then $val_\lambda(\sigma)$ 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 Ω 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 $\langle PrivNames_{I_1}, ObsNames_{I_1}, I_1\rangle$ is a *possible process invariant* for the parameterized memory system $M = \langle C, P\rangle$ if

1. $ObsNames_{I_1} = ObsNames_C \cup ObsNames_P$, and

2. $PrivNames_{I_1} \cap PrivNames_P = \emptyset$.

Definition 5.3 (Process invariant) *Suppose that $\langle privNames_{I_1}, ObsNames_{I_1}, I_1\rangle$ is a possible process invariant for the parameterized memory system M. Suppose the following condition is true for all m and v:*

$[A_{I_1}(m,v)]$ *1.* $C(m,v) \preceq I_1(m,v)$

2. $I_1(m,v) \| P(m,v) \preceq I_1(m,v)$

Then $\langle privNames_{I_1}, ObsNames_{I_1}, I_1\rangle$ is a process invariant *of M.*

Proposition 5.12 *Suppose $\langle PrivNames_{I_1}, ObsNames_{I_1}, I_1\rangle$ 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 Ω for M such that for every sequence σ of memory operations of $M(n,m,v)$, the function Ω produces a rearranged sequence σ' of memory operations such that (1) σ' is serial, and (2) σ and σ' 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 $P(m, v)$.

A parameterized I/O-process $\langle PrivNames_{I_2}, ObsNames_{I_2}, I_2\rangle$ is a *possible merge invariant* for the parameterized memory system M if

1. $ObsNames_{I_2} = ObsNames_P$,
2. $\{R', W'\} \subseteq PrivNames_{I_2}$ and $\{R, W\} \cap PrivNames_{I_2} = \emptyset$, and
3. $PrivNames_{I_2} \cap PrivNames_C = \emptyset$ and $PrivNames_{I_2} \cap PrivNames_P = \emptyset$.

Let $Rd'(m, v) = \{\langle R', j, k\rangle | j < m \text{ and } k < v\}$, and let $Wr'(m, v) = \{\langle W', j, k\rangle | j < m \text{ and } k < v\}$. Let $RdWr'(m, v)$ denote the union of $Rd'(m, v)$ and $Wr'(m, v)$. We define the function *prime* on $RdWr(m, v)$ by $prime(\langle R, j, k\rangle) = \langle R', j, k\rangle$ and $prime(\langle W, j, k\rangle) = \langle W', j, k\rangle$. We define the function *unprime* on $RdWr'(m, v)$ by $unprime(\langle R', j, k\rangle) = \langle R, j, k\rangle$ and $unprime(\langle W', j, k\rangle) = \langle W, j, k\rangle$. We extend *prime* and *unprime* to sequences of actions in the natural way.

Consider the memory system $I_2(m, v)||P(m, v)$ for some value of m and v. Notice that $Obs(I_2(m, v)||P(m, v)) = Obs(I_2(m, v)) = Obs(P(m, v))$. Let $\Upsilon(m, v)$ denote $RdWr'(m, v) \cup Obs(I_2(m, v))$, 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 $(RdWr(m, v) \cup \Upsilon(m, v))^*$. 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) $F(0) = \{x < k | \pi_x \in RdWr(m, v)\}$, (2) $F(1) = \{x < k | \pi_x \in RdWr'(m, v)\}$, (3) $F(2) = \{x < k | \pi_x \in Obs(I_2(m, v))\}$, and (4) 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 σ by first restricting it to actions in $RdWr(m, v) \cup \Upsilon(m, v)$. Formally, for any σ, we have that $h'(\sigma) = h'([\sigma]_{RdWr(m,v)\cup\Upsilon(m,v)})$. We extend h' to operate on sequences of extended actions by operating it on the first component of each extended action. Formally, if σ is a sequence of extended actions, then $h'(\sigma) = h'(First(\sigma))$. We observe that for all runs σ of $P(m, v)\|I_2(m, v)$, the directed graph $\mathcal{G}(h'(\sigma))$ is acyclic.

Let $\pi_0, \pi_1, \ldots, \pi_{k-1}$ be a sequence in $RdWr(m, v) \cup \Upsilon(m, v)^*$ and f be a k-permutation. Then $prime(\pi_{f(0)}, \pi_{f(1)}, \ldots, \pi_{f(k-1)})$ is called a *permutation* of σ. If f is a good order of $\mathcal{G}(h'(\sigma))$ then the sequence $prime(\pi_{f^{-1}(0)}, \pi_{f^{-1}(1)}, \ldots, \pi_{f^{-1}(k-1)})$ is called a *linearization* of σ. Let σ be a run of $I_2(m, v)\|P(m, v)$. A sequence $\beta \in \Upsilon(m, v)^*$ is a permutation of σ if β is a permutation of $[\sigma]_{RdWr(m,v)\cup\Upsilon(m,v)}$. A sequence $\beta \in \Upsilon(m, v)^*$ is a linearization of σ if β is a linearization of $[\sigma]_{RdWr(m,v)\cup\Upsilon(m,v)}$. Let $\Sigma_{I_2\|P} =$

$\bigcup_{m,v} \Sigma(I_2(m,v)||P(m,v))$ and $\Upsilon = \bigcup_{m,v} \Upsilon(m,v)$. A function Θ from $\Sigma_{I_2\|P}$ to Υ 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 σ. For any run σ of $I_2(m,v)$, we define $tr'(\sigma)$ as the restriction of σ to $\Upsilon(m,v)$. Let $\Gamma'(I_2(m,v))$ be the set $\{tr'(\sigma)|\sigma \in \Sigma(I_2(m,v))\}$.

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

[$B0_{I_1,I_2}(m,v)$] For every closed trace $\langle a_1, out\rangle, \langle a_2, out\rangle, \ldots$ of $I_1(m,v)$, we have that the sequence $\langle a_1, in\rangle, \langle a_2, in\rangle, \ldots$ is a run of $I_2(m,v)$.

[$B1_{I_2}(m,v)$] For every closed run σ of $I_2(m,v)||C(m,v)$, the sequence $unprime([\sigma]_{RdWr'(m,v)})$ is serial.

[$B2_{I_2,I_1,\Theta}(m,v)$] For every run σ 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 σ.

We define the following restrictions on the merging function Θ.

- We say that Θ 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 Θ is local if Θ is local on $I_2(m,v)||P(m,v)$ for all m and v.

- We say that Θ is *location symmetric* on $I_2(m,v)||P(m,v)$ if for every m-permutation λ and for all $\sigma \in \Sigma(I_2(m,v)||P(m,v))$, we have that $\Theta(loc_\lambda(\sigma)) = loc_\lambda(\Theta(\sigma))$. We say that Θ is location symmetric if Θ is location symmetric on $I_2(m,v)||P(m,v)$ for all m and v.

- We say that Θ is *data symmetric* on $I_2(m,v)||P(m,v)$ if for every $\perp$-extended v-map λ and for all $\sigma \in \Sigma(I_2(m,v)||P(m,v))$, we have that $\Theta(val_\lambda(\sigma)) = val_\lambda(\Theta(\sigma))$. We say that Θ is data symmetric if Θ is data symmetric on $I_2(m,v)||P(m,v)$ for all m and v.

Theorem 5.13 *Let $\langle PrivNames_{I_1}, ObsNames_{I_1}, I_1\rangle$ be a process invariant for a parameterized memory system M. Let $\langle PrivNames_{I_2}, ObsNames_{I_2}, I_2\rangle$ be a merge invariant of M with respect to $\langle PrivNames_{I_1}, ObsNames_{I_1}, I_1\rangle$. Then M is sequentially consistent.*

Proof: Since I_2 is a merge invariant of M, there is a merging function Θ 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 $\mathcal{B}0_{I_1,I_2}(m,v)$ 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 σ 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 σ_1 of $M(i,m,v)$ and a run σ_2 of $P_{i+1}(m,v)\|I_2(m,v)$ such that σ is a join of σ_1 and σ_2. We get that σ_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 σ_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 σ_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 τ_1 and τ_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 Θ. Using the second part of Theorem 5.3, we have that there is a join σ' of σ_1 and $\Theta(\sigma_2)$. From Theorem 5.1, we get that σ' is a run of $M(i,m,v)\|I_2(m,v)$. Moreover, it is easy to see that σ' is closed. Hence σ' is sequentially consistent from the induction hypothesis. Therefore, there is a linearization of σ' that is serial. Since any linearization of σ' is also a linearization of σ, we have a linearization of σ that is serial. Hence σ is sequentially consistent. ■

Suppose that we manage to come up with possible invariants I_1 and I_2, and a merging function Θ. 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 $\sigma|_{\mathbb{N}_j} \in \Sigma(I_1(j, v))$, and*
2. *if $\sigma \in \Sigma(I_2(m, v))$ then for all $j \leq m$, we have $\sigma|_{\mathbb{N}_j} \in \Sigma(I_2(j, v))$.*

Requirement 5.2 (Location symmetry) *Let λ 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 $loc_\lambda(\sigma) \in \Sigma(I_1(m, v))$, and*
2. *for all $\sigma \in \Sigma(I_2(m, v))$, we have that $loc_\lambda(\sigma) \in \Sigma(I_2(m, v))$.*

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

1. *for all $\sigma \in Act(I_1(m_1, v))^*$, we have $\sigma \in \Sigma(I_1(m_1, v))$ iff $\sigma \in \Sigma(I_1(m_2, v))$, and*
2. *for all $\sigma \in Act(I_2(m_1, v))^*$, 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 $\sigma \in Act(I_1(m, v))^*$, 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 $\sigma \in Act(I_2(m, v))^*$, and for all $j < m$, we have $\sigma|_j \in \Gamma' I_2(m, v)$.*

Requirement 5.5 (Data symmetry) *Let λ be an $\perp$-extended 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* $val_\lambda(\sigma) \in \Sigma(I_1(m,v))$*, and*

2. *for all* $\sigma \in \Sigma(I_2(m,v))$*, we have that* $val_\lambda(\sigma) \in \Sigma(I_2(m,v))$*.*

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

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

2. *for all* $\sigma \in Act(I_2(m,v_1))^*$*, 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* $\perp$*-extended* v*-map with range* $\{0,1,\perp\}$ *such that* $val_\lambda(\sigma) \notin \Gamma(I_1(m,v))$*.*

2. *if* $\sigma \notin \Gamma'(I_2(m,v))$ *then there is a* $\perp$*-extended* v*-map with range* $\{0,1,\perp\}$ *such that* $val_\lambda(\sigma) \notin \Gamma'(I_2(m,v))$*.*

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 σ be a trace of $C(m,v)$. Fix $p < m$ and let λ be the m-permutation such that $\lambda(p) = 0$, $\lambda(0) = p$, and $\lambda(j) = j$ if j is neither 0 nor p. Then $\sigma|_p = loc_{\lambda^{-1}}(loc_\lambda(\sigma)|_0)$. We have that $loc_\lambda(\sigma)|_0$ is a trace of $C(1,v)$ from Assumptions 5.1, 5.2 and 5.3. If $loc_\lambda(\sigma)|_0$ is not a trace of $I_1(1,v)$ then from Requirement 5.7 there is a $\perp$-extended v-map λ with range $\{0,1,\perp\}$ such that $val_\lambda(loc_\lambda(\sigma)|_0)$ 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 $val_\lambda(loc_\lambda(\sigma)|_0)$ is a trace of $C(1,2)$. Therefore $loc_\lambda(\sigma)|_0$ is a trace of $I_1(1,v)$. Thus we get that $\sigma|_p = loc_{\lambda^{-1}}(loc_\lambda(\sigma)|_0)$ is a trace of $I_1(m,v)$ by Requirements 5.3 and 5.2. From Requirement 5.4, we get that σ is a trace of $I_1(m,v)$.

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

■

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 $B0_{I_2}(1,2)$ is true then $B0_{I_2}(m,v)$ is true, and*
2. *if $B1_{I_2}(1,2)$ is true then $B1_{I_2}(m,v)$ is true.*

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

Let Θ be a merging function. Let $\sigma \in \Sigma(I_2(m,v)\|P(m,v)$ and $[\sigma]_{\Upsilon(m,v)} = \pi_0, \pi_1, \ldots, \pi_{k-1}$. Let $\Theta(\sigma) = \pi_{f^{-1}(0)}, \pi_{f^{-1}(1)}, \ldots, \pi_{f^{-1}(k-1)}$ for some k-permutation f. Let h^Θ 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) $F'(j) = \{x \in K | loc(\pi_x) = j\}$, 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 Λ^Θ be the function on $\Sigma(I_2(m,v)\|P(m,v))$ such that $\Lambda^\Theta(\sigma) = \mathcal{G}(h'(\sigma)) \cup \mathcal{G}(h^\Theta(\sigma))$.

Lemma 5.16 *Let Θ be a local merging function. Let σ 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 $\Lambda^\Theta(\sigma)|_{\mathcal{A}(h^\Theta(\sigma),R)}$.*

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

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 Θ satisfying $B2_{I_2,I_1,\Theta}(3, v)$ then for all $m > 0$, there is a merging function Θ' satisfying $B2_{I_2,I_1,\Theta'}(m, v)$.*

Proof: Fix some $v > 0$. Suppose Θ is a local and location symmetric merging function satisfying $B2_{I_2,I_1,\Theta}(3, v)$. Fix some $m > 0$. We now show how to define a merging function Θ' satisfying $B2_{I_2,I_1,\Theta'}(m, v)$. Let $\sigma \in \Sigma(I_2(m, v) \| P(m, v))$.

Fix $p < m$ and let λ 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 $loc_\lambda(\sigma)|_0$ is a run of $I_2(1, v) \| P(1, v)$ and therefore of $I_2(3, v) \| P(3, v)$. Since Θ is a merging function for $I_2(3, v) \| P(3, v)$, we have that $\Theta(loc_\lambda(\sigma)|_0) \in \Gamma'(I_2(3, v))$.

$$
\begin{aligned}
\Theta(loc_\lambda(\sigma)|_0) &= \Theta(loc_\lambda(\sigma))|_0 && \text{since } \Theta \text{ is local,} \\
&= loc_\lambda(\Theta(\sigma))|_0 && \text{since } \Theta \text{ is location symmetric,} \\
&= loc_\lambda(\Theta(\sigma)|_p) && \text{since } loc_\lambda(\gamma)|_0 = loc_\lambda(\gamma|_p) \text{ for all } \gamma.
\end{aligned}
$$

Since $\Theta(loc_\lambda(\sigma)|_0) \in \Gamma'(I_2(3, v))$, we have that $loc_{\lambda^{-1}}(\Theta(loc_\lambda(\sigma)|_0)) \in \Gamma'(I_2(3, v))$. Thus we get that $loc_{\lambda^{-1}}(\Theta(loc_\lambda(\sigma)|_0)) = loc_{\lambda^{-1}}(loc_\lambda(\Theta(\sigma)|_p)) = \Theta(\sigma)|_p \in \Gamma'(I_2(3, v))$. Therefore $\Theta(loc_\lambda(\sigma)|_0) \in \Gamma'(I_2(m, v))$.

Suppose σ is a run of $I_2(3, v) \| P(3, v)$. Since Θ 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 λ 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 σ is a run of $I_2(3, v) \| P(3, v)$, we have that $loc_\lambda(\sigma)|_{\mathbb{N}_{|R|}}$ is a run of $I_2(3, v) \| P(3, v)$ using Assumptions 5.1, 5.2 and 5.3. Therefore $\Lambda^\Theta(loc_\lambda(\sigma)|_{\mathbb{N}_{|R|}})$ is acyclic.

$$
\begin{aligned}
\Lambda^\Theta(loc_\lambda(\sigma)|_{\mathbb{N}_{|R|}}) &= \Lambda^\Theta(loc_\lambda(\sigma))|_{\mathcal{A}(h^\Theta(loc_\lambda(\sigma)),\mathbb{N}_{|R|})} && \text{from Lemma 5.16,} \\
&= \Lambda^\Theta(loc_\lambda(\sigma))|_{\mathcal{A}(h^\Theta(loc_\lambda(\sigma)),\lambda(R))} && \text{since } \lambda(R) = \mathbb{N}_{|R|}, \\
&= \Lambda^\Theta(\sigma)|_{\mathcal{A}(h^\Theta(\sigma),R)} && \text{from Lemma 5.17.}
\end{aligned}
$$

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 σ. ∎

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 Θ is a data symmetric merging function satisfying $B2_{I_2,I_1,\Theta}(m,2)$ then for all $v > 0$, Θ is a merging function satisfying $B2_{I_2,I_1,\Theta}(m,v)$.*

Proof: Consider any $v > 0$. Let $\sigma \in \Sigma(I_2(m,v)\|P(m,v))$. Suppose $\Theta(\sigma)$ does not satisfy $B2_{I_2,I_1,\Theta}(m,v)$.

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

Case 2. $\Theta(\sigma) \notin \Sigma'(I_2(m,v))$. From Requirement 5.7, we have that there is a $\perp$-extended v-map with range $\{0,1,\perp\}$ such that $val_\lambda(\Theta(\sigma)) \notin \Sigma'(I_2(m,v))$. But $val_\lambda(\Theta(\sigma)) = \Theta(val_\lambda(\sigma))$. because Θ is value symmetric. Moreover $val_\lambda(\sigma) \in \Sigma(I_2(m,1)\|P(m,1))$ from Assumptions 5.4 and 5.5, and Requirements 5.5 and 5.6. Therefore we get that $\Theta(val_\lambda(\sigma)) \notin \Sigma'(I_2(m,2))$ which is a contradiction.

∎

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 Θ be a local, location symmetric and data symmetric merging function. Suppose $A_{I_1}(1,2)$, $B0_{I_1,I_2}(1,2)$ and $B1_{I_2}(1,2)$ are true, and Θ is a witness for $B2_{I_2,I_1,\Theta}(3,2)$. 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 $B0_{I_2}(m,v)$ and $B1_{I_2}(m,v)$ are true. From Lemma 5.19, we have that Θ is a witness for $B2_{I_2,I_1,\Theta'}(3,v)$. From Lemma 5.18, we have that there is a witness Θ' for $B2_{I_2,I_1,\Theta'}(m,v)$. From Theorem 5.13, we have that $M(n,m,v)$ is sequentially consistent for all n. ∎

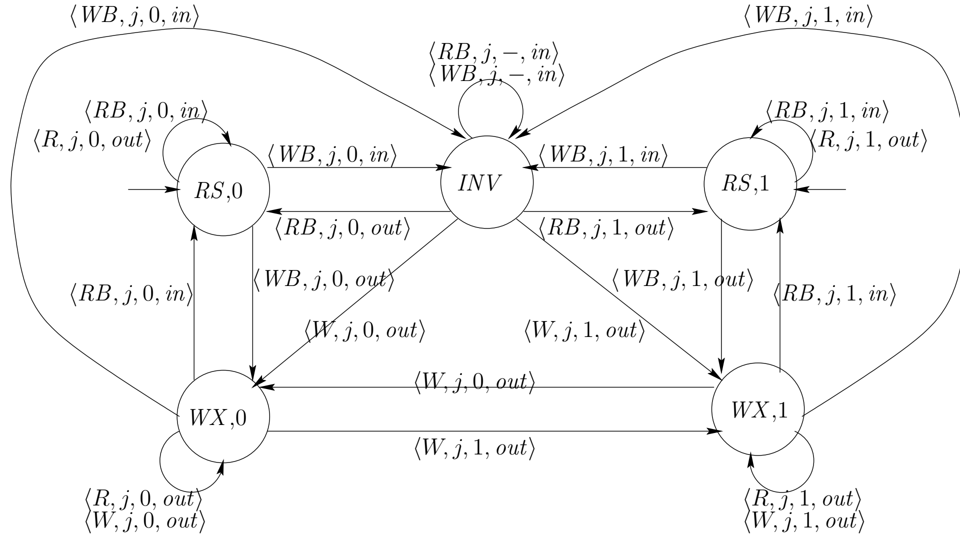


Figure 5.1: Snoopy cache coherence

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 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 $\langle C, P\rangle$, 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 respectively, 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 jth 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 jth 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.[2] 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

[2]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.

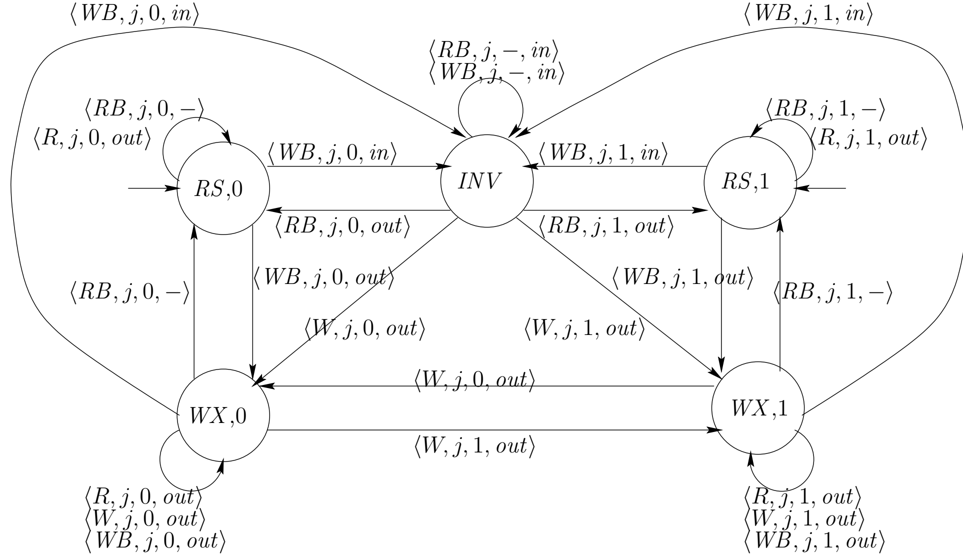


Figure 5.2: Snoopy cache coherence invariant

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 Θ is the identity function.[3] It is easy to see that the merging function is local, location symmetric and data symmetric. We used MOCHA to verify the proof obligations $A_I(1, 2)$, $B0_{I,I}(1, 2)$, $B1_I(1, 2)$ and $B2_{I,I}(3, 2)$. MOCHA required a few minutes to check these obligations.

[3]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 directions 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*-μ calculus, the logic of of emptiness queries over the state sets represented by *post*-μ calculus sentences. 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 from LTL formulas to *post*-μ queries has exponential worst-case complexity. The complexity and alternation depth of the translation match the standard translation to *pre*-μ calculus queries. We also gave a translation to *post*-μ queries of linear size and alternation depth one for co-Büchi automata. Again, the translation matches the corresponding translation to *pre*-μ calculus queries. Finally, we translated safety automata to *post*-μ 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 an example —the sliding window protocol— and obtained encouraging results.

The exact characterization of the intersection of the two logics, query *pre*-μ calculus and query *post*-μ 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*-μ query φ is expressible as a *post*-μ query iff φ is equi-linear. While the intersection identifies the queries that can be model checked by both forward and backward symbolic state traversal, it is the union that identifies the queries that can be model checked at all symbolically, by mixed forward and backward state traversal. A mix of both approaches could potentially be more efficient than either one in isolation. Hence, there is a need for further investigation into the efficacy of mixed symbolic forward and backward reasoning as a heuristic for efficient model checking.

In Chapter 3, we presented 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 illustrated 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 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.

We would like to pursue the concept of "formal design" described in Chapter 4. In "formal design," design and verification activities proceed in parallel and help each other. 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. There are several requirements for making "formal design" a reality. First, our experience with VGI has shown that we need other proof decomposition techniques apart from the assume-guarantee rule. Second, we need design idioms that are general so that realistic designs can be constructed out of combinations and structured so that a library of proof decomposition techniques can be applied. Third, we need tools with a tighter

integration between design and testing [1].

In Chapter 5, we showed how to use model checking to verify sequential consistency of 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 reasonable assumptions on the memory system. 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 local, location symmetric, and data symmetric serializer exists for a memory system with n processors, n memory locations and 2 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 model checked against the serial memory specification. The second method lets us prove sequential consistency for any number of processors, memory locations and data values, although it requires more human effort than the first method. It is based on a novel induction scheme involving two invariant processes —a process invariant and a 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 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. We have found the inductive proof method based on process and merge invariants quite difficult to apply in practice. For instance, it could not be directly applied as described in Chapter 5 to the lazy caching protocol [ABM93]. We had to weaken one of the requirements on the merge invariant 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. With that method, there are two obstacles to achieving the holy grail of fully automatic verification —the manual construction of the serializer and the state explosion problem as the number of processors increases. We believe that the first problem cannot be solved completely. The verifier has to provide some input to describe the serializer. But we would like to

[1] The author biased suggestion would be that the testing should be exhaustive and performed by model checking.

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. This would not only simplify the construction of the serializer, but partition the model checking problem into two sub-problems as well. 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. We would like to investigate special compositional proof strategies that make use of the fact that the processors in the system are identical to each other.

Bibliography

[AB86] J. Archibald and J.-L. Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986. 9

[ABM93] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, 1993. 1, 1, 12

[AG96] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996. 1, 9

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996. 1, 1, 3, 3, 3.2, 3.2, 1, 4.1

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 521–525. Springer-Verlag, 1998. 1, 3, 7, 9

[AHR98] R. Alur, T.A. Henzinger, and S.K. Rajamani. Symbolic exploration of transition hierarchies. In B. Steffen, editor, *TACAS 98: Tools and Algorithms for Construction and Analysis of Systems*, Lecture Notes in Computer Science 1384, pages 330–344. Springer-Verlag, 1998. 7, 7

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. 3

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995. 1, 3

[AMP96] R. Alur, K.L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 219–228, 1996. 1, 9

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property-preserving simulations. In G. von Bochmann and D.K. Probst, editors, *CAV 92: Computer Aided Verification*, Lecture Notes in Computer Science 663, pages 260–273. Springer-Verlag, 1992. 3

[BC96] G. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In *Proc. 11th IEEE Symposium on Logic in Computer Science*, pages 304–312, June 1996. 2, 3, 3

[BCG88] M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988. 2.2

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992. 1, 1, 2, 2

[BD94] J.R. Burch and D.L. Dill. Automatic verification of pipelined microprocessor control. In D.L. Dill, editor, *CAV 94: Computer Aided Verification*, Lecture Notes in Computer Science 818, pages 68–80. Springer-Verlag, 1994. 3.3, 7

[BHSV+96] R.K. Brayton, G.D. Hachtel, A.L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G.M. Swamy, and T. Villa. VIS: A system for verification and synthesis. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 428–432. Springer-Verlag, 1996. 1, 2, 2.4.3, 4.3

[BM79a] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, New York, 1979. 1

[BM79b] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1979. 1

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986. 2

[CD88] E.M. Clarke and I.A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Proc. Workshop on Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer-Verlag, 1988. 2.2.2

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logic of Programs*, Lecture Notes in Computer Science 131, pages 52–71. Springer-Verlag, 1981. 1

[CGH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications*, pages 15–30, 1993. 9

[CGH94] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In D. Dill, editor, *CAV 94: Computer Aided Verification*, Lecture Notes in Computer Science 818, pages 415–427. Springer-Verlag, 1994. 2, 2

[CGL92] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 343–354. ACM Press, 1992. 3

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency: Reflections and Perspectives*, Lecture Notes in Computer Science 803. Springer-Verlag, 1994. 2

[CKS92] R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for the modal μ-calculus. In G. v. Bochmann and D. Probst, editors, *CAV 92: Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 410–422. Springer-Verlag, 1992. 2.1.1, 1

[Dam94] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126:77–96, 1994. 3

[Dil96] D. L. Dill. The Murϕ Verification System. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 390–393. Springer-Verlag, 1996. 1, 2

[DP97] W. Damm and A. Pnueli. Verifying out-of-order executions. In *CHARME 97: IFIP Working Conference on Correct Hardware Design and Verification Methods*, pages 23–47. Chapman & Hall, 1997. 3, 3.4, 3.4

[Eír98] Á.Th. Eíriksson. The formal design of 1M-gate ASICs. In G. Gopalakrishnan and P. Windley, editors, *FMCAD 98: Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science 1522, pages 49–63. Springer-Verlag, 1998. 4.3, 4.3.3

[EL86] E.A. Emerson and C. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proceedings of the 1st Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, 1986. 2, 1, 3, 3

[EM95] Á.Th. Eíriksson and K.L. McMillan. Using formal verification/analysis methods on the critical path in system design: a case study. In P. Wolper, editor, *CAV 95: Computer Aided Verification*, Lecture Notes in Computer Science 939, pages 367–380. Springer-Verlag, 1995. 9

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers, 1990. 1

[GH93] J.V. Guttag and J.J. Horning. *Larch: languages and tools for formal specification.* Springer-Verlag, 1993. 1

[GK94] O. Grumberg and R.P. Kurshan. How linear can branching-time be. In *Proc. 1st International Conference on Temporal Logic*, volume 827 of *Lecture Notes in Artificial Intelligence*, pages 180–194. Springer-Verlag, 1994. 2.4.1, 2.4.1, 6

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL.* Cambridge University Press, 1993. 1

[GMG91] P.B. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 292–303, 1991. 9

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996. 1

[GPVW95] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, August 1995. 2

[Gra94] S. Graf. Verification of a distributed cache memory by using abstractions. In D.L. Dill, editor, *CAV 94: Computer Aided Verification*, Lecture Notes in Computer Science 818, pages 207–219. Springer-Verlag, 1994. 9

[GW91] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pages 406–415. IEEE Computer Society Press, 1991. 1

[HIK98] P.-H. Ho, A.J. Isles, and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *ICCAD 98: IEEE/ACM International Conference on Computer Aided Design*, pages 529–536. IEEE Computer Society Press, 1998. 3.3

[HIKB98] R. Hojati, A. Isles, D. Kirkpatrick, and R.K. Brayton. Verification using uninterpreted functions and finite instantiations. In A. Camilleri and M. Srivas, editors, *FMCAD 96: Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science, pages 218–232. Springer-Verlag, 1998. 3.3

[Hil98] M.D. Hill. Multiprocessors should support simple memory consistency models. *IEEE Computer*, 31(8):28–34, 1998. 9

[HKQ98] T.A. Henzinger, O. Kupferman, and S. Qadeer. From pre-historic to post-modern symbolic model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 195–206. Springer-Verlag, 1998. (document)

[HLQR99] T.A. Henzinger, X. Liu, S. Qadeer, and S.K. Rajamani. Formal specification and verification of a dataflow processor array. In *ICCAD 99: IEEE/ACM International Conference on Computer Aided Design*. IEEE Computer Society Press, 1999. To appear. (document)

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. 1

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991. 1

[HP96a] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann, 1996. 1, 1, 9, 9, 5.6

[HP96b] G.J. Holzmann and D.A. Peled. The State of SPIN. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 385–389. Springer-Verlag, 1996. 1, 2

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: methodology and case studies. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 440–451. Springer-Verlag, 1998. (document), 4, 4.3.3

[HQR99a] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Assume-guarantee refinement between different time scales. In N. Halbwachs and D. Peled, editors, *CAV 99: Computer Aided Verification*, Lecture Notes in Computer Science 1633, pages 208–221. Springer-Verlag, 1999. (document)

[HQR99b] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In N. Halbwachs and D. Peled, editors, *CAV 99: Computer Aided Verification*, Lecture Notes in Computer Science 1633, pages 301–315. Springer-Verlag, 1999. (document)

[HSG98] R. Hosabettu, M.K. Srivas, and G. Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 122–134. Springer-Verlag, 1998. 3.3

[HSG99] R. Hosabettu, M.K. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In N. Halbwachs and D. Peled, editors, *CAV 99: Computer Aided Verification*, Lecture Notes in Computer Science 1633. Springer-Verlag, 1999. 3.3

[ID96] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1–2):41–75, 1996. 9

[IHB98] A.J. Isles, R. Hojati, and R.K. Brayton. Computing reachable control states of systems modeled with uninterpreted functions and infinite memory. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 256–267. Springer-Verlag, 1998. 3.3

[IN97] H. Iwashita and T. Nakata. Forward model checking techniques oriented to buggy designs. In *ICCAD 97: IEEE/ACM International Conference on Computer Aided Design*, pages 400–404. IEEE Computer Society Press, 1997. 2, 2, 2.4.1, 2.4.3

[INH96] H. Iwashita, T. Nakata, and F. Hirose. CTL model checking based on forward state traversal. In *ICCAD 96: IEEE/ACM International Conference on Computer Aided Design*, pages 82–87. IEEE Computer Society Press, 1996. 1, 2, 2, 2.4.1, 2.4.3

[JBJ95] S. Jain, R.E. Bryant, and A. Jain. Automatic clock abstraction from sequential circuits. In *Proceedings of the 32nd Design Automation Conference*, pages 707–711, 1995. 7

[KM89] R.P. Kurshan and K.L. McMillan. A structural induction theorem for processes. In *Proceedings of the 8th Annual Symposium on Principles of Distributed Computing*, pages 239–247, 1989. 1, 9, 5.5.1

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313. IEEE Computer Society Press, 1994. 1

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983. 1, 2, 2.1.1

[KP92] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2):107–120, 1992. 1, 9

[KP95] O. Kupferman and A. Pnueli. Once and for all. In *Proc. 10th IEEE Symposium on Logic in Computer Science*, pages 25–35. IEEE Computer Society Press, 1995. 2.4.2

[KSL95] A. Kuehlmann, A. Srinivasan, and D.P. LaPotin. Verity — A formal verification program for custom CMOS circuits. *IBM Journal on Research and Development*, 39(1-2):149–165, 1995. 7

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994. 3

[KV98a] O. Kupferman and M.Y. Vardi. Freedom, weakness and determinism: from linear-time to branching-time. In *Proc. 13th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1998. 3, 4

[KV98b] O. Kupferman and M.Y. Vardi. Verification of fair transition systems. *Chicago Journal of Theoretical Computer Science*, 1998(2), March 1998. 3

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed program. *Communications of the ACM*, 21(7):558–565, 1978. 9

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979. 1, 9, 5.2

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983. 3

[LD92] P. Loewenstein and D.L. Dill. Verification of a multiprocessor cache protocol using simulation relations and higher-order logic. *Formal Methods in System Design*, 1(4):355–383, 1992. 9

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, 1992. 1

[LLOR99] P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching in TLA. To appear in *Distributed Computing*, 1999. 9

[LPZ85] O. Lichtenstein, A. Pnueli, and L.D. Zuck. The glory of the past. In R. Parikh, editor, *Logics of Programs*, Lecture Notes in Computer Science 193, pages 196–218. Springer-Verlag, 1985. 2.4.2

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual Symposium on Principles of Distributed Computing*, pages 137–151. ACM Press, 1987. 3

[LV95] N.A. Lynch and F. Vaandrager. Forward and backward simulations, Part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995. 3

[Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996. 1

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981. 1, 3

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem*. Kluwer Academic Publishers, 1993. 1, 1, 2

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 24–35. Springer-Verlag, 1997. 1, 3, 3, 3

[McM98] K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 110–121. Springer-Verlag, 1998. 3, 3.3, 3.4, 3.4, 4, 4.3.2, 4.3.3

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989. 1

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992. 1, 2.4.2

[MS72] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proc. 13th IEEE Symp. on Switching and Automata Theory*, pages 125–129, 1972. 1, 3

[MS91] K. L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, pages 242–251, 1991. 9

[NGMG98] R. Nalumasu, R. Ghughal, A. Mokkedem, and G. Gopalakrishnan. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In A.J. Hu and M.Y. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 464–476. Springer-Verlag, 1998. 9

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 411–414. Springer-Verlag, 1996. 1

[PD95] F. Pong and M. Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, 1995. 9

[PD96] S. Park and D.L. Dill. Protocol verification by aggregation of distributed transactions. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 300–310. Springer-Verlag, 1996. 9

[Pel93] D. Peled. All from one, one for all: on model checking using representatives. In C. Courcoubetis, editor, *CAV 93: Computer Aided Verification*, Lecture Notes in Computer Science 697, pages 409–423. Springer-Verlag, 1993. 1

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model checking. In D. Dill, editor, *CAV 94: Computer Aided Verification*, Lecture Notes in Computer Science 818. Springer-Verlag, 1994. 2

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977. 1

[PSCH98] M. Plakal, D.J. Sorin, A.E. Condon, and M.D. Hill. Lamport clocks: verifying a directory cache-coherence protocol. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 67–76, 1998. 9, 9

[QS81] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Fifth International Symposium on Programming*, Lecture Notes in Computer Science 137, pages 337–351. Springer-Verlag, 1981. 1

[Raj99] S.K. Rajamani. *New directions in refinement checking*. PhD thesis, University of California at Berkeley, 1999. 4.3

[SG97] D.J. Scales and K. Gharachorloo. Design and performance of the Shasta distributed shared memory protocol. In *Proceedings of the 11th ACM International Conference on Supercomputing*, pages 245–252. ACM, 1997. 1

[SH97] J. Sawada and W.A. Hunt. Trace table approach for pipelined microprocessor verification. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 364–375. Springer-Verlag, 1997. 3.3

[SH98a] J. Sawada and W.A. Hunt. Processor verification with precise exceptions and speculative execution. In A.J. Hu and M.Y. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 135–146. Springer-Verlag, 1998. 3.3

[SH98b] J. Skakkebæk and W.A. Hunt. Formal verification of out-of-order execution using incremental flushing. In A.J. Hu and M.Y. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 98–109. Springer-Verlag, 1998. 3.3

[SR97] V.P. Srini and J.M. Rabaey. An architecture for web-based image processing. In *Proceedings of the SPIE Conference 3166*, pages 90–101, 1997. 4.3

[Sta85] E.W. Stark. A proof technique for rely/guarantee properties. In *Proceedings of the 5th Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 206, pages 369–391. Springer-Verlag, 1985. 1, 3

[STUR98] V.P. Srini, J. Thendean, S.Z. Ueng, and J.M. Rabaey. A parallel DSP with memory and I/O processors. In *Proceedings of the SPIE Conference 3452*, pages 2–13, 1998. 7, 4.3

[Val90] A. Valmari. A stubborn attack on state explosion. In R.P. Kurshan and E.M. Clarke, editors, *CAV 90: Computer Aided Verification*, Lecture Notes in Computer Science 531, pages 25–42. Springer-Verlag, 1990. 1

[Var98] M.Y. Vardi. Reasoning about the past with two-way automata. In *Proc. 25th International Coll. on Automata, Languages, and Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1998. 6

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994. 3

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 *SEQSIZE* 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
      $[]\, true \rightarrow seq' := 0$

update

$[] window < WINDOW \rightarrow msgP' := nondet; seq' := seq + 1$

endatom

atom controls *sndStore* **reads** *seq*, *sndStore* **awaits** *seq*, *msgP*

init

$[] true \rightarrow$ **forall** i $sndStore'[i] := 0$

update

$[] seq' = seq + 1 \rightarrow$

forall i $sndStore'[i] :=$ **if** $(seq\%WINDOW = i)$ **then** $msgP'$ **else** $sndStore[i]$ **fi**

endatom

atom

controls *msgX*, *seqX*, *msgValid*

reads *index*, *seq*, *msgX*, *seqX*, *msgValid*, *seq*

awaits *sndStore*, *seq*, *busy*, *index*

init

$[] true \rightarrow msgValid' := false$

update

$[] seq' = seq + 1 \rightarrow msgX' := sndStore'[seq\%WINDOW]; msgValid' := true; seqX' := seq$

$[] busy'[index\%WINDOW] \rightarrow$

$msgX' := sndStore'[index\%WINDOW]; msgValid' := true; seqX' := index$

$[] true \rightarrow msgValid' := false; msgX' := nondet; seqX' := nondet$

endatom

atom controls *index* **reads** *index*, *window* **awaits** *busy*

init

$[] true \rightarrow index' := 0$

update

$[]\ busy'[index\%WINDOW] \wedge window > 0 \rightarrow index' := index + 1$

endatom

atom controls *window* **reads** *window*, *index*, *seq* **awaits** *index*, *seq*
init
$[]true \rightarrow window' := 0$
update
$[](seq' = seq + 1) \wedge \ (index' = index + 1) \rightarrow window' := window + 1$
$[] \ (seq' = seq + 1) \wedge (index' = index + 1) \rightarrow window' := window - 1$
endatom

#foreach j = (0..*WINDOW*-1)
atom controls *busy* [$j] **reads** *seq*, *ackX*, *ackValid*, *busy* [$j], *seq*, *ackX* **awaits** *seq*
init
$[]true \rightarrow busy'[\$j] := false$
update
$[]seq' = seq + 1 \wedge (seq\%WINDOW = \$j) \rightarrow busy'[\$j] := true$
$[] \ (seq' = seq + 1) \wedge ackValid \wedge (ackX\%WINDOW = \$j) \rightarrow busy'[\$j] := false$
endatom
#endforeach

lazy atom controls *ack*, *msgC* **reads** *ack*, *recvd*, *msgC*, *recStore*
init
$[]true \rightarrow ack' := 0$
update
$[]recvd[ack\%WINDOW] \rightarrow msgC' := recStore[ack\%WINDOW]; ack' := ack + 1$
endatom

#foreach j = (0..*WINDOW*-1)
atom controls *recvd* [$j] **reads** *recvd* [$j], *ack* **awaits** *seqX*, *msgValid*, *ack*
init
$[]true \rightarrow recvd'[\$j] := false$
update
$[]msgValid' \wedge (seqX' - ack' < \ WINDOW) \wedge (seqX'\%WINDOW = \$j) \rightarrow recvd'[\$j] := true$
$[]ack' = ack + 1 \wedge (ack\%WINDOW = \$j) \rightarrow recvd'[\$j] := false$

endatom

#endforeach

atom controls *recStore* **reads** *recvd*, *recStore* **awaits** *msgX*, *msgValid*, *seqX*, *ack*

init

$[]true \rightarrow$ **forall**$i\,recStore'[i] := 0$

update

$[]msgValid' \wedge (seqX' - ack' < WINDOW) \rightarrow$

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

$[]true \rightarrow ackValid' := false$

update

$[]ack' = ack + 1 \rightarrow ackX' := ack; ackValid' := true$

$[]msgValid' \wedge (ack' - seqX' > 0) \wedge (ack' - seqX' <= WINDOW) \rightarrow$

$ackX' := seqX'; ackValid' := true$

$[]true \rightarrow ackValid' := false; ackX' := nondet$

endatom

endmodule