Mechanical Verification of Reactive Systems

by

Panagiotis Manolios, B.S.,M.A.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2001

The Dissertation Committee for Panagiotis Manolios
Certifies that this is the approved version of the following dissertation:

Mechanical Verification of Reactive Systems

Committee:

J Strother Moore, Supervisor

Lorenzo Alvisi

E. Allen Emerson

Matt Kaufmann

Jayadev Misra

Amir Pnueli

To Helen and Emmanuel

Acknowledgments

First and foremost I want to thank J Moore. J provided financial support when we first started working together, from his Admiral B.R. Inman Centennial Chair in Computing Theory. More importantly, I learned a great deal from J about being a researcher, a teacher, and a human being during our many collaborations and interactions.

The impetus for my dissertation work came from discussions and collaborations with fellow graduate students, including Kedar Namjoshi, Rob Sumners, and Jun Sawada. I met many other wonderful graduate students; the ones who helped shape my view of computing science include Richard Trefler, Rajeev Joshi, Vasilis Samoladas, Emilio Camahort, Yannis Smaragdakis, Nina Amla, John Gunnels, John Havlicek, Will Adams, and Robert Krug.

The teachers that inspired me and taught me things I still remember include Edsger W. Dijkstra, E. Allen Emerson, Jayadev Misra, and, of course, J Moore. I thank Edsger for inviting me to join the Austin Tuesday Afternoon Club, where every week I had the pleasure of reading and analyzing papers on various topics with interesting people.

I was very fortunate that Matt Kaufmann started attending the ACL2 meetings and agreed to be on my committee. Matt has read my work as carefully as anyone else and has been an invaluable resource throughout my graduate career.

My interactions with Warren Hunt, Yuan Yu, and Natarajan Shankar have been enlightening and enjoyable.

The Semiconductor Research Corporation funded my work for several years under contract 99-TJ-685 and I am very grateful.

Finally, my family has been wonderful. Helen and Emmanuel are the source of the greatest joy in my life. I would like to thank my parents and siblings, but especially my sister Fay who left New York and moved to Austin when she found out we were having a baby.

PANAGIOTIS MANOLIOS

The University of Texas at Austin
August 2001

Mechanical Verification of Reactive Systems

Publication No. ____________________

Panagiotis Manolios, Ph.D.

The University of Texas at Austin, 2001

Supervisor: J Strother Moore

It is increasingly crucial to the well-being of our society that safety-critical computing systems behave correctly. Examples of such systems include air traffic control systems, medical monitoring systems, systems for controlling nuclear reactors, communication protocols, and microprocessors. All of the above are examples of *reactive systems*, non-terminating computing systems that maintain an ongoing interaction with their environment. Establishing correctness requires proof. However, due to the complexity of reactive systems, hand proofs are not reliable: just the description of a system can be hundreds of pages long! The only viable solution is *mechanical verification*, where a computer program is employed to check and to help construct proofs of correctness. The main obstacle to the widespread use of mechanical verification is that it requires considerable human effort. In this dissertation, we show how to reduce the effort involved in the mechanical verification of reactive systems.

We start by considering notions of correctness that allow us to relate a reactive system to a simpler system that acts as the specification. Notions of correctness for reactive systems characterize the relationship between infinite computations of the implementation and of the specification. To simplify the effort involved in reasoning mechanically about such notions, we develop a compositional theory of refinement with proof rules that are based on reasoning about single steps of reactive systems, as opposed to infinite computations, which is otherwise required.

In the next part of the dissertation, we present a novel approach to combining theorem proving and model checking, the two leading mechanical verification paradigms. Theorem proving is very general, but requires considerable human interaction; model checking does not require human reasoning, but is only applicable to "small" systems. With our approach, theorem proving is employed to prove the correctness of an abstraction that yields a reduced system. We introduce algorithms for extracting reduced systems which are then analyzed using model checking. This is a very general abstraction technique that allows great flexibility in choosing an appropriate abstraction. The general idea is to reduce the problem to one that can be model checked in a reasonable amount of time. We use the ACL2 (A Computational Logic for Applicative Common Lisp) theorem proving system to implement the ideas, *e.g.*, we develop and verify a model checker for the Mu-Calculus using ACL2.

The dissertation concludes with two case studies. The first case study concerns the verification of a simple communications protocol. This case study highlights the use of our extraction algorithm and our approach to combining theorem proving and model checking. The second case study involves the verification of a simple pipelined machine from the literature. We show that the notion of correctness we use more faithfully represents the informal correctness requirements than other notions of correctness currently in use. In addition, we show that it is possible to automate much of the verification effort in ACL2 by using libraries of general purpose theorems. Finally, we verify several variants of the pipelined machine including machines with exceptions, interrupts, and netlist (gate-level) descriptions.

Contents

List of Symbols

List of Figures

List of Tables

Part I

Introduction and Preliminaries

Chapter 1

Introduction

Society is increasingly dependent on *reactive systems*, non-terminating computing systems that maintain an ongoing interaction with their environment. Examples of safety-critical reactive systems include air traffic control systems, medical monitoring equipment, systems for controlling nuclear reactors, microprocessors, and communication protocols.

The recent PITAC[1] report to the president makes it clear that building dependable computing systems is one of the major challenges facing the computing field ([Pre99], page 4).

> We have become dangerously dependent on large software systems whose behavior is not well understood and which often fail in unpredicted ways.

The correct behavior of these systems depends on the correct behavior of the hardware and software used to implement them. To verify that such systems are indeed correct requires proving that the systems satisfy their

[1] President's Information Technology Advisory Committee

specifications. Due to the complexity of such systems, hand proofs are infeasible and mechanical verification, *i.e.*, computer-aided verification, is the only reliable way of ensuring correctness.

In this dissertation, we examine techniques for reducing the effort required to construct mechanical proofs of correctness. To this end, we extend the theory of stuttering simulation and bisimulation and build libraries of theorems and techniques for reasoning about stuttering simulation and bisimulation in the ACL2[2] theorem proving system. We then present a novel approach to combining theorem proving and model checking (the two leading mechanical verification paradigms). The dissertation ends with two case studies where we apply our theory and techniques to verify a simple communications protocol and a simple pipelined machine.

1.1 Reactive Systems

In this section we examine reactive systems and their verification in more detail. It is instructive to compare reactive systems with the traditional *transformational systems*. Transformational systems accept an input, perform a transformation, and return a result. Such systems are supposed to terminate. Recursive function theory is based on this view of computation. The semantics of transformational systems is given as a relation between initial and final states and underly Hoare logic [Hoa69], the weakest precondition calculus [Dij76], and related reasoning formalisms. In contrast, reactive systems are non-terminating and their semantics is based on their infinite computations.

There are various approaches to reasoning about reactive systems. One

[2] A Computational Logic for Applicative Common Lisp

is to define a notion of equivalence between systems based on their computations. To prove the correctness of a reactive system we show that it is equivalent to another system that serves as a specification. For example, in chapter 12 we show that a pipelined machine is equivalent to the machine defined by the instruction set architecture. This approach is essentially the one used in process algebra [Mil90]. Another approach is to use temporal logic to specify properties of computations: a reactive system is correct if its computations satisfy the set of specification formulas. For example, in chapter 11 we show that, under some fairness conditions, messages sent by a communications protocol are eventually received. Temporal logic was proposed as a formalism for specifying the correctness of reactive systems by Pnueli [Pnu77].

In this dissertation, a computing system is a mathematical object. For example, we think of a microprocessor as a function that given some set of inputs and the current state returns the outputs and the next state. Of course, there are also physical devices that are called microprocessors and someone is responsible for ensuring that the physical devices, when operated in reasonable environments, correspond to the mathematical objects. We ignore such issues as they lie within the purview of the engineers and physicists.

1.2 Mechanical Verification

Since both a reactive system and its specification are mathematical objects, correctness is established by proving that the system satisfies its specification. The proof has to refer to the reactive system under consideration, but such systems tend to be very complicated and can easily require over one hundred pages to define. It is improbable that a hand-constructed "proof" is error-free and it is very difficult and tedious for a human to check. The only reliable way of constructing and checking these proofs is to use a computer program to check the proof and to fill in some of the details.

Unfortunately, commercial systems are very rarely fully verified. To ensure that many of the obvious errors are removed, extensive testing is performed. There are various reasons for this. The most important are that full verification is difficult and requires well-trained scientists. Even so, there have been advances that have made it possible to define very complicated systems. For example, advances in programming languages, including high-level languages, polymorphic type systems, functional programming, libraries of data structures, and domain-specific languages, have all helped. In addition, decision procedures such as model checking, static analysis, and type checking have been very useful for automatically detecting errors. While such methods can be used to detect simple errors, full verification has remained elusive.

Consider the example of microprocessor verification. At Intel, the design of a Pentium-class microprocessor requires about six hundred engineers working for several years. Informal estimates indicate that about 40% of the overall budget is used to ensure the correctness of the microprocessor. This figure has been rising and is expected to keep rising in the future. Even with such resources allocated to ensuring correctness, bugs are common. Some of the bugs are extremely costly, *e.g.*, the Pentium FDIV (Floating point DIVision) bug [Coe95, Ede97] led to a $475 million write-off by Intel. The situation is similar in the software arena. At the 1998 SIGMOD (Special Interest Group on Management of Data) conference, Bill Gates said that Microsoft was getting to the point where they have more software testers than developers [Gat98]. Clearly, the verification of computing systems is one of the major challenges of computing science.

The goal of building a calculus of thought has been a dream from at least the time of Leibniz who wanted to construct a method by which all truths could be reduced to calculation, including the truths of the moral and metaphysical disciplines![3] Leibniz thought that such a program could be carried out in five years. Needless to say, Leibniz was overly optimistic.

It took centuries before Leibniz's dream, restricted to mathematics, was seriously pursued. One of the key figures was Hilbert who, in response to the various paradoxes discovered at the end of the nineteenth century, wanted to place mathematics on a solid foundation. "Hilbert's program" consisted of showing that all of mathematics can be formalized, that there is an effective procedure for deciding if a formula is provable, and the resulting theory can be shown consistent using only finitary methods.

The work of Gödel and Turing showed that Hilbert's program cannot be realized: finitistic consistency proofs are not possible within a given system and there is no algorithm that can decide if a formula is provable (in a rich enough logic). But, there are programs that can be used to check proofs. For example, the ACL2 system, due to Kaufmann and Moore [KM, KMM00b], can be used to check if a proof outline can be turned into a formal proof. Often the proof outline is just the statement of the theorem, in which case ACL2 is responsible for filling in all the details. More often, the user is responsible for giving a fairly detailed proof outline.

[3]Dijkstra and Thomas describe Leibniz's dream in more detail [Dij01, Tho01].

1.3 Contributions and Structure of this Dissertation

In this section, we outline our contributions and the structure of this dissertation. Some of the results reported in this dissertation have previously appeared in various conference proceedings and books [MNS99, Man00a, KMM00b, KMM00a, Man00c, Man00d].

The dissertation consists of the following four parts.

- Introduction and Preliminaries
- Notions of Correctness
- Combining Theorem Proving and Model Checking
- Case Studies and Conclusions

Chapters with technical content have a section that contains bibliographic notes.

In the next chapter of this part we review some preliminaries including notational conventions, theorem proving, model checking, and the notions of simulation and bisimulation. This chapter can be skimmed and consulted as needed. The list of symbols on page xiv and the index at the end of the dissertation can be used to find definitions of symbols and explanations of notational conventions.

Part II deals with notions of correctness. Recall that one method of verifying reactive systems is to show that the system is related to another system that serves as the specification. This involves comparing systems at different levels of abstraction, where a single step of the abstract system may correspond to several steps of the concrete system. For this reason, the relations between systems that we consider are insensitive to finite stuttering. The notions of correctness we consider are stuttering simulation and stuttering bisimulation. Stuttering bisimulation was introduced by Browne, Clarke, and Grumberg [BCG88]. Stuttering simulation is a weaker variant. We develop the theory of these notions in ways that prove useful for mechanical verification. We show that these notions satisfy various algebraic properties, *e.g.*, the relational composition of two stuttering simulations is a stuttering simulation. We also develop a theory of refinement based on these notions and present compositional proof rules for showing stuttering simulations and bisimulation in stages. Proving stuttering simulations and bisimulations directly requires reasoning about infinite computations. We would rather reason about single steps. To this end, we present several sound and complete proof rules, similar to the proof rule of Namjoshi [Nam97]. The proof rules allow us to prove stuttering simulations and bisimulations, but the reasoning is about single steps of the systems in question. Some of our proof rules for stuttering bisimulation are simpler than the one by Namjoshi and some are more general. They shed further light on the structure of stuttering bisimulations, which has allowed us to construct ACL2 libraries that we use to further automate mechanical verification.

Part III describes a novel approach to combining theorem proving and model checking for the verification of reactive systems. The idea is to describe a large system in ACL2 and to reduce the system to a finite-state system by proving a stuttering bisimulation. This is accomplished by defining a relation using what we call representative functions and proving that the relation is a stuttering bisimulation. Such a proof implies that the system is equivalent to an induced quotient structure. The idea is to check the quotient structure, but constructing the quotient structure can be difficult because determining if there is a transition between states in the quotient structure depends on whether there is a transition between some pair of related states in the original system (the number of such pairs may be infinite). Moreover, the quotient structure may be infinite-state, but the set of its reachable states may be finite. To address these two concerns, we introduce on-the-fly algorithms that automatically extract the quotient structure. We present two extraction algorithms. One is based on the simple *state* representative functions but is incomplete since it is possible that a stuttering bisimulation induces a finite quotient structure, but there is no state representative function that can be used to extract it. *Set* representative functions are more complicated, but we prove completeness: for any stuttering bisimulation that induces a finite quotient, there is a set representative function that can be used to extract it. Once extracted, the quotient structure can be model checked. To this end, we use a model checker for the Mu-Calculus that we have written and verified using ACL2.

In part IV we present two case studies and conclusions. The first case study is the verification of the alternating bit protocol, a simple communications protocol. We use our approach to combining theorem proving and model checking to prove a stuttering bisimulation on the alternating bit protocol, which is then used to extract a quotient, which is infinite-state, but has a finite set of reachable states. The quotient is then model checked with our model checker. The second case study is the verification of a simple pipelined machine due to Sawada [Saw00]. We discuss notions of correctness and their importance in some detail and compare our notion of correctness, which is based on stuttering bisimulation, to the variant of the Burch and Dill notion of correctness [BD94] used by Sawada. We show, with mechanical proof, that the Burch and Dill notion can be satisfied by incorrect machines, *e.g.*, machines that deadlock. In contrast, we argue that no incorrect machine satisfies our notion of correctness. In addition, we give an overview of the libraries of ACL2 theorems used and explain how to automate much of the verification, *e.g.*, the verification of the pipelined machine is automatic. We examine various variants of the pipelined machine including machines with exceptions, interrupts (which lead to non-determinism), and netlist (gate-level) descriptions and show that our notion of correctness applies to these extensions. Many of the variant machines are verified using the compositional proof rule for stuttering bisimulations from part II.

Chapter 2

Preliminaries

In this chapter we discuss notational issues and we present some background information including various temporal calculi/logics and the notions of simulation and bisimulation. Most of the notation is standard and some of it is ambiguous, but the intended meaning should be clear from the context.

2.1 Notation and Mathematical Preliminaries

$\mathbb{N}$ and ω both denote the natural numbers, *i.e.*, $\{0, 1, \ldots\}$. The ordered pair whose first component is i and whose second component is j is denoted $\langle i, j\rangle$. $[i..j]$ denotes the closed interval $\{k \in \mathbb{N} \ : \ i \leq k \leq j\}$; parentheses are used to denote open and half-open intervals, *e.g.*, $[i..j)$ denotes the set $\{k \in \mathbb{N} \ : \ i \leq k < j\}$. $Dom.f$ denotes the domain of function f. The disjoint union operator is denoted by $\uplus$. Cardinality of a set S is denoted by $\#S$. $\mathcal{P}(S)$ denotes the powerset of S. Function application is sometimes denoted by an infix dot "." and is left associative. This allows us to use the curried version of a function when it suits us, *e.g.*, we may write $f.x.y$ instead of $f(x, y)$. We use the lambda notation $\lambda x(e.x)$ to denote a function of one argument, x, whose value is $e.x$, for expression e.

$\langle Qx : r : b\rangle$ denotes a quantified expression, where Q is the quantifier, x the bound variable, r the range of x (**true** if omitted), and b the body. We sometimes write $\langle Qx \in X : r : b\rangle$ as an abbreviation for $\langle Qx : x \in X \;\wedge\; r : b\rangle$, where r is **true** if omitted, as before.

For any binary relation R: we abbreviate $\langle s, w\rangle \in R$ by sRw, we write $R(S)$ for the *image* of S under R (*i.e.*, $R(S) = \{y : \langle \exists x : x \in S : xRy\rangle\}$), and $R|_A$ denotes R *left-restricted* to the set A (*i.e.*, $R|_A = \{\langle a, b\rangle : (aRb) \;\wedge\; (a \in A)\}$). The *composition* of binary relations R and T is denoted $R; T$ or $T \circ R$, *i.e.*, $R; T = T \circ R = \{\langle r, t\rangle : \langle \exists x :: rRx \;\wedge\; xTt\rangle\}$. The *inverse* of binary relation R is denoted R^{-1} and is defined to be $\{\langle a, b\rangle : bRa\}$.

A binary relation, $B \subseteq X \times X$, is *reflexive* if $\langle \forall x \in X :: xBx\rangle$. B is *symmetric* if $\langle \forall x, y \in X :: xBy \;\Rightarrow\; yBx\rangle$. B is *antisymmetric* if $\langle \forall x, y \in X :: xBy \;\wedge\; yBx \;\Rightarrow\; x = y\rangle$. B is *transitive* if $\langle \forall x, y, z \in X :: xBy \;\wedge\; yBz \;\Rightarrow\; xBz\rangle$. A binary relation is a *preorder* if it is reflexive and transitive. A preorder that is also symmetric is an *equivalence relation*. A preorder that is antisymmetric is a *partial order*. If $\leq_1$ is a partial order on X, $\leq_2$ is a partial order on Y, and $f : X \to Y$, we say that f is *monotonic* if $a \leq_1 b \;\Rightarrow\; f.a \leq_2 f.b$ for all $a, b \in X$.

A *finite sequence* is a function from $[0..n)$ for some natural number n. An *infinite sequence* is a function from $\mathbb{N}$. When we write $x \in \sigma$, for a sequence σ, we mean that x is in the range of σ.

A *well-founded structure* is a pair $\langle W, \lessdot\rangle$ where W is a set and $\lessdot$ is a binary relation on W such that there are no infinitely decreasing sequences on W, with respect to $\lessdot$. We use $<$ to compare natural numbers and $\prec$ to compare ordinal numbers.

From highest to lowest binding power, we have: parentheses, function application, binary relations (*e.g.*, sBw), equality ($=$) and membership ($\in$), conjunction ($\wedge$) and disjunction ($\vee$), implication ($\Rightarrow$), and finally, binary equivalence ($\equiv$). Spacing is used to reinforce binding: more space indicates lower binding.

2.2 Transition Systems

Definition 1 *(Transition System)*

A *transition system* (TS) is a structure $\langle S, \dashrightarrow, L\rangle$, where S is a set of states, $\dashrightarrow \subseteq S \times S$ is the *transition relation*, L is the *labeling function*: its domain is S and it tells us what is observable at a state. We also require that $\dashrightarrow$ is *left-total*: for any $s \in S$, there is some $u \in S$ such that $s \dashrightarrow u$. Notice that a transition system is a labeled graph where the nodes are states and are labeled by L.

A *path* σ is a sequence of states such that for adjacent states s and u, $s \dashrightarrow u$. A path, σ, is a *fullpath* if it is infinite. $fp.\sigma.s$ denotes that σ is a fullpath starting at state s and σ^i denotes the suffix fullpath $\langle \sigma.i, \sigma(i+1), \ldots\rangle$. We use the symbol ";" for concatenation of paths where the left path is finite, *e.g.*, $a; ab = aab$.

Here is how to think of a program as a transition system. The set of states is the state space of the program. Two states are related by $\dashrightarrow$ if it is possible in one program step to transit from one to the next. The labeling function is the identity function.

2.3 Theorem Proving

By theorem proving we mean mechanical theorem proving. This involves defining a formal logic that is used to state and prove theorems. In addition, the proofs are checked by a computer program.

There are various theorem proving systems in use and their underlying logics vary greatly [KMM00b, KM, COR+95, GM93, CAB+86, DFH+93, CKM+91, Rud92]. For example, there are theorem provers that are based on set theory, higher-order logic, constructive type theory, first order logic, and so on. In addition, there is considerable variability in the amount of automation possible. Some of the theorem proving systems can be thought of as proof checkers, while some can perform a considerable amount of unassisted reasoning. We focus on the ACL2 system which is described in more detail in chapter 8. The ACL2 system consists of a programming language, a first-order logic, and a theorem prover.

One of the advantages of ACL2 is that it is based on a high-level language, namely Common Lisp. The ACL2 programming language is purely functional and, to a first approximation, is applicative Common Lisp. ACL2 also includes macros, which make it possible to extend the syntax and, in effect, define your own notations. This makes it very convenient to define computing systems and is an advantage ACL2 has over most of the languages used in model checking systems, which tend to impose draconian restrictions, *e.g.*, that the computing systems be finite-state. Dijkstra gives a really clear example of how the use of a high-level programming language can make it easy to write programs that were incredibly difficult to write previously ([Dij99], pages 7–8). A computing system, as modeled in ACL2, is just a function composed of other functions. We can represent a transition system $\mathcal{M}$ in ACL2 with three functions, s, r, l, where:

- s is a predicate of one argument that holds when the argument is a state in $\mathcal{M}$.
- r is a predicate of two arguments that holds when the first argument is a state that can transit to the second argument.
- l is a function of one argument that returns the label of its argument, if it is a state.

Notice that we can succinctly represent infinite-state transition systems, *e.g.*, the transition system whose states are the integers and where the only transitions are between i and $i + 1$ for all integers i, is an infinite-state transition system that can be represented in ACL2, with a few lines of code.

The ACL2 logic includes axioms that constrain the meaning of the built-in functions symbols. For example, `cons` and `car` satisfy the axiom `(car (cons` x y`))` `=` x. Since the logic is based on a programming language, it is executable. ACL2 runs on top of Common Lisp and can be compiled, usually into C [KR89] or assembly. From the logical viewpoint, function definitions introduce new axioms. Since the unconstrained introduction of axioms can render the logic unsound, ACL2 has a definitional principle which restricts the functions one can define. This principle guarantees that allowable function definitions do not introduce axioms that render the logic unsound. The logic also includes rules of inference, including an induction principle that is used to reason about recursively defined functions.

The theorem prover is used to check proofs mechanically. More precisely, ACL2 is given a proof outline and checks that the proof outline can be turned into a formal proof, without actually constructing a formal proof. ACL2 uses previously proven theorems when attempting to fill in the gaps of a proof outline and it is possible to have ACL2 automatically prove very complicated theorems by using well-designed libraries of theorems. For example, in chapter 12, we show that with a general-purpose library, ACL2 can automatically verify the correctness of a pipelined machine.

2.4 Temporal Calculi/Logics

In this section, we present variants of some of the standard temporal calculi and logics. This includes the Mu-Calculus, CTL, LTL, and CTL*. The Mu-Calculus is the most expressive, followed by CTL*. LTL and CTL are incomparable. LTL is a linear-time logic, while the others are branching-time calculi and logics. In the linear-time framework, the semantics of a transition system is the set of labeled fullpaths from some set of initial states. In contrast, in the branching-time case, the semantics of a transition system is the set of labeled computation trees from the initial states. Since distinguishable trees can give rise to the same fullpaths, branching-time logics can be more expressive than linear-time logics [Eme90].

2.4.1 Mu-Calculus

The propositional Mu-Calculus [Koz83, Par69, EC80, EL86, EJS93, Eme97] is based on fixpoint operators. We start by recalling the Tarski-Knaster theorem.

Given $f : Z \to Z$, we define $f^\alpha : Z \to Z$, the α-fold composition (iteration) of f as follows, where α ranges over On, the class of ordinals.

- $f^0(A) = A$
- $f^{\alpha+1}(A) = f(f^\alpha(A))$
- $f^\alpha(A) = \{x : \langle \exists \beta :: \langle \forall \gamma : \beta \prec \gamma \prec \alpha : x \in f^\gamma(A) \rangle \rangle \}$, when α is a limit ordinal.

If $f.x = x$ we say that x is a *fixpoint* of f. If f is a monotonic function on the powerset of a set, then by the following version of the Tarski-Knaster theorem [Tar55], it has a least and greatest fixpoint, denoted by $\mu.f$ and $\nu.f$, respectively.

Theorem 1 *Let* $f : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ *such that* $a \subseteq b \quad \Rightarrow \quad f.a \subseteq f.b$ *for all* $a, b \in S$. *Then*

1. $\mu.f \;=\; \langle \cap b : b \subseteq S \;\wedge\; f.b \subseteq b : b \rangle \;=\; \langle \cup \alpha \in On :: f^\alpha(\emptyset) \rangle$, *and*
2. $\nu.f \;=\; \langle \cup b : b \subseteq S \;\wedge\; b \subseteq f.b : b \rangle \;=\; \langle \cap \alpha \in On :: f^\alpha(S) \rangle$,

We say that x is a *pre-fixpoint* of f iff $x \subseteq f.x$; x is a *post-fixpoint* iff $f.x \subseteq x$. The Tarski-Knaster theorem tells us that $\mu.f$ is below all post-fixpoints and that $\nu.f$ is above all pre-fixpoints.

We can replace On by the set of ordinals of cardinality at most $\#S$. When the state space is finite, this gives us an algorithm for computing least and greatest fixpoints. Notice that by the monotonicity of f, $\alpha \preccurlyeq \beta \quad \Rightarrow \quad f^\alpha(\emptyset) \subseteq f^\beta(\emptyset) \quad \wedge \quad f^\beta(S) \subseteq f^\alpha(S)$. Therefore, we can compute $\mu.f$ by applying f to $\emptyset$ until we reach a fixpoint; similarly, we can compute $\nu.f$ by applying f to S until we reach a fixpoint.

We now describe the Mu-Calculus. We are really describing a class of calculi parameterized by a set of variables, VAR, and a language for denoting predicates on labels. The predicate denoted by expression e is written $(\!|e|\!)$. The formulas of the Mu-Calculus are defined recursively as follows.

1. e, where e is an expression;

2. Y, where Y is a variable (*i.e.*, $Y \in \mathit{VAR}$);

3. $\neg f$ and $f \vee g$, where f, g are formulas;

4. $\mathsf{EX} f$, where f is a formula;

5. $\mu Y f$ and $\nu Y f$, where Y is a variable and f is a formula that is monotone in Y, *i.e.*, occurrences of Y in f are under an even number of negations.

Notice that μ and ν are now overloaded, but one can easily distinguish syntactically whether we refer to the fixpoint operator or the Mu-Calculus operator symbol. The semantics of Mu-Calculus formula f is given with respect to a transition system $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ and a valuation $\mathcal{V}$, where $\mathcal{V} : \mathit{VAR} \rightarrow \mathcal{P}(S)$, *i.e.*, $\mathcal{V}$ assigns a subset of S to each variable in VAR. It is denoted $[\![f]\!]_{\mathcal{V}}^{\mathcal{M}}$ and is the set of states in $\mathcal{M}$ satisfying f under valuation $\mathcal{V}$. The semantics of the Mu-Calculus is defined as follows.

1. $[\![e]\!]_{\mathcal{V}}^{\mathcal{M}} \;=\; \{s \in S : L.s \in (\!|e|\!)\}$

2. $[\![Y]\!]_{\mathcal{V}}^{\mathcal{M}} \;=\; \mathcal{V}.Y$

3. $[\![\neg f]\!]_{\mathcal{V}}^{\mathcal{M}} \;=\; S \setminus [\![f]\!]_{\mathcal{V}}^{\mathcal{M}}$
   $[\![f \vee g]\!]_{\mathcal{V}}^{\mathcal{M}} \;=\; [\![f]\!]_{\mathcal{V}}^{\mathcal{M}} \cup [\![g]\!]_{\mathcal{V}}^{\mathcal{M}}$

4. $[\![\mathsf{EX} f]\!]_{\mathcal{V}}^{\mathcal{M}} \;=\; \{s \in S : \langle \exists u \in S :: s \dashrightarrow u \;\;\wedge\;\; u \in [\![f]\!]_{\mathcal{V}}^{\mathcal{M}}\rangle\}$

5. $[\![\mu Y f]\!]^{\mathcal{M}}_{\mathcal{V}} \;=\; \mu.g$ where $g.y = [\![f]\!]^{\mathcal{M}}_{\mathcal{W}}$ and $\mathcal{W} = \mathcal{V}$ except $\mathcal{W}.Y = y$

   $[\![\nu Y f]\!]^{\mathcal{M}}_{\mathcal{V}} \;=\; \nu.g$ where $g.y = [\![f]\!]^{\mathcal{M}}_{\mathcal{W}}$ and $\mathcal{W} = \mathcal{V}$ except $\mathcal{W}.Y = y$

Notice that by the syntactic monotonicity restrictions we place on μ's and ν's, the semantics of μ's and ν's is well defined and correspond to least and greatest fixpoints, respectively. Note also that the semantics of a *sentence* (a formula with no free variables) does not depend on the initial valuation. For f a sentence, by $\mathcal{M}, s \models f$ we mean $s \in [\![f]\!]^{\mathcal{M}}_{\emptyset}$.

2.4.2 Temporal Logic

LTL formulas are formed from expressions (denoting predicates on labels, as above), boolean connectives and the temporal operators X (next time) and U (until). LTL formulas define sets of infinite sequences. CTL* adds the universal and existential branching operators A and E to the LTL syntax. CTL is formed similarly with the restriction that each LTL temporal operator appear paired with its own path quantifier. CTL* and CTL formulas define sets of infinite depth trees.

Most presentations of CTL* define the syntax using mutual recursion, *e.g.*, this is done in the Handbook of Theoretical Computer Science [Eme90]. We present an alternate formulation which does not require mutual recursion. We believe this approach provides a more concise and clear presentation.

The syntax of CTL* formulas follows.

1. e, where e is an expression;

2. $f \wedge g$ and $\neg f$, where f, g are formulas;

3. $\mathsf{E}f, \mathsf{X}f, f\mathsf{U}g$, where f, g are formulas.

The semantics of CTL* is given with respect to $\mathcal{M}$, a transition system and σ, a fullpath of $\mathcal{M}$. For temporal logic formula f, $\mathcal{M}, \sigma \models f$ denotes that f holds on fullpath σ of $\mathcal{M}$.

1. $\mathcal{M}, \sigma \models e$ iff $L(\sigma.0) \in (\!|e|\!)$;

2. $\mathcal{M}, \sigma \models f \wedge g$ iff $\mathcal{M}, \sigma \models f$ and $\mathcal{M}, \sigma \models g$,
   $\mathcal{M}, \sigma \models \neg f$ iff it is not the case that $\mathcal{M}, \sigma \models f$;

3. $\mathcal{M}, \sigma \models \mathsf{E}f$ iff there exists a fullpath $\delta = \langle \sigma.0, ... \rangle$ in $\mathcal{M}$ such that
   $$\mathcal{M}, \delta \models f,$$
   $\mathcal{M}, \sigma \models \mathsf{X}f$ iff $\mathcal{M}, \sigma^1 \models f$,
   $\mathcal{M}, \sigma \models f \mathsf{U} g$ iff there exists $i \in \mathbb{N}$ such that $\mathcal{M}, \sigma^i \models g$
   $$\text{and for all } j < i,\ \mathcal{M}, \sigma^j \models f.$$

We introduce the following useful operators. $\mathsf{A}f$ abbreviates $\neg\mathsf{E}\neg f$; $\mathcal{M}, \sigma \models \mathsf{A}f$ if f holds on all fullpaths from $\sigma.0$. $\mathsf{F}g$ abbreviates $\mathbf{true}\mathsf{U}g$; $\mathcal{M}, \sigma \models \mathsf{F}g$ if eventually g holds along σ. $\mathsf{G}f$ abbreviates $\neg\mathsf{F}\neg f$; $\mathcal{M}, \sigma \models \mathsf{G}f$ if f always holds along σ.

Sometimes a formula depends only on the first state of a fullpath, *e.g.*, this is the case with the formulas $\mathsf{A}f$ and $\mathsf{E}f$. In such cases, we often write $\mathcal{M}, s \models f$ instead of $\mathcal{M}, \sigma \models f$, where $s = \sigma.0$. Such formulas are called *state formulas*.

We now define various sublogics of CTL*. Since the languages of the sublogics are subsets of the language of CTL*, the semantics of the logics has already been given.

The language of LTL is the set of CTL* formulas that start with an A path quantifier and contain no other path quantifiers.

The syntax of CTL can be defined by replacing item 3 in the syntax of CTL* by the following.

3. if f, g are formulas then so are $\mathsf{EX}f, \mathsf{E}(f\mathsf{U}g), \mathsf{E}\neg(f\mathsf{U}g)$.

ECTL* is defined to be CTL*, but with negations applied only to predicates. ECTL is be defined be CTL, but with negations applied only to predicates. ACTL* and ACTL are similar to ECTL* and ECTL, but instead of the operator E, only the operator A is allowed. CTL$^* \setminus \mathsf{X}$, CTL $\setminus \mathsf{X}$, ECTL$^* \setminus \mathsf{X}$, ECTL $\setminus \mathsf{X}$, ACTL$^* \setminus \mathsf{X}$, and ACTL $\setminus \mathsf{X}$ are obtained from CTL*, CTL, ECTL*, ECTL, ACTL*, and ACTL, respectively, by dropping the X operator.

2.5 Model Checking

Model checking algorithms are used to decide if a finite-state system satisfies a temporal formula [CE81, Eme81, QS82]. Model checking is covered in detail in chapter 9; we briefly mention some of the key points here. Many temporal logics, *e.g.*, CTL, LTL, and CTL* can be translated into the Mu-Calculus. In addition, the algorithm that decides the Mu-Calculus is used for symbolic (BDD-based) model checking [CBM89, Pix90, McM93, BCM+92, TSL+90], a technique that has greatly extended the applicability of model checking. Model checking is especially useful for verifying *reactive systems*, systems with nonterminating or concurrent behavior [Pnu77]. Such systems are especially difficult to design and verify. Model checking has been successfully applied to automatically verify many reactive systems and is now being used by hardware companies as part of their verification process.

2.6 Simulation and Bisimulation

R is a *simulation relation* [Mil71] on TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ if $R \subseteq S \times S$ and for s, w such that sRw we have the following.

1. $L.s = L.w$
2. $\langle \forall u : s \dashrightarrow u : \langle \exists v :: w \dashrightarrow v \quad \wedge \quad uRv\rangle\rangle$

B is a *bisimulation relation* [Par81, Mil90] on TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ if $B \subseteq S \times S$ and for s, w such that sBw we have the following.

1. $L.s = L.w$
2. $\langle \forall u : s \dashrightarrow u : \langle \exists v :: w \dashrightarrow v \quad \wedge \quad uBv\rangle\rangle$
3. $\langle \forall v : w \dashrightarrow v : \langle \exists u :: s \dashrightarrow u \quad \wedge \quad uBv\rangle\rangle$

We say that s is *similar* to w if there exists a simulation relation R such that sRw. Similarly we say that s is *bisimilar* to w if there exists a bisimulation relation B such that sBw. It is not the case that if s is similar to w and w is similar to s, then s and w are bisimilar [Pnu85]. If s and w are bisimilar, then they satisfy the same Mu-Calculus formulas. Hence, they satisfy the same CTL*, CTL, and LTL formulas. If s is similar to w then any ACTL* formula (and hence any LTL formula as well) that holds in w also holds in s. There exists a greatest bisimulation and a greatest simulation. The greatest bisimulation is an equivalence relation and the greatest simulation is a preorder.

2.7 Bibliographic Notes

Temporal logic was proposed as a formalism for specifying the correctness of computing systems in a landmark paper by Pnueli [Pnu77]. There are a few differences between our presentation of temporal logic and the standard presentation [Eme90]. First, in our presentation there is only one type of formula, but in the standard presentation of temporal logic there are two types of formulas. There are state formulas and path formulas and they are defined using mutual recursion. This makes the definitions more difficult to understand and the proofs more tedious than is the case with our approach. Second, in the standard presentation, instead of building formulas from expressions denoting predicates on the labels, formulas are built out of atomic propositional constants which denote predicates on the labels. The reason we use expressions is that we define a Mu-Calculus model checker using ACL2 in a later chapter and we want the ability to write temporal logic formulas using the full power of ACL2.

The most efficient algorithm for deciding bisimulation equivalence has time complexity $O(m \log n)$, where n is the number of states and m the number of transitions, and is due to Paige and Tarjan [PT87]. The best known algorithm for checking simulations has time complexity $O(m \cdot n)$ and is due to Henzinger, Henzinger, and Kopke [BP95, HHK95]. In contrast, Stockmeyer and Meyer show that trace equivalence and trace containment are both PSPACE-complete problems [SM73]. One way of understanding this difference is that both simulation and bisimulation are local properties: they can be checked by comparing related states and their successors. In contrast trace containment and trace equivalence are global properties: one has to examine paths through the transition system. We discuss this issue in more detail in part II, especially in section 5.6, on page 66. A very readable discussion of the computational complexity of bisimulation is given by Moller and Smolka [MS95]. Pnueli gives a balanced comparison between the branching-time and linear-time approaches in [Pnu85].

Bisimulation has proved useful even in the foundations of mathematics. Peter Aczel uses bisimulation to develop a theory of non-well-founded sets [Acz88]. This theory differs from ZFC in that the axiom of foundation is replaced by the "Anti-Foundation Axiom", which allows sets that are not well-founded. Non-well-founded sets can be thought of as graphs. Two graphs are the same if they are bisimilar. Non-well-founded set theory even appears in an introductory text on set theory by Devlin [Dev92].

2.8 Summary

In this chapter we presented our notational conventions and gave an overview of transitions systems, theorem proving, temporal calculi and logics, model checking, and simulation and bisimulation.

Part II

Notions of Correctness

Chapter 3

Introduction

In this part we examine two notions of correctness for transition systems. We say that an implementation is correct with respect to a notion of correctness and a specification, if the implementation is related to the specification as prescribed by the notion of correctness. A step of the specification might translate to many steps of the implementation. Since we do not want to limit the kinds of implementations that are considered, stuttering steps should be ignored. In fact, both notions of correctness that we consider are insensitive to stuttering.

Our notions of correctness are based on simulation [Mil71] and bisimulation [Par81, Mil90] and are therefore branching-time notions of correctness. This is in contrast to linear-time notions such as trace containment, trace equivalence, and trace congruence [Pnu85]. There are two reasons why we chose to work in the branching-time framework. First, as was mentioned on page 24, there are polynomial time algorithms for deciding simulation and bisimulation, whereas the corresponding problems for linear time are PSPACE-complete. This is important later when we show how to reduce large (perhaps infinite-state) systems to finite-state systems that we then compare against specifications using bisimulation algorithms; thus, efficiency is important. Second, when we prove refinement theorems about infinite-state systems, we use inductive arguments which depend on the structure of the systems in question. The branching-time notions are structural and local; this leads to simple proof rules. In the linear time case, proof rules are more complicated, *e.g.*, it is sometimes necessary to introduce so-called "prophecy" variables. These issues are discussed more fully in section 5.6.

The first notion we consider is stuttering simulation. With this notion of correctness, an implementation is correct if every one of its computations is a computation of the specification, up to stuttering. Thus, the implementation can resolve some of the non-determinism in the specification. The second notion is stuttering bisimulation [BCG88]. With this notion of correctness, an implementation is correct if it has exactly the same computations as the specification, up to stuttering.

We develop the theory of stuttering simulation and bisimulation in order to simplify mechanical verification. We show that these notions satisfy various algebraic properties, *e.g.*, the relational composition of two stuttering simulations is a stuttering simulation. We also examine proof rules, similar to those of Namjoshi [Nam97]. The proof rules are sound and complete and have the technical advantage that they allow us to prove stuttering simulations and bisimulations by reasoning about single steps of the systems in question. (Proving stuttering simulations and bisimulations directly requires reasoning about infinite computations.) Our proof rule for stuttering bisimulation is more general and simpler than the one by Namjoshi and sheds further light on the structure of stuttering bisimulations. This has helped us construct ACL2 libraries to automate mechanically checked proofs. We also develop a theory of refinement based on these notions and present compositional proof rules for showing stuttering simulation and bisimulation in stages.

In the next section of this chapter, we define what it means for two fullpaths to "match". This notion of matching is fundamental to the development of the theory in the next two chapters. The next two chapters are devoted to stuttering simulation and stuttering bisimulation. At the end of the stuttering bisimulation chapter, we discuss various issues. For example, we compare our notion of refinement with that of Abadi and Lamport [AL91] and we discuss reasoning about performance.

3.1 Matching Fullpaths

Stuttering simulation and bisimulation depend on the notion of matching we now define. We start with an informal account. We are given a relation B on a set S. We say that an infinite sequence σ (of elements from S) matches an infinite sequence δ (of elements from S) if the sequences can be partitioned into non-empty, finite segments such that elements in related segments are related by B. For example, if the first segment of σ has three elements and the first segment of δ has seven elements, then each of the three elements is related by B to each of the seven elements. We use matching, where the infinite sequences are fullpaths of a transition system, to define stuttering simulation and bisimulation.

Definition 2 *(match)*

Let i range over $\mathbb{N}$. Let *INC* be the set of strictly increasing sequences of natural numbers starting at 0; formally, $INC = \{\pi : \pi : \mathbb{N} \to \mathbb{N} \quad \wedge \quad \pi.0 = 0 \ \wedge \ \langle \forall i \in \mathbb{N} :: \pi.i < \pi(i{+}1) \rangle \}$. The i^{th} segment of an infinite sequence σ with respect to $\pi \in INC$, ${}^{\pi}\sigma^{i}$, is given by the sequence $\langle \sigma(\pi.i), \ldots , \sigma(\pi(i{+}1) - 1) \rangle$.

For $B \subseteq S \times S$, $\pi, \xi \in INC$, $i, j \in \mathbb{N}$, and infinite sequences σ and δ, we abbreviate $\langle \forall s, w : s \in {}^{\pi}\sigma^{i} \ \wedge w \in {}^{\xi}\delta^{j} : sBw \rangle$ by $({}^{\pi}\sigma^{i})B({}^{\xi}\delta^{j})$. In addition:

$corr(B, \sigma, \pi, \delta, \xi) \equiv \langle \forall i \in \mathbb{N} :: ({}^{\pi}\sigma^{i})B({}^{\xi}\delta^{i}) \rangle$

and

$match(B, \sigma, \delta) \ \equiv \ \langle \exists \pi, \xi \in INC :: corr(B, \sigma, \pi, \delta, \xi) \rangle$

Lemma 1 *Given set S, $B \subseteq S \times S$, and infinite sequences σ and δ,*

$$\langle \exists \pi, \xi \in INC :: corr(B, \sigma, \pi, \delta, \xi) \rangle$$

$$\equiv$$

$$\langle \exists \pi', \xi' \in INC :: corr(B, \sigma, \pi', \delta, \xi') \ \wedge \ \langle \forall i \in \mathbb{N} :: \#({}^{\pi}\sigma^{i}) = 1 \ \vee \ \#({}^{\xi}\delta^{i}) = 1 \rangle \rangle$$

Proof The $\Leftarrow$ direction is clear. For the other case, π', ξ' are just refinements of π, ξ. Suppose $\#({}^{\pi}\sigma^{i}) = n > 1$ and $\#({}^{\xi}\delta^{i}) = m > 1$. If $n \geq m$ we subdivide ${}^{\pi}\sigma^{i}$ into m segments where the first $m - 1$ segments have 1 element and the last segment has $n - m + 1$ elements; ${}^{\xi}\delta^{i}$ is subdivided into m segments, each consisting of 1 element. As $({}^{\pi}\sigma^{i})B({}^{\xi}\delta^{i})$, each of the refined segments matches its corresponding segment and all of the δ segments are of length 1. If $m > n$ we proceed similarly. π', ξ' have the desired properties. $\square$

The above lemma allows us to reason about segments using case analysis, where the three cases are: both segments are of length 1, the right segment is of length 1 and the left of length greater than 1, and the left segment is of length 1 and the right of length greater than 1. As will be seen shortly, the definitions of well-founded simulation and well-founded bisimulation are to some extent based on these three cases. We henceforth assume that all segments are of this form, as this assumption simplifies some of the arguments.

Chapter 4

Stuttering Simulation

4.1 Stuttering Simulation

A relation on $B \subseteq S \times S$ where $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ is a stuttering simulation, if for any s, w such that sBw, s and w are identically labeled and any fullpath starting at s can be matched by some fullpath starting at w.

Definition 3 *(Stuttering Simulation (STS))*

$B \subseteq S \times S$ is a stuttering simulation on TS $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ iff for all s, w such that sBw:

(Sts1) $\quad L.s = L.w$

(Sts2) $\quad \langle \forall \sigma : \mathit{fp}.\sigma.s : \langle \exists \delta : \mathit{fp}.\delta.w : \mathit{match}(B, \sigma, \delta) \rangle \rangle$

We show that there is a greatest STS. This is accomplished by showing that an arbitrary union of STS's is an STS, hence, there is a greatest STS. We then show that the identity relation is an STS and if A and B are STS's so is $A; B$—the composition of A and B. Together, the above results imply that the reflexive, transitive closure of an STS is an STS and that the greatest STS is a preorder.

Lemma 2 $(B \subseteq C) \quad \Rightarrow \quad [match(B, \sigma, \delta) \quad \Rightarrow \quad match(C, \sigma, \delta)]$

Proof

$$match(C, \sigma, \delta)$$

$\equiv$ { Definition of *match*, *corr* }

$$\langle \exists \pi, \xi \in INC :: \langle \forall i \in \mathbb{N} :: ({}^{\pi}\sigma^{i}) C ({}^{\xi}\delta^{i}) \rangle \rangle$$

$\Leftarrow$ { $B \subseteq C$, monotonicity }

$$\langle \exists \pi, \xi \in INC :: \langle \forall i \in \mathbb{N} :: ({}^{\pi}\sigma^{i}) B ({}^{\xi}\delta^{i}) \rangle \rangle$$

$\equiv$ { Definition of *match*, *corr* }

$$match(B, \sigma, \delta) \ \Box$$

Lemma 3 *Let $\mathcal{C}$ be a set of STS's on TS $\mathcal{M}$, then $G = \langle \cup B : B \in \mathcal{C} : B \rangle$ is an STS on $\mathcal{M}$.*

Proof If sGw, sBw for some $B \in \mathcal{C}$. $L.s = L.w$ since sBw and Sts1. Our remaining proof obligation follows.

$$\langle \forall \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : match(G, \sigma, \delta) \rangle \rangle$$

$\Leftarrow$ { $B \subseteq G$, lemma 2, monotonicity }

$$\langle \forall \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : match(B, \sigma, \delta) \rangle \rangle \ \Box$$

Corollary 1 *For any TS $\mathcal{M}$, there is a greatest STS on $\mathcal{M}$.*

Proof Let $G = \langle \cup B : B \in \mathcal{C} : B \rangle$ where $C = \{B : B \text{ is an STS on } \mathcal{M}\}$. G is an STS by lemma 3, hence, the greatest. $\Box$

Lemma 4 *If R and S are STS's, so is $T = R; S$.*

Proof Let aTb. By definition of T, there is an x such that aRx and xSb. We show Sts1 and Sts2:

1. $L.a = L.b$ by transitivity of equality.

2. We must show $\langle \forall \sigma : fp.\sigma.a : \langle \exists \delta : fp.\delta.b : match(T, \sigma, \delta) \rangle \rangle$. Since R is an STS and aRx, there exist γ, π, ξ such that $fp.\gamma.x$ and $corr(R, \sigma, \pi, \gamma, \xi)$, *i.e.*, $match(R, \sigma, \gamma)$. Similarly, since S is an STS and xSb, there exist δ, ζ, ρ such that $fp.\delta.b$ and $corr(S, \gamma, \zeta, \delta, \rho)$, *i.e.*, $match(S, \gamma, \delta)$. We show $match(T, \sigma, \delta)$, thereby proving Sts2. To do this, we must relate states in σ to states in δ.

   Here is an outline of the proof. In step (a), we show that if a state in σ is related by T to a state in δ, then all the states in the corresponding segments are also related. We then work at the segment level and show how to merge segments in σ and δ to get new segments that match under T. We start in step (b) by defining a function C from segments in σ to sets of segments in δ that can be thought of as performing the merging in δ. In step (c), we use C to define the actual partitions on σ and δ, which are then shown to match under T.

   (a) $L : \mathbb{N} \to \mathbb{N}$ (Lower) and $U : \mathbb{N} \to \mathbb{N}$ (Upper) are defined as follows.

      - $L.i = j$ such that $\xi.i \in [\zeta.j, \zeta(j+1))$
      - $U.i = j$ such that $\xi(i+1) - 1 \in [\zeta.j, \zeta(j+1))$

Note that $\xi.i$ is the index of the first element in ${}^{\xi}\gamma^{i}$ and $\xi(i+1)-1$ is the index of the last element in ${}^{\xi}\gamma^{i}$. These elements are each in some segment of γ with respect to ζ. We show that any element in ${}^{\pi}\sigma^{i}$ is related to any element in ${}^{\rho}\delta^{k}$ for $k \in [L.i, U.i]$, *i.e.*, $\langle \forall i, k : k \in [L.i, U.i] : ({}^{\pi}\sigma^{i})T({}^{\rho}\delta^{k})\rangle$. To see this, let $k \in [L.i, U.i], t \in {}^{\rho}\delta^{k}$. By $corr(S, \gamma, \zeta, \delta, \rho)$, for all $w \in {}^{\zeta}\gamma^{k}$, we have wSt. Since ${}^{\xi}\gamma^{i}$ intersects ${}^{\zeta}\gamma^{k}$—by definition of U and L—there is $u \in {}^{\zeta}\gamma^{k}$ such that $u \in {}^{\xi}\gamma^{i}$ and uSt. Since $corr(R, \sigma, \pi, \gamma, \xi)$, for all $s \in {}^{\pi}\sigma^{i}, sRu$, and by definition of T, $\langle \forall s \in {}^{\pi}\sigma^{i} :: sTt\rangle$.

(b) We define C (corresponds), a function that identifies which segments of δ under ρ correspond to ${}^{\pi}\sigma^{i}$. We define $C(-1)$ to be $\{-1\}$.

- $C.i = \{U.i\}$ if $U.i \in C(i-1)$ or $U.i = max.C(i-1)+1$, otherwise
- if $U.i = U(i+1), C.i = (max.C(i-1), U.i)$, otherwise
- $C.i = (max.C(i-1), U.i]$

(c) To prove $match(T, \sigma, \delta)$ we will exhibit an explicit partition. We do this by defining τ and ϕ as follows.

- $\tau.0 = 0$
- $\tau(i+1) = \tau.i + j$ where $j = min.\{k \in \mathbb{N} : C(\tau.i + k) \neq C(\tau.i)\}$
- $\phi.i = min.C(\tau.i)$

The idea is that τ removes duplicate $C.i$'s (duplicate $C.i$'s indicate that neighboring segments of σ under π should be merged into one segment). We now prove $corr(T, \sigma, \pi \circ \tau, \delta, \rho \circ \phi)$ that expanded is $\langle \forall i, m, n : i \in \mathbb{N} \wedge m \in [\tau.i, \tau(i+1)) \wedge n \in [\phi.i, \phi(i+1)) :$

$({}^{\pi}\sigma^m)T({}^{\rho}\delta^n)\rangle$. Note $m \in [\tau.i, \tau(i+1)) \Rightarrow C.m = C(\tau.i)$ because τ collapses similar $C.i$'s; hence, $({}^{\pi}\sigma^m)T({}^{\rho}\delta^j)$ for any $j \in C(\tau.i)$ since $j \in [L(\tau.i), U(\tau.i)]$, but $n \in [\phi.i, \phi(i+1)) \equiv n \in [min.C(\tau.i), min(C.\tau(i+1))) \equiv n \in C(\tau.i)$, hence, $({}^{\pi}\sigma^m)T({}^{\rho}\delta^n)$. $\square$

Lemma 5 *The reflexive, transitive closure of an STS is an STS.*

Proof The reflexive, transitive closure of B, denoted B^*, can be written as $\langle \cup i \in \mathbb{N} :: B^i \rangle$. The identity relation, B^0, is an STS and by induction on the natural numbers, using lemma 4, so is B^i for any $i > 0$. By lemma 3 so is $\langle \cup i \in \mathbb{N} :: B^i \rangle$. $\square$

Theorem 2 *Given TS $\mathcal{M}$, there is a greatest STS on $\mathcal{M}$, which is a preorder.*

Proof By corollary 1, there is a greatest STS G on $\mathcal{M}$. By lemma 5, G^* is also an STS. Now, $G^* \supseteq G$, by definition of G^* and $G^* \subseteq G$ since G is the greatest STS. Thus, $G = G^*$. $\square$

STSs are not closed under intersection or negation, as the following lemma shows.

Lemma 6 *STSs are not closed under intersection or negation.*

Proof Consider a TS with two states with different labels. The identity relation is an STS, but its negation is not. An example showing that STSs are not closed under intersection appears in figure 4.1.

Theorem 3 *Let B be a STS on $\mathcal{M}$ and let sBw. For any* $\mathrm{ACTL}^* \setminus \mathsf{X}$ *formula f, if $\mathcal{M}, w \models f$ then $\mathcal{M}, s \models f$.*

Proof The proof is by induction on the structure of $\mathrm{ACTL}^* \setminus \mathsf{X}$ formulas and is similar to a proof given by Browne, Clarke, and Grumberg [BCG88]. $\square$

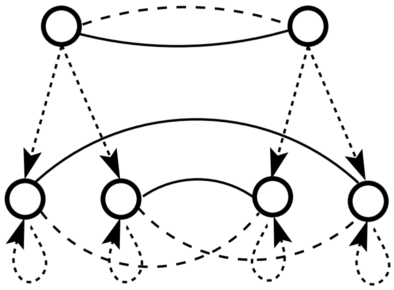

Figure 4.1: An example showing that STSs are not closed under intersection. The transition relation is denoted by a dashed arrow. There are two stuttering simulation relations denoted by a dashed line and a solid line. Any two states in the same strongly connected component induced by the dashed line are in the first stuttering simulation relation. Similarly, any two states in the same strongly connected component induced by the solid line are in the second stuttering simulation relation. Only the top two states are related in the intersection, but they do not have matching children.

4.2 Well-Founded Simulation

In order to check that a relation is an STS, we have to show that infinite sequences "match". This can be problematic when using computer-aided verification techniques. We present the notion of a *well-founded simulation* to remedy this situation. To show that a relation is a well-founded simulation, we need only check local properties; this is analogous to proving program termination by exhibiting a function that maps states into a well-founded relation and showing that the function decreases during every step of the program. As mentioned previously, the intuition is that for any pair of states s, w that are related by an STS and u such that $s \dashrightarrow u$, there are essentially three cases: either there is a v such that $w \dashrightarrow v$ and u is related to v, or u is related to w, or there is a v such that $w \dashrightarrow v$ and s is related to v. In the last two cases, we must also ensure that we do not have an infinite sequence of states, each of which is related to a single state. This is where the well-founded relation comes in: we must show that in these cases there is an appropriate ranking function into a well-founded relation that decreases. Formally, we have:

Definition 4 *(Well-Founded Simulation (WFS)) $B \subseteq S \times S$ is a well-founded simulation on TS $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ iff:*

(Wfs1) $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$

(Wfs2) There exists functions, $\mathit{rankt} : S \times S \rightarrow W$, $\mathit{rankl} : S \times S \times S \rightarrow \mathbb{N}$, such that $\langle W, \lessdot \rangle$ is well-founded, and

$\langle \forall s, u, w \in S : sBw \quad \wedge \quad s \dashrightarrow u :$

(a) $\langle \exists v : w \dashrightarrow v : uBv \rangle \quad \vee$

(b) $(uBw \quad \wedge \quad \mathit{rankt}(u, w) \lessdot \mathit{rankt}(s, w)) \quad \vee$

(c) $\langle \exists v : w \dashrightarrow v : sBv \quad \wedge \quad \mathit{rankl}(v, s, u) < \mathit{rankl}(w, s, u) \rangle \rangle$

4.3 Equivalence

In this section, we show that well-founded simulation completely characterizes stuttering simulation. Thus, we can think of well-founded simulation as a sound and complete proof rule.

In section 4.3.1 we prove that any WFS is an STS. The proof proceeds by showing that given a WFS and a fullpath from state a, where aBb, we can construct a fullpath from b that matches the fullpath from a.

In section 4.3.2 we prove that any STS is a WFS. For the proof, we have to exhibit the rank functions as per the definition of WFS. Here is a high-level overview.

The value of $rankt(s, w)$ is important only if sBw, as otherwise there are no restrictions required by the definition of WFS. If sBw, then consider the largest subtree of the computation tree rooted at s such that no node in the subtree matches a successor of w. The "rank" (a kind of height) of this subtree is the value of $rankt(s, w)$. The "rank" of s is greater than the "rank" of any of its children in the tree, so case Wfs2b is satisfied.

The value of $rankl(w, s, u)$ is important only if sBw and $s \dashrightarrow u$, as otherwise there are no restrictions required by the definition of WFS. If sBw and $s \dashrightarrow u$, then $rankl(w, s, u)$ is the length of the shortest path from w that matches s, u. In the case of Wfs2c, we can choose the next successor of w in this path to satisfy the condition.

4.3.1 Soundness

Proposition 1 *(Soundness) If B is a WFS, then it is an STS.*

Proof Let aBb; we need to show Sts1 and Sts2:

1. $L.a = L.b$ since B is a WFS (Wfs1).

2. $\langle \forall \sigma : fp.\sigma.a : \langle \exists \delta : fp.\delta.b : match(B, \sigma, \delta) \rangle \rangle$. Suppose $fp.\sigma.a$. We define fullpath δ and increasing sequences π, ξ recursively as follows:

   $\delta.0 = b, \pi.0 = 0, \xi.0 = 0$. We write σ^i, δ^i instead of ${}^{\pi}\sigma^i$, ${}^{\xi}\delta^i$, respectively. Also, to make the following more readable, let $s = \sigma(\pi.i), u = \sigma(\pi.i+1)$, and $w = \delta(\xi.i)$. The idea is that from $\pi.i, \xi.i, \delta(\xi.i)$ we can define $\pi(i+1), \xi(i+1), \delta^i, \delta.\xi(i+1)$ with σ^i, δ^i matching. There are three cases:

   (a) Wfs2a holds, *i.e.*, $\langle \exists v : w \dashrightarrow v : uBv \rangle$. Let $\pi(i+1) = \pi.i+1, \xi(i+1) = \xi.i+1, \delta^i = \langle w \rangle, \delta.\xi(i+1) = v$.

(b) Wfs2a does not hold, but Wfs2b does, *i.e.*, $\neg\langle \exists v : w \dashrightarrow v : uBv\rangle \ \wedge\ uBw \ \wedge\ \mathit{rankt}(u,w) \lessdot \mathit{rankt}(s,w)$. Let n be the maximum number such that $\langle \forall l : 1 \leq l \leq n : \sigma(\pi.i+l)Bw \quad \wedge \quad \neg\langle \exists v : w \dashrightarrow v : \sigma(\pi.i+l)Bv\rangle\rangle$. Note $n \geq 1$ since the above holds for $u = \sigma(\pi.i+1)$. Also, by the well-foundedness of *rankt*, n must be finite.

Wfs2 has to hold between $\sigma(\pi.i+n), w, \sigma(\pi.i+n) \dashrightarrow \sigma(\pi.i+n+1)$. By the maximality of n, we have the following two cases:

i. If $\langle \exists v : w \dashrightarrow v : \sigma(\pi.i+n+1)Bv\rangle$, then $\sigma(\pi.i+n+1), v$ mark the beginning of the $(i+1)^{th}$ segments and $\pi(i+1) = \pi.i+n+1, \xi(i+1) = \xi.i+1, \delta^i = \langle w\rangle, \delta.\xi(i+1) = v$.

ii. Otherwise, $\neg(\sigma(\pi.i+n+1)Bw)$, but now Wfs2b cannot hold, but neither can Wfs2c, because if it does $\langle \exists v : w \dashrightarrow v : \sigma(\pi.i+n)Bv\rangle$, but by definition of n, this cannot be; hence, Wfs2a must hold, *i.e.*, $\langle \exists v : w \dashrightarrow v : \sigma(\pi.i+n+1)Bv\rangle$, which is our first case.

(c) Only Wfs2c holds. Let $\vec{v} = \langle v.0 = w, v.1 = v, \ldots, v.n\rangle$ be a maximal (with respect to prefix order) sequence such that $\langle \forall l : 0 \leq l < n : v.l \dashrightarrow v(l+1)\rangle$ and only Wfs2c holds for $s, v.l, s \dashrightarrow u$ for all $l < n$. Such a sequence must contain $v.0 = w$ and $v.1 = v$, so $n \geq 1$; in addition, the sequence is finite by the well-foundedness of *rankl*. Since $sBv.n$, Wfs2 must hold between $s, v.n, s \dashrightarrow u$, specifically either Wfs2a or Wfs2b (by maximality of $\vec{v}$), but Wfs2b cannot hold because if it did we would have $uBv.n$, but then Wfs2a would hold between $s, v(n-1), s \dashrightarrow u$; hence, Wfs2a must hold between $s, v.n, s \dashrightarrow u$, *i.e.*, there exists an x such that $v.n \dashrightarrow x$

and uBx. Let $\pi(i+1) = \pi.i + 1, \xi(i+1) = \xi.i + n + 1, \delta^i = \langle v.0, \ldots, v.n \rangle, \delta.\xi(i+1) = x$; hence, u, x mark the beginning of the $(i+1)^{th}$ segment. $\square$

4.3.2 Completeness

Given a TS $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$, the notion of the *computation tree* rooted at a state $s \in S$ is standard. It is the tree obtained by unfolding $\mathcal{M}$ starting from s and can be defined as follows. The nodes of the tree are finite sequences over S. The tree is defined to be the smallest tree satisfying the following.

1. The root is $\langle s \rangle$.

2. If $\langle s, \ldots, w \rangle$ is a node and $w \dashrightarrow v$, then $\langle s, \ldots, w, v \rangle$ is a node whose parent is $\langle s, \ldots, w \rangle$.

Definition 5 *(tree) Given an STS B, if $\neg(sBw)$, then tree(s,w) is the empty tree, otherwise tree(s,w) is the largest subtree of the computation tree rooted at s such that for any non-root node of the tree, $\langle s, \ldots, x \rangle$, we have that xBw and $\langle \forall v : w \dashrightarrow v : \neg(xBv) \rangle$.*

Lemma 7 *Every path of tree(s,w) is finite.*

Proof Suppose not, then there exists a path σ such that *fp*.$\sigma.s$. Consider σ', the infinite sequence starting at $\sigma.1$, but otherwise identical to σ. We have $(\sigma'.0)Bw$, so there exists δ such that *match*(B, σ', δ). By construction, $\langle \forall v : w \dashrightarrow v : \neg((\sigma'.1)Bv) \rangle$, hence, $\delta.1$ marks the beginning of the second segment in δ. Say $\sigma'.j$ marks the beginning of the second segment of σ', then $(\sigma'.j)B(\delta.1)$, but by construction of *tree*(s,w) this is impossible. $\square$

Since the child relation on nodes in $tree.s$ is well-founded, we can recursively define a labeling function, l, that assigns an ordinal to nodes in the tree as follows: $l.n = \langle \cup c : c \text{ is a child of } n : (l.c) + 1 \rangle$. This is the standard "rank" function encountered in set theory [Kun80]. We use the convention that the label of a tree is the label of its root.

Lemma 8 *If $\#S \preceq \kappa$, where κ is an infinite cardinal (*i.e.*, $\omega \preceq \kappa$) then for all $s, w \in S$, $tree(s, w)$ is labeled with an ordinal of cardinality $\preceq \kappa$.*

Proof The i^{th} level of the tree consists of all nodes of distance i from the root. Since paths are finite, i is a natural number. We can show by induction on $\mathbb{N}$ that for all i, the number of nodes at level i is at most κ. At level 0 it is easy to see as there is only 1 node. Suppose level i contains at most κ nodes. Since each node has at most κ children, level $i+1$ can have at most κ^2 nodes, which is at most κ. We have shown each level has at most κ nodes. Since there is a countable number of levels, the total number of nodes is at most $\omega \times \kappa = max.\{\omega, \kappa\} = \kappa$.

Consider any function $F = \{(n, \alpha) : n \in tree.s \text{ and } \alpha \text{ is an ordinal}\}$ where R, the range of F, is an ordinal (this guarantees that for any α in R, we have $\beta \in R$ for all $\beta \prec \alpha$). Note that the labeling described above is such a function. We have that $F.s$ is an ordinal of cardinality at most κ, otherwise we have a surjection from κ to a larger cardinal, a contradiction. $\square$

Lemma 9 *If $sBw, s \dashrightarrow u, u \in tree(s, w)$ then $l.tree(u, w) \prec l.tree(s, w)$.*

Proof Since $u \in tree(s, w)$, by construction of $tree(s, w)$, the subtree rooted at u is $tree(u, w)$. By definition of l, $l.tree(u, w) \prec l.tree(s, w)$. $\square$

Definition 6 *(length) Given B, an STS, $length(w, s, u) = 0$ if $\neg(sBw)$ or $\neg(s \dashrightarrow u)$, otherwise $length(w, s, u)$ is the length of the shortest initial segment starting at w that matches $\langle s, u\rangle$. Formally:*

$length(w, s, u) = \langle min\ \sigma, \delta, \pi, \xi\ :\ fp.\sigma.s\ \wedge\ \sigma.1 = u\ \wedge\ fp.\delta.w\ \wedge\ \pi, \xi \in INC\ \wedge\ corr(B, \sigma, \pi, \delta, \xi) : \#({}^{\xi}\delta^{0})\rangle$

As sBw and $s \dashrightarrow u$, the above range is non-empty and $length(w, s, u) \in \mathbb{N}$.

Lemma 10 *If $sBw, s \dashrightarrow u$ and $\neg\langle\exists\sigma, \delta, \pi, \xi : fp.\sigma.s\ \wedge\ \sigma.1 = u\ \wedge\ fp.\delta.w\ \wedge\ \pi, \xi \in INC\ :\ corr(B, \sigma, \pi, \delta, \xi)\ \wedge\ {}^{\xi}\delta^{0} = \langle w\rangle\rangle$, then $\langle\exists v\ :\ w \dashrightarrow v\ :\ length(v, s, u) < length(w, s, u)\ \wedge\ sBv\rangle$.*

Proof First, note that by assumption $length(w, s, u) \geq 2$, because for any σ, δ, π, ξ such that $fp.\sigma.s \wedge \sigma.1 = u \wedge fp.\delta.w \wedge \pi, \xi \in INC \wedge corr(B, \sigma, \pi, \delta, \xi)$, we have $\neg(uB\delta.1)$. Hence, ${}^{\pi}\sigma^{0} = \langle s\rangle, \#({}^{\xi}\delta^{0}) \geq 2$, and $length(w, s, u) \geq 2$. Let v be the successor of w in some minimal initial segment, then $length(v, s, u) = length(w, s, u) - 1 \geq 1$ and sBv, as required. $\Box$

Proposition 2 *(Completeness) If B is an STS, then B is a WFS.*

Proof Wfs1 follows from Sts1. Let $W = (\#S + \omega)^{+}$. Note that $+$ denotes cardinal arithmetic; we add ω to $\#S$ to guarantee that we have an infinite cardinal; κ^{+} is the successor cardinal to κ. Clearly, $(W, \prec)$ is well-founded. Let $rankt = l.tree$ and let $rankl = length$. Let sBw and $s \dashrightarrow u$. There are three cases:

1. $\langle \exists v : w \dashrightarrow v : uBv \rangle$.

   By lemma 1, if (1) does not hold, then for any σ, δ, π, ξ such that $\mathit{fp}.\sigma.s \wedge \sigma.1 = u \wedge \mathit{fp}.\delta.w \wedge \pi, \xi \in \mathit{INC} \wedge \mathit{corr}(B, \sigma, \pi, \delta, \xi)$, either s marks the end of ${}^{\pi}\sigma^{0}$ or w marks the end of ${}^{\xi}\delta^{0}$, but not both.

2. $\langle \exists \sigma, \delta, \pi, \xi : \mathit{fp}.\sigma.s \wedge \sigma.1 = u \wedge \mathit{fp}.\delta.w \wedge \pi, \xi \in \mathit{INC} \wedge {}^{\xi}\delta^{0} = \langle w \rangle : \mathit{corr}(B, \sigma, \pi, \delta, \xi) \rangle$ and (1) does not hold. This implies that $\#({}^{\pi}\sigma^{0}) > 1$, uBw, and $u \in \mathit{tree}(s, w)$; hence, $\mathit{rankt}(u, w) \prec \mathit{rankt}(s, w)$ by lemma 9.

3. If (1) and (2) do not hold, we must have $\neg \langle \exists \sigma, \delta, \pi, \xi : \mathit{fp}.\sigma.s \wedge \sigma.1 = u \wedge \mathit{fp}.\delta.w \wedge \pi, \xi \in \mathit{INC} : \mathit{corr}(B, \sigma, \pi, \delta, \xi) \wedge {}^{\xi}\delta^{0} = \langle w \rangle \rangle$. By lemma 10 and the definition of *rankl*, $\langle \exists v : w \dashrightarrow v : \mathit{rankl}(v, s, u) < \mathit{rankl}(w, s, u) \wedge sBv \rangle$.
   $\Box$

Theorem 4 *(Equivalence) B is an STS iff B is a WFS.*

Proof By propositions 1 and 2. $\Box$

A consequence of the above theorem is that all of the properties proved for STSs carry over to WFSs; we use this fact freely, without reference, in the sequel.

4.4 Refinement

Up to this point, we have developed a theory for relating states. We now show how to apply the theory to transition systems. In this section, we define a notion of refinement and show that STSs can be used in a compositional fashion. For states s and w, we write $s \sqsubseteq w$ to mean that there is an STS B

such that sBw. By theorem 2, $s \sqsubseteq w$ iff sGw, where G is the greatest STS. We now lift this idea to transition systems.

Definition 7 *(Simulation Refinement) Let* $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$, $\mathcal{M}' = \langle S', \dashrightarrow', L' \rangle$, *and* $r : S \to S'$. *We say that* $\mathcal{M}$ *is a simulation refinement of* $\mathcal{M}'$ *with respect to refinement map* r, *written* $\mathcal{M} \sqsubseteq_r \mathcal{M}'$, *if there exists a relation,* B, *such that* $\langle \forall s \in S :: sB(r.s) \rangle$ *and* B *is an STS on the TS* $\langle S \uplus S', \dashrightarrow \uplus \dashrightarrow', \mathcal{L} \rangle$, *where* $\mathcal{L}.s = L'(s)$ *for* s *an* S' *state and* $\mathcal{L}.s = L'(r.s)$ *otherwise.*

In the above definition, it helps to think of $\mathcal{M}'$ as the specification and $\mathcal{M}$ as the implementation. That $\mathcal{M}$ is a simulation refinement of $\mathcal{M}'$ with respect to r implies that every visible behavior of $\mathcal{M}$ (where what is visible depends on r) is a behavior of $\mathcal{M}'$. There are often other considerations, *e.g.*, it might be that $\mathcal{M}$ and $\mathcal{M}'$ have certain states that are "initial". In this case one might wish to show that initial states in $\mathcal{M}$ are mapped to initial states in $\mathcal{M}'$.

One has a great deal of flexibility in choosing refinement maps. The danger is that by choosing a complicated refinement map, one can bypass the verification problem all together. To make this point clear, let PRIME be the system whose single behavior is the sequence of primes and let NAT be the system whose single behavior is the sequence of natural numbers. We do not consider NAT to be an implementation of PRIME, but using the refinement map from NAT to PRIME that maps i to the i^{th} prime, we can indeed prove the peculiar theorem that NAT is a refinement of PRIME. The moral is that we must be careful to not bypass the verification problem with the use of such refinement maps. Simple refinement maps with a clear relationship between implementation states and their image under the map are best.

To check that a proof of refinement is meaningful, one has to look at the specification system and the refinement map. It is important to look at the specification system to check that in fact it corresponds to what you expected. It is important to look at the refinement map for the reasons stated above, namely, one has to check that refinement maps map implementation states to "related" specification states. It may occur to the reader that we should require that refinement maps preserve (part of) the labeling function of the implementation states. In fact, we consider such a restriction shortly, but the reason we do not place restrictions on refinement maps is that it is not a priori apparent what the "reasonable" relationships between implementation states and specification states might be, *e.g.*, suppose that the specification system represents numbers in decimal but the implementation system represents numbers in binary, or that numbers in the specification are spread across several registers in the implementation, and so on. See also the discussion in section 5.6, on page 66.

Often refinement maps are especially clear, which makes it easy to check that they are in fact appropriate. Suppose that associated with states is a set of variables, each of a particular type. Furthermore, suppose that the variables in the implementation are a superset of the variables in the specification and that the refinement map just hides the implementation variables that do not appear in the specification. Then, it is clear that the refinement map is a reasonable one. More precisely, given TS $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$, if L has the following structure, we say that $\mathcal{M}$ is *typed*. Let $VARS$ be a set and let $TYPE$ be a function whose domain is $VARS$. Think of $VARS$ as the variables of TS $\mathcal{M}$, where $TYPE$ gives the type of the variables. For all $s \in S$, let $L.s$ be a function from $VARS$ such that $L.s.v \in TYPE.v$. The lemma below shows why the appropriateness of refinement maps that hide some of the implementation variables is easy to ascertain.

Lemma 11 *If* $\mathcal{M} = \langle S, \dashrightarrow, L\rangle \sqsubseteq_r \mathcal{M}' = \langle S', \dashrightarrow', L'\rangle$, *both* $\mathcal{M}$ *and* $\mathcal{M}'$ *are typed TSs, and* $L'(r.s) = L.s|_V$, *then for any pair of states* $s, r.s$ *such that* $s \in S$, *and any* ACTL* $\setminus$ X *formula,* f, *built out of expressions that only depend on variables in* V, *we have* $\mathcal{M}', r.s \models f \;\Rightarrow\; \mathcal{M}, s \models f$.

Proof Since $\mathcal{M} \sqsubseteq_r \mathcal{M}'$ we have that $s \sqsubseteq r.s$ in the disjoint union of $\mathcal{M}$ and $\mathcal{M}'$. Recall that the refinement map r does not alter the value of variables in V. The result follows from theorem 3. $\Box$

Lemma 12 *If* B *is an STS on TS* $\mathcal{M} = \langle S \supseteq S_1 \cup S_2, \dashrightarrow, L\rangle$, $S_1 \cap S_2 = \emptyset$, *any state in* S_1 *can only reach states in* S_1, *any state in* S_2 *can only reach states in* S_2, *then* $\hat{B} = \{\langle s_1, s_2\rangle : s_1 \in S_1 \;\wedge\; s_2 \in S_2 \;\wedge\; s_1 B s_2\}$ *is an STS on* $\mathcal{M}$.

Proof Suppose $s\hat{B}w$, then $s \in S_1$, $w \in S_2$, and sBw. Let σ satisfy $fp.\sigma.s$. Since sBw, there is a δ such that $fp.\delta.w$ and $match(B, \sigma, \delta)$. Since all states reachable from s are in S_1, σ contains only S_1 states and similarly δ contains only S_2 states, therefore, $match(\hat{B}, \sigma, \delta)$ by the definitions of $\hat{B}$ and $match$. $\Box$

Theorem 5 *(Composition) If* $\mathcal{M} \sqsubseteq_r \mathcal{M}'$ *and* $\mathcal{M}' \sqsubseteq_q \mathcal{M}''$ *then* $\mathcal{M} \sqsubseteq_{r;q} \mathcal{M}''$.

Proof Let $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$, $\mathcal{M}' = \langle S', \dashrightarrow', L'\rangle$, and $\mathcal{M}'' = \langle S'', \dashrightarrow'', L''\rangle$. Since $\mathcal{M} \sqsubseteq_r \mathcal{M}'$, we have an STS, A, such that $\langle \forall s \in S :: sA(r.s)\rangle$; by lemma 12 we can assume that $A \subseteq S \times S'$. Since $\mathcal{M}' \sqsubseteq_q \mathcal{M}''$, we have an STS, B, such that $\langle \forall s' \in S' :: s'B(q.s')\rangle$; by lemma 12 we can assume that $B \subseteq S' \times S''$. Letting C be $A;B$, we have, by properties of A and B, that

$C \subseteq S \times S''$ and $\langle \forall s \in S :: sCq(r.s) \rangle$. By lemma 4, we have that C is an STS on the TS $\langle S \uplus S'', \dashrightarrow \uplus \dashrightarrow'', \mathcal{L} \rangle$, where $\mathcal{L}.s = L''(q(r.s))$ for s an S state and $\mathcal{L}.s = L''(s)$ otherwise. $\square$

4.5 Bibliographic Notes

Lamport makes the case that specifications should be invariant under stuttering in [Lam83]. Abadi and Lamport have written a well-cited paper on refinement maps [AL91]. A comparison of their approach to refinement with our approach appears in section 5.6, on page 66.

Stuttering bisimulation was introduced by Browne, Clark, and Grumberg [BCG88]. The definition of stuttering simulation is derived from their definition. The proof of soundness and completeness is based on the proof of soundness and completeness of Namjoshi's proof rule for well-founded bisimulation [Nam97]. We compare our proof rules with Namjoshi's proof rule in the bibliographic notes of the next chapter. The first paper to present a mechanically verified theorem using stuttering simulation is by Sumners [Sum00]. Sumners shows the correctness of a concurrent, wait-free implementation of a double-ended queue, which is used to implement an optimal work stealing algorithm.

See also the bibliographic notes in the next chapter.

4.6 Summary

In this chapter, we introduced the notion of stuttering simulation. We proved that stuttering simulations enjoy several algebraic properties. For example, they are closed under arbitrary union and relational composition. We gave a sound and complete proof rule that allows us to prove a stuttering simulation by reasoning about single transitions. We also introduced the notion of refinement for stuttering simulations and showed that this notion is compositional, *i.e.*, refinement proofs can be decomposed into a sequence of simpler refinement proofs.

Chapter 5

Stuttering Bisimulation

5.1 Stuttering Bisimulation

A relation on $B \subseteq S \times S$ where $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ is a stuttering bisimulation, if for any s, w such that sBw, s and w are identically labeled and any fullpath starting at s can be matched by some fullpath starting at w and conversely.

Definition 8 *(Stuttering Bisimulation (STB))*

$B \subseteq S \times S$ is a *stuttering bisimulation* on TS $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ iff both B and B^{-1} are STSs on $\mathcal{M}$.

Lemma 13 $match(R, \sigma, \delta) \equiv match(R^{-1}, \delta, \sigma)$

Proof

$$match(R, \sigma, \delta)$$

$\equiv$ { Definition of *match*, *corr* }

$\langle \exists \pi, \xi \in INC :: \langle \forall i \in \mathbb{N} :: ({}^{\pi}\sigma^{i}) R ({}^{\xi}\delta^{i}) \rangle \rangle$

$\equiv$ { Expand $({}^{\pi}\sigma^{i}) R ({}^{\xi}\delta^{i})$, definition of R^{-1} }

$\langle \exists \pi, \xi \in INC : \langle \forall i \in \mathbb{N} :: ({}^{\xi}\delta^{i}) R^{-1} ({}^{\pi}\sigma^{i}) \rangle \rangle$

$\equiv$ { Definition of *match*, *corr* }

$match(R^{-1}, \delta, \sigma)$ $\Box$

Lemma 14 *$B \subseteq S \times S$ is a stuttering bisimulation on TS $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ iff for all s, w such that sBw:*

(Stb1) $L.s = L.w$

(Stb2) $\langle \forall \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : match(B, \sigma, \delta) \rangle \rangle$

(Stb3) $\langle \forall \delta : fp.\delta.w : \langle \exists \sigma : fp.\sigma.s : match(B, \sigma, \delta) \rangle \rangle$

Proof This follows by the definition of STB and lemma 13. $\Box$

We show that there is a greatest STB, and that it is an equivalence relation. This is accomplished by showing that an arbitrary union of STB's is an STB, hence, there is a greatest STB. We then show: the identity relation is an STB, if B is an STB, B^{-1}—the inverse of B—is an STB, and if A and B are STB's so is $A; B$—the composition of A and B. Together, the above results imply that the reflexive, symmetric, transitive closure of an STB is an STB and that the greatest STB is an equivalence relation.

Lemma 15 *If R is an STB, so is R^{-1}.*

Proof

R is an STB

$\equiv$ { Definition of STB }

R is an STS and R^{-1} is an STS

$\equiv$ { Definition of STB, R^{-1} }

R^{-1} is an STB $\Box$

Lemma 16 *Let $\mathcal{C}$ be a set of STB's on TS $\mathcal{M}$, then $G = \langle \cup B : B \in \mathcal{C} : B \rangle$ is an STB on $\mathcal{M}$.*

Proof G is an STS by lemma 3. $G^{-1} = \langle \cup B : B \in \mathcal{C} : B \rangle^{-1} = \langle \cup B : B \in \mathcal{C} : B^{-1} \rangle$. By lemma 15 $\langle \forall B \in \mathcal{C} :: B^{-1}$ is an STB $\rangle$ and by lemma 3, G^{-1} is an STS, hence, G is an STB. $\Box$

Corollary 2 *For any TS $\mathcal{M}$, there is a greatest STB on $\mathcal{M}$.*

Proof Let $G = \langle \cup B : B \in \mathcal{C} : B \rangle$ where $C = \{B : B$ is an STB on $\mathcal{M}\}$. G is an STB by lemma 16, hence, the greatest. $\Box$

Lemma 17 *If R and S are STB's, so is $T = R; S$.*

Proof $R; S$ is an STS by lemma 4. By lemma 15, S^{-1} and R^{-1} are STBs, hence, STSs. Thus, by lemma 4, $S^{-1}; R^{-1} = (R; S)^{-1}$ is an STS.

Lemma 18 *The reflexive, symmetric, transitive closure of an STB is an STB.*

Proof The reflexive, symmetric, transitive closure of B, denoted $B^{\equiv}$, by definition is $(B \cup B^{-1})^*$, which is $\langle \cup i \in \mathbb{N} :: (B \cup B^{-1})^i \rangle$. The identity relation, $(B \cup B^{-1})^0$, is an STB, as is $B \cup B^{-1}$ by lemmas 15,16. By induction on the natural numbers, using lemma 17, so is $(B \cup B^{-1})^i$ for any $i > 0$. By lemma 16 so $\langle \cup i \in \mathbb{N} :: (B \cup B^{-1})^i \rangle$. $\Box$

Theorem 6 *Given TS $\mathcal{M}$, there is a greatest STB on $\mathcal{M}$, which is an equivalence relation.*

Proof By corollary 2 there is a greatest STB G on $\mathcal{M}$. By lemma 18 $G^{\equiv}$ is also an STB. Now, $G^{\equiv} \supseteq G$, by definition of $G^{\equiv}$ and $G^{\equiv} \subseteq G$ since G is the greatest STB. Thus, $G = G^{\equiv}$. $\Box$

Lemma 19 *STBs are not closed under intersection or negation.*

Proof The proof of lemma 6 works in this case as well, as the STSs described in the proof and in figure and the figure 4.1 are also STBs, and their negations and intersections are not. $\Box$

Theorem 7 *Let B be an STB on $\mathcal{M}$ and let sBw. For any* $\mathrm{CTL}^* \setminus \mathsf{X}$ *formula f, $\mathcal{M}, w \models f$ iff $\mathcal{M}, s \models f$.*

Proof The proof is by induction on the structure of $\mathrm{CTL}^* \setminus \mathsf{X}$ formulas and is given by Browne, Clarke, and Grumberg [BCG88]. $\Box$

5.2 Well-Founded Bisimulation

As was the case with STS, in order to check that a relation is an STB, we have to show that infinite sequences "match". This can be difficult when using computer-aided verification techniques. We present the notion of a *well-founded bisimulation* to remedy this situation. To show that a relation is a well-founded bisimulation, we need only check local properties; this is analogous to proving program termination by exhibiting a well-founded relation and showing it decreases during every step of the program. Formally, we have:

Definition 9 *(Well-Founded Bisimulation (WFB))*

$B \subseteq S \times S$ is a Well-Founded Bisimulation (WFB) on TS $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ iff both B and B^{-1} are WFSs on $\mathcal{M}$.

Theorem 8 *(Equivalence) B is an STB iff B is a WFB.*

Proof B is an STB iff B is an STS and B^{-1} is an STS iff B is a WFS and B^{-1} is a WFS iff B is a WFB. $\Box$

Corollary 3 *Given any transition system $\mathcal{M}$, there is a greatest WFB on $\mathcal{M}$.*

Proof The greatest WFB corresponds to the greatest STB. $\Box$

5.3 Equivalence Bisimulations

If we modify the definitions of STB, WFB so that B is required to be an equivalence relation, then some simplifications are possible. In this section we explore this idea.

Definition 10 *(Equivalence STB (ESTB))*

$B \subseteq S \times S$ is an equivalence stuttering bisimulation on TS $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ iff B is an equivalence relation and an STS.

Lemma 20 *If B is an STS on TS $\mathcal{M}$ and B is symmetric, then B is an STB on $\mathcal{M}$.*

Proof If B is symmetric then $B = B^{-1}$, thus both B and B^{-1} are STSs, *i.e.*, B is an STB. $\Box$

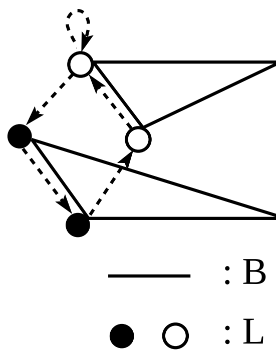

Figure 5.1: The graph denotes a transition system, where circles denote states, the color of the circles denotes their label, and the transition relation is denoted by a dashed line. States related by the equivalence relation B are joined by a solid line. Notice that B is an ESTB as related states have the same infinite paths, up to stuttering.

Lemma 21 *B is an ESTB on TS $\mathcal{M}$ iff it is an STB on $\mathcal{M}$ and an equivalence relation.*

Proof Since B is an equivalence relation, it is symmetric, thus, by lemma 20 it is an STB. For the other direction, note that an STB is an STS. $\square$

An example of an ESTB appears in figure 5.1. ESTBs are very difficult to prove mechanically because one has to reason about infinite sequences. We would much rather reason about single steps. This is the motivation for the following definition.

Definition 11 *(Well-Founded Equivalence Bisimulation (WEB)) $B \subseteq S \times S$ is a well-founded equivalence bisimulation on TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ iff:*

(Web1) B is an equivalence relation on S; and

(Web2) $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$; and

(Web3) There exist functions , *erankt* $: S \to W$, *erankl* $: S \times S \to \mathbb{N}$, such that

$\langle W, \lessdot \rangle$ is well-founded, and

$\langle \forall s, u, w \in S : sBw \quad \wedge \quad s \dashrightarrow u :$

(a) $\langle \exists v : w \dashrightarrow v : uBv \rangle \quad \vee$

(b) $(uBw \quad \wedge \quad \mathit{erankt}.u \lessdot \mathit{erankt}.s) \quad \vee$

(c) $\langle \exists v : w \dashrightarrow v : sBv \quad \wedge \quad \mathit{erankl}(v, u) < \mathit{erankl}(w, u) \rangle \rangle$

We could have defined a WEB to be a WFS that is an equivalence relation, but notice that *erankt* and *erankl* have fewer arguments in the above definition. We call a triple $\langle \mathit{erankt}, \langle W, \lessdot \rangle, \mathit{erankl} \rangle$ satisfying condition Web3 in the above definition, a *well-founded witness.* Note that to prove a relation is a WEB, reasoning about single steps of $\dashrightarrow$ suffices.

The third WEB condition can be thought of as follows. It says that given states s and w in the same class, such that s can step to u, u is either matched by a step from w, or u and w are in the same class and the rank function decreases (to guarantee that w is forced to take a step), or some successor v of w is in the same class as s and the rank function decreases (to guarantee that u is eventually matched). An example of a WEB appears in figure 5.2.

Proposition 3 *(Soundness) If B is a WEB, then it is an ESTB.*

Proof If we satisfy Web1-3, we can obtain a WFS by just defining *erankt* to be a function of 2 arguments and *erankl* to be a function of three arguments.

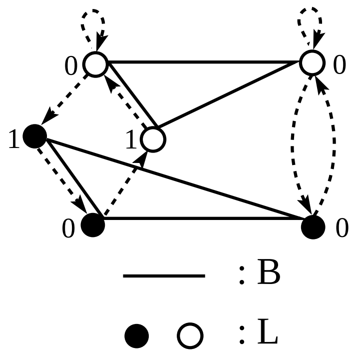

Figure 5.2: The graph denotes a transition system, as in figure 5.1. To check that B is a WEB, let $erankt.v$ = tag of v, $erankl(v, u)$ = tag of v, and use the well-founded witness $\langle erankt, \langle \mathbb{N}, < \rangle, erankl \rangle$.

We then use the soundness of WFS (proposition 1). $\Box$

More interesting is the proof of completeness. For the proof, we have to exhibit the rank functions as per the definition of WEB. Here is a high-level overview.

We define $erankt.s$ to be the rank of the largest subtree of the computation tree rooted at s such that for every node x we have sBx. If Web3b holds, then w does not have a successor in the class of s and the subtree has no infinite path. Notice that we do not have to keep track of w; all we need to know is that it is in the same class as s.

The value of $erankl(w, u)$ is the length of the shortest path from w that stays in the class of w and reaches a state in the class of u. Again, we do not need s as an argument and can define $erankl$ so that it works for any state in the class of w.

Definition 12 *(etree) Given an ESTB B, if* $\langle \forall w : sBw : \langle \exists v : w \dashrightarrow v : sBv \rangle\rangle$, *then etree.s is the empty tree, otherwise etree.s is the largest subtree of* *the computation tree rooted at s such that for any node of the tree, $\langle s, \ldots, x \rangle$, we have that sBx.*

Lemma 22 *Every path of etree.s is finite.*

Proof Suppose not, then there exists a path σ such that $fp.\sigma.s$. As $etree.s$ is not empty, there is a state w such that sBw and for every successor v of w, we have $\neg(sBv)$. Since $sBw, \langle \exists \delta :: match(B, \sigma, \delta) \rangle$, hence, $\delta.1$ marks the beginning of the second segment in δ (because $\langle \forall v : w \dashrightarrow v : \neg(sBv) \rangle$). Say $\sigma.j$ marks the beginning of the second segment of σ, then $(\sigma.j)B(\delta.1)$, but $sB(\sigma.j)$ by the definition of *etree*. Since B is an equivalence relation, $sB(\delta.1)$—a contradiction; hence, σ cannot be infinite. $\square$

We can use the same labeling function defined on page 43, *i.e.*, since the child relation on nodes in *tree.s* is well-founded, we can recursively define a labeling function, l, that assigns an ordinal to nodes in the tree as follows: $l.n = \langle \cup c : c \text{ is a child of } n : (l.c) + 1 \rangle$.

Lemma 23 *If $\#S \preceq \kappa$, where κ is an infinite cardinal (*i.e., *$\omega \preceq \kappa$) then for all $s \in S$, etree.s is labeled with an ordinal of cardinality $\preccurlyeq \kappa$.*

Proof The proof is identical to the proof of lemma 8 (except that *etree* is used instead of *tree*). $\square$

Lemma 24 *If $s \dashrightarrow u, u \in etree.s$ then $l(etree.u) \prec l(etree.s)$.*

Proof Since $u \in etree.s$, $etree.s$ is non-empty. If $etree.u$ is empty, $l(etree.u) \prec l(etree.s)$, otherwise, by construction of $etree.s$, the subtree rooted at u is $etree.u$. By definition of l, $l(etree.u) \prec l(etree.s)$. $\square$

Definition 13 *(elength) Given B, an ESTB, $elength(w,u) = 0$ if $\neg\langle \exists s : sBw : s \dashrightarrow u\rangle$, otherwise $elength(w,u)$ is the length of the shortest initial segment starting at w that matches $s;u$ for some s such that sBw and $s \dashrightarrow u$. Formally:*

$elength(w,u) = \langle min\ s,\sigma,\delta,\pi,\xi : fp.\sigma.s \ \wedge\ \sigma.1 = u \ \wedge\ fp.\delta.w \ \wedge\ \pi,\xi \in INC \ \wedge\ corr(B,\sigma,\pi,\delta,\xi) : \#({}^{\xi}\delta^{0})\rangle$

The above range is non-empty because $\langle \exists s : sBw : s \dashrightarrow u\rangle$, and $elength(w,u) \in \mathbb{N}$.

Lemma 25 *If $sBw, s \dashrightarrow u$ and $\neg\langle \exists \sigma,\delta,\pi,\xi : fp.\sigma.s \ \wedge\ \sigma.1 = u \ \wedge\ fp.\delta.w \ \wedge\ \pi,\xi \in INC : corr(B,\sigma,\pi,\delta,\xi) \ \wedge\ {}^{\xi}\delta^{0} = \langle w\rangle\rangle$, then $\langle \exists v : w \dashrightarrow v : elength(v,u) < elength(w,u) \ \wedge\ sBv\rangle$.*

Proof Note that by assumption $elength(w,u) \geq 2$, because for any s',σ,δ,π,ξ such that $fp.\sigma.s' \ \wedge\ \sigma.1 = u \ \wedge\ fp.\delta.w \ \wedge\ \pi,\xi \in INC \ \wedge\ corr(B,\sigma,\pi,\delta,\xi)$, we have $\neg(uB\delta.1)$; if not, $uB\delta.1$ when $s' = s$, a contradiction. Hence, ${}^{\pi}\sigma^{0} = \langle s'\rangle, \#({}^{\xi}\delta^{0}) \geq 2$, and $elength(w,u) \geq 2$. Let v be the successor of w in some minimal initial segment, then $elength(v,u) = elength(w,u) - 1 \geq 1$; by $sBw, wBs', s'Bv$, and transitivity we get sBv, as required. $\Box$

Proposition 4 *(Completeness) If B is an ESTB, then B is a WEB.*

Proof Web1 holds because an ESTB is an equivalence relation. Web2 follows from Sts1. Let $W = (\#S + \omega)^{+}$. Recall that $+$ denotes cardinal arithmetic and that κ^{+} is the successor cardinal to κ. We add ω to $\#S$ to guarantee that we have an infinite cardinal. Clearly, $(W, \prec)$ is well-founded. Let $erankt = l.etree$ and let $erankl = elength$. Let sBw and $s \dashrightarrow u$. There are three cases:

1. $\langle \exists v : w \dashrightarrow v : uBv \rangle$.

   By lemma 1, if (1) does not hold, then for any σ, δ, π, ξ such that $fp.\sigma.s \wedge \sigma.1 = u \wedge fp.\delta.w \wedge \pi, \xi \in INC \wedge corr(B, \sigma, \pi, \delta, \xi)$, either s marks the end of ${}^{\pi}\sigma^0$ or w marks the end of ${}^{\xi}\delta^0$, but not both.

2. $\langle \exists \sigma, \delta, \pi, \xi : fp.\sigma.s \wedge \sigma.1 = u \wedge fp.\delta.w \wedge \pi, \xi \in INC \wedge {}^{\xi}\delta^0 = \langle w \rangle : corr(B, \sigma, \pi, \delta, \xi) \rangle$ and (1) does not apply. This implies that $\#({}^{\pi}\sigma^0) > 1$, uBw, and $etree.s$ is non-empty (if $\langle \exists v : w \dashrightarrow v : sBv \rangle$ by uBw, wBs, sBv, and transitivity uBv holds, as does case (1)—a contradiction). Hence, $u \in etree.s$ and $erankt.u \prec erankt.s$ by lemma 24.

3. If (1) and (2) do not hold, we must have $\neg\langle \exists \sigma, \delta, \pi, \xi : fp.\sigma.s \wedge \sigma.1 = u \wedge fp.\delta.w \wedge \pi, \xi \in INC : corr(B, \sigma, \pi, \delta, \xi) \wedge {}^{\xi}\delta^0 = \langle w \rangle \rangle$. By lemma 25 and the definition of $erankl$, $\langle \exists v : w \dashrightarrow v : sBv \wedge erankl(v, u) < erankl(w, u) \rangle$. $\square$

Theorem 9 *(Equivalence) B is an ESTB iff B is a WEB.*

Proof By propositions 3 and 4. $\square$

Corollary 4 *B is a WEB on $\mathcal{M}$ iff B is a WFB on $\mathcal{M}$ and an equivalence relation.*

Proof B is a WEB iff (theorem 9) B is an ESTB iff (lemma 21) B is an STB and an equivalence relation iff (theorem 8) B is a WFB and an equivalence relation. $\square$

Corollary 5 *Given any transition system $\mathcal{M}$, there is a greatest WEB and a greatest ESTB on $\mathcal{M}$, and they are equal.*

Proof The greatest STB is an equivalence relation by theorem 6. By lemma 21 it is also an ESTB. Since any ESTB is an STB (lemma 21), there is a greatest ESTB which is equal to the greatest STB. By theorem 9 it is also the greatest WEB. $\square$

5.4 Quotient Structures

For an STB B on $\mathcal{M}$, a *quotient structure* $\mathcal{M}/B$ (read $\mathcal{M}$ "mod" B) can be defined, where the states are equivalence classes (obtained from B) of states of $\mathcal{M}$ and the transition relation is derived from the transition relation of $\mathcal{M}$. Quotient structures can be much smaller than the original: an STB with finitely many classes induces a finite quotient. As we will see in chapter 7, we are often concerned only with the part of the quotient structure that is reachable from a set of initial states.

Let $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ be a TS and let B be an STB on $\mathcal{M}$. The equivalence class of state s, under $B^{\equiv}$, is denoted by $[s]_B$. Define $\mathcal{M}/B$ as the TS $\langle \mathcal{S}, \rightsquigarrow, \mathcal{L}\rangle$ given by:

1. $\mathcal{S} = \{[s]_B \mid s \in S\}$,
2. The transition relation is given by : For $C, D \in \mathcal{S}$, $C \rightsquigarrow D$ iff either
   (a) $C \neq D$ and $\langle \exists s, w : s \in C \wedge w \in D : s \dashrightarrow w\rangle$, or
   (b) $C = D$ and $\langle \forall s : s \in C : \langle \exists w : w \in C : s \dashrightarrow w\rangle\rangle$.

   The case distinction is needed to prevent spurious self-loops in the quotient, arising from stuttering steps in the original.

3. The labeling function is given by $\mathcal{L}.C = L.s$, for some s in C (states in an equivalence class have the same label).

Theorem 10 *If B is an STB on $\mathcal{M}$, then there is an ESTB on the disjoint union of $\mathcal{M}$ and $\mathcal{M}/B$ that relates s with $[s]_B$.*

Proof Let C be $B \cup \{\langle s, [s]_B \rangle : s \in S\}$. The ESTB relating states of $\mathcal{M}$ and $\mathcal{M}/B$ is $C^{\equiv}$, the reflexive, symmetric, transitive closure of C. The rest of the proof can be obtained from the proof of a similar result due to Namjoshi [Nam97].

Corollary 6 *For any* CTL$^* \setminus \mathsf{X}$ *formula f, $\mathcal{M}, s \models f$ iff $\mathcal{M}/B, [s] \models f$.* $\Box$

5.5 Refinement

Up to this point, we have developed a theory for relating states. We now show how to apply the theory to transition systems. In this section, we define a notion of refinement and show that STBs can be used in a compositional fashion. For states s and w, we write $s \approx w$ to mean that there is a WEB B such that sBw. By theorem 6, $s \approx w$ iff sGw, where G is the greatest STB. We now lift this idea to transition systems.

Definition 14 *(WEB Refinement) Let $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$, $\mathcal{M}' = \langle S', \dashrightarrow', L' \rangle$, and $r : S \to S'$. We say that $\mathcal{M}$ is a WEB refinement of $\mathcal{M}'$ with respect to refinement map r, written $\mathcal{M} \approx_r \mathcal{M}'$, if there exists a relation, B, such that $\langle \forall s \in S :: sB(r.s) \rangle$ and B is a WEB on the TS $\langle S \uplus S', \dashrightarrow \uplus \dashrightarrow', \mathcal{L} \rangle$, where $\mathcal{L}.s = L'(s)$ for s an S' state and $\mathcal{L}.s = L'(r.s)$ otherwise.*

In the above definition, it helps to think of $\mathcal{M}'$ as the specification and $\mathcal{M}$ as the implementation. That $\mathcal{M}$ is a WEB refinement of $\mathcal{M}'$ implies that $\mathcal{M}$ and $\mathcal{M}'$ have the same visible behaviors. There are often other considerations, *e.g.*, it might be that $\mathcal{M}$ and $\mathcal{M}'$ have certain states that are "initial". In this case one might wish to show that initial states in $\mathcal{M}$ are mapped to initial states in $\mathcal{M}'$. The points made regarding simulation refinement, *e.g.*, that we should use clear refinement maps, also apply.

Lemma 26 *If $\mathcal{M} = \langle S, \dashrightarrow, L\rangle \approx_r \mathcal{M}' = \langle S', \dashrightarrow', L'\rangle$, both $\mathcal{M}$ and $\mathcal{M}'$ are typed TSs, and $L'(r.s) = L.s|_V$, then any pair of states $s, r.s$ such that $s \in S$ satisfy the same* $\mathrm{CTL}^* \setminus \mathsf{X}$ *formulas built out of expressions that only depend on variables in V.*

Proof Since $\mathcal{M} \approx_r \mathcal{M}'$ we have that $s \approx r.s$ in the disjoint union of $\mathcal{M}$ and $\mathcal{M}'$. Recall that the refinement map r does not alter the value of variables in V. The result follows from theorem 7. $\Box$

We note that the above lemma can be strengthened by replacing "$\mathrm{CTL}^* \setminus \mathsf{X}$" by "stuttering-insensitive."

Theorem 11 *(Composition) If $\mathcal{M} \approx_r \mathcal{M}'$ and $\mathcal{M}' \approx_q \mathcal{M}''$ then $\mathcal{M} \approx_{r;q} \mathcal{M}''$.*

Proof Let $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$, $\mathcal{M}' = \langle S', \dashrightarrow', L'\rangle$, and $\mathcal{M}'' = \langle S'', \dashrightarrow'', L''\rangle$. Since $\mathcal{M} \approx_r \mathcal{M}'$, we have a WEB, A, such that $\langle \forall s \in S :: sA(r.s)\rangle$. Since $\mathcal{M}' \sqsubseteq_q \mathcal{M}''$, we have a WEB, B, such that $\langle \forall s' \in S' :: s'B(q.s')\rangle$. Thus, $A, B, A \cup B$ are STBs on the disjoint union of $\mathcal{M}$, $\mathcal{M}'$, and $\mathcal{M}''$, by the definition of STB and lemma 16. Letting C be the reflexive, symmetric, transitive closure of $A \cup B$, we have, by lemma 18 and the definition of an equivalence relation, that C is a WEB on the disjoint union of $\mathcal{M}$, $\mathcal{M}'$, and $\mathcal{M}''$. Letting

$D = C \cap (S \cup S'')^2$, we have that D is a WEB on the TS $\langle S \uplus S'', \dashrightarrow \uplus \dashrightarrow'', \mathcal{L} \rangle$, where $\mathcal{L}.s = L''(q(r.s))$ for s an S state and $\mathcal{L}.s = L''(s)$ otherwise. $\square$

5.6 Remarks on Refinement

In this section we relate our work on refinement with the well-known and influential work of Abadi and Lamport [AL91]. Abadi and Lamport prove that if one system implements another, then there is a refinement map that can be used to prove this. Their theorem has the following conditions. First, the systems under consideration are from a restricted class: the implementation system is *machine closed* and the specification system has *finite invisible nondeterminism* and is *internally continuous*. Second, they allow the use of *history* and *prophecy* variables, which may be needed to carry out the refinement proof. See their paper for the details. As in our case, the reason why such a theorem is interesting is that it allows one to prove that systems have related infinite computations, by reasoning *locally*, about states and their successors, instead of *globally*, about infinite paths. Our soundness and completeness theorems differ from the Abadi and Lamport theorem in that there are no restrictions on the types of systems considered and there is no need for auxiliary variables. We now give an overview of why this is the case, by discussing some of the key differences between the two approaches.

The most glaring difference is that our notions of correctness are based on the branching-time framework, whereas the work of Abadi and Lamport is based on the linear-time framework. The reason we chose branching-time notions is that refinement proofs are inductive. One shows that an implementation state and the specification state that it is related to by a refinement map have related successors. As was pointed out in section 2.7 on page 24, simulation and bisimulation are structural properties, whereas linear-time notions such as trace containment are not and require global analysis. As a consequence, we do not need prophecy variables to prove soundness and completeness. Consider the example given by Abadi and Lamport to motivate the need for prophecy variables. System $\mathcal{S}$ chooses ten values non-deterministically and displays each in turn, whereas system $\mathcal{I}$ chooses each value as it is displayed. From the linear-time point of view $\mathcal{I}$ refines $\mathcal{S}$ since they have the same traces, but since proofs using refinement maps are structural and local, there is no refinement map that can be used to show this. This is one reason for introducing prophecy variables and they are used to resolve the dilemma as follows. A prophecy variable is added to $\mathcal{I}$ and the variable "guesses" what $\mathcal{I}$ will decide to do in the future. There is now a refinement map, based on this prophecy variable, that can be used to show that $\mathcal{I}$ implements $\mathcal{S}$. What is happening is that the prophecy variables allow one to push all of the branching in the computation tree of $\mathcal{I}$ up to the root, thereby destroying the branching structure of $\mathcal{I}$. In light of this, it is no surprise that we cannot show that $\mathcal{I}$ refines $\mathcal{S}$, because from the branching point of view, it does not. More specifically, from the initial state in $\mathcal{I}$, there is a successor that has more than one possible future, a branching-time expressible property that does not hold in the initial state of $\mathcal{S}$.

Another difference is that in the Abadi and Lamport approach, a refinement map, $r : S_1 \rightarrow S_2$ has to satisfy the following condition. For all $s, u \in S_1$ such that u is a successor of s, we have that $r.u$ is a successor of $r.s$ or $r.s = r.u$. In our case, the analogous condition is: in the disjoint union of the two systems, s and $r.s$ are related (*e.g.*, see definitions 7 and 14). Our formulation allows us to avoid the use of history variables, which are needed in the Abadi and Lamport approach. Consider the example given by Abadi and Lamport to motivate the need for history variables. System $\mathcal{S}$ is a three-bit clock, where only the low-order bit is visible and system $\mathcal{I}$ is a one-bit clock. $\mathcal{I}$ refines $\mathcal{S}$ since they have the same traces (up to stuttering), but there is no refinement map that can be used to show this because there is no way to define the internal state of $\mathcal{S}$ in a way that satisfies the above condition on refinement maps. This is one reason for introducing history variables and they are used to resolve the dilemma as follows. A history variable is added to $\mathcal{I}$ and the variable "remembers" what $\mathcal{I}$ did in the past. The result is that the state space of $\mathcal{I}$ is expanded so that there are enough states to define an appropriate refinement map. In our case, it should be clear that we do not need history variables. We can define a refinement map that maps the state in $\mathcal{I}$ whose counter is 0 to any state in $\mathcal{S}$ whose low-order bit is 0 and similarly with the other state in $\mathcal{I}$. The equivalence relation that relates states with the same low-order bit in the disjoint union of the two systems is a stuttering bisimulation, hence, the relation is also a stuttering simulation.

There is a third example given by Abadi and Lamport that shows why a prophecy variable is needed to slow down an implementation that runs faster than a specification, even though the specification is just stuttering. This is again due to the same condition discussed in the previous paragraph: a refinement map, $r : S_1 \to S_2$ has to satisfy the following: for all $s, u \in S_1$ such that u is a successor of s, we have that $r.u$ is a successor of $r.s$ or $r.s = r.u$. From this condition, one can show that any path in S_1 has to be matched, under r, by a path in S_2, where we can add, but not remove, stuttering steps. In our formulation, for the same reasons outlined in the previous paragraph, we can both add and remove stuttering steps.

A minor difference is that Lamport and Abadi require that systems have the same externally visible states. They make the point that one cannot say whether the value 11111100 corresponds to -3 without knowing how to interpret a sequence of bits as an integer. They go on to say that given such an interpretation, they can translate the externally visible states to the appropriate representation. In our case, instead of having a separate interpretation phase, we allow refinement maps to alter the labels of states directly.

Abadi and Lamport define a specification (what we call a system) as a state machine together with a supplementary property (usually a liveness property) and require *machine closure*: that the supplementary property does not imply any safety property not already implied by the state machine. In our case, we have a single system that includes both safety and liveness properties.

Finally, our theorems seem stronger than the ones given by Abadi and Lamport. For example, they show that even when $\mathcal{S}$ is not internally continuous a refinement map exists to show that $\mathcal{I}$ satisfies the safety property specified by $\mathcal{S}$. They continue "We do not know if anything can be said about proving arbitrary liveness properties." Since our refinement theorems apply to any systems, a simple corollary is that, with our approach, refinement maps can always be used to prove both safety and liveness properties. This is something that we use several times in the sequel, *e.g.*, we show that one can use theorem proving to reduce an infinite-state system to a finite-state system in such a way that stuttering-insensitive properties, including liveness, are preserved. We then model check the reduced system and can lift the results to the original system. The details are outlined later.

5.7 Remarks on Performance

The notions of correctness presented in this part do not address performance. Even though performance is extremely important, there are good reasons for this omission. The main reason is that correctness and performance can be treated separately, thus, it makes sense to separate concerns. Having separated concerns, we find that there are enough interesting problems associated with correctness that, for now, we ignore performance. Another reason is that it is sometimes difficult to reason about performance formally, *e.g.*, in the case of microprocessor performance analysis, the clock rate is a very poor indicator of performance. Instead, "typical" workloads are used. What is typical today might not be typical tomorrow and perhaps the best way of dealing with this kind of performance analysis is to run experiments. Other types of performance analysis are possible. For example, if we are given a cost model, then it is possible to reason mechanically about algorithmic complexity.

5.8 Bibliographic Notes

The process algebra notions of simulation [Mil71] and bisimulation, [Par81, Mil90] have turned out to be of fundamental importance. There are various notions of bisimulation equivalence, *e.g.*, strong bisimulation [Par81, Mil90], weak bisimulation [Mil90], stuttering bisimulation [BCG88], branching bisimulation[vGW96], and many others [Gla01]. We chose to use stuttering simulation and bisimulation as opposed to weak bisimulation because weak bisimulation allows infinite stuttering and we do not consider implementations that allow divergence, where none is present in the specification, to be correct.

Brown, Clarke, and Grumberg describe an algorithm for stuttering bisimulation that for TS with n states has time complexity $O(n^5)$ [BCG88]. This was improved by Groote and Vaandrager who give an $O(m \cdot n)$ algorithm [GV90], where m is the number of edges, *i.e.*, the size of the transition relation.

Our proofs rules for stuttering bisimulation are based on a proof rule by Namjoshi [Nam97]. Namjoshi gives a sound and complete proof rule for symmetric stuttering bisimulations. Besides developing the theory of stuttering simulations, our work extends and simplifies Namjoshi's work as follows. First, we consider more general definitions. For example, we remove the restriction that stuttering bisimulations are symmetric and we do not require transition systems to be countably branching. In addition, we prove that the reflexive, symmetric, transitive closure of a stuttering bisimulation is again a stuttering bisimulation. As a consequence, nothing essential is lost in requiring stuttering bisimulations to be equivalence relations, but simpler proofs rules are now possible. For example, our definition of WEB contains rank functions with less arguments than previously required, but is still a sound and complete proof rule. This requires different constructions that shed further light on the structure of stuttering bisimulations. The practical consequence of this analysis is that it allowed us to construct ACL2 libraries that are used to more fully automate mechanical verification. How this is done is described in chapter 12.

Most of the results presented in the last two chapters were first proved for bisimulations. We later realized that simulations are important, as bisimulation is often too strong a notion. In addition, starting with simulations led to cleaner proofs and a more streamlined presentation. Mechanical verification efforts based on stuttering bisimulation are reported in [MNS99, Man00a, Man00d, Sum00]. The case studies in this dissertation are based on the first three references. The work by Sumners is described in the bibliographic notes section of the previous chapter (page 50).

Reasoning about simulations and bisimulations can be tricky. For example, Basten shows that the "simple proof" claimed for the folklore result stating that branching bisimulation (a type of bisimulation closely related to stuttering bisimulation) is an equivalence relation is wrong [Bas96].

5.9 Summary

This chapter was devoted to stuttering bisimulations. We proved that stuttering bisimulations enjoy several algebraic properties. For example, they are closed under arbitrary union and relational composition. We presented sound and complete proof rules that allow us to prove a stuttering bisimulation by reasoning about single transitions. When stuttering bisimulations are equivalence relations, further simplification is possible; this was explored in the chapter. Finally, we introduced the notion of refinement for stuttering bisimulations and showed that this notion is compositional, *i.e.*, refinement proofs can be decomposed into a sequence of simpler refinement proofs.

Part III

Combining Theorem Proving and Model Checking

Chapter 6

Introduction

In this part we present a novel way of combining theorem proving and model checking. To prove that a system satisfies its specification, given as a set of temporal logic properties, we first use theorem proving to show a WEB on the system. Another way of saying this is that theorem proving is employed to prove the correctness of an abstraction that yields a reduced system. The reduced system can then be model checked. This is a very general abstraction technique that allows great flexibility in choosing an appropriate abstraction. If more theorem proving effort is applied, more reduction is possible. At one extreme, all the work is done with the theorem prover and at the other, all the work is done with the model checker. The right balance between theorem proving and model checking depends on the problem at hand. The general idea is to reduce the problem to one that can be model checked in a reasonable amount of time. Since we often apply this technique to infinite-state systems, some amount of theorem proving is required.

We now examine this idea in more detail. The definition of WEB allows us to carry out the theorem proving task by reasoning only about single steps of the system, a considerable simplification. A WEB induces a quotient structure that is equivalent (up to stuttering) with the original system. Quotient structures can be much smaller than the original: a bisimulation with finitely many classes induces a finite quotient (of a possibly infinite-state system). The idea is to check the quotient structure, but constructing the quotient structure can be difficult because determining if there is a transition between states in the quotient structure depends on whether there is a transition between some pair of related states in the original system (the number of such pairs may be infinite). Moreover, the quotient structure may be infinite-state, but the set of its reachable states may be finite. To address these two concerns, we introduce an on-the-fly procedure that automatically extracts the quotient structure, if the set of reachable states is finite. Once the quotient structure is extracted, we model check it using a model checker for the Mu-Calculus written in ACL2.

In chapter 7 we present two quotient extraction procedures. The first one is based on state representative functions: functions that assign a state to each equivalence class in a WEB. The second quotient extraction procedure is based on set representative functions: functions that assign a set to each equivalence class in a WEB. State representative functions are simpler than set representative functions, thus, when applicable, they are preferable. However, it is possible that a WEB induces a finite quotient structure, but there is no state representative function that can be used to extract it. Using set representative functions, we prove that for any WEB that induces a finite quotient, there is a set representative function that can be used to extract it.

In chapter 8, we give an informal and quick overview of ACL2. ACL2 is explained in a textbook by Kaufmann, Manolios, and Moore [KMM00b] and there is extensive online documentation available [KM]. We include only enough to make this dissertation self-contained.

In chapter 9, we show how to embed the Mu-Calculus into ACL2. We give the syntax and semantics of the Mu-Calculus in ACL2. The result is that we have an executable model checker that we use to model check the quotients, as extracted by our extraction procedure.

Chapter 7

Quotient Extraction

7.1 State Representative Functions

Definition 15 *(State Representative Function) Let $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ be a TS and let B be a WEB on $\mathcal{M}$, with well-founded witness $\langle rankt, \langle W, \prec \rangle, rankl \rangle$. Let $rep : S \rightarrow S$; then rep is a state representative function for $\mathcal{M}$ with respect to B if for all $s, w \in S$ all of the following hold.*

1. $sBw \quad \equiv \quad rep.s = rep.w$
2. $rep(rep.s) = rep.s$
3. $rankt(rep.s) \preccurlyeq rankt.s$
4. $rankl(rep.s, w) \leq rankl(s, w)$

In this and the next section, by "condition i", we mean condition i in the above definition. Notice that the function *rep* can be used to define B, the WEB, because condition 1 can be thought of as defining B from *rep*.

From conditions 1 and 2 we have that $sBrep.s$, *i.e.*, $rep.s$ is an element of the equivalence class of s in the induced quotient structure. Conditions 3 and 4 are "minimality" conditions which, as will be shown shortly, guarantee that the *rep* of a state has all of the branching behaviors of its class.

Theorem 12 *Let rep be a state representative function for TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ with respect to WEB B. Let $\mathcal{M}' = \langle S', \rightrightarrows, L'\rangle$, where*

1. *$S' = rep(S)$; and*

2. *$L' = L|_{S'}$; and*

3. *$s \rightrightarrows u$ iff $\langle \exists v : s \dashrightarrow v : rep.v = u\rangle$.*

Then $\mathcal{M}'$ is $\mathcal{M}/B$, up to a renaming of states.

Proof We start by defining f, a function from states in $\mathcal{M}'$ to states in $\mathcal{M}/B = \langle \mathcal{S}, \rightsquigarrow, \mathcal{L}\rangle$, as follows: $f(rep.s) = [s]$.[1] From conditions 1 and 2 we have that $sBrep.s$ and it then follows that f is a bijection between the states of $\mathcal{M}'$ and $\mathcal{M}/B$ that respects the labeling functions. We now show that f respects the transition relations, *i.e.*, $c \rightrightarrows d \quad \Rightarrow \quad f.c \rightsquigarrow f.d$ and $p \rightsquigarrow q \quad \Rightarrow \quad f^{-1}(p) \rightrightarrows f^{-1}(q)$.

If $c \neq d$ and $c \rightrightarrows d$, then by the definition of $\rightrightarrows$, there is a w such that $c \dashrightarrow w$ and wBd. Since $c, w \in S$ and $c \dashrightarrow w$, by the definition of the quotient, $[c] \rightsquigarrow [w]$; but $[w] = [d]$, thus, $[c] \rightsquigarrow [d]$. We now have: $(c \neq d \ \wedge \ c \rightrightarrows d) \quad \Rightarrow \quad f.c \rightsquigarrow f.d$.

If $c \rightrightarrows c$, then by the definition of $\rightrightarrows$, there is a u such that $c \dashrightarrow u$ and uBc. For any s such that cBs, Web3 must hold. If Web3a or Web3c holds,

[1]In this chapter, in order to simplify notation, we will write $[s]$ instead of $[s]_B$, as it is clear from the context what B, the relation omitted, is.

then s has a successor in $[c]$ and by the definition of the quotient, $[c] \rightsquigarrow [c]$. Web3b does not hold because if it did, we would have $\mathit{rankt}.u \prec \mathit{rankt}.c$, which violates condition 3. Combining this with the previous result gives: $c \rightrightarrows d \quad \Rightarrow \quad f.c \rightsquigarrow f.d$.

If $p \neq q$ and $p \rightsquigarrow q$, then by the definition of quotient, there is an s and a u such that $[s] = p$, $[u] = q$, and $s \dashrightarrow u$. Since $sB\mathit{rep}.s$, by the definition of WEB, Web3 holds for $s, u, \mathit{rep}.s$. Web3b does not hold as $\neg(uB\mathit{rep}.s)$. Web3c does not hold as this violates condition 4. Thus, Web3a holds, *i.e.*, $\langle \exists v : \mathit{rep}.s \dashrightarrow v : uBv \rangle$. By transitivity, $\neg(\mathit{rep}.sBv)$; by the definition of $\rightrightarrows$, $\mathit{rep}.s \rightrightarrows \mathit{rep}.u$, but $\mathit{rep}.s = f^{-1}(p)$ and $\mathit{rep}.u = f^{-1}(q)$. We now have $(p \neq q \quad \wedge \quad p \rightsquigarrow q) \quad \Rightarrow \quad f^{-1}(p) \rightrightarrows f^{-1}(q)$.

If $p \rightsquigarrow p$, then by the definition of quotient, for any s such that $[s] = p$, there is a u such that $[u] = p$ and $s \dashrightarrow u$; since $[\mathit{rep}.s] = p$, $\langle \exists u : \mathit{rep}.s \dashrightarrow u \quad \wedge \quad sBu \rangle$; by the definition of $\rightrightarrows$, $\mathit{rep}.s \rightrightarrows \mathit{rep}.s$. Combining this with the previous result gives: $p \rightsquigarrow q \quad \Rightarrow \quad f^{-1}(p) \rightrightarrows f^{-1}(q)$. $\square$

Note that in systems where all states have self-loops, $\mathcal{M}/B$ and $\mathcal{M}'$ will have self-loops; hence, condition 3 is not required.

State representative functions are very useful (when they exist) because they identify states that have all of the branching behavior of their class. They allow one to view the quotient as a submodel of the original structure, and they are used in the on-the-fly procedure for constructing quotient structures defined in figure 7.1.

$S',\ \rightrightarrows,\ open\ :=\ rep(I),\ \emptyset,\ rep(I)$
while $open \neq \emptyset$
    choose $s \in open$
    $open\ :=\ open \setminus \{s\}$
    for $u \in rep(next.s)$
        if $u \notin S'$ then $open\ :=\ open \cup \{u\}$
        $S',\ \rightrightarrows\ :=\ S' \cup \{u\},\ \rightrightarrows \cup \{\langle s, u\rangle\}$

Figure 7.1: Procedure for extracting quotient structures using state representative functions.

7.2 Quotient Extraction for State Representative Functions

Theorem 13 *Let rep be a state representative for TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ with respect to WEB B. If*

1. *$rep(I)$ is finite and computable, where I is a set of initial states; and*
2. *$\mathcal{M}/B$ has a finite number of reachable states from $\{[i] : i \in I\}$; and*
3. *for all $s \in S$ reachable from I, $rep(next.s)$ is finite and computable, where $next.s$ is $\{u : s \dashrightarrow u\}$*

then the procedure in figure 7.1 will construct a TS isomorphic to $\mathcal{M}/B$, restricted to states reachable from $\{[i] : i \in I\}$.

Proof We explore the set of states reachable from $rep(I)$ using the the procedure in figure 7.1.

The variables *open* and S' contain a set of representative states. Exactly the states that correspond to reachable equivalence classes in $\mathcal{M}/B$ are generated. We can show this by induction: the set of generated states includes the initial states (line 0) and is closed under successors (line 4). Since there are a finite number of reachable representatives and the while loop of line 1 is taken at most that many times, the procedure terminates. Since the three conditions of theorem 12 are satisfied, the above procedure returns the reachable part of $\mathcal{M}'$, which by theorem 12 is the reachable part of $\mathcal{M}/B$, up to a renaming of states. $\Box$

We now analyze the complexity of the above procedure. One of the difficulties is that it is not clear what a reasonable cost model is. Specifically, we do not know the cost of computing $rep(A)$, where A is a set to which rep is applied above; all we know is that $rep(A)$ is finite and computable. We therefore consider two possibilities. The first is that the cost of $rep(A)$ (this includes computing A and $rep(A)$) is $O(\#(rep(A)))$. In this case, the time complexity of the above procedure is $O(m)$, where m is the number of edges in the quotient structure.

Lemma 27 *Under the cost model above, the procedure in figure 7.1 has (linear) time complexity $O(m + n) = O(m)$, on the average, where n is the number of reachable states in the quotient structure and m is the number of transitions (edges) in the reachable part of the quotient structure.*

Proof Since a transition system is left-total, $n \leq m$. Line 0 requires $O(n)$ time since $\#(rep(I)) \leq n$. The variable *open* can be implemented as a linked list with a pointer to the last item as an auxiliary field. This makes it possible to perform the operations on *open* ($\cup$ and $\setminus$) in $O(1)$ time. S' is implemented as a hash table, which makes it possible to perform the search and insert operations in $O(1)$ average time. Also, operations on $\rightrightarrows$ (union) can be implemented in

$O(1)$ time. The while loop in line 1 is executed once per node (*i.e.*, n times) and the for loop is executed once per edge (*i.e.*, m times). Since $n \leq m$, the time complexity is $O(m)$. $\square$

For the second cost model, we assume that $\#A$ is finite for any A to which *rep* is applied (as would be the case if states in $\mathcal{M}$ are finite branching) and that the cost of *rep.s* is constant. Under this, more realistic, assumption, the time complexity of the procedure is given by the following lemma.

Lemma 28 *Let R be the image under rep of the set of reachable states in $\mathcal{M}$. Under the cost model above, the procedure in figure 7.1 has time complexity $O(\#I + \sum_{\{r \in R\}} \#(next.r))$ on the average.*

Proof Line 0 requires $\#I$ time, as *rep* is applied to each element in I. The variables *open*, S', and $\rightrightarrows$ are implemented as in the proof of lemma 27. The while loop in line 1 is executed once per state for each element r in R and the inner for loop is executed at most once per element in $next.r$, thus the time complexity is $O(\#I + \sum_{\{r \in R\}} \#(next.r))$ on the average. $\square$

In the (we expect common) case where $\#I$ is a constant and the branching factor of any state is at most b (as is the case when a transition system is given by a set of rules that when applied to a state determine its successors), then by the above lemma, we have time complexity, $O(n \cdot b)$, where n is the number of reachable states in the quotient structure. This follows from the observation that $\#R = n$ and $\#(next.r) \leq b$.

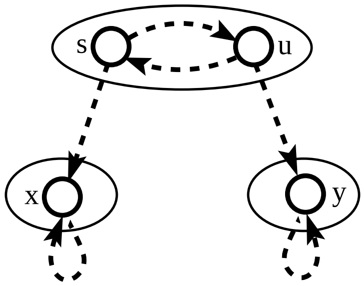

Figure 7.2: The graph denotes a transition system, where circles denote states, ovals denote equivalence classes, and the transition relation is denoted by a dashed line. Notice that neither state s nor state u can be the representative of their class because neither state has all the branching behaviors of its class.

7.3 Set Representative Functions

It is possible to define a WEB for which there is no state representative. The reason is that there may be an equivalence class such that no element of the class has all of the behaviors of the class. This is the case for the simple transition system in figure 7.2. In the figure, circles denote states, ovals denote equivalence classes, and the transition relation is denoted by a dashed line. To see that the partitioning of states results in a WEB, consider the well-founded witness where *rankt* applied to any state is 0 and the *rankl* of any pair is 0, except that both $rankl(s, y)$ and $rankl(u, x)$ are 1. Thus, neither s nor u can be the *rep* of the class, as in either case condition 4 of the definition of state representative is violated. The intuition is that neither s nor u has all the branching behaviors of their class. We introduce a more general type of representative function and show that there is an extraction method which can be used with any WEB that induces a finite quotient.

Definition 16 *(Set Representative Function) Let* $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ *be a TS and let* B *be a WEB on* $\mathcal{M}$*, with well-founded witness* $\langle rankt, \langle W, \prec \rangle, rankl \rangle$*. Let* $rep : S \to \mathcal{P}(S)$*; then* rep *is a set representative function for* $\mathcal{M}$ *with respect to* B *if for all* $s, w \in S$ *all of the following hold.*

1. $sBw \quad \equiv \quad rep.s = rep.w$
2. $\langle \forall u \in rep.s :: sBu \rangle$
3. $\langle \exists u \in rep.s :: rankt.u \preccurlyeq rankt.s \rangle$
4. $\langle \exists u \in rep.s :: rankl(u, w) \leq rankl(s, w) \rangle$

In this and the next section, by "condition i", we mean condition i in the above definition.

Lemma 29 $u \in rep.s \quad \Rightarrow \quad rep.u = rep.s$

Proof $rep.u = rep.s$ follows from uBs by condition 1, which follows from $u \in rep.s$, by condition 2. $\square$

Theorem 14 *Let rep be a set representative function for TS* $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ *with respect to WEB* B*. Let* $\mathcal{M}' = \langle S', \rightrightarrows, L' \rangle$*, where:*

1. $S' = rep(S)$*; and*
2. $L'.C = L.s$*, for some* s *in* C *(equivalent states have the same label); and*
3. *The transition relation is given by: For* $C, D \in S'$*,* $C \rightrightarrows D$ *iff either*
   (a) $C \neq D$ *and* $\langle \exists s, w : s \in C \wedge rep.w = D : s \dashrightarrow w \rangle$*, or*
   (b) $C = D$ *and* $\langle \forall s \in C :: \langle \exists w : rep.w = C : s \dashrightarrow w \rangle \rangle$*.*

Then $\mathcal{M}'$ is $\mathcal{M}/B$, up to a renaming of states.

Proof We start by defining f, a function from states in $\mathcal{M}'$ to states in $\mathcal{M}/B = \langle \mathcal{S}, \rightsquigarrow, \mathcal{L} \rangle$, as follows: $f(rep.s) = [s]$. f is a bijection between the states of $\mathcal{M}'$ and $\mathcal{M}/B$ as $([s] = [u] \quad \equiv \quad sBu) \wedge (rep.s = rep.u \quad \equiv \quad sBu)$. In addition, f respects the labeling functions, *i.e.*, the label of $rep.s$ in $\mathcal{M}'$ equals the label of s, as $rep.s$ is non-empty (condition 3 of the definition of set representative function) and the labels of the states in $rep.s$ equal the label of s; the label of $[s]$ in $\mathcal{M}/B$ also equals the label of s, by the definition of quotient structures. We now show that f respects the transition relations, *i.e.*, $c \rightrightarrows d \quad \Rightarrow \quad f.c \rightsquigarrow f.d$ and $p \rightsquigarrow q \quad \Rightarrow \quad f^{-1}(p) \rightrightarrows f^{-1}(q)$.

If $c \neq d$ and $c \rightrightarrows d$, then by the definition of $\rightrightarrows$, there is an $s \in c$ such that $s \dashrightarrow u$ and $rep.u = d$. Since $s, u \in S$ and $s \dashrightarrow u$, by the definition of the quotient, $[s] \rightsquigarrow [u]$. Now $c = rep.s$ by lemma 29, thus $f.c = f(rep.s) = [s]$. Also, $f.d = f(rep.u) = [u]$ and we have $(c \neq d \quad \wedge \quad c \rightrightarrows d) \quad \Rightarrow \quad f.c \rightsquigarrow f.d$.

If $rep.s \rightrightarrows rep.s$, then choose y such that $y \in [s]$ and $\langle \forall x \in [s] :: rankt.y \preccurlyeq rankt.x \rangle$. Since $\prec$ is well-founded this is possible. By condition 3 (of the definition of set representative functions), we can choose $b \in rep.s$ such that $rankt.b \preccurlyeq rankt.y$. Now, let $a \in [s]$. We show that a has a successor in $[s]$. Since bBa and $b \dashrightarrow z$ for some $z \in [s]$, Web3 holds. If Web3a or Web3c holds, then a has a successor in $[s]$ and by the definition of the quotient, $[s] \rightsquigarrow [s]$. Web3b does not hold because if it did, we would have $rankt.z \prec rankt.b$, and since $rankt.b \preccurlyeq rankt.y$, $rankt.z \prec rankt.y$, contradicting our choice of y. Combining this with the previous result gives: $c \rightrightarrows d \quad \Rightarrow \quad f.c \rightsquigarrow f.d$.

If $p \neq q$ and $p \rightsquigarrow q$, then by the definition of quotient, there exist a and b such that $[a] = p$, $[b] = q$, and $a \dashrightarrow b$. Choose $x \in [a]$ such that

$\langle \forall y \in [a] :: rankl(x,b) \leq rankl(y,b) \rangle$. Since $<$ is well-founded, this is possible. By condition 4, there is a $u \in rep.a$ such that $rankl(u,b) \leq rankl(x,b)$. We therefore have that $\langle \forall y \in [a] :: rankl(u,b) \leq rankl(y,b) \rangle$. Since aBu, by the definition of WEB, Web3 holds for a, b, u. If Web3a holds then u has a successor in $[b]$. Web3b does not hold as $[b] \neq [u]$. Web3c does not hold as otherwise, u has a successor, say v, such that uBv and $rankl(v,b) < rankl(u,b)$, contradicting the minimality of $rankl(u,b)$. We thus have $(p \neq q \quad \wedge \quad p \rightsquigarrow q) \quad \Rightarrow \quad f^{-1}(p) \rightrightarrows f^{-1}(q)$.

If $[s] \rightsquigarrow [s]$, then by the definition of quotient, $\langle \forall x \in [s] :: \langle \exists y \in [s] :: x \dashrightarrow y \rangle \rangle$ which implies $\langle \forall x \in rep.s :: \langle \exists y : rep.y = rep.s : x \dashrightarrow y \rangle \rangle$ as $x \in rep.s \quad \Rightarrow \quad x \in [s]$, and $y \in [s] \quad \equiv \quad rep.y = rep.s$. Combining this with the previous result gives: $p \rightsquigarrow q \quad \Rightarrow \quad f^{-1}(p) \rightrightarrows f^{-1}(q)$. $\Box$

In light of the above theorem, we can view set representative functions as a generalization of quotient structures. If we define $rep.s = [s]$, then we get the quotient structure. If $\#(rep.s) = 1$ for all s, then, in essence, we have a state representative. However, it may be possible to define rep such that $rep.s \subset [s]$; it may even be possible to define rep such that $\#(rep.s) < \omega$, in which case, we may be able to use the definition of $\mathcal{M}'$ to construct the quotient automatically. In fact, this definition allows us to prove a completeness result: for any WEB that induces a finite quotient, it is possible to define a representative function that can be used to automatically extract the quotient. As before, only the reachable part of the quotient needs to be finite.

7.4 Quotient Extraction for Set Representative Functions

A set s is *hereditarily finite* if s has a finite number of elements and each set in s is hereditarily finite.

Theorem 15 *Let rep be a set representative for TS $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ with respect to WEB B. If*

1. *$rep(I)$ is hereditarily finite and computable, where I is a set of initial states; and*
2. *$\mathcal{M}/B$ has a finite number of reachable states from $\{[i] : i \in I\}$; and*
3. *for all $s \in S$ reachable from I, $rep(next.s)$ is hereditarily finite and computable, where $next.s$ is $\{u : s \dashrightarrow u\}$*

then the procedure in figure 7.3 will construct a TS isomorphic to $\mathcal{M}/B$, restricted to states reachable from $\{[i] : i \in I\}$.

Proof We explore the set of states reachable from $rep(I)$ using the procedure in figure 7.3.

The variables *open* and S' contain a set of representative state sets. Exactly the states in $\mathcal{M}'$ (which are sets of states in $\mathcal{M}$) that correspond to reachable equivalence classes in $\mathcal{M}/B$ are generated. We can show this by induction: the set of generated states includes the initial states (line 0) and is closed under successors (line 6). There are a finite number of reachable representatives and the while loop on line 1 is taken that many times. In addition, the two for loops on lines 4 and 6 are bounded as $rep(I)$ and $rep(next.w)$ are

```
0   S′, ⇉, open := rep(I), ∅, rep(I)
1   while open ≠ ∅
2       choose srep ∈ open
3       open, sloop := open \{ srep }, true
4       for w ∈ srep
5           wloop := false
6           for urep ∈ rep(next.w)
7               if urep ∉ S′ then open := open ∪{ urep }
8               if urep = srep
9                   then wloop := true
10                  else S′, ⇉ := S′ ∪ { urep }, ⇉ ∪{⟨ srep, urep ⟩}
11          if (¬ wloop) then sloop := false
12      if sloop then ⇉ := ⇉ ∪{⟨ srep, srep ⟩}
```

Figure 7.3: Procedure for extracting quotient structures using set representative functions.

hereditarily finite. Thus, the procedure terminates. We have made sure to define the transition relation as prescribed by condition 3 of theorem 14, thus the above procedure computes the reachable part of $\mathcal{M}'$. By theorem 14, this is the reachable part of $\mathcal{M}/B$, up to a renaming of states. $\square$

Notice that all we really needed for the proof of theorem 15 is that *rep* is defined on reachable states, as the above procedure does not attempt to evaluate *rep* on unreachable states.

We now analyze the complexity of the above procedure. We assume that the cost of $rep.s$ is $O(\#(rep.s))$. This is a reasonable cost model, as we charge for each element in $rep.s$. We also assume that *rep* is applied only to finite sets, *i.e.*, I is finite as is $next.w$ on line 6 of the procedure. We use the same data structures as before to represent S', *open*, and $\rightrightarrows$. The time complexity of the procedure is given by the following lemma.

Lemma 30 *Let R be the image under rep of the set of reachable states in $\mathcal{M}$. Under the cost model outlined above, the procedure in figure 7.3 has time complexity* $O([\sum_{\{i \in I\}} \#(rep.i)] + [\sum_{\{r \in R\}} \sum_{\{w \in r\}} \sum_{\{u \in next.w\}} \#(rep.u)])$

Proof Line 0 requires $O(\sum_{\{i \in I\}} \#(rep.i))$ time as *rep* is applied to each element in I. The variables *open*, S', and $\rightrightarrows$ are implemented as in the proof of lemma 27. The <u>while</u> loop in line 1 is executed once for each element r in R; the <u>for</u> loop on line 4 is executed once for each element w of r; the <u>for</u> loop on line 6 is executed at most once for each element u of $next.w$ and has time complexity $O(\#(rep.u))$ under our cost model; thus, the total time complexity is $O([\sum_{\{i \in I\}} \#(rep.i)] + [\sum_{\{r \in R\}} \sum_{\{w \in r\}} \sum_{\{u \in next.w\}} \#(rep.u)])$ on the average. $\Box$

In the (we expect common) case where $\#I$ is a constant, the branching factor of any state is at most b (as is the case when a transition system is given by a set of rules that when applied to a state determine its successors), and the size of a representative is p, then by the above lemma, we have time complexity, $O(n \cdot b \cdot p^2)$, where n is the number of reachable states in the quotient structure. This follows from the observation that $\#R = n$, $\#(next.w) \leq b$, and $\#(rep.u) = \#r = p$. Notice that if $p = 1$, then the complexity is $O(n \cdot b))$, which matches the complexity of the state representative procedure.

We now show a completeness result for set representative functions. For any WEB that induces a finite quotient, there is a representative function that can be used to extract it.

Theorem 16 *Let B be a WEB for $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ with well-founded witness $\langle rankt, \langle \alpha, \prec\rangle, rankl\rangle$ such that for all $s \in S$, $next.s$ is finite and computable and $\mathcal{M}/B$ has a finite number of reachable states. Then it is possible to define* *a set representative function for $\mathcal{M}$ with respect to B such that the extraction procedure in figure 7.3 automatically extracts the reachable part of the quotient.*

Proof We show that the conditions for theorem 15 can be satisfied, thus the procedure in figure 7.3 can be used to extract the reachable part of the quotient. Let c be a reachable class in $\mathcal{M}/B$ and define $Next.c = \{e : c \rightsquigarrow e \text{ in } \mathcal{M}/B\}$. For any $s \in c$, we define $rep.s = \{rmin(c,e) : e \in Next.c\}$, where $rmin(c,c)$ is a state $u \in c$ such that $\langle \forall w \in c :: rankt.u \preccurlyeq rankt.w \rangle$ and otherwise (*i.e.*, $e \neq c$) $rmin(c,e)$ is a state $u \in c$ such that $\langle \exists w \in e : u \dashrightarrow w : \langle \forall x, y : x \in c \;\;\wedge\;\; y \in e : rankl(u,w) \leq rankl(x,y) \rangle\rangle$. This defines a set representative function, as conditions 1–4 of the definition of a set representative function are now easily verified. Furthermore, the representative function is hereditarily finite as $\#(rep.s) \leq \#(Next.[s])$, which is bounded by the number of reachable states in $\mathcal{M}/B$, which by assumption, is finite. Finally, by assumption, for all $s \in S$, $next.s$ is finite and computable, thus, $rep(next.s)$ is hereditarily finite and computable and the conditions on theorem 15 are satisfied.

We sometimes refer to the two extraction procedures as "algorithms" in the dissertation. Although this is a slight abuse of notation, in this chapter we made clear under what conditions they terminate.

7.5 Bibliographic Notes

A sketch of some of the results in this chapter appears in [MNS99]. Related work includes the approach by Havelund and Shankar [HS96]. See the bibliographic notes section of chapter 11 for further references.

7.6 Summary

In this chapter, an approach to verification combining the strengths of model checking and theorem proving was presented. An overview of the approach follows. A theorem prover is used to show a WEB (well-founded bisimulation) on a (potentially infinite-state) system. The WEB induces a quotient structure that has the same $\mathrm{CTL}^* \setminus \mathsf{X}$ properties as the original system. An on-the-fly method can be used to extract the (reachable part of the) quotient induced by the WEB (for finite quotients).

Chapter 8

ACL2

The ACL2 system consists of a programming language, a logic, and a theorem prover. ACL2 was developed by Kaufmann and Moore [KM] and the sources are freely available on the Web, under the GNU General Public License. The ACL2 homepage is `http://www.cs.utexas.edu/users/moore/acl2`. Extensive documentation, including tutorials, a user's manual, and related papers are available from the ACL2 homepage. ACL2 is also described in a textbook by Kaufmann, Manolios, and Moore [KMM00b]. There is also a book of case studies [KMM00a]. Supplementary material for both books, including all the solutions to the exercises (over 200 in total) can be found on the Web [KMM00d, KMM00c]. In this chapter we give an informal overview of the ACL2 system, in order to make this dissertation more self-contained.

ACL2, the language, is an applicative, or purely functional programming language. One consequence is that the rule of Leibniz, *i.e.*, $x = y \Rightarrow f.x = f.y$, written in ACL2 as `(implies (equal x y) (equal (f x) (f y)))`, is a theorem. This effectively rules out side effects. The ACL2 data types and expressions are presented in sections 8.1 and 8.2, respectively. ACL2 code can be made to execute efficiently. One way is to compile ACL2 code, which can be done with any Common Lisp [Ste90] compiler. Another way is to use stobjs, single-threaded objects. Logically, stobjs have applicative semantics, but syntactic restrictions on their use allow ACL2 to produce code that *destructively* modifies stobjs. Stobjs have been very useful when efficiency is paramount, as is the case when modeling complicated computing systems such as microprocessors. For example, Hardin, Wilding, and Greve compare the speeds of a C model and an ACL2 model of the JEM1 (a silicon Java Virtual Machine designed by Rockwell Collins). They found that the ACL2 model runs at about 90% of the speed of the C model [HWG98].

ACL2, the logic, is a first-order logic. The logic can be extended with *events*; examples of events are function definitions, constant definitions, macro definitions, and theorems. Logically speaking, function definitions introduce new axioms. Since new axioms can easily render the theory unsound, ACL2 has a definitional principle which limits the kinds of functions one can define. For example, the definitional principle guarantees that functions are *total*, *i.e.*, that they terminate. In section 8.3.1 we discuss the issues. ACL2 also has macros, which allow one to customize the syntax. Macros are described in section 8.3.2. In section 8.4.1 we describe encapsulation, a mechanism for introducing constrained functions, functions that satisfy certain constraints, but that are otherwise undefined. We end by describing books in section 8.4.2. Books are files of events that often contain libraries of theorems and can be loaded by ACL2 quickly, without having to prove theorems.

ACL2, the theorem prover is described in a textbook written by Kaufmann, Manolios, and Moore [KMM00b]. In this dissertation, we do not explore the details of the theorem prover.

8.1 Data Types

The ACL2 universe consists of *atoms* and *conses*. Atoms are atomic objects and include the following.

1. *Numbers* includes integers, rationals, and complex rationals. Examples include `-1`, `3/2`, and `#c(-1 2)`.

2. *Characters* represent the ASCII characters. Examples include `#\2`, `#\a`, and `#\Space`.

3. *Strings* are finite sequences of characters; an example is `"Hello World!"`.

4. *Symbols* consist of two strings: a *package name* and a *symbol name*. For example, the symbol `FOO::BAR` has package name `"FOO"` and symbol name `"BAR"`. ACL2 is case-insensitive with respect to symbol and package names. If a package name is not given, then the current package name is used, *e.g.*, if the current package is `"FOO"`, then `BAR` denotes the symbol `FOO::BAR`. The symbols `t` and `nil` are used to denote **true** and **false**, respectively.

Conses are ordered pairs of objects. For example, the ordered pair consisting of the number `1` and the string `"A"` is written `(1 . "A")`. The left component of a cons is called the *car* and the right component is called the *cdr*. You can think of conses as binary trees; the cons `(1 . "A")` is depicted in figure 8.1(a). Of special interest are a class of conses called *true lists*. A true list is either the symbol `nil`, which denotes the empty list and can be written `()`, or a cons whose cdr is a true list. For example, the true list containing the numbers 1 and 2, written `(1 2)`, is depicted in figure 8.1(b). Also of interest

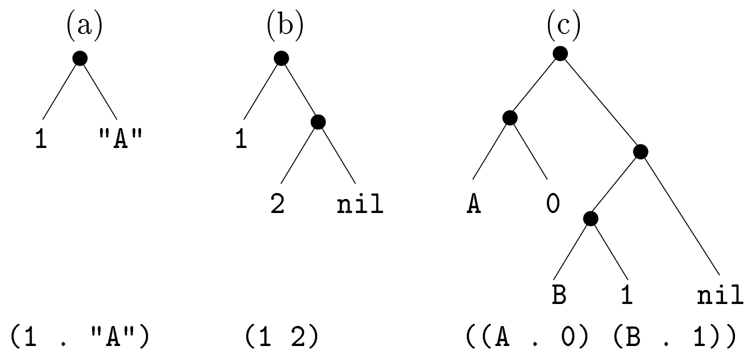

Figure 8.1: Examples of conses.

are *association lists* or *alists*. An alist is a true list of conses and is often used to represent a mapping that associates the car of an element in the list with its cdr. The alist `((A . 0) (B . 1))` is shown in figure 8.1(c).

8.2 Expressions

Expressions, which are also called terms, represent ACL2 programs and evaluate to ACL2 objects. We give an informal overview of expressions in this section. This allows us to suppress many of the details while focusing on the main ideas. Expressions depend on what we call a *history*, a list recording events. One reason for this dependency is that it is possible to define new functions (see section 8.3 on page 102) and these new functions can be used to form new expressions. User interaction with ACL2 starts in what we call the *ground-zero* history which includes an entry for the built-in functions. As new events arise, the history is extended, *e.g.*, a function definition extends the history with an entry, which includes the name of the function and its arity. We are now ready to discuss expressions. Essentially, given history h, an expression is:

- A constant symbol, which includes the symbols `t`, `nil`, and symbols in the package `"KEYWORD"`; constant symbols evaluate to themselves.

- A constant expression, which is a number, a character, a string, or a *quoted constant*, a single quote (`'`) followed by an object. Numbers, characters, and strings evaluate to themselves. The value of a quoted constant is the object quoted. For example, the values of `1`, `#\A`, `"Hello"`, `'hello`, and `'(1 2 3)` are `1`, `#\A`, `"Hello"`, (the symbol) `hello`, and (the list) `(1 2 3)`, respectively.

- A variable symbol, which is any symbol other than a constant symbol. The value of a variable symbol is determined by an environment.

- `(`f e_1 $\ldots$ e_n`)`, where f is a function expression of arity n in history h and e_i, for $1 \leq i \leq n$, is an expression in history h. A function expression of arity n is a symbol denoting a function of arity n (in history h) or a lambda expression of the form `(lambda (`$v_1 \ldots v_n$`)` *body*`)`, where $v_1, \ldots, v_n$ are distinct, *body* is an expression (in history h), and the only variables occurring freely in *body* are $v_1, \ldots, v_n$. The value of the expression is obtained by evaluating function f in the environment where the values of $v_1, \ldots, v_n$ are $e_1, \ldots, e_n$, respectively.

ACL2 contains many built-in, or primitive, functions. For example, `cons` is a built-in function of two arguments that returns a cons whose left element is the value of the first argument and whose right element is the value of the second argument. Thus, the value of the expression `(cons 'x 3)` is the cons `(x . 3)` because the value of the quoted constant `'x` is the symbol `x` and the value of the constant `3` is itself. Similarly, the value of expression `(cons (cons nil '(cons a 1)) (cons 'x 3))` is `((nil . (cons a 1)) . (x . 3))`. There are built-in functions for manipulating all of the ACL2 data types. Some of the built-in functions are described in table 8.2.

Comments are written with the use of semicolons: anything following a semicolon, up to the end of the line on which the semicolon appears, is a comment. Notice that an expression is an (ACL2) object and that an object is the value of some expression. For example, the object `(if (consp x) (car x) nil)` is the value of the expression `'(if (consp x) (car x) nil)`.

Expressions also include *macros*, which are discussed in more detail in section 8.3.2. Macros are syntactic sugar and can be used to define what seem to be functions of arbitrary arity. For example, `+` is a macro that can be used as if it is a function of arbitrary arity. We can write `(+)`, `(+ x)`, and `(+ x y z)` which evaluate to 0, the value of `x`, and the sum of the values of `x`, `y`, and `z`, respectively. The way this works is that `binary-+` is a function of two arguments and expressions involving `+` are abbreviations for expressions involving 0 or more occurrences of `binary-+`, *e.g.*, `(+ x y z)` is an abbreviation for `(binary-+ x (binary-+ y z))`.

Commonly used macros include the ones listed in table 8.2.

An often used macro is `cond`. `Cond` is a generalization of `if`. Instead of deciding between two expressions based on one test, as happens with `if`, one can decide between any number of expressions based on the appropriate number of tests. Here is an example.

Expression	**Value**
(`equal` x y)	`T` if the value of `x` equals the value of `y`, else `nil`
(`if` x y z)	The value of `z` if the value of `x` is `nil`, else the value of `y`
(`implies` x y)	`T` if the value of `x` is `nil` or the value of `y` is not `nil`, else `nil`
(`not` x)	`T` if the value of `x` is `nil`, else `nil`
(`acl2-numberp` x)	`T` if the value of `x` is a number, else `nil`
(`integerp` x)	`T` if the value of `x` is an integer, else `nil`
(`rationalp` x)	`T` if the value of `x` is a rational number, else `nil`
(`atom` x)	`T` if the value of `x` is an atom, else `nil`
(`endp` x)	Same as `(atom x)`
(`zp` x)	`T` if the value of `x` is 0 or is not a natural number, else `nil`
(`consp` x)	`T` if the value of `x` is a cons, else `nil`
(`car` x)	If the value of `x` is a cons, its left element, else `nil`
(`cdr` x)	If the value of `x` is a cons, its right element, else `nil`
(`cons` x y)	A cons whose car is the value of `x` and whose cdr is the value of `y`
(`binary-append` x y)	The list resulting from concatenating the value of `x` and the value of `y`
(`len` x)	The length of the value of `x`, if it is a cons, else 0

Table 8.1: Some built-in function symbols and their values.

Expression	**Value**
(`caar` x)	The car of the car of `x`
(`cadr` x)	The car of the cdr of `x`
(`cdar` x)	The cdr of the car of `x`
(`cddr` x)	The cdr of the cdr of `x`
(`first` x)	The car of `x`
(`second` x)	The cadr of `x`
(`append` x1 ... xn)	The `binary-append` of `x1 ... xn`
(`list` x1 ... xn)	The list containing `x1 ... xn`
(`+` x1 ... xn)	Addition
(`*` x1 ... xn)	Multiplication
(`-` x y)	Subtraction
(`and` x1 ... xn)	Logical conjunction
(`or` x1 ... xn)	Logical disjunction

Table 8.2: Some commonly used macros and their values.

(`cond` ($test_1$ exp_1)

...

($test_n$ exp_n)

(`t` exp_{n+1}))

The above `cond` is an abbreviation for the following expression.

(`if` $test_1$ exp_1

...

(`if` $test_n$ exp_n

exp_{n+1}) ...)

Another important macro is `let`. `Let` expressions are used to (simultaneously) bind values to variables and expand into lambdas. For example

```
(let ((v1   e1)
        ...
        (vn   en))
   body)
```

is an abbreviation for

```
((lambda (v1 ... vn)
    body)
 e1 ... en)
```

Consider the expression `(let ((x '(1 2)) (y '(3 4))) (append x y))`. It is an abbreviation for `((lambda (x y) (binary-append x y)) '(1 2) '(3 4))`, whose value is the list `(1 2 3 4)`.

Finally, `let*` is a macro that is used to sequentially bind values to variables and can be defined using `let`, as we now show.

```
(let* ((v1   e1)
         ...
         (vn   en))
   body)
```

is an abbreviation for

```
 (let ((v1   e1))
   (let* (...
            (vn   en))
          body))
```

8.3 Definitions

In this section, we give an overview of how one goes about defining new functions and macros in ACL2.

8.3.1 Functions

Functions are defined using `defun`. For example, we can define the successor function, a function of one argument that increments its argument by 1, as follows.

```
(defun succ (x)
  (+ x 1))
```

The form of a `defun` is `(defun` f *doc* $dcl_1 \ldots dcl_m$ `(`$x_1 \ldots x_n$`)` *body*`)`, where:

- $x_1 \ldots x_n$ are distinct variable symbols
- the free variables in *body* are in $x_1 \ldots x_n$
- *doc* is a documentation string and is optional
- $dcl_1 \ldots dcl_m$ are declarations and are optional
- functions, other than f, used in *body* have been previously introduced
- if f is recursive we must prove that it terminates

A common use of declarations is to declare *guards*. Guards are used to indicate the expected domain of a function. Since ACL2 is a logic of *total* functions, all functions, regardless of whether there are guard declarations or not, are defined on all ACL2 objects. However, guards can be used to increase efficiency because proving that guards are satisfied allows ACL2 to directly use the underlying Common Lisp implementation to execute functions. For example, `endp` and `eq` are defined as follows.

```
(defun endp (x)
  (declare (xargs :guard (or (consp x) (equal x nil))))
  (atom x))
(defun eq (x y)
  (declare (xargs :guard (if (symbolp x) t (symbolp y))))
  (equal x y))
```

Both `endp` and `eq` are logically equivalent to `atom` and `equal`, respectively. The only difference is in their guards, as `atom` and `equal` both have the guard `t`. If `eq` is only called when one of its arguments is a symbol, then it can be implemented more efficiently than `equal` which can be called on anything, including conses, numbers, and strings. Guard verification consists of proving that defined functions respect the guards of the functions they call. If guards are verified, then ACL2 can use efficient versions of functions.

Another common use of declarations is to declare the measure used to prove termination of a function. Consider the following function definition.

```
(defun app (x y)
  (declare (xargs :measure (len x)))
  (if (consp x)
      (cons (car x) (app (cdr x) y))
    y))
```

`App` is a recursive function that can be used to concatenate lists `x` and `y`. Such a definition introduces the axiom `(app x y)` = *body* where *body* is the body of the function definition. The unconstrained introduction of such axioms can render the theory unsound, *e.g.*, consider the "definition" `(defun bad (x) (not (bad x)))`. The axiom introduced, namely, `(bad x)` = `(not (bad x))` allows us to prove `nil` (**false**). To guarantee that function definitions are meaningful, ACL2 has a definitional principle which requires that the we prove that the function terminates. This requires exhibiting a *measure*, an expression that decreases on each recursive call of the function. For many of the common recursion schemes, ACL2 can guess the measure. In the above example, we explicitly provide a measure for function `app` using a declaration. The measure is the length of `x`. Notice that `app` is called recursively only if `x` is a cons and it is called on the cdr of `x`, hence the length of `x` decreases. For an expression to be a measure, it must evaluate to an ACL2 ordinal on any argument. ACL2 ordinals correspond to the ordinals up to ϵ_0 in set theory. They allow one to use many of the standard well-founded structures commonly used in termination proofs, *e.g.*, the lexicographic ordering on tuples of natural numbers.

8.3.2 Macros

Macros are really useful for creating specialized notation and for abbreviating commonly occurring expressions. Macros are functions on ACL2 objects, but they differ from ACL2 functions in that they map the objects given as arguments to expressions, whereas ACL2 functions map the values of the objects given as arguments to objects. For example, if m is a macro then `(`m x_1 $\ldots$ x_n`)` evaluates to an expression obtained by evaluating the function corresponding to the macro symbol m on arguments $x_1, \ldots, x_n$ (not their values, as happens with function evaluation), obtaining an expression *exp*. *Exp* is the *immediate expansion* of `(`m x_1 $\ldots$ x_n`)` and is then further evaluated until no macros remain, resulting in the *complete expansion* of the term. The complete expansion is then evaluated, as described previously.

Suppose that we are defining recursive functions whose termination can be shown with measure `(len x)`, where `x` is the first argument to the function. Instead of adding the required declarations to all of the functions under consideration, we might want to write a macro that generates the required `defun`. Here is one way of doing this.

```
(defmacro defunm (name args body)
  `(defun ,name
     ,args
     (declare (xargs :measure (len ,(first args))))
     ,body))
```

Notice that we define macros using `defmacro`, in a manner similar to function definitions. Notice the use of what is called the *backquote* notation. The value of a backquoted list is a list that has the same structure as the backquoted list except that expressions preceded by a comma are replaced by their values. For example, if the value of `name` is `app`, then the value of `` `(defun ,name) `` is `(defun app)`.

We can now use `defunm` as follows.

```
(defunm app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
    y))
```

This expands to the following.

```
(defun app (x y)
  (declare (xargs :measure (len x)))
  (if (consp x)
      (cons (car x) (app (cdr x) y))
    y))
```

When the above is processed, the result is that the function `app` is defined. In more detail, the above macro is evaluated as follows. The macro formals `name`, `args`, and `body` are bound to `app`, `(x y)`, and `(if (consp x) (cons (car x) (app (cdr x) y)) y)`, respectively. Then, the macro body is evaluated. As per the discussion on the backquote notation, the above expansion is produced.

We consider a final example to introduce ampersand markers. The example is the `list` macro and its definition follows.

```
(defmacro list (&rest args)
  (list-macro args))
```

Recall that `(list 1 2)` is an abbreviation for `(cons 1 (cons 2 nil))`. In addition, `list` can be called on an arbitrary number of arguments; this is accomplished with the use of the `&rest` ampersand marker. When this marker is used, it results in the next formal, `args`, getting bound to the list of the remaining arguments. Thus, the value of `(list 1 2)` is the value of the expression `(list-macro '(1 2))`. In this way, an arbitrary number of objects are turned into a single object, which is passed to the function `list-macro`, which in the above case returns the expression `(cons 1 (cons 2 nil))`.

8.4 Theorems

We now discuss how to prove theorems with ACL2. The theorem prover contains a database of rules, included in the *logical world* or *world*, that are used to prove theorems. The user can add a rule to the world by proving a theorem. The user can specify what kind of rules are generated by a theorem and this affects the future behavior of the theorem prover.

The command for submitting theorems to ACL2 is `defthm`. Here is an example.

```
(defthm app-is-associative
  (equal (app (app x y) z)
         (app x (app y z))))
```

ACL2 proves this theorem automatically, given the definition of `app`, but with more complicated theorems ACL2 often needs help. One way of providing help is to prove lemmas which are added to the world and can then be used in future proof attempts. For example, ACL2 does not prove the following theorem automatically.

```
(defthm app-is-associative-with-one-arg
  (equal (app (app x x) x)
```

```
          (app x (app x x))))
```

However, if `app-is-associative` is in the world, then ACL2 recognizes that the above theorem follows (it is a special case of `app-is-associative`). Another way of providing help is to give explicit hints, *e.g.*, one can specify what induction scheme to use, or what instantiations of previously proven theorems to use, and so on. More generally, the form of a `defthm` is

(`defthm` *name* *formula*
  `:rule-classes` ($class_1 \ldots class_n$)
  `:hints` ...)

where both the `:rule-classes` and `:hints` parts are optional.

8.4.1 Encapsulation

ACL2 provides a mechanism called *encapsulation* by which one can introduce constrained functions. For example, the following event can be used to introduce a function that is constrained to be associative and commutative.

```
(encapsulate
 ((ac (x y) t))
 (local (defun ac (x y) (+ x y)))
 (defthm ac-is-associative
   (equal (ac (ac x y) z)
          (ac x (ac y z))))
 (defthm ac-is-commutative
   (equal (ac x y)
```

```
            (ac y x))))
```

This event adds the axioms `ac-is-associative` and `ac-is-commutative`. The sole purpose of the local definition of `ac` in the above encapsulate form is to establish that the constraints are satisfiable. In the world after admission of the encapsulate event, the function `ac` is undefined; only the two constraint axioms are known.

There is a derived rule of inference called *functional instantiation* that is used as follows. Suppose f is a constrained function with constraint ϕ and suppose that we prove theorem ψ. Further suppose that g is a function that satisfies the constraint ϕ, with f replaced by g, then replacing f by g in ψ results in a theorem as well. That is, any theorem proven about f holds for any function satisfying the constraints on f. For example, we can prove the following theorem about `ac`.

```
(defthm commutativity-2-of-ac
  (equal (ac y (ac x z))
         (ac x (ac y z)))
  :hints (("Goal"
            :in-theory (disable ac-is-associative)
            :use ((:instance ac-is-associative)
                  (:instance ac-is-associative
                             (x y) (y x))))))
```

We can now use the above theorem and the derived rule of inference to show that any associative and commutative function satisfies the above theorem. For example, here is how we show that `*` satisfies the above theorem.

```
(defthm commutativity-2-of-*
  (equal (* y (* x z))
         (* x (* y z)))
  :hints (("Goal"
            :by (:functional-instance
                  commutativity-2-of-ac
                  (ac (lambda (x y) (* x y)))))))
```

ACL2 generates and establishes the necessary constraints, that `*` is associative and commutative.

Encapsulation and functional instantiation allow quantification over functions and thus have the flavor of a second order mechanism, although they are really first-order. For the full details see [BGKM91, KM01].

8.4.2 Books

A *book* is a file of ACL2 events analogous to a library. The ACL2 distribution comes with many books, including books for arithmetic, set theory, data structures, and so on. The events in books are *certified* as admissible and can be loaded into subsequent ACL2 sessions without having to replay the proofs. This makes it possible to structure large proofs and to isolate related theorems into libraries. Books can include *local* events that are not included when books are *included*, or loaded, into an ACL2 session.

8.5 Bibliographic Notes

ACL2 is the successor to the Boyer-Moore theorem prover Nqthm. It has been developed by Kaufmann and Moore with significant early contributions by Boyer. From the ACL2 Web page, `http://www.cs.utexas.edu/users/-moore/acl2`, the sources are freely available under the GNU General Public License. There are also over 3 megabytes of online information available, including extensive documentation, tutorials, a user's manual, and related papers. ACL2 is also described in a textbook [KMM00b] and a book of case studies [KMM00a]. ACL2 has been used on various large projects, including the verification of floating point [MLK98, Rus97, Rus99, Rus98, RF00], microprocessor verification [Hun89, HB92, BH97, HB97, SH97, SH98, Saw99, Gre98, GWH00, HWG98, WGHar, BKM96], and programming languages [Moo96, BT00]. There are various papers describing aspects the internals of ACL2, including single-threaded objects [BM99], encapsulation [KM01], and the base logic [KM97].

8.6 Summary

In this chapter, we described the ACL2 system, which consists of a programming language, a first-order logic, and a theorem prover. We discussed the data types residing in the ACL2 universe in section 8.1. In section 8.2 we discussed expressions, also called terms, which represent ACL2 programs and evaluate to ACL2 objects. We then explored the issues surrounding definitions in section 8.3. For example, we showed why allowing arbitrary definitions can lead to unsoundness and outlined the definitional principle, a principle which guarantees that defined functions do not render the theory unsound, in section 8.3.1. Macros are syntactic sugar and are really useful for creating specialized notation and for abbreviating commonly occurring expressions. Macros were briefly described in section 8.3.2. In section 8.4 we showed how one proves theorems in ACL2. In section 8.4.1 we discussed encapsulation, a mechanism for introducing constrained functions, and functional instantiation, a derived rule of inference that can be used with constrained functions and can be thought of as allowing a kind of quantification over functions. In section 8.4.2 we discussed ACL2 books, files of events analogous to libraries. There is extensive documentation on ACL2, including many large-scale case studies; some of relevant references are given in section 8.5.

Chapter 9

Model Checking

In this chapter, we define a model checker for the Mu-Calculus and CTL in ACL2. The model checker is used for the case studies in part IV. We start by developing ACL2 books on set theory, fixpoint theory, and relation theory. We then encode transition systems and the Mu-Calculus into ACL2. This includes a definition of the syntax and semantics of the Mu-Calculus, as well as proofs that the fixpoint operators of the Mu-Calculus actually compute fixpoints. Finally, we embed CTL into ACL2 and define a translator from CTL to the Mu-Calculus.

9.1 Set Theory

In this section, we develop some set theory. We represent sets as lists and define an equivalence relation on lists that corresponds to set equality. It turns out that we do not have to develop a "general" theory of sets; a theory of *flat* sets, *i.e.*, sets whose elements are compared by `equal`, will do. For example, in our theory of sets, `'(1 2)` is set equal to `'(2 1)`, but `'((1 2))` is not set equal to `'((2 1))`.

We develop some of the set theory in the package `SETS` and the rest in the package `FAST-SETS`, in subsections labeled by the package names.

9.1.1 SETS

We use the simplest definitions of sets and operations on them that we can think of in order to simplify the theorem proving process. We then define functions that are more efficient and prove the rewrite rules that allow us to rewrite the efficient functions into the simpler ones. The more efficient versions are often not `equal` to the original functions, but they are equal with respect to set equality. Using congruence-based reasoning in ACL2, we can rewrite the complicated versions to the simpler ones, if the context is right. In this way, once rewritten, all the theorem proving is about the simple functions, but the execution uses the efficient versions.

The definitions of `in` (set membership), `=<` (subset), and `==` (set equality) follow.

```
(defun in (a X)
  (cond ((endp X) nil)
        ((equal a (car X)) t)
        (t (in a (cdr X)))))

(defun =< (X Y)
  (cond ((endp X) t)
        (t (and (in (car X) Y)
                (=< (cdr X) Y)))))
```

```
(defun == (X Y)
  (and (=< X Y)
       (=< Y X)))
```

We prove that `==` is an equivalence relation. This is communicated to ACL2 with the following event, using the macro `defequiv`.

```
(defequiv ==)
```

The `defequiv` form generates the required `defthm`s. The proof depends on lemmas about `=<` (*e.g.*, that `=<` is transitive). In this chapter, we present what we consider to be the main lemmas, suppressing many of the details.

We use ACL2's congruence-based reasoning, which is an extension of the substitution of equals for equals where arbitrary equivalence relations can be used instead of equality. For example, suppose we define `set-union` to be `append`. As the name implies, we plan to use `set-union` to compute the union of two sets. We might want to prove

```
(implies (== X Z)
         (equal (set-union X Y) (set-union Z Y)))
```

so that ACL2 can replace x by z in `(set-union` x y`)`, if it can establish `(==` x z`)`. Letting x be `(1 1)` and z be `(1)`, it is easy to see that this is not a theorem. However, the following is a theorem.

```
(implies (== X Z)
         (== (set-union X Y) (set-union Z Y)))
```

If stored as a congruence rule, ACL2 can use this theorem to substitute z for (a set equal) x in `(set-union` x y`)`, in a context where it is enough to preserve `==`. A congruence rule can be proven using the macro `defcong`. The above theorem can be written `(defcong == == (set-union x y) 1)`. The general form of a `defcong` is

`(defcong` *eq1* *eq2* `(`f $x_1 \ldots x_n$`)` i`)`

where $1 \leq i \leq n$ and *eq1* and *eq2* are known equivalence relations. The `defcong` abbreviates the theorem

```
(implies (eq1 x y)
         (eq2 (f x1 ... x ... xn)
              (f x1 ... y ... xn)))
```

where x, y replace x_i.

We prove the following.

1. `(defcong == equal (in a X) 2)`
2. `(defcong == equal (=< X Y) 1)`
3. `(defcong == equal (=< X Y) 2)`
4. `(defcong == == (cons a X) 2)`

We now define `set-union` (which is equivalent to `binary-append`).

```
(defun set-union (X Y)
  (if (endp X)
      Y
```

```
    (cons (car X) (set-union (cdr X) Y))))
```

We prove the following results.

1. `(equal (in a (set-union X Y)) (or (in a X) (in a Y)))`
2. `(=< X (set-union Y X))`
3. `(== (set-union X Y) (set-union Y X))`
4. `(equal (== (set-union X Y) Y) (=< X Y))`
5. `(defcong == == (set-union X Y) 1)`
6. `(equal (=< (set-union Y Z) X) (and (=< Y X) (=< Z X)))`

The definition of `intersect`, a function which computes the intersection of two sets, follows.

```
(defun intersect (X Y)
  (cond ((endp X) nil)
        ((in (car X) Y)
         (cons (car X) (intersect (cdr X) Y)))
        (t (intersect (cdr X) Y))))
```

We prove the following results.

1. `(equal (in a (intersect X Y)) (and (in a X) (in a Y)))`
2. `(== (intersect X Y) (intersect Y X))`
3. `(implies (=< X Y) (== (intersect X Y) X))`

4. `(implies (or (=< Y X) (=< Z X))`
   `(=< (intersect Y Z) X))`

The definition of `minus`, a function which computes the set difference of two sets, follows.

```
(defun minus (X Y)
  (cond ((endp X) nil)
        ((in (car X) Y)
         (minus (cdr X) Y))
        (t (cons (car X) (minus (cdr X) Y)))))
```

We prove the following results.

1. `(implies (=< X Y) (equal (minus X Y) nil))`
2. `(implies (=< X Y) (=< (minus X Z) Y))`

The functions `set-complement`, `remove-dups`, `cardinality`, and `s<` (strict subset) are defined below.

```
(defun set-complement (X U) (minus U X))


(defun remove-dups (X)
  (cond ((endp X) nil)
        ((in (car X) (cdr X))
         (remove-dups (cdr X)))
        (t (cons (car X)
                 (remove-dups (cdr X))))))
```

```
(defun cardinality (X) (len (remove-dups X)))

(defun s< (X Y) (and (=< X Y) (not (=< Y X))))
```

We prove:

```
(implies (s< X Y)
         (< (len (remove-dups X)) (len (remove-dups Y))))
```

9.1.2 FAST-SETS

Although the definitions of the basic set operations defined above are simple (thus, good for reasoning about sets), some are not appropriate for execution. For example, `set-union` is not tail-recursive, hence, even if compiled, we can easily get stack overflows. In this section, we define functions that are more appropriate for execution and prove rewrite rules that transform the new, efficient versions to the old, simpler versions in the appropriate context (specifically, when it is enough to preserve `==`). This approach is *compositional*, *i.e.*, it allows us to decompose proof obligations of a system into proof obligations of the components of the system. Compositional reasoning is routinely used by ACL2 experts and is essential to the success of large verification efforts.

The functions we define below have the same names as their analogues, but are in the package `FAST-SETS`. The definition of `set-union`, in the package `FAST-SETS`, follows.

```
(defun set-union (X Y)
  (cond ((endp X) Y)
```

```
        ((in (car X) Y)
         (set-union (cdr X) Y))
        (t (set-union (cdr X) (cons (car X) Y)))))
```

We prove the following.

```
(== (set-union X Y) (sets::set-union X Y))
```

Notice that `set-union` differs from `sets::set-union`, *e.g.*, `(set-union '(1 1) '(1))` is `(1)` whereas `(sets::set-union '(1 1) '(1)` is `(1 1 1)`, but they are `==`. The above rule allows ACL2 to replace occurrences of `set-union` by `sets::set-union` in a context where it is enough to preserve `==`.

The definition of `intersect` follows. Note that its auxiliary function is tail recursive.

```
(defun intersect-aux (X Y Z)
  (cond ((endp X) Z)
        ((in (car X) Y)
         (intersect-aux (cdr X) Y (cons (car X) Z)))
        (t (intersect-aux (cdr X) Y Z))))


(defun intersect (X Y) (intersect-aux X Y nil))
```

We prove the following.

```
(== (intersect X Y) (sets::intersect X Y))
```

We also define `minus`, a tail-recursive version of `sets::minus`, and prove `(== (minus X Y) (sets::minus X Y))`.

Alternate definitions of `remove-dups` and `cardinality` are given below.

```
(defun remove-dups (X) (set-union X nil))
```

```
(defun cardinality (X) (len (remove-dups X)))
```

We prove the following.

```
(equal (cardinality X) (sets::cardinality X))
```

9.2 Fixpoint Theory

In this section, we develop an ACL2 book on the theory of fixpoints, in the package `SETS`. We do this by using encapsulation to reason about a constrained function, `f`, of one argument. Later, we show that certain functions compute fixpoints by using functional instantiation. An advantage of using encapsulation and functional instantiation is that we can ignore irrelevant issues, *e.g.*, in a later section we show that certain functions of several arguments compute fixpoints by functional instantiation using `f`, a function of one argument.

We start by constraining functions `f` and `S` so that `f` is monotonic and when `f` is applied to a subset of `S`, it returns a subset of `S`. Since functions defined in ACL2 are total, we cannot say that `f` is a function whose domain is the powerset of `S`. We could add hypotheses stating that all arguments to `f` are of the right type to the theorems that constrain `f`, but this generality is not needed and makes it slightly more cumbersome to prove theorems about `f`. The definitions of the constrained functions follow.

```
(encapsulate
 ((f (X) t)
```

```
  (S () t))
 (local (defun f(X) (declare (ignore X)) nil))
 (local (defun S() nil))
 (defthm f-is-monotonic
   (implies (=< X Y)
            (=< (f X) (f Y))))
 (defthm S-is-top
   (=< (f X) (set-union X (S)))))
```

We now define `applyf`, a function that applies `f` a given number of times.

```
(defun applyf (X n)
  (if (zp n)
      X
    (if (== X (f X))
        X
      (applyf (f X) (1- n)))))
```

From the Tarski-Knaster theorem (on page 18) and the remark following it, we expect that `lfpf` and `gfpf`, defined below, are the least and greatest fixpoints, respectively.

```
(defabbrev lfpf () (applyf nil (cardinality (S))))
```

```
(defabbrev gfpf () (applyf (S) (cardinality (S))))
```

We now prove the ACL2 version of the Tarski-Knaster theorem. We start by proving that `lfpf` is the least fixpoint.

1. `(== (f (lfpf)) (lfpf))`

2. `(implies (=< (f X) X) (=< (lfpf) X))`

We also prove that `gfpf` is the greatest fixpoint:

1. `(== (f (gfpf)) (gfpf))`

2. ```
   (implies (and (=< X (S)) (=< X (f X)))
            (=< X (gfpf)))
   ```

9.3 Relation Theory

In this section we develop a book, in the package `RELATIONS`, on the theory of relations. We represent relations as alists which map an element to the set of elements it is related to. A recognizer for relations is the following.

```
(defun relationp (r)
  (cond ((atom r) (eq r nil))
        (t (and (consp (car r))
                (true-listp (cdar r))
                (relationp (cdr r))))))
```

The definition of `image`, a tail-recursive function that computes the image of a set under a relation, follows.

```
(defun value-of (x alist)
  (cdr (assoc-equal x alist)))
```
```

```
(defun image-aux (X r tmp)
  (if (endp X)
      tmp
    (image-aux (cdr X) r
               (set-union (value-of (car X) r) tmp))))

(defun image (X r)
  (image-aux X r nil))
```

Similarly, we define `range`, a function that determines the range of a relation.

```
(defun range-aux (r tmp)
  (if (consp r)
      (range-aux (cdr r) (set-union (cdar r) tmp))
    tmp))

(defun range (r)
  (range-aux r nil))
```

We also define `inverse` so that it is tail recursive and computes the inverse of a relation.

```
(defun add (x Y)
  (if (in x Y)
      Y
    (cons x Y)))
```

```
(defun inverse-step-aux (st r tmp)
  (if (endp r)
      tmp
    (inverse-step-aux
     st
     (cdr r)
     (if (in st (cdar r))
         (add (caar r) tmp)
       tmp))))

(defun inverse-step (st r)
  (inverse-step-aux st r nil))

(defun inverse-aux (r ran tmp)
  (if (endp ran)
      tmp
    (inverse-aux
     r
     (cdr ran)
     (acons (car ran) (inverse-step (car ran) r) tmp))))

(defun inverse (r)
  (inverse-aux r (range r) nil))
```

The following function checks if the domain of its first argument (a relation) is a subset of its second argument.

```
(defun rel-domain-subset (r X)
  (cond ((endp r) t)
        (t (and (in (caar r) X)
                (rel-domain-subset (cdr r) X)))))
```

The following function checks if the range of its first argument (a relation) is a subset of its second argument.

```
(defun rel-range-subset (r X)
  (cond ((endp r) t)
        (t (and (=< (cdar r) X)
                (rel-range-subset (cdr r) X)))))
```

We prove the following.

1. `(implies (rel-range-subset r X) (=< (image Y r) X))`

2. `(implies (and (rel-range-subset r X) (=< X Y))`
   `(rel-range-subset r Y))`

9.4 Transition Systems

We define the notion of a transition system, or model, in ACL2. The functions defined in this section, as well as the next two sections, are in the package `MODEL-CHECK`. An ACL2 model is a five-tuple because it is useful to precompute the inverse relation of the transition relation and the cardinality of the set of states. The inverse transition relation relates a pair of states if, in one step, the first state can be reached from the second. A function that creates a model is defined below.

```
(defun make-model (s r l)
  (list s r l (inverse r) (cardinality s)))
```

We define `modelp`, a recognizer for models. We also define the accessor functions `states`, `relation`, `s-labeling`, `inverse-relation`, and `size` to access the states, transition relation, labeling relation, inverse transition relation, (atomic proposition) labeling relation, and cardinality of the states, respectively.

9.5 Mu-Calculus Syntax

We are now ready to start embedding the Mu-Calculus in ACL2. As defined on page 17, there is a restriction on formulas of the form $\mu Y f$ and $\nu Y f$ that f be monotone in Y; we do not require this. We return to the issue of monotonicity in the next section.

In figure 9.1, we define the syntax of the Mu-Calculus (`v` corresponds to the set of variables). `Mu-symbolp` is used because we do not want to decide the meaning of formulas such as `'(mu + f)` and we tag predicates by using `eval`.

We also define `translate-f`, a function that allows us to write formulas in an extended language, by translating its input into the Mu-Calculus. The extended syntax contains `AX` (`'(AX f)` is an abbreviation for `'(~ (EX (~ f)))`) and the infix operators `|` (which abbreviates `+`), `=>` and `->` (both denote implication), and `=`, `<->`, and `<=>` (all of which denote equality).

We define `(M-calc-sentencep f)`, which recognizes sentences (formulas with no free variables) in the extended syntax.

```
(defun mu-symbolp (s)
  (and (symbolp s)
       (not (in s '(+ & MU NU eval true false)))))

(defun basic-m-calc-formulap (f v)
  (cond ((symbolp f)
         (or (in f '(true false))
             (and (mu-symbolp f)
                  (in f v))))
        ((and (consp f)
              (equal (car f) 'eval))
         t)
        ((equal (len f) 2)
         (and (in (first f) '(~ EX))
              (basic-m-calc-formulap (second f) v)))
        ((equal (len f) 3)
         (let ((first (first f))
               (second (second f))
               (third (third f)))
           (or (and (in second '(& +))
                    (basic-m-calc-formulap first v)
                    (basic-m-calc-formulap third v))
               (and (or (in first '(MU NU)))
                    (mu-symbolp second)
                    (basic-m-calc-formulap
                     third (cons second v)))))))))
```

Figure 9.1: The syntax of the Mu-Calculus.

9.6 Mu-Calculus Semantics

The semantics of a Mu-Calculus formula, as explained in section 2.4.1, page 17, is given with respect to a model and a valuation assigning a subset of the states to variables. The semantics of an expression denoting a predicate is the set of states that satisfy the predicate. We use ACL2 expressions to denote predicates. The semantics of a variable is its value under the valuation. Conjunctions, disjunctions, and negations correspond to intersections, unions, and complements, respectively. $\mathsf{EX}f$ is true at a state if the state has some successor that satisfies f. Finally, μ's and ν's correspond to least and greatest fixpoints, respectively. Note that the semantics of a sentence (a formula with no free variables) does not depend on the initial valuation. The formal definition is given in figure 9.2; some auxiliary functions and abbreviations used in the figure follow. We point out that `defabbrev` generates a macro, but one that expands the way a function definition would. In addition the `mutual-recursion` form is used to define mutually-recursive functions in ACL2.

The function `evl` used in `semantics-EVAL-aux`, below, is an *evaluator*, a function that can evaluate terms constructed from a fixed set of functions symbols. The idea is that once a set of predicates, to be used for specifying temporal properties, is chosen, then `evl` is defined to be an evaluator for these predicates. The macro `defevaluator`, provided by ACL2, can be used for this purpose.

```
(defun semantics-EVAL-aux (l f fs)
  (if (endp l)
      fs
```

```
    (let ((s  (caar l))
          (ls (cdar l)))
      (if (evl (list f (list 'quote ls)) nil)
          (semantics-EVAL-aux (cdr l) f (cons s fs))
        (semantics-EVAL-aux (cdr l) f fs)))))

(defun semantics-EVAL (m f)
  (semantics-EVAL-aux (s-labeling m) (second f) nil))

(defabbrev semantics-EX (m f val)
  (image (mu-semantics m (second f) val)
         (inverse-relation m)))

(defabbrev semantics-NOT (m f val)
  (set-complement (mu-semantics m (second f) val)
                  (states m)))

(defabbrev semantics-AND (m f val)
  (intersect (mu-semantics m (first f) val)
             (mu-semantics m (third f) val)))

(defabbrev semantics-OR (m f val)
  (set-union (mu-semantics m (first f) val)
             (mu-semantics m (third f) val)))

(defabbrev semantics-fix (m f val s)
  (compute-fix-point
```

```
   m (third f) (put-assoc-equal (second f) s val)
   (second f) (size m)))

(defabbrev semantics-MU (m f val)
  (semantics-fix m f val nil))

(defabbrev semantics-NU (m f val)
  (semantics-fix m f val (states m)))
```

Now, we are ready to define the main function:

```
(defun semantics (m f)
  (if (m-calc-sentencep f)
      (mu-semantics m (translate-f f) nil)
    "not a valid mu-calculus formula"))
```

`Semantics` returns the set of states in `m` satisfying `f`, if `f` is a valid Mu-Calculus formula, otherwise, it returns an error string.

As an example, consider a Mu-Calculus formula that holds exactly in those states where it is possible to reach an e-state (*i.e.*, a state whose label satisfies the predicate denoted by expression e). The idea is to start with e-states, then add states that can reach an e-state in one step, two steps, and so on. When you are adding states, this corresponds to a least fixpoint computation. One way to denote this is $\mu Y(e \;\vee\; \mathsf{EX}Y)$; it may help to think about "unrolling" the fixpoint.

As another example contemplate a formula that holds exactly in those states where every reachable state is an e-state. The idea is to start with e-states, then remove states that can reach a non e-state in one step, two steps,

```
(mutual-recursion
(defun mu-semantics (m f val)
  (cond ((eq f 'true) (states m))
        ((eq f 'false) nil)
        ((mu-symbolp f)
         (value-of f val))
        ((and (consp f)
              (equal (car f) 'eval))
         (semantics-EVAL m f))
        ((equal (len f) 2)
         (cond ((equal (first f) 'EX)
                (semantics-EX m f val))
               ((equal (first f) '~)
                (semantics-NOT m f val))))
        ((equal (len f) 3)
         (cond ((equal (second f) '&)
                (semantics-AND m f val))
               ((equal (second f) '+)
                (semantics-OR m f val))
               ((equal (first f) 'MU)
                (semantics-MU m f val))
               ((equal (first f) 'NU)
                (semantics-NU m f val))))))

(defun compute-fix-point (m f val y n)
  (if (zp n)
      (value-of y val)
    (let ((x (value-of y val))
          (new-x (mu-semantics m f val)))
      (if (== x new-x)
          x
        (compute-fix-point
         m f (put-assoc-equal y new-x val) y (- n 1))))))
          ; note that the valuation is updated
)
```

Figure 9.2: The semantics of the Mu-Calculus.

and so on. When you are removing states, this corresponds to a greatest fixpoint computation. One way to denote this is $\nu Y(e \;\wedge\; \neg\mathsf{EX}\neg Y)$; as before it may help to think about unrolling the fixpoint.

The model checking algorithm we presented is *global*, meaning that it returns the set of states satisfying a Mu-Calculus formula. Another approach is to use a *local* model checking algorithm. The difference is that the local algorithm is also given as input a state and checks whether that particular state satisfies the formula; in some cases this can be done without exploring the entire structure, as is required with the global approach.

The model checking algorithm we presented is *extensional*, meaning that it represents both the model and the sets of states it computes explicitly. If any of these structures gets too big—since a model is exponential in the size of the program text, *state explosion* is common—resource constraints make the problem practically unsolvable. Symbolic model checking [CBM89, Pix90, McM93, BCM+92, TSL+90] is a technique that has greatly extended the applicability of model checking. The idea is to use compact representations of the model and of sets of states. This is done by using BDDs (binary decision diagrams), which on many examples have been shown to represent states and models very compactly [Bry92]. BDDs can be thought of as deterministic finite-state automata (see any book covering Automata Theory, *e.g.*, [HU79]). A Boolean function, f, of n variables can be thought of as a set of n-length strings over the alphabet $\{0, 1\}$. We start by ordering the variables; in this way an n-length string over $\{0, 1\}$ corresponds to an assignment of values to the variables. We can represent f by an automaton whose language is the set of strings that make f true. We can now use the results of automata theory, *e.g.*, deterministic automata can be uniquely minimized in $O(n \log n)$ time (the reason why nondeterministic automata are not used is that minimizing them is a PSPACE-complete problem), hence, we have a canonical representation of Boolean functions. Automata that correspond to Boolean functions have a simpler structure than general automata (*e.g.*, they do not have cycles); BDDs are a data structure that exploits this structure. Sets of states as well as transition relations can be thought of as Boolean functions, so they too can be represented using BDDs. Finally, note that the order of the variables can make a big (exponential) difference in the size of the BDD corresponding to a Boolean function. Symbolic model checking algorithms, even for temporal logics such as CTL whose expressive power compared with the Mu-Calculus is quite limited, are based on the algorithm we presented (except that BDDs are used to represent sets of states and models).

Now that we have written down the semantics of the Mu-Calculus in ACL2, we have an executable model checker. But have we embedded the Mu-Calculus into ACL2 correctly? To answer this question, we need to prove theorems in a system that includes both ACL2 and set theory. No such (mechanical) system exists, but if it did, the theorem relating the set-theoretic presentation of the Mu-Calculus to the ACL2 version would be somewhat complicated by the logical differences between ACL2 and set theory, *e.g.*, functions in ACL2 are total. However, we can check certain "saneness" properties. We have already shown that our ACL2 set-theoretic functions satisfy some of the standard properties (*e.g.*, that `set-union` is associative and commutative with respect to `==`). We also show that `MU` formulas are least fixpoints (if the formulas are monotonic in the variable of the `MU` and certain "type" conditions hold), and similarly `NU` formulas are greatest fixpoints. We start by defining what it means to be a fixpoint.

```
(defun fixpointp (m f val x s)
  (== (mu-semantics m f (put-assoc-equal x s val)) s))

(defun post-fixpointp (m f val x s)
  (=< (mu-semantics m f (put-assoc-equal x s val)) s))

(defun pre-fixpointp (m f val x s)
  (=< s (mu-semantics m f (put-assoc-equal x s val))))
```

We use encapsulation to constrain the functions `sem-mon-f`, `good-val`, `good-var`, and `good-model` so that the following hold.

- `Sem-mon-f` is semantically monotone in `good-var`.
- `Good-val` is "reasonable", *i.e.*, the value of each variable is a subset of the state space.
- `Good-var` is "reasonable", *i.e.*, it satisfies `mu-symbolp`.
- `Good-model` is "reasonable", *i.e.*, its size field corresponds to the size of the state space, the range of the transition relation is a subset of the state space, the domain of the labeling relation is a subset of the state space, and the range of the inverse transition relation is a subset of the state space.

We prove the following fixpoint theorems by functionally instantiating the main theorems in the book `fixpoints`.

```
(fixpointp (good-model) (sem-mon-f) (good-val) (good-var)
```

```
          (mu-semantics (good-model)
                        (list 'mu (good-var) (sem-mon-f))
                        (good-val)))

(fixpointp (good-model) (sem-mon-f) (good-val) (good-var)
           (mu-semantics (good-model)
                         (list 'nu (good-var) (sem-mon-f))
                         (good-val)))
```

We prove that MU formulas are least fixpoints and that NU formulas are greatest fixpoints.

```
(implies
 (post-fixpointp (good-model) (sem-mon-f)
                 (good-val) (good-var) x)
 (=< (mu-semantics (good-model)
                   (list 'mu (good-var) (sem-mon-f))
                   (good-val))
     x))

(implies
 (and (=< x (states (good-model)))
      (pre-fixpointp (good-model) (sem-mon-f)
                     (good-val) (good-var) x))
 (=< x (mu-semantics (good-model)
                     (list 'nu (good-var) (sem-mon-f))
                     (good-val))))
```

9.7 Temporal Logic

The syntax of CTL formulas (see section 2.4 on page 17) is embedded in ACL2 as shown in figure 9.3. Notice that the ACL2 syntax is extended in order to simplify the writing of specifications, *e.g.*, the operators A, F, and G are allowed.

We define a translator from CTL into the Mu-Calculus in figure 9.4. The translator is based on the following facts relating CTL and the Mu-Calculus.

- $\mathsf{EF}f \quad \equiv \quad \mu Y(f \quad \vee \quad \mathsf{EX}Y)$
- $\mathsf{EG}f \quad \equiv \quad \nu Y(f \quad \wedge \quad \mathsf{EX}Y)$
- $\mathsf{AF}f \quad \equiv \quad \mu Y(f \quad \vee \quad \mathsf{AX}Y)$
- $\mathsf{AG}f \quad \equiv \quad \nu Y(f \quad \wedge \quad \mathsf{AX}Y)$
- $\mathsf{E}(f\mathsf{U}g) \quad \equiv \quad \mu Y(g \quad \vee \quad (f \quad \wedge \quad \mathsf{EX}Y))$
- $\mathsf{A}(f\mathsf{U}g) \quad \equiv \quad \mu Y(g \quad \vee \quad (f \quad \wedge \quad \mathsf{AX}Y))$

9.8 Bibliographic Notes

Model checking algorithms were introduced by Clarke and Emerson and Queille and Sifakis [CE81, Eme81, QS82]. The propositional Mu-Calculus is due to Kozen [Koz83, Par69, EC80, EL86, EJS93, Eme97]. Many temporal logics, *e.g.*, CTL, LTL, and CTL* can be translated into the Mu-Calculus. In addition, the algorithm that decides the Mu-Calculus is used for symbolic (BDD-based) model checking [CBM89, Pix90, McM93, BCM+92, TSL+90],

```
(defabbrev u-formulap (f)
  (and (equal (len f) 3)
       (ctl-formulap (first f))
       (ctl-formulap (third f))
       (equal 'u (second f))))

(defun ctl-formulap (f)
  (cond ((symbolp f)
         (in f '(true false)))
        ((and (consp f)
              (equal (car f) 'eval))
         t)
        ((equal (len f) 2)
         (and (in (first f) '(~ EX AX EF EG AF AG))
              (ctl-formulap (second f))))
        ((equal (len f) 3)
         (let ((first (first f))
               (second (second f))
               (third (third f)))
           (or (and (in second '(& +))
                    (ctl-formulap first)
                    (ctl-formulap third))
               (and (in first '(E A))
                    (equal second '~)
                    (u-formulap third)))))
        ((equal (len f) 4)
         (and (in (first f) '(A E))
              (u-formulap (cdr f))))))
```

Figure 9.3: The syntax of CTL.

```
(defun ctl-2-muc (f)
  (cond
   ((or (symbolp f)
        (and (consp f) (equal (car f) 'eval)))
    f)
   ((equal (len f) 2)
    (let ((first (first f))
          (second (second f)))
      (cond ((in first '(~ EX AX))
             (list first (ctl-2-muc second)))
            ((equal first 'EF)
             `(mu y (,(ctl-2-muc second) + (EX y))))
            ((equal first 'EG)
             `(nu y (,(ctl-2-muc second) & (EX y))))
            ((equal first 'AF)
             `(mu y (,(ctl-2-muc second) + (AX y))))
            ((equal first 'AG)
             `(nu y (,(ctl-2-muc second) & (AX y)))))))
   ((equal (len f) 3)
    (let ((first (first f))
          (second (second f))
          (third (third f)))
      (cond ((in second '(& +))
             (list (ctl-2-muc first) second (ctl-2-muc third)))
            ((equal first 'E) ; translate (E ~ (f U g))
             (list '~ (ctl-2-muc (cons 'A third))))
            (t  ; translate (A ~ (f U g))
             (list '~ (ctl-2-muc (cons 'E third)))))))
   ((equal (len f) 4)
    (let ((second (second f))
          (fourth (fourth f)))
      (cond ((equal (first f) 'E)  ; translate (E f U g)
             `(mu y (,(ctl-2-muc fourth) +
                     (,(ctl-2-muc second) & (EX y)))))
            (t  ; translate (A f U g)
             `(mu y (,(ctl-2-muc fourth) +
                     (,(ctl-2-muc second) & (AX y))))))))))
```

Figure 9.4: A translator from CTL to the Mu-Calculus.

a technique that has greatly extended the applicability of model checking. The idea of using temporal logic as a specification language is due to Pnueli [Pnu77]. Model checking has been successfully applied to automatically verify many reactive systems and is now being used by hardware companies as part of their verification process. CTL is used as a specification language in model checkers such as SMV [McM93] and VIS [RGA+96].

The work reported in this chapter has been published in [Man00c].

9.9 Summary

We defined the Mu-Calculus, CTL, and a translator from CTL to the Mu-Calculus in ACL2. The result is a model checker for both the Mu-Calculus and CTL. The model checker is used in the case studies in part IV. We proved that the model checker is correct. This required developing ACL2 books on set theory, fixpoint theory, and relation theory. These books were used to show that μ's and ν's really do compute least and greatest fixpoints, respectively.

Part IV

Case Studies and Conclusions

Chapter 10

Introduction

In this part, we describe two case studies. In chapter 11 we describe the verification of the alternating bit protocol, a simple communications protocol. We use our approach to combining theorem proving and model checking to prove a stuttering bisimulation on the alternating bit protocol, which is then used to extract a quotient, which is infinite-state, but has a finite set of reachable states. The quotient is then model checked with our model checker.

The second case study is the verification of a simple pipelined machine due to Sawada [Saw00]. We discuss notions of correctness and their importance in some detail and compare our notion of correctness, which is based on stuttering bisimulation, to the variant of the Burch and Dill notion of correctness [BD94] used by Sawada. We show, with mechanical proof, that the Burch and Dill notion can be satisfied by incorrect machines, *e.g.*, machines that deadlock. In contrast, we argue that no incorrect machine satisfies our notion of correctness. In addition, we give an overview of the libraries of ACL2 theorems used and explain how to automate much of the verification, *e.g.*, the verification of the pipelined machine is automatic. We examine various variants of the pipelined machine including machines with exceptions, interrupts (which lead to non-determinism), and netlist (gate-level) descriptions and show that our notion of correctness applies to these extensions. Many of the variant machines are verified using the compositional proof rule for stuttering bisimulations from part II.

Chapter 11

Alternating-Bit Protocol

11.1 Introduction

We use our approach to combining theorem proving and model checking to verify the alternating bit protocol [BSW69]. This protocol cannot be directly model checked because it has an infinite-state space; however, using the theorem prover ACL2, we show that the protocol is stuttering bisimilar to a small finite-state system, which we model check. We also show that the alternating bit protocol is a refinement of a non-lossy system.

We chose the alternating bit protocol because it has been used as a benchmark for verification efforts. The alternating bit protocol has a simple description but lengthy hand proofs of correctness (*e.g.*, [BG94]), it is infinite-state, and its specification involves a complex fairness property. We have found it to be surprisingly difficult to verify mechanically; many previous papers verify various versions of the protocol (*e.g.*, [Mil90, CE81, COR+95, BG96, MN95]), but all make simplifying assumptions, either by restricting channels

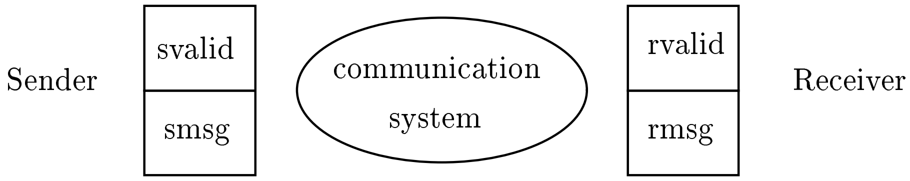

Figure 11.1: Protocol from sender's and receiver's view.

to be bounded buffers, by ignoring data, or by ignoring fairness issues.

In section 11.2, we present the ACL2 formalization of the alternating bit protocol. In section 11.3, we present the proof of correctness and in section 11.4, we discuss related work.

11.2 Protocol

The alternating bit protocol is used to implement reliable communication over faulty channels. We present the protocol from the view of the sender and receiver first and then in complete detail. The sender interacts with the communication system via the register *smsg* and the flag *svalid*. The sender can assign a message to *smsg* provided it is invalid, *i.e.*, *svalid* is false. The receiver interacts with the communication system via the register *rmsg* and the flag *rvalid*. The receiver can read *rmsg* provided it is valid, *i.e.*, *rvalid* is not false; when read, *rmsg* is invalidated. Figure 11.1 depicts the protocol from this point of view.

The communication system consists of the flags *sflag* and *rflag* as well as the two lossy, unbounded, and FIFO channels *s2r* and *r2s*. The idea behind the protocol is that the contents of *smsg* are sent across *s2r* until an acknowledgment for the message is received on *r2s*, at which point a new message can

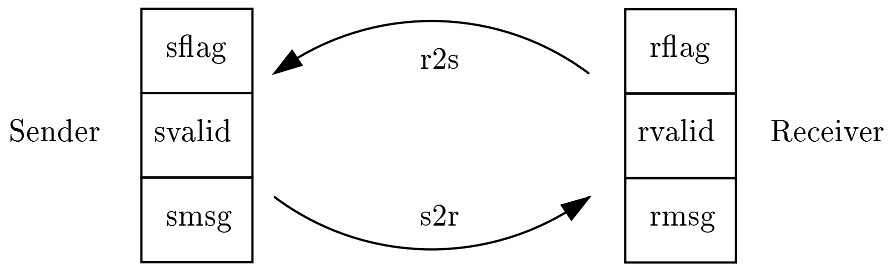

Figure 11.2: Alternating Bit Protocol.

be transmitted. Similarly, acknowledgments for a received message are sent across *r2s* until a new message is received. In order for the receiving end to distinguish between copies of the same message and copies of different messages, each message is tagged with *sflag* before being placed on *s2r*. When a new message is received, *rflag* is assigned the value of the message tag and gets sent across *r2s*; this also allows the sending end to distinguish acknowledgments. There may be an arbitrary number of copies of a message (or an acknowledgment) on the channels, and it turns out that there are at most two distinct messages (or acknowledgments) on the channels, hence binary flags suffice. Figure 11.2 depicts the protocol.

The above discussion is informal; a formal description follows, but first we discuss notation. We have formalized the protocol and its proof in ACL2, however, for presentation purposes we describe the formalization using standard notation. We remain faithful to the ACL2 formalization, *e.g.*, we do not use types: functions that appear typed are really under-specified, but total. The concatenation operator on sequences is denoted by ":", but sometimes we use juxtaposition; "ϵ" denotes the empty sequence; $head.s$ is the first element of sequence s; $tail.s$ is the sequence resulting from removing the first element

Rule	Definition
Skip	$\underline{\mathsf{skip}}$
Accept.m	$\neg svalid \;\rightarrow\; smsg, svalid \;:=\; m, \mathbf{true}$
Send-msg	$svalid \;\rightarrow\; s2r \;:=\; s2r : \langle smsg, sflag \rangle$
Drop-msg	$s2r \neq \epsilon \;\rightarrow\; s2r \;:=\; tail.s2r$
Get-msg	$s2r \neq \epsilon \wedge \neg rvalid \;\rightarrow$
	$\underline{\mathsf{if}}\;\; flag(head.s2r) = rflag$
	$\quad \underline{\mathsf{then}}\;\; s2r \;:=\; tail.s2r$
	$\quad \underline{\mathsf{else}}\;\; s2r, rmsg, rvalid, rflag \;:=$
	$\qquad\qquad tail.s2r, info(head.s2r), \mathbf{true}, flag(head.s2r)$
Send-ack	$r2s \;:=\; r2s : rflag$
Drop-ack	$r2s \neq \epsilon \;\rightarrow\; r2s \;:=\; tail.r2s$
Get-ack	$r2s \neq \epsilon \;\rightarrow$
	$\underline{\mathsf{if}}\;\; head.r2s = sflag$
	$\quad \underline{\mathsf{then}}\;\; r2s, svalid, sflag \;:=\; tail.r2s, \mathbf{false}, \neg sflag$
	$\quad \underline{\mathsf{else}}\;\; r2s := \; tail.r2s$
Reply	$rvalid \;:=\; \mathbf{false}$

Table 11.1: Rules defining the transition relation.

from s; $\#s$ is the size of the sequence. Messages are pairs; *info* returns the first component of a message and *flag* returns the second.

A state is an 8-tuple $\langle sflag, svalid, smsg, s2r, r2s, rflag, rvalid, rmsg \rangle$; *state* is a predicate that recognizes states. The *sflag* of state s is denoted $sflag.s$ and similarly for the other fields. Rules are functions from states into states; they are listed in Table 11.1 and are of the form $G \rightarrow A$; if A is used as a rule, it abbreviates $\mathbf{true} \rightarrow A$. Rule $G \rightarrow A$ defines the function $\lambda s(\underline{\mathsf{if}}\ G.s\ \underline{\mathsf{then}}\ A.s\ \underline{\mathsf{else}}\ s)$. We now define the transition relation, R: sRw iff s is a state and w can be obtained by applying some rule to s.

We have defined the states and transition relation of the alternating bit protocol. The states are labeled with an 8-tuple, as mentioned above. Therefore, the alternating bit protocol defines a TS, $\mathcal{ABP}$.

11.3 Protocol Verification

We give an overview of the verification of the alternating bit protocol. $\mathcal{ABP}''$ is an isomorphic copy of $\mathcal{ABP}$, with the channel values distorted. Thus $\mathcal{ABP} \approx_r \mathcal{ABP}''$, where r is the refinement map that performs the distortion. We define *rep*, a state representative function on $\mathcal{ABP}''$ and prove that B, the equivalence relation induced by *rep*, is a WEB on $\mathcal{ABP}''$. We then use our extraction procedure, which by theorems 12 and 13, gives (the reachable part of) $\mathcal{ABP}''/B$. We now have $\mathcal{ABP}'' \approx_{rep} \mathcal{ABP}''/B$. $\mathcal{ABP}'$ is $\mathcal{ABP}''/B$ restricted to the non-channel variables (which were left untouched by r), hence, $\mathcal{ABP}''/B \approx_q \mathcal{ABP}'$, where q is the refinement map that hides the channels. By theorem 11, $\mathcal{ABP} \approx_{r;rep;q} \mathcal{ABP}'$. $\mathcal{ABP}$ is a typed transition system and $r; rep; q$ is a refinement map that hides variables. Thus, we have a well-behaved refinement map and by lemma 26 (on page 65) we can model check $\mathcal{ABP}'$ and lift the results to $\mathcal{ABP}$.

We also show that $\mathcal{ABP}'$ is a refinement of a non-lossy protocol. Refinement checking is more convincing than model checking when it is easier to describe a correct system than it is to specify a set of temporal logic formulas that implies correctness.

11.3.1 Well-Founded Equivalence Bisimulation

In this subsection we define a relation B and outline the ACL2 proof that B is a WEB. We start with some definitions.

For the following definitions, a and b are sequences of length 1, $a \neq b$, and x is an arbitrary finite sequence. The function *compress* acts on sequences to remove adjacent duplicates. Formally,

$$compress.\epsilon = \epsilon \qquad compress.a = a$$

$$compress(aax) = compress(ax) \qquad compress(abx) = a : compress(bx)$$

The predicate *good-s2r* recognizes sequences that define valid channel contents. Formally,

$$good\text{-}s2r.\epsilon = \mathbf{true} \quad good\text{-}s2r(ax) = (a = \langle info.a, flag.a\rangle) \wedge good\text{-}s2r.x$$

The function *s2r-state* compresses the *s2r* field of a state, except that already received messages at the head of *s2r* are omitted. Formally,

$$s2r\text{-}state.s = compress(relevant\text{-}s2r(s2r.s, \langle rmsg.s, rflag.s\rangle))$$

where the function *relevant-s2r* is defined by:

$$relevant\text{-}s2r(\epsilon, a) = \epsilon \qquad relevant\text{-}s2r(bx, a) = bx$$

$$relevant\text{-}s2r(ax, a) = relevant\text{-}s2r(x, a)$$

The function *r2s-state* compresses the *r2s* field of a state, except that acknowledgments at the head of *r2s* with a flag different from *sflag* are omitted. Formally,

$$r2s\text{-}state.s = compress(relevant\text{-}r2s(r2s.s, sflag.s))$$

where the function *relevant-r2s* is defined by:

$$relevant\text{-}r2s(\epsilon, a) = \epsilon \qquad relevant\text{-}r2s(ax, a) = ax$$

$$relevant\text{-}r2s(bx, a) = relevant\text{-}r2s(x, a)$$

The main idea behind the WEB is to relate states that have similar compressed channels—*i.e.*, are equivalent under *s2r-state* and *r2s-state*—and are otherwise identical. We define a state representative function that will be used to define the WEB, using the rule notation described in section 11.2, as follows.

$$rep: \quad good\text{-}s2r.s2r \quad \rightarrow \quad s2r, r2s := s2r\text{-}state, r2s\text{-}state$$

We now define our proposed WEB B: $sBu \quad \text{iff} \quad rep.s = rep.u$. It is easy to see that B is an equivalence relation that, except for *s2r* and *r2s*, preserves the labeling of states. We define *rankt* to always return 0 and *rank*, a function on states, as follows: $rank.s = \#(s2r.s) + \#(r2s.s)$.

We show that $\langle rankt, \langle \{0\}, < \rangle, rank \rangle$ is a well-founded witness. (To be pedantic we should define *rank* so that it has two arguments. This can be done be defining *rank* as follows: $rank(u, s) = \#(s2r.s) + \#(r2s.s)$.) Note that if sBw, sRu, and sBu, then uBw and by rule *Skip*, wRw, therefore, we need only concern ourselves with the case where $\neg sBu$. To show B is a WEB, it suffices to show:

$$sBw \wedge sRu \wedge \neg sBu \;\Rightarrow\; \langle \exists v : wRv : uBv \vee (sBv \wedge rank.v < rank.w) \rangle$$

We break up the proof (that B is a WEB) into the eight cases in Table 11.2 by expanding R, *i.e.*, by considering all the ways in which s can be related to u. The cases have the form: Rule Lemma; when u or v appear in Lemma they abbreviate the terms Rule.s and Rule.w, respectively. We prove the cases in ACL2.

In order to tie up the case analysis, we define a function *step* that takes three states, s, u, and w, as arguments. If sBu, *step* returns w, else if $u = A.s$,

Rule	Lemma
Accept.m	$sBw \quad \Rightarrow \quad uBv$
Send-msg	$sBw \wedge \neg sBu \quad \Rightarrow \quad uBv$
Drop-msg	$sBw \wedge \neg sBu \quad \Rightarrow \quad (uBv) \vee (sBv \wedge rank.v < rank.w)$
Get-msg	$sBw \wedge \neg sBu \wedge u \neq Drop\text{-}msg.s$
	$\Rightarrow \quad (uBv) \vee (sBv \wedge rank.v < rank.w)$
Send-ack	$sBw \wedge \neg sBu \quad \Rightarrow \quad uBv$
Drop-ack	$sBw \wedge \neg sBu \quad \Rightarrow \quad (uBv) \vee (sBv \wedge rank.v < rank.w)$
Get-ack	$sBw \wedge \neg sBu \wedge u \neq Drop\text{-}ack.s$
	$\Rightarrow \quad (uBv) \vee (sBv \wedge rank.v < rank.w)$
Reply	$sBw \quad \Rightarrow \quad uBv$

Table 11.2: WEB case analysis.

for A, a rule from Table 11.1, *step* returns $A.w$, else *step* returns w. Since we proved that B is an equivalence relation, the following theorem implies that B is a WEB (existential quantification is replaced by the witness function *step*):

$$sBw \wedge sRu \wedge v = step(s, u, w) \quad \Rightarrow \quad wRv \wedge (uBv \vee (sBv \wedge rank.v < rank.w))$$

11.3.2 Quotient Extraction

In this subsection we prove the following ACL2 theorems which show that *rep* is a representative function satisfying the requirements of theorem 12; hence, the quotient structure induced by *rep* is isomorphic to the quotient structure with respect to B: $sBw \equiv rep.s = rep.w$, $rep(rep.s) = rep.s$, and $rank(rep.s) \leq rank.s$. We extract the quotient structure (induced by *rep*) of the alternating bit protocol restricted to binary messages. In the following subsections, we describe the use of model checking and WEB equivalence checking to analyze this structure.

We now have enough machinery to describe how refinement is used in the verification of the alternating bit protocol. $\mathcal{ABP}$ is the model of the alternating bit protocol in ACL2. $\mathcal{ABP}''$ is $\mathcal{ABP}$ with *s2r*, *r2s* relabeled by *s2r-state* and *r2s-state*, respectively. Note that $\mathcal{ABP}$ is a refinement of $\mathcal{ABP}$ with respect to the refinement map that performs the relabeling. B is a WEB on $\mathcal{ABP}''$ with well-founded witness $\langle rankt, \langle \{0\}, < \rangle, rank \rangle$, such that $rank(u, s) = \#(s2r(f^{-1}.s)) + \#(r2s(f^{-1}.s))$ (f is the bijection between $\mathcal{ABP}$ and $\mathcal{ABP}''$; recall that *rank* is defined on states of $\mathcal{ABP}''$). The quotient structure of $\mathcal{ABP}''$ with respect to B is isomorphic to the structure induced by *rep*. $\mathcal{ABP}'$ is this structure, except with *s2r* and *r2s* hidden. It is $\mathcal{ABP}'$ that we analyze in the next two subsections. By theorem 11, $\mathcal{ABP}$ is a refinement of $\mathcal{ABP}'$ and properties of $\mathcal{ABP}'$ can be lifted to $\mathcal{ABP}$.

11.3.3 Model Checking

We model check the quotient structure extracted by the above mentioned procedure using the Mu-Calculus model checker defined in chapter 9. We check the following formulas:

1. $\mathsf{AG}(sending1 \Rightarrow \mathsf{A}(sending1 \;\mathsf{W}\; rmsg = 1))$
2. $\mathsf{AG}(receiving1 \Rightarrow \mathsf{A}(receiving1 \;\mathsf{W}\; delivered1))$
3. $\mathsf{AG}\ \mathsf{EF} svalid$ (acceptance of a new message is always eventually possible)

where *sending1*, *receiving1*, and *delivered1* are abbreviations for $svalid \wedge smsg = 1$, $rvalid \wedge rmsg = 1$, and $\neg rvalid \wedge rmsg = 1$, respectively; formulas analogous to 1 and 2 are proved for message 0. W is the weak until operator which is defined as follows $f \mathsf{W} g \equiv f \mathsf{U} g \vee \mathsf{G} f$. All of the above formulas hold on the extracted structure, which is what one would expect.

The property $\mathsf{AG}\ \mathsf{AF}\mathit{svalid}$ (acceptance of a new message is always eventually guaranteed), however, does not hold without further fairness assumptions.

The liveness properties are as follows. Each property is shown under a set of fairness assumptions on the actions of the process. These are either weak fairness (infinitely often disabled or infinitely often executed) or strong fairness (infinitely often enabled implies infinitely often executed).

1. $\mathsf{AG}(\mathit{sendingNew1} \Rightarrow \mathsf{A}(\mathit{sending1}\ \mathsf{U}\ \mathit{rmsg} = 1))$ ($\mathit{sendingNew1}$ represents the sending of a new copy of message 1): This holds under weak fairness on the Send-msg and Reply actions, and strong fairness on the receipt of a new message by the action Get-msg. A similar property holds for message 0.

2. $\mathsf{AG}\ \mathsf{AF}\mathit{svalid}$: This holds under the fairness assumptions for the previous property, along with weak fairness on the Send-ack action and strong fairness on the receipt of a new acknowledgment by the action Get-ack.

Since the fairness conditions mention actions, we compose Büchi automata accepting fair paths with the quotient structure and model check the resulting structure on formulas which refer both to the propositions of the quotient structure and the accepting states of the automata.

We use an argument based on bisimulation to derive sufficient conditions for data-independence [Wol86] of the protocol. These are verified in ACL2; as a consequence, the properties shown above for the data domain $\{0, 1\}$ suffice to show similar properties for *arbitrary* data domains.

Rule	Definition
Accept.m	$\neg svalid \;\rightarrow\; smsg, svalid \;:=\; m, \mathbf{true}$
Send-msg	$svalid \;\wedge \neg rvalid \;\wedge \neg sent \;\rightarrow$
	$rvalid, sent, rmsg \;:=\; \mathbf{true}, \mathbf{true}, smsg$
Ready	$sent \;\rightarrow\; svalid, sent \;:=\; \mathbf{false}, \mathbf{false}$
Reply	$rvalid \;:=\; \mathbf{false}$

Table 11.3: Rules defining the transition relation of the non-lossy protocol.

11.3.4 Stuttering Bisimulation Checking

In many cases, the correctness proof is more convincing if we can show that the extracted model is stuttering bisimilar to a model that is so simple, it is correct by inspection. In the case of the alternating bit protocol, we can show that the extracted model is stuttering bisimilar to a simple, non-lossy version of the protocol, presented in Table 11.3.

We use a WEB equivalence checker (based on the algorithm in [BCG88]) written in ACL2 to verify that the non-lossy protocol in Table 11.3 and the extracted protocol are related by a WEB. The main idea is that we create the disjoint union of the transition systems corresponding to the extracted protocol and the non-lossy protocol. The algorithm computes the coarsest WEB on a structure; hence, if the initial states of the two systems are in the same class, the two systems are WEB. In computing the coarsest WEB, we examine only *svalid*, *smsg*, *rvalid*, and *rmsg*. Notice that this view is exactly the one presented in figure 11.1.

11.3.5 Remarks

We tried to prove the alternating bit protocol correct using only theorem proving techniques. This required constructing an invariant and was more tedious because it required understanding the precise relationship between the flags, valid bits, and channels. The reason an invariant was not required with our combined approach is that our algorithm extracted the reachable states, thereby generating the strongest invariant automatically. Thus, using our approach, it is possible to reduce the amount of manual effort required for the verification of reactive systems.

Mechanical verification is necessary. In our case, we managed to convince ourselves that a candidate relation was a WEB for the alternating bit protocol, even though it was not; this became clear only when we tried to prove it mechanically.

11.4 Bibliographic Notes

Among related work, [MN95] prove safety properties of the alternating bit protocol by using Isabelle/HOL to prove that a manually constructed finite-state system contains all of the traces of the alternating bit protocol and then model check the finite-state system. [COR+95] show the correctness of an infinite-state system by using PVS to verify that a simple manually constructed finite-state system is a conservative approximation of the infinite-state system. The work described in this paper improves upon such methods by (i) using a (verified) representative function to *automatically* construct a quotient structure, and (ii) using WEBs instead of simulations or trace containment: this allows us to check properties *exactly*, *i.e.*, if a property holds (fails) on the simple system, then it holds (fails) on the original system.

There are several known types of infinite-state systems (*e.g.*, [ACD90, GS92, AJ96, EN95]) for which the model checking problem is decidable, but these types of systems often turn out to be too specialized for many cases where it is possible to devise finite abstractions. There have been several approaches to automatically verifying the alternating bit protocol: safety properties of such lossy channel systems are decidable [AJ96]; however, in order to construct automatic abstractions that demonstrate liveness properties, most other verifications of the alternating bit protocol (*e.g.*, [GS97]) consider channels to be bounded. The work in this chapter appeared in [MNS99].

11.5 Summary

In this chapter, we showed how to use our approach to combining theorem proving and model checking to verify the alternating bit protocol, an infinite-state protocol. The idea was to reduce the size of the buffers by showing that states with the same compressed channels have the same behaviors. This amounts to a WEB proof, which was carried out in ACL2. The proof did not require that we prove auxiliary invariants. We then used our extraction algorithm to extract the quotient structure, which we model checked. We compared our proof to one using only theorem proving methods and found that a significant reduction in manual effort was achieved with our approach of combining theorem proving and model checking.

Chapter 12

Correctness of Pipelined Machines

12.1 Introduction

The correctness of pipelined machines is a subject that has been studied extensively. Most of the recent work has used variants of the Burch and Dill notion of correctness [BD94]. As new features are modeled, *e.g.*, interrupts, new notions of correctness are developed. Given the plethora of correctness conditions, the question arises: what is a reasonable notion of correctness? To motivate the need for a new notion of correctness we show that the variant of the Burch and Dill notion of correctness [BD94] used by Sawada can be satisfied by incorrect machines. We propose a notion of correctness based on WEBs.

To test the utility of the idea, we use ACL2 to verify several variants of a simple pipelined machine described by Sawada [Saw99, Saw00]. Our variants extend the basic machine by adding exceptions (to deal with overflows), interrupts, and fleshed-out 128-bit ALUs (one of which is described in a netlist language). In all cases, we prove the same final theorem. We develop a methodology with mechanical support that we use to verify Sawada's machine. Our proofs contain no intermediate abstractions and are almost automatic. In fact, for Sawada's example, our proof is substantially shorter than the original: given the definitions and some general-purpose books (collections of theorems), the proof is automatic. For some of the variants, we use the compositionality of WEB refinements (theorem 11 on page 65) to prove correctness in stages. An advantage of proving correctness in this way is that we can limit the difference between the machine descriptions from one stage to the next. This allows the proofs go through automatically because there is enough structural similarity between machines that ACL2 can establish equivalence forthwith. Yet another advantage is that changes to the lower-level machines can be localized.

In section 12.2 we present a simple machine and motivate our notion of correctness. In section 12.3 we describe several variants of a simple pipelined machine described by Sawada in [Saw99, Saw00]. The variants include exceptions, fleshed-out ALUs, interrupts, and combinations of these features. In all cases, we prove the same final theorem. In section 12.4 we discuss the books and methodology we developed to automate proofs of pipelined machines. Bibliographic notes appear in section 12.5 and a summary in section 12.6.

12.2 A Simple Pipelined Machine

The specification used to prove a pipelined machine correct is an instruction set architecture (ISA). The ISA describes the interface between the hardware and software and contains the programmer visible components of the machine. A pipelined machine is correct if it satisfies a certain relationship with the ISA. There is no wide agreement on the "right" notion of correctness, but perhaps the most common approach is that of Burch and Dill [BD94]. One of the difficulties with specifying correctness is that we want to account for non-terminating behavior. If we were to restrict ourselves to terminating programs, we could say that a pipelined machine is correct if for any terminating program, both the pipelined machine and the ISA machine halt in the same final state. However, there are interesting non-terminating programs such as operating systems and transmission protocols that run on these machines and the traditional approach of stating correctness as a relationship between initial and final states cannot be used, as there is no final state. We are therefore forced to think about infinite computations.

We start with an example. Consider a simple ISA machine with instructions that are four-tuples consisting of an opcode, a target register, and two source registers. The state components of the ISA machine that are of interest are the program counter and the contents of registers ra and rb. The MA (micro architecture) machine is a pipelined machine with three stages. A pipeline is analogous to an assembly line. The pipeline consists of several stages each of which performs part of the computation required to complete an instruction. When the pipeline is full many instructions are in various degrees of completion. A diagram of the MA machine appears in figure 12.1. The three stages are fetch, set-up, and write. During the fetch stage, the instruction pointed to by the PC (program counter) is retrieved from memory and placed into latch 1. During the set-up stage, the contents of the source registers (of the instruction in latch 1) are retrieved from the register file and sent

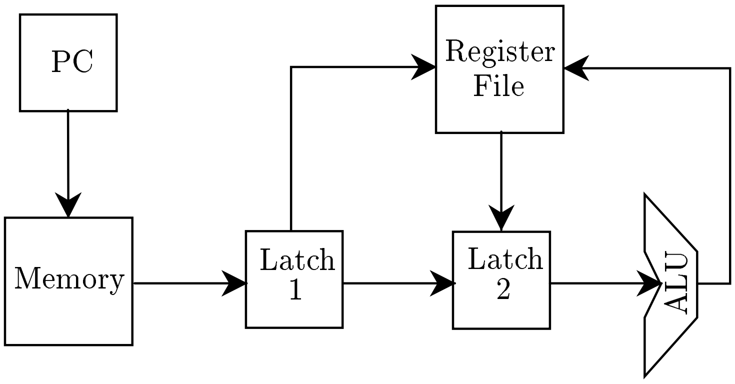

Figure 12.1: A simple three-stage pipelined machine.

Inst				
0	add	rb	ra	ra
1	add	ra	rb	ra

Table 12.1: The contents of memory.

to latch 2 along with the rest of the instruction in latch 1. During the write stage, the appropriate ALU (arithmetic logic unit) operation is performed and the result is used to update the value of the target register.

Suppose that the contents of memory are as shown in table 12.1.

When this simple two-line code fragment is executed on the ISA and MA machines, we get the traces shown in table 12.2.

The rows correspond to steps of the machines, *e.g.*, row Clock 0 corresponds to the initial state, Clock 1 to the next state, and so on. The ISA and MA columns contain the relevant parts of the state of the machines: a pair consisting of the PC and the register file (itself a pair consisting of registers ra and rb). The contents of the register file of the ISA machine are numbers in

Clock	**ISA**	**MA**	**Inst 0**	**Inst 1**
0	$\langle 0, \langle 1,1\rangle\rangle$	$\langle 0, \langle 01,01\rangle\rangle$		
1	$\langle 1, \langle 1,2\rangle\rangle$	$\langle 1, \langle 01,01\rangle\rangle$	Fetch	
2	$\langle 2, \langle 3,2\rangle\rangle$	$\langle 2, \langle 01,01\rangle\rangle$	Set-up	Fetch
3		$\langle 2, \langle 01,10\rangle\rangle$	Write	Stall
4		$\langle \_, \langle 01,10\rangle\rangle$		Set-up
5		$\langle \_, \langle 11,10\rangle\rangle$		Write

Table 12.2: ISA and MA traces.

decimal and the contents of the register file of the MA machine are bit-vectors (we show only the two low-order bits). The final two columns indicate what stage the instructions are in (only applicable to the MA machine).

In the initial state (in row Clock 0) the PCs of the ISA and MA machines contain the value 0 (indicating that the next instruction to execute is Inst 0) and both registers have the value 1. In the next ISA state (in row Clock 1), the PC is incremented and the add instruction performed, *i.e.*, register rb is updated with the value ra + ra = 2. The final entry in the ISA column contains the state of the ISA machine after executing Inst 1.

After one step of the MA machine, Inst 0 completes the fetch phase and the PC is incremented to point to the next instruction. After step 2 (in row Clock 2), Inst 0 completes the set-up stage, Inst 1 completes the fetch phase, and the PC is incremented. After step 3, Inst 0 completes the write-back phase and the register file is updated for the first time with rb set to 10 (2 in binary). However, Inst 1 is stalled during step 3 because one of its source registers is rb, the target register of the previous instruction. Since the previous instruction has not completed, the value of rb is not available and Inst 1 is stalled for one cycle. In the next cycle, Inst 1 enters the set-up stage and Inst 2 enters the

ISA	**MA**		**MA**		**MA**
$\langle 0, \langle 1,1 \rangle\rangle$	$\langle 0, \langle 1,1 \rangle\rangle$		$\langle 0, \langle 1,1 \rangle\rangle$		$\langle 0, \langle 1,1 \rangle\rangle$
$\langle 1, \langle 1,2 \rangle\rangle$	$\langle 1, \langle 1,1 \rangle\rangle$	$\longrightarrow$	$\langle 0, \langle 1,1 \rangle\rangle$	$\longrightarrow$	$\langle 1, \langle 1,2 \rangle\rangle$
$\langle 2, \langle 3,2 \rangle\rangle$	$\langle 2, \langle 1,1 \rangle\rangle$	Commit	$\langle 0, \langle 1,1 \rangle\rangle$	Remove	$\langle 2, \langle 3,2 \rangle\rangle$
	$\langle 2, \langle 1,2 \rangle\rangle$	PC	$\langle 1, \langle 1,2 \rangle\rangle$	Stutter	
	$\langle \_, \langle 1,2 \rangle\rangle$		$\langle 1, \langle 1,2 \rangle\rangle$		
	$\langle \_, \langle 3,2 \rangle\rangle$		$\langle 2, \langle 3,2 \rangle\rangle$		

Table 12.3: How to relate the ISA and MA traces.

fetch stage (not shown). Finally, after step 5, Inst 1 is completed and register ra is updated.

Comparing the partial traces of the ISA and MA machines and thinking about how to relate them makes it clear that we should stick to one representation of numbers. In table 12.3 the partial traces of the ISA and MA machines appear in the first two columns, with numbers represented in decimal.

Notice that the PC differs in the two traces and this occurs because the pipeline, initially empty, is being filled and the PC points to the next instruction to fetch. If the PC were to point to the next instruction to commit (*i.e.*, the next instruction to complete), then we would get the trace shown in column 3. Notice that in column 3, the PC does not change from 0 to 1 until Inst 0 is committed in which case the next instruction to commit is Inst 1. We now have a trace that is the same as the ISA trace except for stuttering; after removing the stuttering we have, in column 4, the ISA trace.

To state correctness we use a refinement map. In the above example we mapped MA states to ISA states by transforming bit-vectors into decimal numbers and by transforming the PC. Proving correctness amounts to relating MA states with the ISA states they map to under the refinement map and proving a WEB. Proving a WEB guarantees that MA states and related ISA states have related computations up to finite stuttering. This is a strong notion of equivalence. As we have seen, a consequence is that the two machines satisfy the same $\mathrm{CTL}^* \setminus \mathsf{X}$ properties *e.g.*, one such property is that the MA machine cannot deadlock (because the ISA machine cannot deadlock).

Why "up to finite stuttering"? Because we are comparing machines at different levels of abstraction: the pipelined machine is a low-level implementation of the high-level ISA specification. When comparing systems at different levels of abstraction, it is often the case that the low-level system requires several steps to match a single step of the high-level system.

Why use a refinement map? Because data can be represented in different ways, *e.g.*, the MA machine represents numbers in binary whereas the ISA machine uses a decimal representation. In addition, there may be components in one system that do not appear in the other, *e.g.*, the MA machine has latches but the ISA machine does not. Yet another reason is that components present in both systems may have different behaviors, as is the case with the PC above. Notice that the refinement map affects how MA and ISA states are related, not the behavior of the MA machine.

Some key observations to keep in mind as you read the chapter follow. Note that we prove the same theorem for each of the pipelined machine variants, including the variant with interrupts and exceptions. Other approaches introduce new notions of correctness to deal with such features [Saw99]. Our characterization of correctness allows us to prove an MA machine correct with respect to an ISA machine by considering only single steps of the machines. For the examples we consider, this leads to dramatically shorter proofs (as already mentioned, some proofs are automatic) and does not require intermediate abstractions.

12.3 Pipelined Machine Verification

In this section, we describe various versions of Sawada's simple pipelined machine [Saw99, Saw00] and the correctness criteria proved. We discuss correctness, the deterministic variants, and finally the non-deterministic ones.

Machines are modeled as functions in ACL2, *e.g.*, the first machine we define is `ISA` and this amount to defining `ISA-step`, a function that given an `ISA` state returns the next state. For all the machines, an instruction is a four-tuple consisting of an opcode, a target register, and two source registers.

12.3.1 Correctness

In the introduction, we made the case that any notion of correctness can be thought of as a constraint. Pipelined machines that satisfy the constraint are "correct" implementations of the ISA, with respect to this notion of correctness. We can judge the merits of a notion of correctness by checking that no obviously incorrect machine satisfies the related constraint. We argued that the refinement maps should be understandable, *e.g.*, if we map MA states to ISA states then there should be a clear relationship between related states. Applying these criteria to the Burch and Dill variant under consideration, we find that in both respects, this notion of correctness is incomplete. We discuss the issues in the next section.

The notion of correctness that we use is simple to state: we show that the pipelined machine is a refinement of the ISA specification. Notice that any notion of correctness has to account for stuttering, *e.g.*, a pipelined machine requires several cycles to fill the pipeline, whereas an ISA machine executes an instruction per cycle; any notion of correctness also has to account for refinement, *e.g.*, a pipelined machine may represent numbers as bit-vectors, whereas the ISA machine may represent them directly. Our notion of correctness is the strongest notion that we can think of which accounts for stuttering and refinement. We discuss the verification of a number of machines, but, regardless of the proof details, we always prove the same theorem, *viz.*, that a pipelined machine has the same infinite paths (up to stuttering and refinement) as an ISA machine.

12.3.2 Deterministic Machines

The deterministic machines are named `ISA`, `MA`, `ISA128`, `MA128`, `MA128serial`, and `MA128net`. `ISA` and `MA` correspond to the machines in [Saw00] and the simple machines in [Saw99]. We start with descriptions of `ISA` and `MA` and compare the Burch and Dill approach to correctness with ours.

ISA

An `ISA` state is a three-tuple consisting of a program counter (pc), a register file, and a memory. The next state of an `ISA` state is obtained by fetching the instruction pointed to by the pc from memory, checking the opcode, and performing the appropriate instruction. The possible instructions are addition, subtraction, and noop. In the case of addition, the target register is updated with the sum of the values in the source registers and the program counter is incremented. Subtraction is treated similarly. In the case of a noop, only the program counter is incremented. `ISA` is the specification for `MA`.

MA

`MA` is a three-stage pipelined machine. An `MA` state is a five-tuple consisting of a pc, a register file, a memory, and two latches. The three stages are the fetch stage, the set-up stage, and the write-back stage. During the fetch stage, instructions are fetched from memory and stored in the first latch; during the set-up stage, the instruction in the first latch is passed to the second latch, but with the values of the source registers; during the write-back stage the values and the opcode are fed to the ALU which performs the appropriate instruction and the result is written into the target register, if the instruction was not a noop. The `MA` machine can execute one instruction per cycle once the pipeline is full, except when there are successive arithmetic instructions where the second instruction uses the target register of the first instruction as a source register. In this case, the machine is stalled for one cycle in order for the target register to be updated.

One difference between `MA` as defined above and the version given by Sawada (`SMA`) is that `SMA` has an extra input signal which determines whether the machine can fetch an instruction. With the use of this signal, `SMA` can be flushed, whereas we have no way of flushing `MA`.

Comparison with the Burch and Dill Notion of Correctness

The proof of `SMA` given in [Saw00, Saw99] uses a variant of the Burch and Dill notion of correctness. The main theorem proved is that if the `SMA` starts in SMA_0, a flushed state, and takes n steps to arrive at state SMA_n, also a flushed state, then there is some number m such that stepping the projection of SMA_0 m steps results in the projection of SMA_n. The projection of an `SMA` state is the ISA state obtained from the program counter, register file, and memory of the SMA state. Since this notion of correctness requires a pipelined machine that can be flushed, the machine defined by Sawada has an extra input signal which can be used to flush the machine. We have no way (and no need) to flush MA.

Sawada also proves the "liveness" theorem that any pipelined state can be flushed. These two theorems constitute his notion of correctness. We ask the informal question, "Are there any pipelined machines that are obviously incorrect but satisfy this notion of correctness?" If you consider deadlock an abhorrent behavior, the answer if yes. Using Sawada's proof scripts, we provide a mechanically checked proof that the trivial pipelined machine with a next-state function which invalidates the latches and keeps the PC, memory, and register file intact satisfies this notion of correctness. The proof is straightforward. The first theorem is established by choosing m to be 0. The second theorem holds because the next state function invalidates the latches; therefore, the next state of any state is flushed. More insidious machines also satisfy this notion of correctness (but we do not provide mechanical proofs). For example, a machine that sometimes enters a deadlock state can be proven correct using a similar argument. As a further example, consider a machine that sometimes enters a livelock cycle by performing the following three operations forever.

1. The two latches are invalidated and everything else remains the same.

2. The next instruction is fetched, but the program counter is not changed. After this step latch1 is valid and latch2 is invalid.

3. No new instruction is fetched, but the instruction in latch1 is sent to

latch2.

This cycle modifies MA components, but does not make any progress at the ISA level.

Flushing Proof of MA

Even though there is no way to flush MA, we can use flushing to prove that MA is a refinement of ISA. This digression allows us to discuss issues related to refinement maps. One approach is to modify MA so that it can be flushed (which would give us SMA), but this is a different machine. The approach we take is to define an auxiliary function that flushes an MA state. Using this function we show that MA is a refinement of ISA. Notice that in contrast to the proof of SMA, there is no trivial pipelined machine that satisfies this notion of correctness. This is because proving a WEB between a pipelined machine and ISA implies that any ISA behavior can be matched by the pipelined machine; since ISA has non-trivial behaviors so does the pipelined machine. Even so, this proof is not entirely satisfactory and this has to do with the use of flushing as a refinement map.

When we define systems at different levels of abstraction, we often find that there are inconsistent states, *e.g.*, consider an MA state in which the first latch contains an instruction that is not in memory. This is usually dealt with by considering only “good” (reachable) MA states. The flushing-based refinement imposes no such restriction because pipelined machines are self-stabilizing: from any state they eventually reach a good state and flushing guarantees this. As a result, the refinement map—which is supposed to show us how to view MA states as ISA states—relates inconsistent MA states with consistent `ISA` states (all `ISA` states are "good"). Since both `MA` and `ISA` are typed transition systems (see page 48), we can apply lemma 26 to see that `ISA` states and related `MA` states have the same behaviors, up to stuttering, when the labeling function is restricted to the program counter, this is because refinement maps based on flushing modify important programmer visible components such as the register file. In contrast, the refinement maps based on our approach only modify the program counter, so that using lemma 26, we can show that `ISA` and related `MA` states satisfy the same behaviors, up to stuttering, when the labeling function is used to hide the program counter, but leaves the rest of the `ISA` visible components untouched. Therefore, we find the use of refinement maps based on flushing objectionable.

Proof of MA

The approach we take to pipelined machine verification in the rest of this paper is to prove a WEB where the refinement map relates pipelined machine states to the `ISA` states obtained by retaining the programmer visible components of the committed part of the pipelined state. The definition of the refinement map is based on the function `committed-MA` which takes an `MA` state and returns the `MA` state obtained by invalidating all partially completed instructions and moving the program counter back based on the number of partially completed instructions. As mentioned previously, "good" MA states are the ones reachable from a committed state. The function `good-MA` recognizes `MA` state s if `committed-MA`.s, stepped the appropriate number of times, is s. Notice that as with flushing, this function is easy to define because we can use the definition of `MA`. In fact, it is simpler to define than the flushing operation because we are not trying to get the machine into a special state: we are just stepping it. Notice that the use of `good-MA` allows us to avoid defining an invariant (an error prone process), hence, we maintain this methodological feature of the Burch and Dill approach. We call this approach the "commitment approach."

ISA128

An `ISA128` state is a four-tuple consisting of a program counter (pc), a register file, a memory, and an exception flag. The next state of an `ISA128` state is obtained by fetching the instruction pointed to by the pc from memory, checking the opcode, and performing the appropriate instruction. The possible instructions are addition, multiplication, and noop. In the case of addition, the program counter is incremented and the target register is modified to contain *sum*, the sum of the values in the source registers if the sum is less than 2^{128}. Otherwise, if an overflow occurs, the exception flag is checked; if it is off, then the program counter is incremented and the target register is assigned *sum* (mod 2^{128}); if the exception flag is on, the exception handler is called. The exception handler is a constrained function of the program counter, register file, and memory that returns a new program counter, register file, memory, and exception flag. (A function about which we know only that it satisfies some specified properties is called a constrained function. An uninterpreted function is a special case of a constrained function.) Multiplication is treated similarly. In the case of a noop, only the program counter is incremented. `ISA128` is the specification for `MA128`, `MA128serial`, and `MA128net`.

MA128, MA128serial, and MA128net

`MA128` is a three-stage pipelined machine. An `MA128` state is a six-tuple consisting of a program counter, a register file, a memory, two latches, and an exception flag. As with `MA`, the three stages are the fetch stage, the set-up stage, and the write-back stage. If an overflow occurs during an arithmetic operation, then the partially executed instructions are invalidated and the exception handler is called. The resulting state is constrained to be flushed (*i.e.*, both latches are invalid). We prove that `MA128` is a refinement of `ISA128` using the commitment approach.

`MA128serial` is the same as `MA128`, except that the ALU is defined in terms of a serial adder and a multiplier based on the adder. The adder, multiplier, and proof of their correctness are taken from [KMM00b]. We used the commitment approach to prove that `MA128serial` refines `MA128`. Since the ALU of `MA128` operates on bit-vectors, the refinement map used maps the bit-vectors in the register file and the second latch to numbers. By theorem 11 (composition), we get that `MA128serial` is a refinement of `ISA128`. Although the use of the composition theorem here was not essential, it was nice to be able to break up the proof into these two logically separate concerns. ACL2 can take advantage of the structural similarity between `MA128serial` and `MA128`, hence the proof of correctness is pretty fast. This is covered in more detail later.

`MA128net` is the same as `MA128`, except that the ALU is defined in terms of an adder described in a netlist language. The netlist adder is a 128-bit adder and is described in terms of primitive functions on bits. We have a function that generates an adder of any size and we prove that the adder generated is correct by relating it to the serial adder (as in the FM8501 [Hun94]). We prove that `MA128net` is a refinement of `MA128serial`, hence by composition, a refinement of `ISA128`.

12.3.3 Non-Deterministic Machines

We now consider the non-deterministic versions of the six deterministic machines described above. The names of these machines are: `ISAint`, `MAint`, `ISA128int`, `MA128int`, `MA128intserial`, and `MA128intnet`. They are elaborations of the similarly named deterministic machines, except that they can be interrupted. Whereas the next state of the deterministic machines is a function of the current state (even in the presence of exceptions), the next state of the machines described in this section also depends on the interrupt signal, which is free. Therefore, the machines in this section are non-deterministic.

The approach in [Saw99] to dealing with interrupts is different. There, the correctness criterion is: if M_0 is a flushed state and if taking n steps where the interrupts at each step are specified by the list l results in a flushed state M_n, then there is a number n' and a list l' such that stepping the projection of M_0 n' steps with interrupt list l' results in the projection of M_n. Notice that a machine which always ignores interrupts satisfies this specification.

In our approach, we have to show that the pipelined machine is a refinement of the specification, as before. Note that this is the same final theorem we proved in the deterministic case, as WEBs can be used to relate non-deterministic systems. Therefore, our notion of correctness cannot be satisfied by a pipelined machine that ignores interrupts. Another advantage is that our proof obligation is still about single steps of the machines, as opposed to finite behaviors. The problem with the finite behaviors approach was highlighted above: when comparing finite executions of a pipelined machine and of its specification, there are executions with different lengths; how does one relate interrupts in one execution with interrupts in the other?

ISAint

An `ISAint` state is a four-tuple consisting of a program counter (pc), a register file, a memory, and an interrupt register. The next state of an `ISAint` state is obtained by first checking the interrupt register. If non-empty, the interrupt handler is called. The interrupt handler is a constrained function of the register file, memory, and interrupt register and returns a state with the same pc and register file, but may modify memory, and clears the interrupt register. If the interrupt register is empty, we check if an interrupt has been raised. If so, we record the interrupt type in the interrupt register. If not, we proceed by fetching the instruction pointed to by the pc from memory, checking the opcode, and performing the appropriate instruction. The possible instructions are addition, multiplication, and noop. In the case of addition, the target register is modified to contain the sum of the values in the source registers and the program counter is incremented. Multiplication is treated similarly. In the case of a noop, only the program counter is incremented. `ISAint` is the specification for `MAint`.

MAint

`MAint` is a three-stage pipelined machine. An `MAint` state is a six-tuple consisting of a pc, a register file, a memory, two latches, and an interrupt register. The three stages are the fetch stage, the set-up stage, and the write-back stage, as before. The next state of an `MAint` state is obtained by first checking the interrupt register. If non-empty, partially executed instructions are aborted and the interrupt handler is called. Otherwise we check if an interrupt has been raised, in which case we abort partially executed instructions and set the interrupt register. Otherwise, execution proceeds in a fashion similar to the execution of `MA`. The refinement map used to show that `MAint` is a refinement of `ISAint` is almost identical to the one used to show that `MA` is a refinement of `ISA`, except that the interrupt register is also retained.

ISA128int

As the name implies this is the ISA-level specification of 128-bit ALU machine with exceptions and interrupts. Interrupts are given priority, and this machine is defined the way you would expect: it is similar to `ISAint`, except that arithmetic operations are checked for overflows, in which case the exception handler is called. This machine is the specification used for the machines `MA128int`, `MA128intserial`, and `MA128intnet`.

MA128int, MA128intserial, and MA128intnet

`MA128int`, `MA128intserial`, and `MA128intnet` are three-stage pipelined machines analogous to `MA128`, `MA128serial`, and `MA128net`, respectively, but with exceptions. As before, we show that `MA128int` is a refinement of `ISA128int`, that `MA128intserial` is a refinement of `MA128int` (where the refinement map converts bit-vectors to integers) and finally that `MA128intnet` is a refinement of `MA128intserial`. By the composition theorem, we get that all of these machines are refinements of `ISA128int`.

12.4 Proof Decomposition

To reduce the amount of guidance that has to be manually given to the theorem prover, we develop a methodology with mechanical support which we use to verify the various variants of Sawada's machine. In section 12.4.1, we describe some of the supporting books used. In section 12.4.2 we discuss the macros used to automate the proofs.

12.4.1 Supporting Books

We use several general-purpose books to support automation. These include the standard `"top-with-meta"` and `"ihs"` books for reasoning about arithmetic as well as the books `"nth-thms"`, `"alist-thms"`, and `"defun-weak-sk"` for reasoning about `nth` and `update-nth`, alists, and quantification, respectively.

To prove correctness, we define interpreters for the ISA and MA machines and prove that the MA machine is a refinement of the ISA machine. Machine states are represented as lists and components of states are accessed and updated with `nth` and `update-nth`, respectively. The proof requires that we compare components of stepped states and much of this can be done automatically with rewrite rules that simplify and normalize `nth` and `update-nth` expressions. The book `"nth-thms"` contains the rewrite rules we found useful for this purpose and is based on the approach taken by Greve, Wilding, and Hardin on page 131 of reference [GWH00]. We also use alists (*e.g.*, register files are represented as lists of register name, value pairs) and the book `"alist-thms"` contains some simple rules, similar to those in `"nth-thms"`, for reasoning about alists.

The book `"defun-weak-sk"` is used to reason about existential quantification. Recall that the macro `defun-sk` is used to implement quantification in ACL2 by introducing witness functions and constraints. For example, the quantified formula $\langle \exists x :: P(x,y) \rangle$ can be rendered in ACL2 as the function `EP` with the constraints `(P x y)` $\Rightarrow$ `(EP y)` and `(EP y)` $=$ `(P (W y) y)`. To see that this corresponds to quantification, notice that the first constraint gives us one direction of the argument: it says that if any value of `x` makes `(P x y)` true (*i.e.*, if $\langle \exists x :: P(x,y) \rangle$) then `(EP y)` is true. This constraint allows us to establish an existentially quantified formula by exhibiting a witness, but the constraint can be satisfied if `EP` always returns `t`. The second constraint gives us the other direction. It introduces the witness function `W` and requires that `(EP y)` is true iff `(P (W y) y)` is true. As a result, if `(EP y)` is true, then some value of `x` makes `(P x y)` true. As is mentioned in the ACL2 documentation [KM], this idea was known to Hilbert. An ACL2 script corresponding to the above follows.[1] Notice that the constraints on `EP` are the constraints on `EP-witness` (which corresponds to our `W`).

```
(defstub P(x y) t)
(defun-sk EP (y)
  (exists (x) (P x y)))
:props EP
:props EP-witness
```

We wish to use quantification and encapsulation in the following way. We prove that a set of constrained functions satisfies a quantified formula. We then use functional instantiation [BGKM91, KM00] to show that a set of functions satisfying these constraints also satisfies the (analogous) quantified formula. We want this proof obligation to be generated by macros but have found that the constraints generated by the quantified formulas complicate the

[1] `Props` shows all of the properties in the ACL2 world that are associated with a symbol.

design of such macros. The following observation has allowed us to simplify the process. The quantified formulas are established using witness functions, as is often the case. Therefore, only the first constraint generated by `defun-sk` is required for the proof. We defined the macro `defun-weak-sk` which generates only this constraint, *e.g.*, executing the ACL2 script

```
(defstub P(x y) t)
(defun-weak-sk E (y)
  (exists (x) (P x y)))
:props E
```

shows that the only constraint on `E` is `(P x y)` $\Rightarrow$ `(E y)`. By functional instantiation, any theorem proven about `E` also holds when `E` is replaced by `EP` (since `EP` satisfies the constraint on `E`). We use `defun-weak-sk` in our scripts and at the very end we prove the `defun-sk` versions of the main results by functional instantiation (a step taken to make the presentation of the final result independent of our macros).

12.4.2 Proof Outline

The books specific to the proof are `"ISA"`, `"MA"`, and `"MA-ISA"`. `"ISA"` and `"MA"` contain the definitions of `ISA-step` and `MA-step`, the functions to step (*i.e.*, compute the next state of) the `ISA` and `MA` machines, respectively. The `"MA-ISA"` book contains the definition of the refinement map, `MA-to-ISA`, and the function recognizing "good" `MA` states, `good-MA`. These functions are described in section 12.3. Also included is the definition of a rank function, `MA-rank`. The rank function is very simple: it is either 0, 1, or 2 based on how many steps it takes MA to commit an instruction.

To complete the proof it seems we have to: define the machine corresponding to the disjoint union of ISA and MA, define a WEB that relates a (good) MA state s to (MA-to-ISA s), define the well-founded witness, and prove that indeed the purported WEB really is a WEB. We have implemented macros which automate this. The macros are useful not only for this example, but also for the verification of the rest of the deterministic machines we present in this paper and can be used to show a WEB between other types of deterministic systems. (Non-deterministic versions are described later in this section.) The proof of correctness is completed with the following three macro calls (in the book "MA-ISA").

```
(generate-full-system isa-step isa-p ma-step ma-p
                      ma-to-isa good-ma ma-rank)
(prove-web isa-step isa-p ma-step ma-p ma-to-isa ma-rank)
(wrap-it-up isa-step isa-p ma-step ma-p
            good-ma ma-to-isa ma-rank)
```

The first macro, generate-full-system, generates the definition of B, the purported WEB as well as R, the transition relation of the disjoint union of the ISA and MA machines. The macro translates to the following. (Some declarations and forward-chaining theorems used to control the theorem prover have been elided. In addition, in the context below, bor is equal to or.)

```
(progn
  (defun B-core (x y) ...
    (and (ISA-p x)
```

```
        (MA-p y)
        (good-MA y)
        (equal x (MA-to-ISA y))))
  (defun B (x y) ...
    (bor (B-core x y)
         (B-core y x)
         (equal x y)
         (and (MA-p x)
              (MA-p y)
              (good-MA x)
              (good-MA y)
              (equal (MA-to-ISA x) (MA-to-ISA y)))))
  (defun rank (x)  ...
    (if (MA-p x) (MA-rank x) 0))
  (defun R (x y) ...
    (cond ((ISA-p x) (equal y (ISA-step x)))
          (t (equal y (MA-step x)))))
  ...)
```

What is left is to prove that `B`—the reflexive, symmetric, transitive closure of `B-core`—is a WEB with well-founded witness `rank`. We do this in two steps. First, the macro `prove-web` is used to prove the "core" theorem (as well as some "type" theorems not shown).

```
(defthm B-is-a-wf-bisim-core
  (let ((u (ISA-step s))
        (v (MA-step w)))
```

```
(implies (and (B-core s w)
              (not (B-core u v)))
         (and (B-core s v)
              (e0-ord-< (MA-rank v) (MA-rank w))))))
```

Comparing `B-is-a-wf-bisim-core` with the definition of WEBs, we see that `B-is-a-wf-bisim-core` does not contain quantifiers and it mentions neither `B` nor `R`. This is on purpose as we use "domain-specific" information to construct a simplified theorem that is used to establish the main theorem. To that end we removed the quantifiers and much of the case analysis. For example, in the definition of WEBs, u ranges over successors of s and v is existentially quantified over successors of w, but because we are dealing with deterministic systems, `u` and `v` are defined to be *the* successors of `s` and `w`, respectively. Also, `B-core` is not an equivalence relation as it is not reflexive, symmetric, or transitive. Finally, we ignore the second disjunct in the third condition of the definition of WEBs because `ISA` does not stutter. The justification for calling this the "core" theorem is that we have proved in the book `"det-encap-wfbisim"` that a constrained system which satisfies a theorem analogous to `B-is-a-wf-bisim-core` (and some "type" theorems) also satisfies a WEB. Using functional instantiation we can now prove `MA` correct. The use of this domain-specific information makes a big difference, *e.g.*, when we tried to prove the theorem obtained by a naive translation of the WEB definition (sans quantifiers), ACL2 ran out of memory after 30 hours, yet the above theorem is now proved in about 11 seconds.

The final macro call generates the events used to finish the proof. We present the generated events germane to this discussion below. The first step is to show that `B` is an equivalence relation. This theorem is proved by functional instantiation of a theorem in the book "det-encap-wfbisim".

```
(defequiv B
  :hints (("goal" :by (:functional-instance
                        encap-B-is-an-equivalence ...))))
```

The second WEB condition, that related states have the same label, is taken care of by the refinement map. We show that rank is a well-founded witness.

```
(defthm rank-well-founded
  (e0-ordinalp (rank x)))
```

We use functional instantiation and theorem B-is-a-wf-bisim-core as described above to prove the following.

```
(defun-weak-sk exists-w-succ-for-u-weak (w u)
  (exists (v) (and (R w v) (B u v))))
(defun-weak-sk exists-w-succ-for-s-weak (w s)
  (exists (v)
    (and (R w v)
         (B s v)
         (e0-ord-< (rank v) (rank w)))))
(defthm
  B-is-a-wf-bisim-weak
  (implies (and (B s w)
                (R s u))
           (or (exists-w-succ-for-u-weak w u)
```

```
                (and (B u w)
                     (e0-ord-< (rank u) (rank s)))
                (exists-w-succ-for-s-weak w s)))
  :hints
  (("goal" :by (:functional-instance b-is-a-wf-bisim-sk ...)))
  :rule-classes nil)
```

We use `defun-weak-sk` for these definitions and for the proofs in the book `"det-encap-wfbisim"` for the reasons outlined in section 12.4.1. To make it easier for the general ACL2 community to understand the results, we state them in terms of the built-in macro `defun-sk`. The proof is a trivial functional instantiation, since the single constraint generated by `defun-weak-sk` is one of the constraints generated by `defun-sk`.

```
(defun-sk exists-w-succ-for-u (w u)
  (exists (v) (and (R w v) (B u v))))
(defun-sk exists-w-succ-for-s (w s)
  (exists (v)
    (and (R w v)
         (B s v)
         (e0-ord-< (rank v) (rank w)))))
...
(defthm
  B-is-a-wf-bisim
  (implies (and (B s w)
                (R s u))
           (or (exists-w-succ-for-u w u)
```

```
              (and (B u w)
                   (e0-ord-< (rank u) (rank s)))
              (exists-w-succ-for-s w s)))
:hints
(("goal"
   :by (:functional-instance B-is-a-wf-bisim-weak
          (exists-w-succ-for-u-weak exists-w-succ-for-u)
          (exists-w-succ-for-s-weak exists-w-succ-for-s))))
:rule-classes nil))
```

Theorem Proving Effort

Finally, we compare the complexity of the proofs. We consider only the books required for the proof; this includes neither the books used to define the machines nor supporting general-purpose books. We consider the books containing our macros to be part of the supporting books because they were designed for general-purpose use and have been used with the dozen or so proofs described in this paper. Since we proved the correctness of `MA` without the use of any intermediate abstractions, invariants, or user-supplied theorems, the size of the book containing the definitions required for the proof is about 3K; the size of the files containing Sawada's proof is about 94K. The time required for our proof (including the loading of related books) is about 30 seconds on a 600MHz Pentium III; the time required for Sawada's proof is about 460 seconds (on the same machine).

Non-Deterministic Machines

Non-determinism has led to new notions of correctness in the literature. For example, to deal with interrupts, a notion of correctness (still based on the Burch and Dill notion, but different from the one used for deterministic machines) is presented by Sawada [Saw99]: if M_0 is a flushed state and if taking n steps where the interrupts at each step are specified by the list l results in a flushed state M_n, then there is a number n' and a list l' such that stepping the projection of M_0 n' steps with interrupt list l' results in the projection of M_n. Notice that a machine which always ignores interrupts satisfies this specification and is therefore considered correct.

In contrast, since WEBs apply to non-deterministic systems, our notion of correctness in the presence of interrupts remains the same, *i.e.*, we prove the pipelined machine is a refinement of its ISA specification. As a consequence, a pipelined machine which ignores interrupts cannot be proven correct. Another advantage is that our proof obligation is still about single steps of the machines, as opposed to finite behaviors. As before, this makes the proof much simpler as no intermediate abstractions are required.

We end this section by describing a clear, compositional path from the verification of term-level descriptions of pipelined machines to the verification of low-level descriptions (*e.g.*, netlist descriptions). In our proof that `MA128` is a refinement of `ISA128`, the definition of the ALU is "disabled"; the result is that ACL2 treats the ALU as an uninterpreted function. However, to verify that `MA128net` is a refinement of `MA128`, we "enable" the definition of the ALU and prove theorems relating a netlist description of the circuit to a serial adder which is then related to addition on integers. This allows us to relate term-level descriptions of machines to lower-level descriptions in a compositional way.

12.5 Bibliographic Notes

Various approaches to pipelined machine verification appear in the literature. Some of the early work on pipelined machine verification was based on skewed abstraction functions [SB90, Cyr93, SM95]. The Burch and Dill notion of correctness, based on flushing and commuting diagrams, was introduced later [BD94]. Theorem-proving approaches include the work by Sawada and Hunt [SH97, SH98, Saw99, Saw00]. They use an intermediate abstraction called MAETT to verify some very complicated machines. Our machine is based on a simple machine described by Sawada [Saw99, Saw00]. We used this machine because the proof scripts are publicly available and because of the ubiquity of the Burch and Dill approach to pipelined machine verification. Sawada uses this machine to explain issues of correctness and proof techniques. It is a toy version of the FM9801, the final machine verified in Sawada's thesis [Saw99]. The verification of the FM9801 is substantial and impressive. A line of research that includes both hand proofs and mechanical proofs of various types of pipelined machines has been conducted by Pneuli and others [DP97, PA98, AP99, AP00]. Another theorem-proving approach is presented in [HSG98, HSG99, HGS99], where "completion functions" are used to decompose the abstraction function. Yet another theorem-proving approach where a synthesizable design is verified at the gate level is [Kro01]. In [WC94], a related notion of correctness based on state and temporal abstraction is used to verify a pipelined machine. Model checking approaches include the use of symmetry reductions and compositional model checking [McM98] and the use of assume-guarantee reasoning [HQR99]. In addition, decision procedures for Boolean logic with equality and uninterpreted function symbols [BGV99, PRSS99] have been used to verify pipelined machines [BGV99].

The work reported in this chapter has been published in two papers. One paper deals with notions of correctness for pipelined machines [Man00a] and the other discusses the ACL2 techniques and books used to automate the process [Man00d].

12.6 Summary

We have presented an approach to pipelined machine correctness based on WEBs and argued that only machines which truly implement the instruction set architecture satisfy our notion of correctness. In contrast, we showed with mechanical proof that the Burch-Dill variant of correctness used in [Saw99, Saw00] can be satisfied by pipelined machines which deadlock. We verified various extensions to the simple pipelined machine presented in [Saw99, Saw00]. Our extensions include exception handling, interrupts, and ALUs described in part at the netlist level. In every case, we proved the same final theorem. In addition, we showed how to use the compositionality of WEBs and ACL2's functional instantiation to relate term-level descriptions to netlist-level descriptions. All of the proofs were done within one logical system and we thus avoided the semantic gaps that could otherwise result. To automate the proofs, we developed a methodology with mechanical support that is used to generate and discharge the proof obligations. The main proof obligation generated contains no quantifiers and has minimal case analysis, but once proven is used to automatically infer the WEB. This is done with the functional instantiation of a decomposition theorem that is part of the mechanical support for WEBs that we provide. The reasons for the simplicity of our the proofs include:

1. We prove a theorem about a constrained system that is then invoked with functional instantiation. This allows us to bypass much of the case analysis and the reasoning about quantifiers that would otherwise be required.

2. We implemented macros that generate the disjoint union of the ISA and MA machines and that generate proof obligations which take advantage of the analysis mentioned immediately above.

3. Our notion of correctness can be proved by reasoning about single steps of the machines (as opposed to reasoning about an arbitrary number of steps). This helps us avoid the use of intermediate abstractions.

4. We used the compositionality of WEBs to decompose proofs.

5. We used encapsulation to model exceptions and interrupts. As a result, we can use functional instantiation to apply our proofs to any specific exception and interrupt handlers.

All of the proof scripts are available from my Web page [Man00b].

We have also exhibited a clear, compositional path from the verification of term-level descriptions of pipelined machines to the verification of low-level descriptions (*e.g.*, netlist descriptions). For example, in the proof that `MA128` is a refinement of `ISA128`, the definition of the ALU is disabled; the result is that ACL2 treats the ALU as an uninterpreted function. However, to verify that `MA128net` is a refinement of `MA128`, we enable the definition of the ALU and prove theorems relating a netlist description of the circuit to a serial adder which is then related to addition on integers. This allows us to relate term-level descriptions of machines to lower-level descriptions in a compositional way.

Chapter 13

Conclusions

In this dissertation we showed how to reduce the effort involved in the mechanical verification of reactive systems. Reactive systems are ubiquitous and often safety-critical. Thus, it is increasingly crucial for our society that they behave correctly. Due to the complexity of reactive systems, mechanically checked proofs are the most reliable way of ensuring correctness. However, mechanical verification is not in wide use currently, partly because it is a labor-intensive process. In this dissertation we presented various methods that can be used to alleviate the problem.

We developed the theory of stuttering simulation and stuttering bisimulation in ways that allowed us to partially automate mechanical verification. Stuttering simulation and bisimulation are notions of correctness that are used to relate systems at different levels of abstraction and are therefore insensitive to finite stuttering. We proved various useful algebraic properties, developed a compositional theory of refinement, and presented sound and complete proof rules that are particularly amenable to mechanization. We also added mechanical support for reasoning about stuttering bisimulation to the ACL2 system.

The two leading mechanical verification paradigms are theorem proving and model checking; we presented a novel approach to combining them. Theorem proving is used to reduce a large (possibly infinite-state) system to a small, finite-state system by proving a stuttering bisimulation. Algorithms that we developed are then used to extract the reduced system. We showed that every stuttering insensitive property that holds in the reduced system also holds in the original system, thus we can analyze the reduced system using automatic methods such as model checking and can lift the results to the original system. The algorithms have been implemented, *e.g.*, we have implemented and verified a model checker for the Mu-Calculus in the ACL2 system.

We have applied the theory and algorithms by conducting two case studies with the ACL2 system. One of the case studies is the verification of a simple communications protocol. This case study highlights the use of our extraction algorithm and our approach to combining theorem proving and model checking. The other case study involves the verification of a simple pipelined machine from the literature and several variants, including machines with exceptions, interrupts, and netlist (gate-level) descriptions. We argued that correctness based on stuttering bisimulation is more thorough than other notions of correctness currently used. In addition, we demonstrated how to automate much of the process, thereby reducing the effort required for mechanical verification.

Bibliography

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model checking for real time systems. In *5th IEEE Symp. on Logic in Computer Science*, 1990.

[Acz88] Peter Aczel. *Non-Well-Founded Sets*. CSLI Publications, Stanford, 1988.

[AJ96] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, June 1996.

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

[AP99] Tamarah Arons and Amir Pnueli. Verifying tomasulo's algoithm by refinement. In *Proc. 12th International Conference on VLSI Design*, 1999.

[AP00] Tamarah Arons and Amir Pnueli. A comparison of two verification methods for speculative instruction execution. In *TACAS00: Tools and Algorithms for the Construction and Analysis of Systems*, pages 487–502, 2000.

[Bas96] Twan Basten. Branching bisimilarity is an equivalence indeed. *Information Processing Letters*, 58(3):141–147, 1996.

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

[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, June 1992.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[BG94] M. A. Bezem and J. F. Groote. A correctness proof of a one bit sliding window protocol in mCRL. *The Computer Journal*, 1994.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDD's. In *Conference on Computer Aided Verification*, volume 1102 of *LNCS*, 1996.

[BGKM91] Robert Stephen Boyer, D. Goldschlag, Matt Kaufmann, and J Strother Moore. Functional instantiation in first order logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

[BGV99] Randal E. Bryant, Steve German, and Miroslav N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*, pages 470–482. Springer-Verlag, 1999.

[BH97] Bishop Brock and Warren A. Hunt, Jr. Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP. In *1997 IEEE International Conference on Computer Design*, pages 31–36. IEEE Computer Society, October 1997.

[BKM96] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag, 1996.

[BM99] Robert Stephen Boyer and J Strother Moore. Single-threaded objects in ACL2, 1999. See URL `http://www.cs.utexas.edu/-users/moore/publications/acl2-papers.html#Foundations`.

[BP95] Bard Bloom and Robert Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Science of Computer Programming*, 24(3):189–220, 1995.

[Bry92] R. E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 1992.

[BSW69] K. A. Barlett, R. A. Scantlebury, and P. C. Wilkinson. A note on reliable full duplex transmission over half duplex links. In *Communications of the ACM*, volume 12, 1969.

[BT00] Piergiorgio Bertoli and Paolo Traverso. Design verification of a safety-critical embedded verifier. In Kaufmann et al. [KMM00a], pages 233–245.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[CBM89] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 365–373, 1989.

[CE81] E. M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, May 1981.

[CKM+91] D. Craigen, S. Kromodimoeljo, I. Meisels, W. Pase, and M. Saaltink. Eves: An overview. In *VDM'91 Formal Software Development Methods*, volume 551 of *LNCS*. Springer-Verlag, 1991.

[Coe95] Tim Coe. Inside the Pentium FDIV bug. *Dr. Dobb's Journal of Software Tools*, 20(4):129–135, 1995.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Proceedings of the Workshop on Industrial-*

Strength Formal Specification Techniques. Boca Raton, FL, April 1995.

[Cyr93] David Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, SRI, December 1993.

[Dev92] Keith Devlin. *The Joy of Sets: Fundamentals of Contemporary Set Theory.* Springer-Verlag, second edition, 1992.

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Technical Report 154, INRIA, Rocquencourt, France, 1993.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.

[Dij99] Edsger Wybe Dijkstra. Computing science: Achievements and challenges, March 1999. EWD 1284.

[Dij01] Edsger W. Dijkstra. Under the spell of Leibniz's dream. *Information Processing Letters*, 77(2–4):53–61, February 2001.

[DP97] Werner Damm and Amir Pnueli. Verifying out-of-order executions. In *CHARME*, pages 23–47, 1997.

[EC80] E. Allen Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *Proceedings 7th International Colloquium on Automata, Languages, and Programming*, volume 85 of *LNCS*. Springer-Verlag, 1980.

[Ede97] Alan Edelman. The mathematics of the Pentium division bug. *SIAM Review*, 39(1):54–67, 1997.

[EJS93] E. Allen Emerson, C. S. Jutla, and A. P. Sistla. On model checking for fragments of the Mu-Calculus. In *Proceedings 5th International Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 385–396. Springer-Verlag, 1993.

[EL86] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional Mu-Calculus (extended abstract).

In *Proceedings, Symposium on Logic in Computer Science*, pages 267–278, Cambridge, Massachusetts, 16–18 June 1986. IEEE Computer Society.

[Eme81] E. Allen Emerson. *Branching time temporal logics and the design of correct concurrent programs.* PhD thesis, Division of Applied Sciences, Harvard University, August 1981.

[Eme90] E. Allen Emerson. Temporal and modal logic. In van Leeuwen [vL90], pages 995–1072.

[Eme97] E. Allen Emerson. Model checking and the Mu-Calculus. In N. Immerman and P. Kolaitis, editors, *Proceedings of the DIMACS Symposium on Descriptive Complexity and Finite Models*, pages 185–214, 1997.

[EN95] E. Allen Emerson and K. S. Namjoshi. Reasoning about rings. In *ACM Symposium on Principles of Programming Languages*, 1995.

[Gat98] Bill Gates. Remarks by Bill Gates, June 1998. Presented at SIGMOD 98 (Special Interest Group on Management of Data). See URL `http://www.microsoft.com/billgates/speeches/-SIGMOD98.asp`.

[Gla01] R. J. van Glabbeek. The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[Gre98] David A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design – FMCAD*, LNCS. Springer-Verlag, 1998.

[GS92] S. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 1992.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification*, volume 1254 of *LNCS*, 1997.

[GV90] J. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M. S. Paterson, editor, *ICALP*, volume 443 of *LNCS*, pages 626–638. Springer-Verlag, 1990.

[GWH00] David Greve, Matthew Wilding, and David Hardin. High-speed, analyzable simulators. In Kaufmann et al. [KMM00a], pages 113–135.

[HB92] Warren A. Hunt, Jr. and Bishop Brock. A formal HDL and its use in the FM9001 verification. *Proceedings of the Royal Society*, 1992.

[HB97] Warren A. Hunt, Jr. and Bishop Brock. The `DUAL-EVAL` hardware description language and its use in the formal specification and verification of the FM9001 microprocessor. *Formal Methods in Systems Design*, 11:71–105, 1997.

[HGS99] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG10.5 Advanced Research Working Conference, (CHARME '99)*, volume 1703 of *LNCS*, pages 8–22. Springer-Verlag, 1999.

[HHK95] Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 453–462. IEEE Computer Society Press, 1995.

[HJ00] Warren A. Hunt, Jr. and Steven D. Johnson, editors. *Formal Methods in Computer-Aided Design–FMCAD 2000*, volume 1954 of *LNCS*. Springer-Verlag, 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[HQR99] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Assume-guarantee refinement between different time scales. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*, pages 208–221. Springer-Verlag, 1999.

[HS96] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe (FME)*, volume 1051 of *LNCS*. Springer-Verlag, 1996.

[HSG98] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Decomposing the proof of correctness of a pileplined microprocessors. In Hu and Vardi [HV98].

[HSG99] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*. Springer-Verlag, 1999.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[Hun89] Warren A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[Hun94] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795. Springer-Verlag, 1994.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification – CAV '98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.

[HWG98] David Hardin, Matthew Wilding, and David Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In Hu and Vardi [HV98]. See URL `http://-pobox.com/users/hokie/docs/concept.ps`.

[KM] Matt Kaufmann and J Strother Moore. ACL2 homepage. See URL `http://www.cs.utexas.edu/users/moore/acl2`.

[KM97] Matt Kaufmann and J Strother Moore. A precise description of the ACL2 logic. Technical report, Department of Computer Sciences, University of Texas at Austin, 1997. See URL `http://www.cs.utexas.edu/users/moore/-publications/acl2-papers.html#Foundations`.

[KM00] Matt Kaufmann and J Strother Moore, editors. *Proceedings of the ACL2 Workshop 2000*. The University of Texas at Austin, Technical Report TR-00-29, November 2000.

[KM01] Matt Kaufmann and J Strother Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, February 2001.

[KMM00a] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[KMM00b] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.

[KMM00c] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. Supporting files for "Computer-Aided Reasoning: ACL2 Case Studies". See the link from URL `http://www.cs.utexas.edu/users/-moore/acl2`, 2000.

[KMM00d] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. Supporting files for "Computer-Aided Reasoning: An Approach". See the link from URL `http://www.cs.utexas.edu/users/moore/-acl2`, 2000.

[Koz83] D. Kozen. Results on the propositional Mu-Calculus. *Theoretical Computer Science*, pages 334–354, December 1983.

[KR89] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1989.

[Kro01] Daniel Kroning. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Universität des Saarlandes, 2001.

[Kun80] Kenneth Kunen. *Set Theory - an Introduction to Independence Proofs*, volume 102 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1980.

[Lam83] Leslie Lamport. What good is temporal logic? *Information Processing*, 83:657–688, 1983.

[Man00a] Panagiotis Manolios. Correctness of pipelined machines. In Hunt and Johnson [HJ00], pages 161–178.

[Man00b] Panagiotis Manolios. Homepage of Panagiotis Manolios, 2000. See URL `http://www.cs.utexas.edu/users/pete`.

[Man00c] Panagiotis Manolios. Mu-calculus model-checking. In Kaufmann et al. [KMM00a], pages 93–111.

[Man00d] Panagiotis Manolios. Verification of pipelined machines in ACL2. In Kaufmann and Moore [KM00].

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[McM98] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 110–121. Springer-Verlag, 1998.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *In Proceedings of the Second Internation Joint Conference on Artificial Intelligence*, pages 481–489, 1971.

[Mil90] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1990.

[MLK98] J Strother Moore, T. Lynch, and Matt Kaufmann. A mechanically checked proof of the AMD5$_K$86 floating-point division program. *IEEE Trans. Comp.*, 47(9):913–926, September 1998. See URL `http://www.cs.utexas.edu/users/moore/publications/-acl2-papers.html#Floating-Point-Arithmetic`.

[MN95] O. Müller and T. Nipkow. Combining model checking and deduction for I/O-Automata. In *Proceedings of TACAS*, 1995.

[MNS99] Panagiotis Manolios, Kedar Namjoshi, and Robert Sumners. Linking theorem proving and model-checking with well-founded bisimulation. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*, pages 369–379. Springer-Verlag, 1999.

[Moo96] J Strother Moore. *Piton : A Mechanically Verified Assembly-Level Language.* Kluwer Academic Press, Dordrecht, The Netherlands, 1996.

[MS95] F. Moller and S. A. Smolka. On the complexity of bisimulation. *ACM Computing Surveys*, 27(2):287–289, June 1995.

[Nam97] K. S. Namjoshi. A simple characterization of stuttering bisimulation. In *17th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *LNCS*, pages 284–296, 1997.

[PA98] Amir Pnueli and Tamarah Arons. Verification of data-insensitive circuits: An in-order-retirement case study. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methodsin Computer-Aided Design, FMCAD '98*, volume 1522 of *LNCS*. Springer-Verlag, 1998.

[Par69] David Park. Fixpoint induction and proofs of program properties. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, volume 5, pages 59–78. Edinburgh University Press, 1969.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer-Verlag, 1981.

[Pix90] Carl Pixley. A computational theory and implementation of sequential hardware equivalence. In *CAV'90 DIMACS series*, volume 3, June 1990. Also DIMACS Tech. Report 90-31.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Providence, Rhode Island, 31 October–2 November 1977. IEEE.

[Pnu85] Amir Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In W. Brauer, editor, *Proceedings of 12th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 194 of *LNCS*, pages 15–32. Springer-Verlag, 1985.

[Pre99] President's Information Technology Advisory Committee. Information technology research: Investing in our future. National Coordination Office for Computing, Information, and Communications. See URL `http://www.ccic.gov/ac/report/`, February 1999.

[PRSS99] Amir Pnueli, Yoav Rodeh, Ofer Shtrichman, and Michael Siegel. Deciding equality formulas by small domain instantiations. In

Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*, pages 455–469. Springer-Verlag, 1999.

[PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, volume 137 of *LNCS*, 1982.

[RF00] David M. Russinoff and Arthur Flatau. RTL verification: A floating-point multiplier. In Kaufmann et al. [KMM00a], pages 201–231.

[RGA+96] R. K. Brayton, G. D. Hachtel, A. 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. Swamy, and T. Villa. VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 428–432. Springer-Verlag, July 1996.

[Rud92] P. Rudnicki. An overview of the MIZAR project. In *1992 Workshop on Types for Proofs and Programs*, 1992.

[Rus97] David M. Russinoff. A mechanically checked proof of correctness of the AMD5_K86 floating-point square root microcode. *Formal Methods in System Design Special Issue on Arithmetic Circuits*, 1997.

[Rus98] David M. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998.

[Rus99] David M. Russinoff. A mechanically checked proof of correctness of the AMD-K5 floating-point square root microcode. *Formal Methods in System Design*, 14:75–125, 1999.

[Saw99] Jun Sawada. *Formal Verification of an Advanced Pipelined Machine.* PhD thesis, University of Texas at Austin, December 1999. See URL `http://www.cs.utexas.edu/users/sawada/-dissertation/`.

[Saw00] Jun Sawada. Verification of a simple pipelined machine model. In Kaufmann et al. [KMM00a], pages 137–150.

[SB90] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, pages 52–64, September 1990.

[SH97] Jun Sawada and Warren A. Hunt, Jr. Trace table based approach for pipelined microprocessor verification. In *Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 364–375. Springer-Verlag, 1997.

[SH98] Jun Sawada and Warren A. Hunt, Jr. Processor verification with precise exceptions and speculative execution. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 135–146. Springer-Verlag, 1998.

[SM73] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *STOC: ACM Symposium on Theory of Computing (STOC)*, pages 1–9, 1973.

[SM95] Mandayam K. Srivas and Steven P. Miller. Formal verification of an avionics microprocessor. Technical Report CSL-95-04, SRI International, 1995.

[Ste90] G. L. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, Burlington, MA, 1990.

[Sum00] Rob Sumners. An incremental stuttering refinement proof of a concurrent program in ACL2. In Kaufmann and Moore [KM00].

[Tar55] A. Tarski. A lattice theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 55:285–309, 1955.

[Tho01] Wolfgang Thomas. Logic for computer science: The engineering challenge. In Reinhard Wilhelm, editor, *Informatics - 10 Years Back. 10 Years Ahead.*, volume 2000 of *Lecture Notes in Computer Science*, pages 257–267. Springer, 2001.

[TSL+90] H. J. Touati, H. Savoj, B. Lin, R. S. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *IEEE /ACM International Conference on CAD*, pages 130–133, 1990.

[vGW96] Rob J. van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*. Elsevier, Amsterdam, 1990.

[WC94] Phillip J. Windley and Michael L. Coe. A correctness model for pipelined microprocessors. In *Theorem Provers in Circuit Design*, volume 901 of *LNCS*, pages 33–52. Springer-Verlag, 1994.

[WGHar] Matthew Wilding, David Greve, and David Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, to appear. Draft TR available as `http://pobox.com/-users/hokie/docs/efm.ps`.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 184–193. ACM Press, 1986.

Index

Vita

Panagiotis Manolios was born in Athens, Greece on December 22, 1967 to Emmanuel and Sofia Manolios. Panagiotis spent the first seven years of his life in Karpathos, an island at the southeastern tip of Greece. In 1974, his family moved to Brooklyn, New York. Panagiotis received a Bachelor of Science degree from Brooklyn College in 1991. In 1992, he received a Master of Arts degree, again from Brooklyn College. He entered the PhD program at the University of Texas at Austin in 1994. In 1995, he married Helen Nikolopoulos and in 2000, their first child, Emmanuel Aristotelis Manolios, was born. In August 2001, Panagiotis will join the faculty at the Georgia Institute of Technology as an assistant professor in the College of Computing.

Permanent Address: Panagiotis Manolios
3365 Lake Austin Blvd, D
Austin, TX 78703

This dissertation was set by the author using the $\LaTeX\,2_\varepsilon$ typesetting system.