Copyright

by

Sandip Ray

2005

The Dissertation Committee for Sandip Ray

certifies that this is the approved version of the following dissertation:

Using Theorem Proving and Algorithmic Decision Procedures for Large-Scale System Verification

Committee:

J Strother Moore, Supervisor

E. Allen Emerson

Mohammed G. Gouda

Warren A. Hunt, Jr.

C. Greg Plaxton

John R. Matthews

Using Theorem Proving and Algorithmic Decision Procedures for Large-Scale System Verification

by

Sandip Ray, B.E., M.E.

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

December 2005

To the few people who believed I could do it even when I myself didn't

Acknowledgments

This dissertation has been shaped by many people, including my teachers, collaborators, friends, and family. I would like to take this opportunity to acknowledge the influence they have had in my development as a person and as a scientist.

First and foremost, I wish to thank my advisor J Strother Moore. J is an amazing advisor, a marvellous collaborator, an insightful researcher, an empathetic teacher, and a truly great human being. He gave me just the right balance of freedom, encouragement, and direction to guide the course of this research. My stimulating discussions with him made the act of research an experience of pure enjoyment, and helped pull me out of many low ebbs. At one point I used to believe that whenever I was stuck with a problem one meeting with J would get me back on track. Furthermore, my times together with J and Jo during Thanksgivings and other occasions always made me feel part of his family. There was no problem, technical or otherwise, that I could not discuss with J, and there was no time when he hesitated to help me in any difficulty that I faced. Thank you, J.

The person whom this dissertation owes most is Rob Sumners. Rob aroused my interest in the subject matter here at a time when I was frustrated by my incapability to do research. His influence marks every chapter, even work that is

not in direct collaboration with him. He clarified my understanding, critiqued my approach, and shaped the style of my presentation. Frankly, this work would not have been done without Rob's active involvement.

I am grateful to my committee members for helpful feedback. Special thanks are due to Warren A. Hunt Jr. and John Matthews. Warren forced me to focus on techniques of practical value and taught me the essence of practical research. My association with John started when I did a summer internship at HP Labs Cambridge in 2002. My collaborations with him have been very refreshing to me, and have significantly broadened the scope of this dissertation.

I have been fortunate to come across several great teachers who taught me things that added breadth to my knowledge. They include Lorenzo Alvisi, Allen Emerson, Anna Gal, Mohamed Gouda, Greg Plaxton, and of course, J Moore.

I am thankful to the members of the Automatic Theorem Proving research group at the University of Texas for helping me at every step of this work. I specifically thank Matt Kaufmann, Jeff Golden, and Erik Reeber. Matt's promptness in answering queries and the clarity and succinctness of his responses are legendary and I need not add anything. Jeff lent a critical ear to my ideas by playing the "devil's advocate" with the common initial response to anything I said being "I do not think you are right!". With such a skeptic around, it is impossible not to have a clear understanding of the material under discussion. Erik has been my office-mate during the last four years of graduate school, and my conversations with him have helped clarify many of my concepts. In addition, I acknowledge with thanks the numerous discussions I had with Fei Xie, Pete Manolios, Jun Sawada, Robert Krug, Hanbing Liu, Jared Davis, and Thomas Wahl. Talking to all these people helped me

get a clear view of the structure of my research. Thomas in particular also provided a very careful critique of an earlier draft of this dissertation.

At the risk of sounding anthropomorphic, I must thank the ACL2 theorem prover for being my constant companion for the last five years and the most ruthless critic of my ideas. It never let me get away with any hand-waving, and shares equal credit with me for whatever novelty exists in the research documented here. It forced me to spend long hours and sleepless nights with it, trying to convince it of the correctness of some of my arguments. Its stubborn refusal to believe me has shown me the fallacies and flaws in many of my ideas, and its final "affirmative nod" has led to some of the most joyful moments of my doctoral study.

My days in Austin have been made worth living because of the lively interaction with a number of friends without whose active involvement (to steal a joke from P. G. Wodehouse and Rajmohan Rajaraman) "this dissertation would have been completed in half the time". They include Dwip Narayan Banerjee, Jayanta Bhadra, Sutirtha Bhattacharya, Seldron Geziben, Anubrati Mukherjee, Abhijit Jas, Shovan Kanjilal, Sumit Ghosh, Sagnik Dey, Subhadeep Choudhury, Ashis Tarafdar, Jayabrata Ghosh Dastidar, Arindam Banerjee, Sreangshu Acharya, Samarjit Chakraborty, Sugato Basu, Shalini Ghosh, C. V. Krishna, Hari Mony, Anindya Patthak, and Arnab Bandyopadhayay. In addition, my outlook to life and work has been influenced by my lifelong friends that include Avijit Chakraborty, Abanti Dasgupta, Pinaki Datta, Aditya Nori, Arnab Paul, and Prabodh Saha. I have missed the names of several others, but even a mention of the innumerable contributions my friends have had in shaping me will probably make this dissertation twice as long. Thank you all.

The material in this dissertation is based on work supported by the National Science Foundation (NSF) under Grant No. 0417413 and the Semiconductor Research Consortium (SRC) under Grant No. 02-TJ-1032, and I am grateful for the support. In particular, I am grateful to the SRC for the valuable feedback they provided me annually which guided the course of the dissertation towards issues of practical interest.

The very competent staff of the Computer Sciences department at the University of Texas helped me in every infrastructural problem in the course of this dissertation. Gloria Ramirez, Kata Carbone, Katherine Utz, Patti Spencer, Carol Hyink, and Chris Kortla deserve a special mention for the extensive help they provided on numerous occasions during my stay in graduate school.

Finally, I would like to thank my parents without whose patience, perseverence, and immense personal sacrifice this dissertation would not have seen the light of the day.

SANDIP RAY

The University of Texas at Austin

December 2005

Using Theorem Proving and Algorithmic Decision Procedures for Large-Scale System Verification

Publication No. ____________________

Sandip Ray, Ph.D.
The University of Texas at Austin, 2005

Supervisor: J Strother Moore

The goal of formal verification is to use mathematical methods to prove that a computing system meets its specifications. When applicable, it provides a higher assurance than simulation and testing in correct system execution. However, the capacity of the state of the art in formal verification today is still far behind what is necessary for its widespread adoption in practice. In this dissertation, we devise methods to increase the capacity of formal verification.

Formal verification techniques can be broadly divided into two categories, namely deductive techniques or theorem proving, and decision procedures such as model checking, equivalence checking, and symbolic trajectory evaluation. Neither deductive nor algorithmic techniques individually scale up to the size of modern industrial systems, albeit for orthogonal reasons. Decision procedures suffer from *state explosion*. Theorem proving requires manual assistance. Our methods involve a sound, efficient, and scalable integration of deductive and algorithmic techniques.

There are four main contributions in this dissertation. First, we present several results that connect different deductive proof styles used in the verification of sequential programs. The connection allows us to efficiently combine theorem proving with symbolic simulation to prove the correctness of sequential programs without requiring a verification condition generator or manual construction of a global invariant. Second, we formalize a notion of correctness for reactive concurrent programs that affords effective reasoning about infinite computations. We discuss several reduction techniques that reduce the correctness proofs of concurrent programs to the proof of an invariant. Third, we present a method to substantially automate the process of discovering and proving invariants of reactive systems. The method combines term rewriting with reachability analysis to generate efficient *predicate abstractions*. Fourth, we present an approach to integrate model checking procedures with deductive reasoning in a sound and efficient manner.

We use the ACL2 theorem prover to demonstrate our methods. A consequence of our work is the identification of certain limitations in the logic and implementation of ACL2. We recommend several augmentations of ACL2 to facilitate deductive verification of large systems and integration with decision procedures.

Contents

Acknowledgments **v**

Abstract **ix**

Contents **xi**

List of Figures **xviii**

I Introduction and Preliminaries 1

Chapter 1 Introduction **2**

Chapter 2 Overview of Formal Verification **11**

2.1 Theorem Proving 12

2.2 Temporal Logic and Model Checking 17

2.3 Axiomatic Semantics and Verification Conditions 25

2.4 Bibliographic Notes 32

Chapter 3 Introduction to ACL2 **35**

3.1 Basic Logic of ACL2 36

3.2 Ground Zero Theory 39
3.2.1 Terms, Formulas, Functions, and Predicates 43
3.2.2 Ordinals and Well-founded Induction 45
3.3 Extension Principles 49
3.3.1 Definitional Principle 51
3.3.2 Encapsulation Principle 55
3.3.3 Defchoose Principle 57
3.4 Bibliographic Notes 58

II Sequential Program Verification **61**

Chapter 4 Sequential Programs **62**
4.1 Modeling Sequential Programs 62
4.2 Proof Styles 65
4.2.1 Step Invariants 66
4.2.2 Clock Functions 67
4.3 Discussions 68
4.3.1 Composition 72
4.3.2 Over-specification 75
4.4 Summary 76
4.5 Bibliographic Notes 76

Chapter 5 Mixing Step Invariant and Clock Function Proofs **78**
5.1 Proof Style Equivalence 79
5.1.1 Step Invariants to Clock Functions 79

5.1.2 Clock Functions to Step Invariants 81

5.2 Generalized Proof Styles 82

5.3 Verifying Program Components 86

5.4 Mechanically Switching Proof Styles 88

5.5 Summary and Comments 89

5.6 Bibliographic Notes 92

Chapter 6 Operational Semantics and Assertional Reasoning 94

6.1 Cutpoints, Assertions, and VCG Guarantees 95

6.2 VCG Guarantees and Symbolic Simulation 100

6.3 Examples 103

6.3.1 An Iterative Program: Fibonacci on the TINY Machine . . . 103

6.3.2 A Recursive Program: Factorial on the JVM 106

6.4 Comparison with Related Approaches 109

6.5 Summary 111

6.6 Bibliographic Notes 112

III Verification of Reactive Systems 114

Chapter 7 Reactive Systems 115

7.1 Modeling Reactive Systems 117

7.2 Stuttering Trace Containment 119

7.3 Fairness Constraints 121

7.4 Discussions 128

7.5 Summary 134

7.6 Bibliographic Notes 135

Chapter 8 Verifying Concurrent Protocols Using Refinements 137

8.1 Reduction via Stepwise Refinement 139

8.2 Reduction to Single-step Theorems 139

8.3 Equivalences and Auxiliary Variables 146

8.4 Examples 149

8.4.1 An **ESI** Cache Coherence Protocol 149

8.4.2 An Implementation of the Bakery Algorithm 154

8.4.3 A Concurrent Deque Implementation 162

8.5 Summary 170

8.6 Bibliographic Notes 172

Chapter 9 Pipelined Machines 173

9.1 Simulation Correspondence, Pipelines, and Flushing Proofs 174

9.2 Reducing Flushing Proofs to Refinements 178

9.3 A New Proof Rule 181

9.4 Example 183

9.5 Advanced Features 189

9.5.1 Stalls 190

9.5.2 Interrupts 190

9.5.3 Out-of-order Execution 191

9.5.4 Out-of-order and Multiple Instruction Completion 191

9.6 Summary 193

9.7 Bibliographic Notes 194

IV Invariant Proving **197**

Chapter 10 Invariant Proving **198**

10.1 Predicate Abstractions 203

10.2 Discussions 206

10.3 An Illustrative Example 208

10.4 Summary 210

10.5 Bibliographic Notes 211

Chapter 11 Predicate Abstraction via Rewriting **213**

11.1 Features and Optimizations 220

11.1.1 User Guided Abstraction 221

11.1.2 Assume Guarantee Reasoning 222

11.2 Reachability Analysis 223

11.3 Examples 224

11.3.1 Proving the **ESI** 224

11.3.2 German Protocol 227

11.4 Summary and Comparisons 230

11.5 Bibliographic Notes 233

V Verification of RTL Designs **235**

Chapter 12 RTL Systems **236**

Chapter 13 A Verilog to ACL2 Translator **240**

13.1 Overview of Verilog 241

13.2 Deep and Shallow Embeddings 245

13.3 An RTL Library 248

13.4 Translating Verilog Language Constructs 252

13.5 Summary and Comments 263

13.6 Bibliographic Notes 264

Chapter 14 Verification of a Pipelined RTL Microprocessor 266

14.1 The Y86 Processor Design 267

14.2 The Y86 Implementation 270

14.2.1 The **seq** Implementation 270

14.2.2 The **pipe** Implementation 273

14.3 Verification Objectives and Simplifications 277

14.4 Verification Methodology and Experience 279

14.5 Summary and Comments 287

14.6 Bibliographic Notes 290

VI Formal Integration of Decision Procedures 291

Chapter 15 Integrating Deductive and Algorithmic Reasoning 292

Chapter 16 A Compositional Model Checking Procedure 298

16.1 Formalizing a Compositional Procedure 300

16.1.1 Finite State Systems 300

16.1.2 Temporal Logic formulas 302

16.1.3 Compositional Procedure 302

16.2 Modeling LTL Semantics 305

16.3 Verification 311
16.4 Discussion 316
16.4.1 Function Objects 320
16.5 Summary 324
16.6 Bibliographic Notes 326

Chapter 17 Theorem Proving and External Oracles **328**
17.1 Integrating Oracles with ACL2 331
17.2 External Oracles and Clause Processors 335
17.3 Summary 336
17.4 Bibliographic Notes 337

VII Conclusion and Future Directions **339**

Chapter 18 Summary and Conclusion **340**

Chapter 19 Future Directions **343**
19.1 Real-time Systems and Peer-to-peer Protocols 343
19.2 Counterexamples with Predicate Abstraction 345
19.3 Integrating GSTE with Theorem Proving 346
19.4 Certifying Model Checkers 347

Bibliography **349**

Vita **388**

List of Figures

2.1 A Simple One-Loop Program 28

3.1 Some Functions Axiomatized in GZ 41

3.2 Examples of Ordinal Representation in ACL2 47

3.3 Example of a Mutually Recursive Definition 54

4.1 Step Invariant for the One-Loop Program 69

4.2 Clock Function for the One-Loop Program 70

4.3 A Key Lemma for the One-loop Program 71

6.1 TINY Assembly Code Computing Fibonacci 103

6.2 Assertions for the Fibonacci Program on TINY 105

6.3 Java Program for Computing Factorial 106

6.4 M5 Byte-code for the Factorial Method 107

7.1 Definition of a Stuttering Trace 121

8.1 A Model of the **ESI** Cache Coherence Protocol. 151

8.2 Pseudo-code for State Transition of System **mem** 153

8.3 The Bakery Program Executed by Process *p* with Index `j`. 156

8.4 Methods for the Concurrent Deque Implementation 163

9.1 Pictorial Representation of Simulation Proofs Using Projection . . . 175

9.2 Pictorial Representation of Flushing Proofs 177

9.3 Using Flushing to Obtain a Refinement Theorem 180

9.4 A Simple 5-stage Pipeline 183

10.1 Equations showing the transitions of the Two Component System . . 209

10.2 Finite-state Abstraction of the Two Component System 211

11.1 Chopping a Term τ 216

11.2 Procedure for Generation of Predicates 217

11.3 Generic Rewrite Rules for Set and Record Operations 225

11.4 State Predicates Discovered for the **ESI** Model 227

13.1 A 4-bit Counter Implemented in Verilog 242

13.2 Functions Representing Bit Vector Operations 250

13.3 Translating Module Instantiation of 4-bit Counter 260

14.1 Hardware Structure of the **seq** Processor 271

14.2 Hardware Structure of the **pipe** Processor 274

14.3 Definition of *rank* for showing ($\mathbf{seq} \trianglerighteq \mathbf{pipe}^{+}$) 283

14.4 Some Theorems about Bit Vector Manipulation 285

16.1 Formalization of the Compositional Model Checking Procedure . . . 304

16.2 A Periodic Path and Its Match 315

16.3 A Truth Predicate 319

16.4 Path Semantics in terms of Function Objects 323

Part I

Introduction and Preliminaries

Chapter 1

Introduction

Computing systems are ubiquitous in today's world. They control medical monitoring equipments, banking, traffic control and transportation, and many other operations. Many of these systems are *safety critical*, and the failure of a system might cause catastrophic loss of money, time, and even human life. It is crucial for our well-being, then, that computing systems behave correctly and reliably.

Ensuring reliable behavior of a modern computing system, however, is a challenging problem. Most critical computing systems are incredibly complex artifacts. The complexity is induced by the exacting efficiency needs of current applications, along with advances in the areas of design and fabrication technologies. A modern micro-controller laid out in a small silicon die today has, and needs to have, computing power several times that of a large supercomputer of thirty years back. Implementations of such systems typically involve megabytes of program code, and even a description of their desired properties, when written down, goes to hundreds of pages. Given such complexity, it is not surprising that modern computing sys-

tems are error-prone, often containing bugs that are difficult to detect and diagnose. It is impossible for a designer to keep track of the different possible cases that can arise during execution, and an "innocent" optimization made with an inaccurate mental picture of the system might lead to a serious error. The currently practiced methods for ensuring reliable executions of most system designs principally involve extensive simulation and testing. However, essential as they are, they are now proving inadequate due to the computational demands of the task. For example, it is impossible to simulate in any reasonable time the execution of a modern microprocessor on all possible input sequences or even a substantial fraction of possible inputs. Furthermore, simulation and testing are usually designed to detect only certain well-defined types of errors. They can easily miss a subtle design fault that may cause unexpected trouble only under a particular set of conditions.

Formal verification has emerged as an attractive approach for ensuring correctness of computing systems. In this approach, one models the system in some mathematical logic and formally proves that it satisfies its desired specifications. Formal verification in practice makes use of some mechanical reasoning tool, that is, a trusted computer program which is responsible for guiding the proof process and checking the validity of the constructed proof. When successful, the approach provides a high assurance in the reliability of the system, namely a mathematical guarantee of its correctness up to the accuracy of the model and the soundness of the computer program employed in the reasoning process. The approach is particularly enticing since, unlike simulation and testing, the guarantee is provided for *all* system executions.

Formal verification has enjoyed several successes recently, in proving the

correctness of industrial-scale hardware and software systems. Verification is now part of the tool flow in microprocessor companies like AMD, IBM, Intel, and Motorola. For example, many floating point operations of the AMD K5™ [MLK98], Athlon™ [Rus98] and Opteron™, and the Intel Pentium® Pro [OZGS99] microprocessors have been formally proven to be IEEE compliant. However, in spite of these spectacular successes, the capacity of the state of the art in formal verification is far below what is required for its widespread adoption in commercial designs. In this dissertation we identify some of the difficulties in the applying the current verification techniques to large-scale systems, and devise tools and methodologies to increase the capacity of formal verification.

Research in formal verification is broadly divided into two categories, namely the use of deductive verification or theorem proving [KMM00b, GM93, Har00, NPW02, ORS92], and the use of algorithmic methods like model checking [CE81, QS82], equivalence checking [MM04], symbolic simulation [Jon02], and symbolic trajectory evaluation [Cho99]. The key difference between the deductive and algorithmic methods is in the expressiveness of the formal logic employed. Theorem provers typically use an expressive logic which allows the user to apply a variety of proof techniques. For example, almost any practical theorem prover allows proofs by mathematical induction, term rewriting, some form of equality reasoning, and so on. However, theorem proving is not automatic in general, and its successful use for proving non-trivial theorems about complicated systems depends on significant interaction with a trained user. The user must both be familiar with the formal logic of the theorem prover and conversant with the nuances of the system being verified. The problem with the use of this technology on industrial-scale system

verification is that the manual effort necessary for the task might be prohibitively expensive. In the algorithmic approach, on the other hand, we employ a decidable logic to model system properties. While such logics are less expressive than those employed by a theorem prover, their use affords the possibility of implementing decision procedures for system verification, with (at least in theory) no requirement for any user interaction. When applicable, algorithmic methods are therefore more suitable than theorem proving for integration with an industrial tool flow. However, the key obstacle to the widespread applicability of decision procedures in practice lies in the well-known *state explosion* problem. Most decision procedures involve a (symbolic or explicit) exploration of the different states that a system can reach in the course of its execution, and modern systems have too many states for an effective exploration to be practicable.

Given the individual but somewhat orthogonal limitations of both algorithmic and deductive reasoning techniques, our quest is for an effective combination of these techniques that scales better than each individual technique. The preceding discussion suggests that such a combination should satisfy the following two criteria.

1. The manual effort involved must be significantly less than what would be needed to verify systems using theorem proving alone.

2. Decision procedures should be carefully applied so that state explosion can be effectively averted.

In this dissertation, we develop tools and techniques to facilitate the conditions above. Since we wish to preserve the expressiveness, flexibility, and diversity of proof techniques afforded by deductive reasoning, our approach is principally based on theorem proving. We intend to use decision procedures as efficient and automatic proof rules within a theorem prover to decide the validity of certain conjectures that

can be posed as formulas in a decidable fragment of the logic of the theorem prover, and require significant user expertise to verify using theorem proving alone.

Our approach, stated in these terms, might seem simple or trivial, but it is neither. In addition to the engineering challenges involved in effectively integrating different procedures, such combinations bring in new *logical* challenges which are absent in the application of either technique individually. For example, combining theorem proving with model checking must necessarily involve working simultaneously with different logics and formal systems. How do we then know that the composite system is sound? In the context of formal verification the question is not merely academic, and we will see in Chapter 15 that it is inextricably linked with the question of *interpreting* an affirmative answer produced by a decision procedure as a theorem in the logic of the theorem prover.

Computing systems in practical use range from implementations of sequential programs to distributed protocols and pipelined microarchitectures. The verification problems differ significantly as we consider different systems within this spectrum. This diversity must be kept in mind as we integrate decision procedures with a theorem prover for solving different problems. We do so by carefully studying the inefficiencies of deductive reasoning as a stand-alone technique for the problem domain under consideration, and determining the appropriate decision procedure that can assist in resolving the inefficiency. It is not always necessary to integrate an *external* decision procedure. For example, in Chapter 6, we will see how we can merely *configure* a theorem prover to perform symbolic simulation by repeated simplification. However, as we consider more and more complicated systems, we need sophisticated integration methods.

Our approach requires that we work inside a practical general-purpose theorem prover. We use ACL2 [KMM00a, KMM00b] for this purpose. ACL2 is a theorem prover in the Boyer-Moore tradition, and has been used in the verification of some of the largest theorem proving problems [MLK98, Rus98, BKM96]. In the context of our work, the use of ACL2 has both advantages and disadvantages. On the positive side, the language of the theorem prover is a large subset of an applicative programming language, namely Common Lisp [Ste90]. The theorem prover provides a high degree of support for fast execution, efficient term rewriting, and inductive proofs, which are necessary ingredients for modeling and reasoning about large-scale computing systems. On the negative side, the logic of ACL2 is essentially first order. Thus any specification or proof technique that involves higher-order reasoning is unavailable to us in ACL2. This has some surprising repercussions and we will explore some of its consequences later in the dissertation. However, this dissertation is not about ACL2; we use ACL2 principally as a mechanized formal logic with which the author is intimately familiar. We believe that many of our integration techniques can be used or adopted for increasing the capacity of verification in other general-purpose theorem provers such as HOL [GM93] or PVS [ORS92]. Our presentation in this dissertation does not assume any previous exposure to ACL2 on the part of the reader, although it does assume some familiarity with first order logic.

This dissertation consists the following seven parts.

- Introduction and Preliminaries
- Sequential Program Verification

- Verification of Reactive Systems
- Invariant Proving
- Verification of RTL Designs
- Formal Integration of Decision Procedures
- Conclusion and Future Directions

Most of the technical chapters contain bibliographic notes. In addition, if the research described in a chapter represents collaborative work with other researchers, then the bibliographic notes contain references to the co-authors and the published version of the work wherever applicable.

The rest of this part provides the foundational groundwork for the dissertation. In Chapter 2, we review selected verification techniques, namely theorem proving, model checking, and verification condition generation. In Chapter 3, we describe the logic of the ACL2 theorem prover. These two chapters are intended to provide the necessary background to keep the dissertation self-contained; no new research material is presented. Nevertheless the reader is encouraged to refer to them to become familiar with our treatment of different formalisms.

Parts II-VI contain our chief technical contributions. In Part II, we discuss techniques for verification of sequential programs. Proving the correctness of sequential programs is one of the most well-studied aspects of formal verification. We show how we can use symbolic simulation effectively with theorem proving to simplify and substantially automate such proofs. Furthermore, we show how we can use the simplification engine of the theorem prover itself for effective symbolic simulation.

In Part III, we extend our reasoning to reactive systems. Contrary to sequential programs, for which all correctness notions are based on terminating computations, reactive systems are characterized by processes performing ongoing, non-terminating executions. Hence the notion of correctness for a reactive system involves properties of infinite sequences. We present a refinement framework to facilitate the verification of reactive systems. We argue that the notion of refinement we present corresponds to the intuitive notion of correctness of such systems. Traditional notions of non-determinism and fairness can be easily formalized in the framework. We determine several proof rules to facilitate verification of such refinements, and show how an effective orchestration of the rules affords significant automation in the verification of reactive systems. A key bottleneck we face in achieving the automation is in discovering relevant inductive invariants of systems. We conclude that the capacity of deductive verification can be improved by integrating decision procedures for automatic discovery and proof of invariants.

In Part IV, we consider a novel approach for proving invariants of reactive systems using a combination of theorem proving and model checking. Our method reduces an invariant proof to model checking on a finite graph that represents a *predicate abstraction* [GS97] of the system. Predicate abstractions are computed by *term rewriting* [BN98] which can be controlled and augmented by proving rewrite rules using the theorem prover. We improve the efficiency of verification by lightweight integration of industrial-strength model checking tools such as VIS [BHSV$^+$96] and SMV [McM93].

In Part V, we use refinement techniques and predicate abstraction in the verification of systems modeled at the level of register-transfer language (RTL). Of

course, RTL designs are typically specified not in a formal logic but using a Hardware Description Language (HDL) such as VHDL [Bha92] or Verilog [TM96]. To facilitate reasoning about such designs, we implement a translator to convert designs written in a well-defined subset of Verilog to the logic of ACL2, and build a library of rewrite rules for reasoning about such formalized designs. We demonstrate the efficacy of our approach by verifying a pipelined RTL implementation of a version of the Y86 processor [BO03], an academic processor designed by Bryant and O'Hallaron based on the IA32 instruction set architecture.

In Part VI, we explore a generic method for integrating decision procedures with theorem proving to automate checking expressive temporal properties. We study the use of the method to integrate a model checker with a theorem prover in a sound and efficient manner. The study exposes certain surprising obstacles that need to be overcome in order to make the integration effective. This results in a collection of recommendations to strengthen the logic and design of the ACL2 theorem prover.

This dissertation contains several case studies. However, the focus of our presentation is on verification techniques, and not on the details of the individual systems verified; case studies are used principally for demonstrating the applicability of the techniques presented. As such, some of the proof details are skipped in the interest of clarity and space. The models of all systems discussed here, along with instructions for using the tools presented for their verification, are available from the author's home page [Ray].

Chapter 2

Overview of Formal Verification

Verification of a computing system entails furnishing a mathematical proof showing that the executions of the system satisfy some desired properties or specification. To do this, we must use some mathematical structure to model the system of interest and derive the desired properties of the system as theorems about the structure. The principal distinction between the different formal verification approaches stems from the choice of the mathematical formalism used in the reasoning process. In this chapter we survey some of the key techniques for formal verification and understand some of their strengths and limitation.

Formal verification is a very active area of research, and numerous promising techniques and methodologies have been invented in recent years for streamlining and scaling its application on large-scale computing systems. Given the vastness of the subject, it is impossible to do justice to any of the techniques involved in this short review; we only focus on the aspects of the different methods that are relevant to our work. The bibliographic notes in Section 2.4 lists some of the books

and papers that provide a more comprehensive treatment of the materials discussed here.

2.1 Theorem Proving

Theorem proving represents one of the key approaches to formal verification. A theorem prover is a computer program for constructing and checking derivations in some formal logic. A formal logic comprises of a *formal language* to express formulas, a collection of formulas called *axioms*, and a collection of *inference rules* for deriving new formulas from existing ones. To use formal logic in order to reason about some mathematical artifact, one considers a logic such that the formulas representing the axioms are *valid*, that is, can be interpreted as self-evident truths about the artifact, and the inference rules are validity preserving. Thus the formulas derived by applying a sequence of inference rules from the axioms must also be valid. Such formulas are referred to as *formal theorems* (or simply *theorems*). The sequence of formulas such that each is either an axiom or obtained from a collection of previous formulas in the sequence by applying an inference rule is called a *derivation* or *deduction*. In the context of theorem proving, verifying a formula is tantamount to showing that a deduction of the formula exists in the logic of the theorem prover.

There are many theorem provers in active use today. Some of the popular ones include ACL2 [KMM00b], Coq [DFH+91], Forte [AJK+00], HOL [GM93], Isabelle [Pau], and PVS [ORS92]. The underlying logics of theorem provers vary considerably. There are theorem provers for set theory, constructive type theory, first order logic, higher order logic, and so on, to name only a few. There is also substantial difference in the amount of automation provided by the different theorem

provers; some are proof checkers while others can do a considerable amount of unassisted reasoning. In the next chapter, we will see some details of one theorem prover, namely ACL2. In spite of diversity of features, however, a common aspect is that most logics supported by theorem provers are rich and expressive. The expressiveness allows one to use the technology for mechanically checking deep and interesting theorems in a variety of mathematical domains. For instance, Gödel's incompleteness theorem and Gauss' Law of Quadratic Reciprocity have been mechanically verified in the Nqthm theorem prover [Sha94, Rus92], and Ramsey's Theorem and Cantor's Theorem have been proved in Isabelle [Pau93, Pau95].

However, this expressive power comes at a cost. Foundational research during the first half of the last century showed that any sufficiently expressive logic that is consistent must be undecidable [Göd31, Tur37]. That means that there is no automatic procedure (or algorithm) that, given a formula in the logic, can always determine if there exists a derivation of the formula in the logic, or in other words, if the formula is a theorem. Thus the successful use of theorem proving for deriving non-trivial theorems typically involves substantial interaction with a trained user. Nevertheless, theorem provers are still invaluable tools for formal verification. Some key aspects of theorem proving are the following:

- A theorem prover can mechanically *check* a proof, that is, verify that a sequence of formulas does correspond to a legal derivation in the proof system. This is usually an easy matter; the theorem prover needs to merely check that each derivation in the sequence is either an axiom or follows from the previous ones by a legal application of the rules of inference.

- A theorem prover can assist the user in the construction of a proof. Most theorem provers in practice implement several heuristics for proof search. Such heuristics include generalizing the formula for applying mathematical induction, using appropriate

instantiation of previously proven theorems, judicious application of term rewriting, and so on.

- If the formula to be proved as a theorem is expressible in some well-identified decidable fragment of the logic, then the theorem prover can invoke a decision procedure for the fragment to determine whether it is a theorem. Most theorem provers integrate decision procedures for several logic fragments. For instance, PVS has procedures for deciding formulas in Presburger arithmetic, and formulas over finite lists [Sho79, NO79], and ACL2 has procedures for deciding linear inequalities over rationals [BM88b, HKM03].

In spite of approaches designed to automate the search of proofs, it must be admitted that undecidability poses an insurmountable barrier in automating such derivations. In practice, construction of a non-trivial derivation via theorem proving is a creative process requiring substantial interaction between the theorem prover and a trained user. The user usually provides an outline of the derivation and the theorem prover is responsible for determining if the outline can indeed be turned into a formal proof. At what level of detail the user needs to give the outline depends on the complexity of the derivation and the implementation and architecture of the theorem prover. In general, when the theorem prover fails to deduce the derivation of some formula given a proof outline, the user needs to refine the outline, possibly by proving some intermediate lemmas. In a certain sense, interacting with a theorem prover for verifying a complex formula "feels" like constructing a *very* careful mathematical argument for its correctness, with the prover checking the correctness of the low level details of the argument.

How do we apply theorem proving to prove the correctness of a computing system? The short answer is that the basic approach is exactly the same as what we

would do to prove the correctness of any other mathematical statement in a formal logic. We determine a formula that expresses the correctness of the computing system in the logic of the theorem prover, and derive the formula as a theorem. Throughout this dissertation we will see several examples of computing systems verified in this manner. However, the formulas we need to manipulate for reasoning about computing systems are extremely long. A formal specification of even a relatively simple system might involve formulas ranging over 100 pages. Doing formal proof involving formulas at such scale, even with mechanical assistance from a theorem prover, requires a non-trivial amount of discipline and attention to detail. Kaufmann and Moore [KMc] succinctly mention this issue as follows:

> Minor misjudgments that are tolerable in small proofs are blown out of proportions in big ones. Unnecessarily complicated function definitions or messy, hand-guided proofs are things that can be tolerated in small projects without endangering success; but in large projects, such things can doom the proof effort.

Given that we want to automate proofs of correctness of computing systems as much as possible, why should we apply theorem proving for this purpose? Why not simply formalize the desired properties of systems in a decidable logic and use a decision procedure to check if such properties are theorems? There are indeed several approaches to formal verification based on decidable logics and use of decision procedures; in the next section we will review one such representative approach, namely *model checking*. Nevertheless, theorem proving, undecidable as it is, has certain distinct advantages over decision procedures. In some cases, one needs an expressive logic simply to state the desired correctness properties of the system of interest, and theorem proving is the only technology that one can resort to in proving such properties. Even when the properties can be expressed in a decidable logic,

for example when one is reasoning about a finite state system, theorem proving has the advantage of being both succinct and general. For instance consider a system $\mathcal{S}_3$ of 3 processes executing some mutual exclusion protocol. As we will see in the next section, it is possible to state the property of mutual exclusion for the system in a decidable logic, and indeed, model checking can be used to prove (or disprove) the property for the system. However, if we now implement the *same* protocol for a system $\mathcal{S}_4$ with 4 processes, the verification of $\mathcal{S}_3$ does not give us any assurance in the correctness of this new system, and we must re-verify it using model checking. In general, we would probably like to verify the implementation for a system $\mathcal{S}_n$ of n processes. However, the mutual exclusion property for such a parameterized system $\mathcal{S}_n$ might not be expressible in a decidable logic. Some advances made in *Parameterized Model Checking* allow us to apply automated decision procedures to parameterized systems, but there is strong restriction on the kind of systems on which they are applicable [EK00]. Also, even when decision procedures are applicable in principle (say for the system $\mathcal{S}_{100}$ with 100 processes), they might be computationally intractable. On the other hand, one can easily and succinctly represent mutual exclusion for a parameterized system as a formula in an expressive logic, and use theorem proving to reason about the formula. While this will probably involve some human effort, the result is reusable for any system irrespective of the number of processes, and thereby solves a family of verification problems in one fell swoop.

There is one very practical reason for applying theorem proving in the verification of computing systems. Most theorem provers afford a substantial degree of

control in the process of derivation of complex theorems.[1] This can be exploited by the user in different forms, typically by proving key intermediate lemmas that assist the theorem prover in its proof search. By manually structuring and decomposing the verification problem, the user can guide the theorem prover into proofs about very complex systems. For instance, using ACL2, Brock and Hunt [BH99] proved the correctness of an industrial DSP processor. The main theorem was proved by crafting a subtle generalization which was designed from the user understanding of the high-level concepts behind the workings of the system. This is a standard occurrence in practical verification. Paulson [Pau01] shows examples of how verification can be streamlined if one can use an expressive logic to succinctly explain the high-level intuition behind the workings of a system. Throughout this dissertation, we will see how user control is effectively used to simplify formal verification.

2.2 Temporal Logic and Model Checking

Theorem proving represents one approach to reasoning about computing systems. In using theorem proving, we trade automation in favor of expressive power of the underlying logic. Nevertheless, automation, when possible, provides a key benefit in the application of formal reasoning to computing systems. A substantial amount of research in formal verification is aimed at designing decidable formalisms in which interesting properties of computing systems can be formulated, so that one can check the truth and falsity of such formulas by decision procedures.

It should be noted that in addition to providing automation, the use of a

[1]By theorem provers in this dissertation we normally mean the so-called *general-purpose* theorem provers. There are other more automated theorem provers, for instance Otter [McC97a], which afford much less user control and substantially more automation. We do not discuss these theorem provers here since they are not easily applicable in the verification of computing systems.

decidable formalisms for formal verification has one other significant benefit. When the system has a bug, that is its desired properties are not theorems, the use of a decision procedure can provide a counterexample. Such counterexamples can be invaluable in tracing the source of such errors in the system implementation.

In this section, we study in some detail one such decidable formalism, namely *propositional linear temporal logic* or PLTL (LTL for short). LTL has found several applications for specifying properties of reactive systems, that is, computing systems which are characterized by non-terminating computations. Learning about LTL and decision procedures for checking properties of computing systems specified as LTL formulas will give us some perspective of how decision procedures work and the key difference between them and deductive reasoning. Later in the dissertation, in Chapter 16, we will consider formalizing and embedding LTL into the logic of ACL2.

Before going any further, let us point out one major difference between theorem proving and decision procedures. When using theorem proving, we use derivations in a formal logic, which contains axioms and rules of inferences. If we prove that some formula expressible in the logic is a theorem, then we are asserting that for *any* mathematical artifact such that we can provide an *interpretation* of the formulas in the logic such that the axioms are true facts (or *valid*) about the artifact under the interpretation, and the inference rules are validity preserving, the interpretation of the theorem must also be valid. An interpretation under which the axioms are valid and inference rules are validity preserving is also called a *model* of the logic. Thus the consequence of theorem proving is often succinctly described as: "Theorems are valid for all models." For our purpose, theorems are the desired properties of executions of computing systems of interest. Thus, a successful verifi-

cation of a computing system using theorem proving is a proof of a theorem which can be interpreted to be a statement of the form: "All legal executions of the system satisfy a certain property." What is important in this is that the legal executions of the system of interest must be expressed as some set of formulas in the formal language of the logic. Formalisms for decision procedures, on the other hand, typically describe syntactic rules for specifying *properties* of executions of computing systems, but the executions of the systems themselves are not expressed inside the formalism. Rather, one defines the *semantics* of a formula, that is, a description of when a system can be said to satisfy the formula. For instance, we will talk below about the semantics of LTL. The semantics of LTL are given by properties of paths through a *Kripke Structure*. But the semantics themselves, of course, are not expressible in the language of LTL. By analogy from our discussion about theorem proving, we can then say that the semantics provides an *interpretation* of the LTL formulas and the Kripke Structure is a model of the formula under that interpretation. In this sense, one often refers to the properties checked by decision procedures as saying that "they need to be valid under one interpretation." Thus a temporal logic formula representing some property of a computing system, when verified, *does not* correspond to a formal theorem in the sense that a property verified by a theorem prover does. Indeed, it has been claimed [CGP00] that by restricting the interpretation to one model instead of all as in case of theorem proving, formalisms based on decision procedures succeed in proving interesting properties while still being amenable to algorithmic methods. Of course it is possible to think of a logic such that the LTL formulas, Kripke Structures, and the semantics of LTL with respect to Kripke Structures can be represented as formulas inside the logic. We can

then express the formula: "Kripke Structure κ satisfies formula ψ" as a theorem. Indeed, this is exactly what we will seek to do in Chapter 16 where we choose ACL2 to be the logic. However, in that case the claim of decidability is not for the logic in which such formulas can be expressed but only on the *fragment* of formulas which designate temporal logic properties of Kripke Structures.

So what do (propositional) LTL formulas look like? The formulas are described in terms of a set $\mathcal{AP}$ called the set of *atomic propositions*, standard Boolean operators "$\wedge$", "$\vee$", and "$\neg$", and four *temporal operators* "X", "G", "F", and "U". The structure of an LTL formula is then recursively defined as follows.

- If $\psi \in \mathcal{AP}$ then ψ is an LTL formula.
- If ψ_1 and ψ_2 are LTL formulas, then so are $\neg\psi_1$, $\psi_1 \vee \psi_2$, $\psi_1 \wedge \psi_2$, $\mathsf{X}\psi_1$, $\mathsf{G}\psi_1$, $\mathsf{F}\psi_1$, and $\psi_1 \mathsf{U} \psi_2$.

As mentioned above, the semantics of LTL is specified in terms of paths through a Kripke Structure. A Kripke Structure κ is a triple $\langle S, R, L, s_0 \rangle$, where S is a set of *states*, R is a relation over $S \times S$ that is assumed to be left-total called the *transition relation*, $L : S \rightarrow 2^{\mathcal{AP}}$ is called the *state labeling function*, and $s_0 \in S$ is called the *initial state*. It is easy to model computing systems in terms of Kripke Structures. A system usually comprises of several *components* which can take up values in some range. To represent a system as a Kripke Structure we take the set of states to be the set of all possible valuations of the different components of the system. A state is simply one such valuation. The system is called *finite state* if the set of states is finite. The *initial state* corresponds to the valuation of the components at the time of system initiation or reset. Two states s_1 and s_2 are related by the transition relation if it is possible for the system to transit from state s_1 to s_2 in one step. Notice that by specifying the transition as a *relation* we can talk about systems that

make non-deterministic transitions from a state. The state labeling function (*label* for short) maps a state s to the atomic propositions that are true of s.

Given a Kripke Structure κ, we can talk about *paths* through κ. An *infinite path* (or simply, a path) π is an infinite sequence of states such that any two consecutive states in the sequence are related by the transition relation. Given any infinite sequence π, we will refer to the i-th element in π by π^i and the subsequence of π starting from π^i by π_i. Given an LTL formula ψ and a path π we then define what it means for π to satisfy ψ as follows:

1. If $\psi \in \mathcal{AP}$ then π satisfies ψ if and only if $\psi \in L(\pi^0)$.
2. π satisfies $\neg\psi$ if and only if π does not satisfy ψ.
3. π satisfies $\psi_1 \vee \psi_2$ if and only if π satisfies ψ_1 or π satisfies ψ_2.
4. π satisfies $\psi_1 \wedge \psi_2$ if and only if π satisfies ψ_1 and π satisfies ψ_2.
5. π satisfies $\mathsf{X}\psi$ if and only if π_1 satisfies ψ.
6. π satisfies $\mathsf{G}\psi$ if and only if for each i, π_i satisfies ψ.
7. π satisfies $\mathsf{F}\psi$ if and only if there exists some i such that π_i satisfies ψ.
8. π satisfies $\psi_1 \mathsf{U} \psi_2$ if and only if there exists some i such that (i) π_i satisfies ψ_2, and (ii) for each $j < i$, π_j satisfies ψ_1.

Not surprisingly, "F" is called the *eventuality* operator, "G" is called the *always* operator, "X" is called the *next time* operator, and "U" is called the *until* operator. A Kripke Structure $\kappa \doteq \langle S, R, L, s_0 \rangle$ will be said to satisfy formula ψ if and only if every path π of κ such that $\pi^0 \doteq s_0$ satisfies ψ.

Given this semantics, it is easy to specify different interesting properties of reactive systems using LTL. For instance, consider the mutual exclusion property for the 3 process system we referred to in the previous section. Here a state of the

system is the tuple of the *local states* of each of the component processes, together with the valuation of the shared variables, communication channels, etc. A local state of a process is given by the valuation of its local variables, such as program counter, local stack, etc. Let P_1, P_2, and P_3 be atomic propositions that specify that the program counter of processes 1, 2, and 3 are in the critical section respectively. That is, in the Kripke Structure, the label maps a state s to P_i if and only if the program counter of process i is in the critical section in state s. Then the LTL formula for mutual exclusion is given by:

$$\psi \doteq \mathsf{G}(\neg(P_1 \wedge P_2) \wedge \neg(P_2 \wedge P_3) \wedge \neg(P_3 \wedge P_1))$$

As an aside, notice the use of "$\doteq$". When we write $A \doteq B$ we mean that A is a shorthand for writing B, and we use this notation throughout the dissertation. In other treatises, one might often write this as $A = B$. But since much of the work in this dissertation is based on a fixed formal logic, namely ACL2, we restrict the use of the symbol "$=$" to only formulas in ACL2.

It should be clear that the LTL formulas can become big and cumbersome as the number of processes increases. In particular, if the system has an unbounded number of processes then specification of mutual exclusion by the above approach is not possible; one needs to have a stronger atomic proposition that can talk about quantification over all processes.

Given an LTL formula ψ and a Kripke Structure κ how do we decide if κ satisfies ψ? This is possible if κ has a finite number of states, and the method is known as *model checking*.[2] There are several interesting model checking algorithms and a complete treatment of such is beyond the scope of this dissertation. We merely

[2]Model checking is the generic name for decision procedures for formalisms based on temporal logics, μ-calculus, etc. We only talk about LTL model checking in this dissertation.

sketch one algorithm that is based on the construction of a Büchi automaton, since we will use properties of this algorithm in Chapter 16.

A Büchi automaton $\mathcal{A}$ is given by a 5-tuple $\langle \Sigma, Q, \Delta, q_0, F \rangle$, where Σ is called the *alphabet* of the automaton, Q is the set of *states* of the automaton, $\Delta \subseteq Q \times \Sigma \times Q$ is called the *transition relation*, $q_0 \in Q$ is called the *initial state*, and $F \subseteq Q$ is called the *set of accepting states*.[3] An infinite sequence of symbols from Σ constitutes a *word*. We say that $\mathcal{A}$ *accepts* a word σ if there exists an infinite sequence ρ of states of $\mathcal{A}$ with the following properties:

- ρ^0 is the initial state of $\mathcal{A}$,
- For each i, $\langle \rho^i, \sigma^i, \rho^{i+1} \rangle \in \Delta$, and
- Some accepting state occurs in ρ infinitely often.

The *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of words accepted by $\mathcal{A}$.

What has all this got to do with model checking? Both LTL formulas and Kripke Structures can be translated to Büchi automata. Here is the construction of a Büchi automaton $\mathcal{A}_\kappa$ for a Kripke Structure $\kappa \doteq \langle S, R, L, s_0 \rangle$, such that every word σ accepted by $\mathcal{A}_\kappa$ corresponds to paths in κ. That is, for every word σ accepted by $\mathcal{S}$, $\sigma^i \subseteq \mathcal{AP}$ and there is a path π in κ such that σ^i is equal to $L(\pi^i)$.

- The set of states of $\mathcal{A}$ is the set S. Each state is an accepting state.
- $\Sigma \doteq 2^{\mathcal{AP}}$ where $\mathcal{AP}$ is the set of atomic propositions.
- $\langle s, \alpha, s' \rangle \in \Delta$ if and only if $\langle s, s' \rangle \in R$ and $L(s)$ is equal to α.

Similarly, given an LTL formula ψ we can construct a Büchi automaton $\mathcal{A}_\psi$ such that every word accepted by this automaton satisfies ψ. This construction, often referred

[3] The terms *state*, *transition relation*, etc. are used with different meanings when talking about automata and Kripke Structures, principally because they are used to model the same artifacts of a computing system. In discussing Kripke Structures and automata together, this “abuse” of notation might cause ambiguity. We hope that the structure we are talking about will be clear from the context.

to as *tableau construction*, is complicated but well-known [CGP00, CES86]. Checking if κ satisfies ψ now reduces to checking $\mathcal{L}(\mathcal{A}_\kappa) \subseteq \mathcal{L}(\mathcal{A}_\psi)$. It is well-known that the languages recognized by a Büchi automaton are closed under complementation and intersection. It is also known that given an automaton $\mathcal{A}$, there is an algorithm to check if $\mathcal{L}(\mathcal{A})$ is empty. Thus, we can check the language containment question above as follows. Create a Büchi automaton $\mathcal{A}_{\kappa,\psi}$ such that $\mathcal{L}(\mathcal{A}_{\kappa,\psi}) \doteq \mathcal{L}(\mathcal{A}_\kappa) \cap \overline{\mathcal{L}(\mathcal{A}_\psi)}$. Then check if $\mathcal{L}(\mathcal{A}_{\kappa,\psi})$ is empty.

No practical model checker actually constructs the above automata and checks emptiness explicitly. Nevertheless, the above construction suggests some of the key limitations of model checking. Note that if a system contains n Boolean variables then the number of possible states in κ is 2^n. Thus the number of states in the automaton for κ is exponential in the number of variables in the original system. Practical model checkers perform a number of optimizations to prevent this "blow-up". One of the key approaches involves methods for efficient representation of states using BDDs [Bry86], which allows model checking to scale up to systems containing thousands of state variables. Nevertheless, in practice, model checking suffers from the well-known *state explosion* problem.

Several algorithmic techniques have been devised recently to ameliorate the state explosion problem. These are primarily based on the idea that in many cases one can reduce the problem of checking of a temporal formula ψ on a Kripke Structure κ to checking some formula possibly in a smaller structure κ'. We will consider two very simple reductions in Chapter 16. Modern model checkers perform a host of reductions to make the model checking problem tractable. They include identification of *symmetry* [CEJS98], reductions based on partial order [KP88], assume-

guarantee reasoning [Pnu84], and many others. In addition, one popular approach is based on iterative refinement of abstractions based on counterexamples [CGJ+00]. Since model checking is a decision procedure, if a formula does not satisfy a Kripke Structure then model checking can return a counterexample, that is, a path through the Kripke Structure that does not satisfy the formula. In counterexample-based refinements, one starts with a Kripke Structure $\hat{\kappa}$ such that every execution of κ can be appropriatelyviewed as an execution of $\hat{\kappa}$ (but not necessarily vice versa). Then $\hat{\kappa}$ is referred to as an *abstraction* of κ. It is possible to find abstractions of a Kripke Structure with very small number of states. One then applies model checking to check if $\hat{\kappa}$ satisfies ψ. If the model checking succeeds, then κ must satisfy ψ as well. If the model checking fails, the counterexample produced might be spurious since $\hat{\kappa}$ has more execution paths than κ. One then uses this counterexample to iteratively *refine* the abstraction. This process concludes when either (i) the counterexample provided by model checking the abstraction is also a counterexample on the "real" system, that is, κ, or (ii) one finds an abstraction such that the model checking succeeds.

2.3 Axiomatic Semantics and Verification Conditions

The use of decision procedures is one approach to scaling up formal verification of computing systems. The idea is to automate the verification process by expressing the verification problem in a decidable formalism. The use of *assertions* and *axiomatic semantics* forms another approach. The goal of this approach is to simplify program verification by factoring out the details of the machine executing the program from the verification process.

How do we talk about a program formally? If we use Kripke Structures to model the program executions, then we must talk in terms of *states*. The states, of course, are valuations of the different components of the *machine* executing the program. That is, in using Kripke Structures as a formalism to talk about the program, we must think in terms of the underlying machine. Specifying the semantics of a program by describing the effect of its instructions on the machine state is termed the *operational approach* to modeling the program and the semantics so defined is called the *operational semantics* [McC62]. Thus operational semantics forms the basis of applying model checking to reason about programs. We will later see that in many theorem proving approaches we use operational semantics as well. Indeed, operational models have often been lauded for their clarity and concreteness. Nevertheless it is cumbersome to reason about the details of the executing machine when proving the correctness of a program. The goal of axiomatic semantics [Hoa69, Dij75] is to come to grips with treating the program text itself as a mathematical object.

To do this, we will think of each instruction of the program as performing a transformation of predicates. To see how this is done, assume that I is any sequence of instructions in the programming language. The axiomatic semantics of the language are specified by a collection of formulas of the form $\{\mathcal{P}\}I\{\mathcal{Q}\}$, where $\mathcal{P}$ and $\mathcal{Q}$ are (first order) predicates over the program variables. Such a formula can be read as: "If $\mathcal{P}$ holds for the state of the machine when the program is poised to execute $\mathcal{I}$ then, after the execution of I, $\mathcal{Q}$ holds." Predicates $\mathcal{P}$ and $\mathcal{Q}$ are called the *precondition* and *postcondition* for I respectively. For example, if I is a single instruction specifying an assignment statement `x := a`, then its axiomatic semantics is given by the following schema:

- $\{\mathcal{P}\}\mathtt{x} := \mathtt{a}\{\mathcal{Q}\}$ holds if $\mathcal{P}$ is obtained by replacing every occurrence of the variable $\mathtt{x}$ in $\mathcal{Q}$ by $\mathtt{a}$.

This schema is known as the axiom of assignment. We can use this schema to derive, for example, $\{\mathtt{a} > 1\}\mathtt{x} := \mathtt{a}\{\mathtt{x} > 1\}$ which says that if the machine is in a state s in which the value of $\mathtt{a}$ is greater than 1, then in the state s' reached after executing `x:=a` from s, the value of `x` must be greater than 1. Notice that although the axiom is *interpreted* to be a statement about machine states, the schema itself, and its application involve syntactic manipulation of the program constructs without requiring any insight about the operational details of the machine.

Hoare [Hoa69] provides a collection of 5 schemas like the above to specify the semantics of a simple programming language. In addition, he provides the following inference rule, often referred to as the *rule of composition*.

- Infer $\{\mathcal{P}\}\langle i_1; i_2\rangle\{\mathcal{Q}\}$ from $\{\mathcal{P}\}i_1\{\mathcal{R}\}$ and $\{\mathcal{R}\}i_2\{\mathcal{Q}\}$.

Here $\langle i_1; i_2\rangle$ represents the sequential execution of the instruction sequences i_1 and i_2. Another rule, which allows generalization and the use of logical implication, is the following.

- Infer $\{\mathcal{P}\}i\{\mathcal{Q}\}$ from $\{\mathcal{R}_1\}i\{\mathcal{R}_2\}$, $\mathcal{P} \Rightarrow \mathcal{R}_1$, and $\mathcal{R}_2 \Rightarrow \mathcal{Q}$

Here, "$\Rightarrow$" is simply logical implication. It should be clear from the examples given above, that one can define such axiom schema and inference rules to capture the semantics of a programming language in terms of how *predicates* on states change on execution of instructions of a program. First order logic, together with the Hoare axioms and inference rules, form a proof system in which we can now talk about the correctness of programs. Suppose we are given a program Π and we want to prove that if the program starts from some state satisfying $\mathcal{P}$, then the state reached on

```
1:  X:=0;                          {T}
2:  Y:=10;
3:  if (Y ≤ 0) goto 7;             {(X + Y) = 10}
4:  X:=X+1;
5:  Y:=Y-1;
6:  goto 3;
7:  HALT                           {X = 10}
```

Figure 2.1: A Simple One-Loop Program

termination satisfies $\mathcal{Q}$. This can be succinctly written using axiomatic semantics as $\{\mathcal{P}\}\Pi\{\mathcal{Q}\}$. $\mathcal{P}$ and $\mathcal{Q}$ are called the precondition and postcondition *of the program*. One then derives this formula as a theorem.

How do we verify a program given axiomatic semantics for the language? One typically annotates the program with predicates (called *assertions*) at certain locations (that is, certain values of the program counter). These locations correspond to the entry and exit of the basic blocks of the program such as loop tests and program entry and exit points. These annotated program points are also called *cutpoints*. The entry point of the program is annotated with the precondition, and the exit point is annotated with the postcondition. One then shows that if the program control is in an annotated state satisfying the corresponding assertion, then the next annotated state it reaches will also satisfy the assertion. This is achieved by using the axiomatic semantics for the programming language.

Let us see how all this is done using a simple one-loop program that is shown in Figure 2.1. The program consists of two variables X and Y, and simply loops 10 times incrementing X in each iteration. In the figure, the number to the left of each instruction is the corresponding program counter value for the loaded program. The

cutpoints for this program correspond to program counter values 1 (program entry), 3 (loop test), and 7 (termination). The assertions associated with each cutpoint are shown to the right. Here the precondition T is assumed to be the predicate that is universally true. The postcondition says that the variable X has the value 10. Notice that in writing the assertions we have ignored type considerations such as that the variables X and Y store natural numbers.

How do we now show that every time the control reaches a cutpoint the assertion holds? Take for a simple example the cutpoints given by the program counter values 1 and 3. Let us call this fragment of the execution $\mathtt{1} \rightarrow \mathtt{3}$, identifying the beginning and ending values of the program counter along the execution of the basic block. Then we must show the following formula to be a theorem:

$$\{\mathtt{T}\}\langle \mathtt{X} := \mathtt{0}; \mathtt{Y} := \mathtt{10}\rangle\{(\mathtt{X} + \mathtt{Y}) = 10\}$$

Applying the assignment axiom and composition and first order implication above, we can now derive the following proof obligation:

$$\mathtt{T} \Rightarrow (0 + 10) = 10$$

This proof obligation, of course, is trivial. But one thing to observe about it is that by applying the Hoare axioms — in this case the axiom of assignment — we have obtained a formula that is free from the constructs of the programming language. Such a formula is known as a *verification condition*. To finish the proof of correctness of the program here, we generate such verification conditions for each of the execution paths $\mathtt{1} \rightarrow \mathtt{3}$, $\mathtt{3} \rightarrow \mathtt{3}$, and $\mathtt{3} \rightarrow \mathtt{7}$, and show that they are logical truths.

One must observe that in addition to the precondition and postcondition, we have added an assertion at the loop test. The process of annotating a loop

test is often colloquially referred to as "cutting the loop". It is difficult to provide a syntactic characterization of loops in terms of predicates (as was done for the assignment statement); thus if we omit annotation of a loop, then the Hoare axioms are normally not sufficient to generate verification conditions.

The careful reader will notice that if we do prove the verification conditions based on an axiomatic semantics the way we described above, it only guarantees *partial correctness* of the program. That is, it guarantees that if the control ever reaches the exit point then the postcondition holds. It does not guarantee *termination*, that is, the control eventually reaches such a point. *Total correctness* provides both the guarantees of partial correctness and termination. To have total correctness one needs an argument based on well-foundedness. We will carefully look at such arguments in the context of the ACL2 logic in the next chapter. In the context of program termination, well-foundedness means that there must be a function of the program variables whose value decreases as the control goes from one cutpoint to the next until it reaches the exit point, and the value of this function cannot decrease indefinitely. Such a function is called a *ranking function*. For total correctness one attaches, in addition to assertions, ranking functions at every cutpoint, and the axiomatic semantics of the programming language is augmented so as to be able to reason about such ranking functions.

In practice, the verification conditions might be complicated formulas and their proofs might not be trivial. Practical applications of axiomatic semantics depend on two tools, namely a *verification condition generator* (VCG) that takes an annotated program and generates the verification conditions, and a theorem prover that proves the verification conditions. In this approach it is not necessary

to formalize the semantics of the program in a theorem prover or reason about the operational details of the machine executing the program. The VCG, on the other hand, is a tool that manipulates assertions based on the axiomatic semantics of the language.

Notice that one downside to applying this method is that one requires two trusted tools, namely a VCG and a theorem prover. Furthermore, one has to provide an axiomatic semantics (and implement a VCG) for every different programming language that one is interested in, so that the axioms capture the language constructs as formula manipulators. As new programming constructs are developed, the axiomatic semantics have to be changed to interpret such constructs. For instance, Hoare axioms are insufficient if the language contains pointer arithmetic, and additional axioms are necessary. With the plethora of axioms it is often difficult to see if the proof system itself remains sound. Early versions of *separation logic* for example, that was introduced to augment axiomatic semantics to reason about pointers, were later found to be inconsistent [Rey00]. In addition, implementing a VCG for a practical programming language is a substantial enterprise. For example, method invocation in a language like JVM involves complicated non-syntactic issues like method resolution with respect to the object on which the method is invoked, as well as side effects in many parts of the machine state such as the call frames of the caller and the callee, thread table, heap, and class table. Coding all this in terms of predicate transformation, instead of state transformation as required when reasoning about an operational model of the program, is difficult and error-prone. VCGs also need to do some amount of logical reasoning in order to keep the size of the generated formula reasonable. Finally, the axiomatic semantics of the program are not

usually specified as a formula in a logic but rather encoded in the VCG which makes them difficult to inspect. Of course, one answer to all these concerns is that one can model the VCG itself in a logic, verify the VCG with respect to the logic using a theorem prover, and then use such a verified VCG to reason about programs. Some recent research has focused on formally verifying VCGs using a theorem prover with respect to the operational semantics of the corresponding language [HM95, Glo99]. However, formal verification of a VCG is a substantial enterprise and most VCGs for practical languages are not verified.

Nevertheless, axiomatic semantics and assertions have been popular both in program verification theory and in its practical application. It forms the basis of several verification projects [BR01, HJMS02, DLNS98, Nec98]. A significant benefit of using the approach is to factor out the *machine details* from the program. One reasons about the program using assertions (and ranking functions), without concern for the machine executing it, except for the axiomatic semantics.

2.4 Bibliographic Notes

The literature on formal verification is vast, with several excellent surveys. Some of the significant surveys of the area include those by Kern and Greenstreet [KG99], and Gupta [Gup92]. In addition, a detailed overview on model checking techniques is presented in a book on the subject by Clarke, Grumberg and Peled [CGP00].

Theorem proving was started arguably with the pioneering *Logic Theorist System* of Newell, Shaw, and Simon [NSS59]. One of the key advocates of using theorem proving for verification of computing systems was McCarthy [McC62] who wrote: “Instead of debugging programs one should prove that it meets its specifi-

cation and the proof should be checked by a computer program." McCarthy also suggested the use of *operational semantics* for reasoning about programs. Many of the early theorem provers were based on the principle of resolution [Rob65] that forms a sound and complete proof rule for first order predicate calculus. The focus on resolution was motivated by the goal to implement fully automatic theorem provers. Theorem provers based on resolution are used today in many contexts; for instance, EQP, a theorem prover for equational reasoning has recently "found" a proof of the Robbin's problem which has been an open problem in mathematics for about 70 years [McC97b]. In the context of verification of computing systems, however, non-resolution theorem provers have found more applications. Some of the early work on non-resolution theorem proving were done by Wang [Wan63], Bledsoe [Ble77], and Boyer and Moore [BM79]. The latter, also known as the Boyer-Moore theorem prover or Nqthm, is the precursor of the ACL2 theorem prover that is the basis of the work in this dissertation. Nqthm and ACL2 have been used in reasoning about some of the largest computing systems ever verified. The bibliographic notes for the next chapter lists some of their applications. Other significant theorem provers in active use for verification of computing systems include Forte [AJK$^+$00], HOL [GM93], HOL Light [Har00], Isabelle [NPW02], and PVS [ORS92].

The idea of using temporal logics for specifying properties of reactive systems was suggested by Pnueli [Pnu77]. Model checking was discovered independently by Clarke and Emerson [CE81], and Queille and Sifakis [QS82]. It is one of the most widely used formal verification technology used in the industry today. Some of the important model checkers include SMV [McM93], VIS [BHSV$^+$96], NuSMV [CCGR99], SPIN [Hol03], and Murϕ [Dil96]. Most model checkers in prac-

tice include several reductions and optimizations [CEJS98, Pnu84, KP88]. Model checkers have achieved amazing results on industrial problems. Indeed, model checking in some form is used in formal reasoning in almost every hardware industry. While other decision procedures such as *symbolic trajectory evaluation* [Cho99] and its generalization [YS02] have found applications in specification and verification of computing systems in recent times, they can be shown to be logically equivalent to instances of the model checking algorithms [SSTV04].

The notion of assertions was made explicit in a classic paper by Floyd [Flo67], although the idea of attaching assertions to program points appears much earlier, for example in the work of Goldstein and von Neumann [GvN61], and Turing [Tur49]. Program logics were introduced by Hoare [Hoa69] and Dijkstra [Dij75]. Assertional reasoning was extended to concurrent programs by Owicki and Gries [OG76]. King [Kin69] wrote the first mechanized VCG. Implementations of VCGs abound in the program verification literature. Some of the recent substantial projects involving complicated VCG constructions include ESC/Java [DLNS98], proof carrying code [Nec98], and SLAM [BR01].

Chapter 3

Introduction to ACL2

The name "ACL2" stands for *A Computational Logic for Applicative Common Lisp*. It is used to denote (i) a programming language based on Common Lisp, (ii) a logic, and (iii) a mechanical theorem prover for the logic. ACL2 is an industrial-strength theorem prover that has been used successfully in a number of formal verification projects both in the industry and academia. As a logic, ACL2 is a first order logic of recursive functions with equality and induction. As a theorem prover, ACL2 is a complex software implementing a wide repertoire of heuristics and decision procedures aimed at effectively proving large and complicated theorems about mathematical artifacts and computing systems. The work in this dissertation is based on the logic of ACL2 and all the theorems we claim to have mechanically verified have been derived using the ACL2 theorem prover.[1] In this chapter, we present ACL2 *as a logic*, and briefly touch upon how computing systems can be defined in ACL2 and how the logic can be used to prove theorems about them. To

[1] At the time of this writing, the latest official release of ACL2 is version 2.9. Our comments on ACL2 are pertinent to this release and all the theorems we describe have been certified with this version.

facilitate the understanding of the logic, we discuss its connections with traditional first order logic. We omit description of the other facets of ACL2, namely as a programming language and as a theorem prover. Readers interested in a more thorough understanding of ACL2 are referred to the ACL2 home page [KMb]. In addition, we list several books and papers about ACL2 in Section 3.4.

3.1 Basic Logic of ACL2

Recall from Chapter 2 that a formal logic consists of the following three components:

- A *formal language* for describing formulas.
- A set of formulas called *axioms*.
- A set of *inference rules* that allow derivation of new formulas from old.

As a logic, ACL2 is essentially a quantifier-free first order logic of recursive functions with equality. Formulas are built out of *terms*, and terms are built out of *constants*, *variables*, and *function symbols*. More precisely, a term is either a constant, or a variable, or the application of an n-ary function symbol f on a list of n terms. The syntax of ACL2 is based on the prefix-normal syntax of Common Lisp. Thus, the application of f on arguments $x_1, \ldots, x_n$ is represented as (f x_1 ... x_n) instead of the more traditional $f(x_1, \ldots, x_n)$. However, for this dissertation, we will use the more traditional syntax. We will also write some binary functions in the traditional infix form, thus writing $x + y$ instead of $+(x, y)$.

The constants in ACL2 comprise what is known as the *ACL2 universe*. The universe is open but contains *numbers*, *characters*, *strings*, certain types of *constant symbols*, and *ordered pairs*. We quickly recount their representations below.

- Numbers are represented as in traditional mathematics, for example 2, -1, 22/7, etc. The universe contains rational and complex rational numbers.

- Characters are represented in a slightly unconventional syntax, for example the character `a` is represented by `#\a`. We will not use characters in this dissertation.

- A string is represented as a sequence enclosed within double quotation marks, such as `"king"`, `"queen"`, `"Alice"`, etc. Note that the first character of the string `"king"` is the *character* `#\k`.

- ACL2 has a complicated mechanism for representing constant symbols, that is derived from Common Lisp. We will not worry about that representation. It is sufficient for our purpose to know that the universe contains two specific constant symbols `T` and `NIL`, which will be interpreted as Boolean true and false respectively, and certain other symbols called *keywords*. Keywords are clearly demarcated by a beginning "`:`" (colon). Thus `:abc`, `:research`, etc. are keywords.

- An ordered pair is represented by a pair of constants enclosed within parenthesis and separated by a "`.`" (dot). Examples of ordered pairs are `(1 . 2)`, `(#\a . :abc)`, and `("king" . 22/7)`. ACL2 and Common Lisp use ordered pairs for representing a variety of data structures. One of the key data structures that is extensively used is the *list* or *tuple*. A list containing x, y, and z is represented as the ordered pair `(`x `. (`y `. (`z `. NIL)))`. In this dissertation, we will use the notation $\langle x, y, z \rangle$ to denote such tuples.

Note that we have not talked about the syntax of variables. Throughout this dissertation we will talk about terms (and formulas), which will contain constant, variable and function symbols. In any formal term that we show, any symbol that is not a constant symbol (according to our description above) or a function symbol (which should be identifiable from the context) will be taken to be a variable symbol.

Formulas are built out of terms by the use of the equality operator "=", and logical operators "$\vee$" and "$\neg$". Formally, if τ_1 and τ_2 are terms, then $\tau_1 = \tau_2$ is an *atomic formula*. Formulas are then recursively defined as follows:

- Every atomic formula is a formula.
- If Φ_1 and Φ_2 are formulas then so are $(\neg\Phi_1)$ and $(\Phi_1 \vee \Phi_2)$.

We drop parenthesis whenever it is unambiguous to do so. We also freely use the logical operators "$\wedge$", "$\Rightarrow$", etc. as well, in talking about formulas. Formally speaking, they are abbreviations. That is, $\Phi_1 \wedge \Phi_2$ is an abbreviation for $\neg(\neg\Phi_1 \vee \neg\Phi_2)$, $\Phi_1 \Rightarrow \Phi_2$ for $\neg\Phi_1 \vee \Phi_2$, and $\Phi_1 \Leftrightarrow \Phi_2$ for $(\Phi_1 \Rightarrow \Phi_2) \wedge (\Phi_2 \Rightarrow \Phi_1)$.

The logical axioms of ACL2 constitute the standard first order axioms, namely **Propositional Axiom**, **Identity Axiom**, and **Equality Axiom**. These are described below. Notice that all the logical axioms are *axiom schemas*.

Propositional Axiom: For each formula Φ, $\neg\Phi \vee \Phi$ is an axiom.

Identity Axiom: For each term τ, the formula $\tau = \tau$ is an axiom.

Equality Axiom: If $\alpha_1, \ldots, \alpha_n$, and $\beta_1, \ldots, \beta_n$ are terms, then the following formula is an axiom, where f is an n-ary function symbol:

$$((\alpha_1 = \beta_1) \Rightarrow \ldots ((\alpha_n = \beta_n) \Rightarrow (f(\alpha_1, \ldots, \alpha_n) = f(\beta_1, \ldots, \beta_n))) \ldots)$$

In addition to axioms, the logic must provide inference rules to derive theorems in the logic. The inference rules of the ACL2 logic constitute the inference rules of Propositional Calculus, the first order rule of instantiation, and well-founded induction up to ϵ_0. The propositional inference rules are the following:

Expansion Rule: Infer $\Phi_1 \vee \Phi_2$ from Φ_2.

Contraction Rule: Infer Φ_1 from $\Phi_1 \vee \Phi_1$.

Associative Rule: Infer $(\Phi_1 \vee \Phi_2) \vee \Phi_3$ from $\Phi_1 \vee (\Phi_2 \vee \Phi_3)$.

Cut Rule: Infer $\Phi_2 \vee \Phi_3$ from $\Phi_1 \vee \Phi_2$ and $\neg\Phi_1 \vee \Phi_3$.

To describe the **Instantiation Rule**, we need some more terminology. For a term τ, we refer to the variable symbols in τ by $\nu(\tau)$. A *substitution* is a mapping from a set of variables to terms. A substitution σ that maps the variable v_1 to τ_1 and v_2 to τ_2 will be written as $\sigma \doteq [v_1 \rightarrow \tau_1, v_2 \rightarrow \tau_2]$, where the domain of σ, referred to as $dom(\sigma)$ is the set $\{v_1, v_2\}$. For a term τ, we write τ/σ to denote the term obtained by replacing every variable in $\nu(\tau) \cap dom(\sigma)$ by $\sigma(v)$. For a formula Φ, Φ/σ is defined analogously. Then the **Instantiation Rule** is as specified below.

Instantiation Rule: Infer Φ/σ from Φ for any substitution σ.

As is customary given these axiom schemas, abbreviations, and inference rules, we will always interpret the operators "$\vee$", "$\wedge$", "$\neg$", "$\Rightarrow$", "$\leftrightarrow$", and "$=$" as disjunction, conjunction, negation, implication, equivalence, and equality respectively. In addition, ACL2 also has an **Induction Rule** that allows us to derive theorems using well-founded induction. The **Induction Rule** is a little more complex than the rules we have seen so far, and we will look at it after we understand how well-foundedness arguments are formalized in ACL2.

3.2 Ground Zero Theory

It should be clear from the description so far that the logic of ACL2 is a fairly traditional first order logic. First order logic and its different extensions have been studied by logicians for more than a century. However, ACL2 has been designed not for the study of logic but for *using* it to reason about different mathematical artifacts. To achieve this goal, ACL2 provides a host of additional axioms aimed at capturing properties of different mathematical artifacts. Such axioms, together with the axiom schemas and inference rules we presented above, form what is known as

the ACL2 Ground Zero Theory (GZ for short). Since ACL2 is based on Common Lisp, the axioms of GZ formalize many of the Lisp functions. For example, here is an axiom that relates the functions *car* and *cons*.

$$\mathit{car}(\mathit{cons}(x, y)) = x$$

The axiom can be interpreted as: "For any x and y, the function *car*, when applied to *cons* of x and y, returns x." Notice that formulas are implicitly universally quantified over free variables, although the syntax is quantifier-free. The implicit universal quantification occurs as a consequence of the **Instantiation Rule**; given the above axiom, we can apply this rule to prove, for instance, the theorem $\mathit{car}(\mathit{cons}(2, 3)) = 2$.

GZ provides axioms formalizing about 200 applicative (that is, side-effect free) functions described in the Common Lisp Reference Manual [Ste90]. There are axioms for all the arithmetic functions, and functions for manipulating strings, characters, and lists. A description of all the axioms in GZ is beyond the scope of this dissertation. In Figure 3.1, we provide a brief list of some of the important functions, and how they can be interpreted given the axioms. It is not necessary at this point to understand the meaning of every function; we will come back to many of them later.

The axioms in GZ have an important property, which we can call *evaluability.* That means that for any term τ with no variables, we can determine the "value" of τ using the axioms. More precisely, a term τ is said to be *expressible* in GZ if for each function symbol f in τ, GZ has some axiom referring to f. A term τ is called a *ground term* if and only if it contains no variable, that is, $\nu(\tau)$ is empty. The axioms of GZ have the property that given any ground term τ expressible in GZ we can determine a constant c such that $(\tau = \mathsf{c})$ is a theorem. The constant c is then

Function Symbols	Interpretation
equal(x, y)	Returns `T` if x is equal to y, else `NIL`
if(x, y, z)	Returns z if x is equal to `NIL`, else y
and(x, y)	Returns `NIL` if x is equal to `NIL`, else y
or(x, y)	Returns y if x is equal to `NIL`, else x
not(x)	Returns `T` if x is equal to `NIL`, else `NIL`
consp(x)	Returns `T` if x is an ordered pair, else `NIL`
cons(x, y)	Returns the ordered pair of x and y
car(x)	If x is an ordered pair returns its first element, else `NIL`
cdr(x)	If x is an ordered pair returns its second element, else `NIL`
nth(i, l)	Returns the i-th element of l if l is a list, else `NIL`
update-nth(i, v, l)	Returns a copy of list l with the i-th element replaced by v
len(x)	Returns the length of the list x
acl2-numberp(x)	Returns `T` if x is a number, else `NIL`
integerp(x)	Returns `T` if x is an integer, else `NIL`
rationalp(x)	Returns `T` if x is a rational number, else `NIL`
natp(x)	Returns `T` if x is a natural number, else `NIL`
zp(x)	Returns `NIL` if x is a natural number greater than 0, else `T`
$(x + y)$	Returns the sum of x and y. Treats non-numbers as 0
$(x - y)$	Returns the difference of x and y. Treats non-numbers as 0
$(x \times y)$	Returns the product of x and y. Treats non-numbers as 0
(x/y)	Returns the quotient of x and y. Treats non-numbers as 0
nfix(x)	Returns x if x is a natural number, else 0

Figure 3.1: Some Functions Axiomatized in GZ

called the *value* of the term τ.

Since the functions axiomatized in GZ are described in the Common Lisp Manual, we can ask about the relation between the value of the ground term τ as specified by the axioms and the value returned by evaluating the term in Lisp. There is one major difference. Functions in Common Lisp are *partial*; each function has an intended domain of application in which the standard specifies the return value of the function. For example, the return value of the function *car* is specified by Common Lisp only when its argument is either NIL or an ordered pair. Thus the value of *car*(2) is undefined, and evaluating this term can produce arbitrary results, including different return values for different evaluations. On the other hand, all functions axiomatized in GZ are *total*, that is, the axioms specify what each function returns on every possible argument. This is done as follows. For each Common Lisp function axiomatized in GZ, there is also a function that "recognizes" its intended domain, that is, returns T if the arguments are in the intended domain, and NIL otherwise. For instance, a unary function *consp* is axiomatized to return T if and only if its argument is an ordered pair, and NIL otherwise. The intended domain of $\mathit{car}(x)$, then, is given by the formula $(\mathit{consp}(x) = \mathtt{T}) \vee (x = \mathtt{NIL})$. The axioms of GZ specify the same return value for the function as Common Lisp does for arguments in the intended domain. In addition, GZ provides *completion axioms* that specifies the value of a function on arguments outside the intended domain. The completion axiom for *car*, shown below, specifies that if x is outside the intended domain then $\mathit{car}(x)$ returns NIL.[2]

$$\neg((\mathit{consp}(x) = T) \vee (x = \mathtt{NIL})) \Rightarrow \mathit{car}(x) = \mathtt{NIL}$$

Similarly, in case of arithmetic functions such as $+$, $-$, $<$, etc., if one of the arguments is not a number, the completion axiom allows us to interpret that argument

[2]One should note that based on the completion axiom and the intended interpretation of *car*, we can interpret NIL as both the Boolean false as well as the empty list. This interpretation is customary both in ACL2 and in Common Lisp.

to be 0.

As an aside, we remark that since the axioms of GZ and Common Lisp agree on the return value of each axiomatized function on arguments in its intended domain, ACL2 can sometimes reduce variable-free terms to constants by using the Common Lisp execution engine to evaluate the term [KM94]. This "execution capability" allows ACL2 to deal with proofs of theorems containing large constants. Fast execution is one of the key reasons for the success of the ACL2 theorem prover in the formal verification of large computing systems, allowing it to be used for simulation of formal models in addition to reasoning about them [GWH00].

3.2.1 Terms, Formulas, Functions, and Predicates

Before proceeding further, we should point out a duality between terms and formulas in ACL2. So far in the presentation, we have distinguished between terms and formulas. In ACL2, however, we often use terms *in place of* formulas. When a term τ is used in place of a formula, then the intended formula is $\neg(\tau = \texttt{NIL})$. Indeed, a user of the theorem prover only writes terms and never writes formulas. If we prove the term τ as a theorem, then we can thus interpret the theorem as follows: "For any substitution σ that maps each variable in $\nu(\tau)$ to an object in the ACL2 universe, the value of the term τ/σ does not equal NIL."

How can we always write terms instead of formulas? This is achieved in GZ by providing certain "built-in" axioms. One of the important built-in functions axiomatized in GZ is the binary function *equal*. We have shown this function and its interpretation in Figure 3.1. We show the formal built-in axioms below. By the above convention, we can interpret *equal*(τ_1, τ_2) as the formula $\tau_1 = \tau_2$.

- $(x = y) \Rightarrow \mathit{equal}(x, y) = \texttt{T}$

- $\neg(x = y) \Rightarrow \mathit{equal}(x, y) = \mathtt{NIL}$

Logical operators are specified in terms of *equal* and another built-in function, namely the ternary function *if*. This function can be interpreted as "if-then-else" based on the following axioms.

- $(x = \mathtt{NIL}) \Rightarrow \mathit{if}(x, y, z) = z$
- $\neg(x = \mathtt{NIL}) \Rightarrow \mathit{if}(x, y, z) = y$

Using *if*, we can now talk about "function versions" of the logical operators "$\wedge$", "$\vee$", "$\neg$", "$\Rightarrow$", "$\Leftrightarrow$", etc. The functions, namely *and*, *or*, etc., together with their interpretations, are shown in Figure 3.1. Here we show the axioms that allow such interpretation.

- $\mathit{and}(x, y) = \mathit{if}(x, y, \mathtt{NIL})$
- $\mathit{or}(x, y) = \mathit{if}(x, x, y)$
- $\mathit{not}(x) = \mathit{if}(x, \mathtt{NIL}, \mathtt{T})$
- $\mathit{implies}(x, y) = \mathit{if}(x, \mathit{if}(y, \mathtt{T}, \mathtt{NIL}), \mathtt{T})$
- $\mathit{iff}(x, y) = \mathit{if}(x, \mathit{if}(y, \mathtt{T}, \mathtt{NIL}), \mathit{if}(y, \mathtt{NIL}, \mathtt{T}))$

Now that we have functions representing all logical operators and equality, we can write terms representing formulas. For instance, the completion axiom of *car* above can be represented as follows:

$$\mathit{implies}(\mathit{not}(\mathit{or}(\mathit{consp}(x), \mathit{equal}(x, \mathtt{NIL}))), \mathit{equal}(\mathit{car}(x), \mathtt{NIL}))$$

In this dissertation, we will find it convenient to use both terms and formulas depending on context. Thus, when we want to think about $\mathit{equal}(\tau_1, \tau_2)$ as a formula will write it as $\tau_1 = \tau_2$.

As a consequence of the duality between terms and formulas, there is also a duality between *functions* and *predicates*. In a formal presentation of first-order

logic [Sho67], one distinguishes between the function and predicate symbols in the following way. Terms are built out of constants and variables by application of the function symbols, and atomic formulas are built out of terms using the predicate symbols. Thus according to our description above, the logic of ACL2 has a single predicate symbol, namely "="; *equal*, as described above, is a function and *not* a predicate. Nevertheless, since *equal* only returns the values `T` and `NIL`, we will often find it convenient to call it a predicate symbol. We will refer to an n-ary function symbol P as a predicate when, for any ground term $P(\tau_1, \ldots, \tau_n)$, the value of the term is equal to either `T` or `NIL`. If the value is `NIL` we will say that P does not hold on $\tau_1, \ldots, \tau_n$, and otherwise we will say that it does. Thus we can refer to *consp* above as a predicate that holds if its argument is an ordered pair. Two other important unary predicates that we will use are (1) *natp* that holds if and only if its argument is a natural number, and (2) *zp* that does not hold if and only if its argument is a positive natural number. In some contexts, however, we will "abuse" this convention and also refer to a *term* τ as a predicate. In such situations, if we say that τ holds, all we mean is that $\neg(\tau = \texttt{NIL})$ is a theorem.

3.2.2 Ordinals and Well-founded Induction

Among the functions axiomatized in GZ are functions manipulating ordinals. Ordinals have been studied extensively for the last 100 years and form the basis of Cantor's set theory [Can95, Can97, Can52]. Ordinals are extensively used in ACL2, and afford the application of well-founded induction as an inference rule. The use of well-founded induction in proving theorems is one of the strengths of the ACL2 theorem prover. To understand the rule, we briefly review the theory of well-foundedness

and ordinals, and see how they allow induction.

In classical set theory, a *well-founded structure* consists of a (possibly infinite) set W, and a total order $\prec$ on W such that there is no infinite sequence $\langle \ldots w_2 \prec w_1 \prec w_0 \rangle$ where each $w_i \in W$. Thus, the set $\mathbb{N}$ of natural numbers forms a well-founded structure under the ordinary arithmetic "$<$".

Ordinals form another example of a well-founded structure, which is created by extending the set $\mathbb{N}$ and the interpretation of "$<$" as follows. We start extending $\mathbb{N}$ with a new element ω and extend the interpretation of "$<$" so that $0 < 1 < \ldots < \omega$. The "number" ω is called the first infinite ordinal. We add an infinite number of such "numbers" using a positional ω-based notation namely, $\omega + 1$, $\omega + 2$, $\omega \times 2$, ω^2, ω^ω, etc. We extend the interpretation of "$<$" analogously. This set of extended "numbers" is called the set of *ordinals*. The first few ordinals, in order, (with omissions) are $0, 1, \ldots, \omega, \omega+1, \omega+2, \ldots, \omega \times 2, \omega \times 3, \ldots, \omega^2, \omega^2+1, \ldots, \omega^2+\omega, \omega^2+\omega+1, \ldots, \omega^3, \omega^4, \ldots, \omega^\omega, \omega^{(\omega^\omega)}, \omega^{(\omega^{(\omega^\omega)})}, \ldots$ The limit of this sequence, namely $\omega^{\omega^{\omega^{\cdots}}}$ containing a tower of height ω, is called ϵ_0. This set forms a well-founded structure under the linear ordering which is the extension of "$<$" over the ordinals. The set of ordinals up to ϵ_0 forms a very small initial segment of ordinals; nevertheless we will be only interested in ordinals less than ϵ_0 since this is the set of ordinals that are represented in ACL2. For this dissertation whenever we talk about an ordinal we mean a member of this set.

How do we talk about ordinals in ACL2? To do so, one must provide a representation of the ordinals as constants in the ACL2 universe. The initial segment of ordinals, namely the natural numbers, are of course available as constants. The ordinals from ω and larger are represented as ordered pairs. We show some examples of ordinals and their representations as constants in Figure 3.2. It is not necessary

Ordinal	ACL2 Representation
0	0
1	1
2	2
3	3
$\ldots$	...
ω	((1 . 1) . 0)
$\omega + 1$	((1 . 1) . 1)
$\omega + 2$	((1 . 1) . 2)
$\ldots$	...
$\omega \times 2$	((1 . 2) . 0)
$\omega \times 2 + 1$	((1 . 2) . 1)
$\ldots$	...
$\omega \times 3$	((1 . 3) . 0)
$\omega \times 3 + 1$	((1 . 3) . 1)
$\ldots$	...
ω^2	((2 . 1) . 0)
$\ldots$	...
$\omega^2 + \omega \times 4 + 3$	((2 . 1) (1 . 4) . 3)
$\ldots$	...
ω^3	((3 . 1) . 0)
$\ldots$	...
ω^ω	((((1 . 1) . 0) . 1) . 0)
$\ldots$	...
$\omega^\omega + \omega^{99} + \omega \times 4 + 3$	((((1 . 1) . 0) . 1) (99 . 1) (1 . 4) . 3)
$\ldots$	...
ω^{ω^2}	((((2 . 1) . 0) . 1) . 0)
$\ldots$	...
$\omega^{(\omega^\omega)}$	((((((1 . 1) . 0) . 1) . 0) . 1) . 0)
$\ldots$	...

Figure 3.2: Examples of Ordinal Representation in ACL2

to understand the exact representation. What is important for our purpose is that GZ axiomatizes two predicates, namely a unary predicate *o-p* and a binary predicate "$\prec_o$",[3] which can be interpreted as follows.

- *o-p*(x) holds if x is the ACL2 representation of an ordinal. In particular of course *o-p*(x) holds if x is a natural number.
- Given two ordinals x and y, $(x \prec_o y)$ holds if and only if x is below y in the linear ordering of ordinals. In particular if x and y are natural numbers then $(x \prec_o y)$ is equal to $(x < y)$.

The existence of ordinals and the fact that they are well-founded allows ACL2 to prove formulas by induction. Here is the **Induction Rule** of ACL2.

Induction Rule: Infer Φ from

> **Base Case:** $(\neg\mathcal{C}_1 \wedge \ldots \wedge \neg\mathcal{C}_k) \Rightarrow \Phi$, and
>
> **Induction Step:** $(\mathcal{C}_i \wedge \Phi/\sigma_{i1} \wedge \ldots \wedge \Phi/\sigma_{ik_i}) \Rightarrow \Phi$ for each $1 \leq i \leq k$.
>
> if there exists some term m such that the following are theorems.
>
> 1. *o-p*(m)
> 2. $\mathcal{C}_i \Rightarrow m/\sigma_{ik} \prec_o m$, for each $1 \leq i \leq k$, $1 \leq j \leq k_i$.

Notice that the rule allows us to choose a (finite) number of induction hypotheses in the **Induction Step**. We can interpret the rule as follows: "Φ holds, if (1) Φ holds in the "base case" when $(\neg\mathcal{C}_1 \wedge \ldots \wedge \neg\mathcal{C}_k)$ holds, and (2) Φ holds whenever $\mathcal{C}_i$ holds, and some "smaller instance" of the formula, namely Φ/σ, holds." The fact that Φ/σ is a "smaller instance" of Φ is shown by conditions 1 and 2. Well-foundedness of the set of ordinals under "$\prec_o$" guarantees that the sequence cannot decrease indefinitely, justifying the rule.

[3] "$\prec_o$" is referred to as "`o<`" in ACL2.

It should be noted that we did not explicitly need the ordinals but *some well-founded structure*. In GZ, the only structure axiomatized to be well-founded is the set of ordinals under relation "$\prec_0$". In order to use any other well-founded structure, we will need to embed the structure inside the ordinals. More precisely, we will say that a pair $\langle \mathit{o\text{-}p}_{\prec}, \prec \rangle$ defines a well-founded structure if and only if there is a unary function $E_{\prec}$ such that the following are theorems:

- $\mathit{o\text{-}p}_{\prec}(x) \Rightarrow \mathit{o\text{-}p}(E_{\prec}(x))$
- $\mathit{o\text{-}p}_{\prec}(x) \wedge \mathit{o\text{-}p}_{\prec}(y) \wedge (x \prec y) \Rightarrow (E(x) \prec_o E(y))$

For instance, $\langle \mathit{natp}, < \rangle$ can be shown to define a well-founded structure by choosing E to be the identity function: $E(x) = x$. If $\langle \mathit{o\text{-}p}_{\prec}, \prec \rangle$ has been proved as above to define a well-founded structure, then we can replace the $\mathit{o\text{-}p}$ and "$\prec_o$" in the proof obligations 1 and 2 above by $\mathit{o\text{-}p}_{\prec}$ and "$\prec$" respectively.

3.3 Extension Principles

GZ axiomatizes many Lisp functions. One can prove as theorems formulas expressing properties of such functions. However, just having GZ fails to accommodate the intent of *using* ACL2 to reason about other mathematical artifacts or computing systems. Except in the unlikely case where (one of) the functions axiomatized in GZ already models the artifact we care about, we must be able to extend GZ by adding axioms to represent such models. Of course indiscriminate addition of axioms can render the logic inconsistent. To prevent this, ACL2 provides *extension principles* that allow us to extend GZ in a disciplined manner.

Before talking about the extension principles, we will fix some more terminology. A proof system with the first order axiom schema (namely **Propositional**

Axiom, **Identity Axiom**, and **Equality Axiom**) and inference rules, together with a collection of axioms specifying individual properties of some of the functions, is referred to as a *first order theory*, or simply *theory* [Sho67]. Thus GZ is a theory. The function symbols individually axiomatized in a theory $\mathcal{T}$ are said to have been *introduced* in $\mathcal{T}$, and the individual axioms are often called *nonlogical axioms* of $\mathcal{T}$. A term (or formula) is *expressible* in $\mathcal{T}$ if and only if all function symbols in the formula have been introduced in $\mathcal{T}$. We already talked about terms and formulas expressible in GZ. We say that a theory $\mathcal{T}'$ *extends* $\mathcal{T}$ if and only if all the nonlogical axioms of $\mathcal{T}$ are also nonlogical axioms of $\mathcal{T}'$.

The extension principles of ACL2 (discussed below) allow us to extend a theory by new nonlogical axioms. Given a theory $\mathcal{T}$, the nonlogical axioms introduced by the extension principles have the property that at least one function symbol is introduced in the extended theory $\mathcal{T}'$.[4] We will then say that the extension principle has *introduced* the new functions. A theory $\mathcal{T}$ is a *legal* theory if and only if it is obtained by a series of extensions from GZ using the extension principles. When we talk about a theory $\mathcal{T}$ in this dissertation, we always mean a legal theory.

The chief extension principles in ACL2 are (1) the *Definitional Principle* for introducing total functions, (2) the *Encapsulation Principle* for introducing constrained or partial functions, and (3) the *Defchoose Principle* for introducing Skolem functions. These three principles are used in *any* practical application of ACL2, and we make extensive use of them in modeling computing systems and their properties throughout this dissertation.

[4] ACL2 does allow addition of an arbitrary formula as an axiom. The use of this principle is discouraged because of the obvious risk of inconsistency. We do not treat the addition of arbitrary axioms as an extension principle, although we will make use of this facility in Chapter 17.

3.3.1 Definitional Principle

The *definitional principle* affords the extension of a theory by introducing a new total function. For instance, assume that one wants to extend GZ with a unary function symbol *mfact* that computes the factorial of its argument. It is possible to invoke the definitional principle to add such an axiom as follows.

$$\mathit{mfact}(x) = \mathit{if}(\mathit{zp}(x), 1, (x \times \mathit{mfact}(x-1)))$$

The axiom is called the *definitional axiom* (or simply *definition*) of *mfact*. For clarity, we will use a more familiar mathematical notation, namely:

$$\mathit{mfact}(x) \triangleq \begin{cases} 1 & \mathit{if}\ \mathit{zp}(x) \\ x \times \mathit{mfact}(x-1) & \text{otherwise} \end{cases}$$

Notice that we have used the symbol "$\triangleq$" in writing the axiom above, rather than "$=$". We write $f(x_1, \ldots, x_n) \triangleq \tau$ to mean the formula $f(x_1, \ldots, x_n) = \tau$ when we want to remind ourselves that the formula is an axiom that can be introduced in ACL2 by the extension principles.

In general, given a theory $\mathcal{T}$, the definitional principle is used to extend $\mathcal{T}$ by adding axioms of the form $f(x_1, \ldots, x_n) \triangleq \tau$. To ensure that the axiom does not make the resulting theory inconsistent, ACL2 checks that the purported axiom satisfies certain *admissibility requirements*. These are listed below.

- f is not a function symbol in $\mathcal{T}$,
- each x_i is a distinct variable symbol,
- $\nu(\tau) \subseteq \{x_1, \ldots, x_n\}$,
- τ is expressible in the theory $\mathcal{T}'$ obtained by extending $\mathcal{T}$ with the function symbol f of arity n and no axioms, and
- certain *measure conjectures* (described below) can be proved in $\mathcal{T}'$.

The measure conjectures are formulas, which, if proven as theorems, guarantee that a certain well-founded measure of the arguments decreases in each recursive call of f in τ. This guarantees that there is one unique function satisfying the definition [BKM95]. Assume that τ contains k recursive calls. Then there are $k+1$ measure conjectures. Let $\langle \textit{o-p}_{\prec}, \prec\rangle$ define a well-founded structure where $\textit{o-p}_{\prec}$ and "$\prec$" have been introduced in $\mathcal{T}$. Further, let m be a term expressible in $\mathcal{T}$ and $\nu(m) \subseteq \{x_1, \ldots, x_n\}$. The first measure conjecture is simply the following formula:

- $\textit{o-p}_{\prec}(m)$

The remaining k conjectures correspond to the k recursive calls. Let the i-th recursive call be of the form $f(\alpha_1, \ldots, \alpha_n)$, and let $\beta_1, \ldots, \beta_l$ be terms that represent the conditions under which the i-th recursive call is made. Let $\sigma_i \doteq [x_1 \rightarrow \alpha_1, \ldots, x_n \rightarrow \alpha_n]$. Then the i-th measure conjecture is given by:

- $\beta_1 \wedge \ldots \wedge \beta_l \Rightarrow m/\sigma \prec m$

In practice, the user often has to provide the measure term m and the well-founded structure to be used to prove the measure conjectures.

How does the definitional principle work with our example *mfact*? We will use the term $\textit{nfix}(x)$ for our measure term and the well-founded structure defined by $\langle \textit{natp}, <\rangle$. The following are the measure conjectures, which are easy to prove in GZ. Recall (Figure 3.1) that $\textit{nfix}(x)$ returns x if x is a natural number, else 0.

- $\textit{natp}(\textit{nfix}(x))$
- $\neg \textit{zp}(x) \Rightarrow \textit{nfix}(x-1) < \textit{nfix}(x)$

One should note that there is a connection between the measure conjectures proved in admitting a function definition, and the **Induction Rule**; both involve arguments based on well-foundedness. In particular, one proves theorems about recursive definitions using induction. For instance, consider proving that *mfact* always returns a natural number. This can be stated as:

- $\mathit{natp}(\mathit{mfact}(x))$

To prove this, we will use induction as follows:

Base Case: $\mathit{zp}(x) \Rightarrow \mathit{natp}(\mathit{mfact}(x))$

Induction Step: $(\neg \mathit{zp}(x) \wedge \mathit{natp}(\mathit{mfact}(x-1))) \Rightarrow \mathit{natp}(\mathit{mfact}(x))$

Both these obligations are trivial. But what is important for us is that the *justification* of the induction (as specified by conditions 1 and 2 of the **Induction Rule**) is exactly the same as the measure theorem for introducing $\mathit{mfact}(x)$. We will thus refer to this proof as *proof by induction based on* $\mathit{mfact}(x)$. Throughout this dissertation when we talk about proofs by induction we will present the term on which the induction is based.

We note that the functions introduced by the definitional principle are total. That is, the definitional axiom for a function f specifies the value of f on every input as long as every other function g referenced in the axiom is total. From the axioms defining *mfact* and *even* above, we can determine that the value of $\mathit{mfact}(6)$ is 120 and the value of $\mathit{even}(3)$ is `NIL`. Perhaps surprisingly, the axioms also specify that the value of $\mathit{mfact}(\mathtt{T})$ is 1, and the value of $\mathit{even}(1/2)$ is `T`.

The definitional principle can also introduce mutually recursive definitions. For example, we can define two functions *odd* and *even* as shown in Figure 3.3. To admit mutually recursive definitions, one must provide separate measure terms for each function, such that each measure is well-founded and decreases along each (mutually) recursive call.

We now have the terminology to talk about modeling some computing system as a formal theory extending GZ. Here is a trivial example. Consider a system that consists of a single component `s`, and at every instant it executes the following

$$\mathit{odd}(x) = \begin{cases} \texttt{NIL} & \text{if } \mathit{zp}(x) \\ \mathit{even}(x-1) & \text{otherwise} \end{cases}$$

$$\mathit{even}(x) = \begin{cases} \texttt{T} & \text{if } \mathit{zp}(x) \\ \mathit{odd}(x-1) & \text{otherwise} \end{cases}$$

Figure 3.3: Example of a Mutually Recursive Definition

instruction `s:=s+1`. We can talk about the executions of this program by defining a simple function *step*:

- $\mathit{step}(s) \triangleq s + 1$

The definition is not recursive and therefore does not need any measure theorem. Notice that we are modeling the execution of a program instruction by specifying what the effect of the instruction is on the state of the machine executing the program. The definition of *step* above can be read as: "At each step, the machine state is incremented by 1." This, as we discussed in the last chapter, is termed the *operational semantics* of the program. We saw there that specification of systems using model checking involves operational models. Now we see that using theorem proving we use operational models as well. The function *step* is often referred to as the *state transition function* of the machine.

We can now reason about this trivial system. For instance, we can prove that if the system is at some state s at a certain time then after two transitions it reaches the state $s + 2$. In the theory in which *step* has been defined, this can be represented as the formula:

- $\mathit{step}(\mathit{step}(s)) = s + 2$

The theorem is trivial. Nevertheless, we point out one aspect of its proof. To prove this theorem, we consider the left hand side of the equality, use the definition

of *step* to simplify *step*(*step*(s)) to *step*($s+1$), and do this again to simplify this term to $s+2$. This way of simplifying a term which involves a composition of the state transition function of a machine is reminiscent of executing (or *simulating*) the machine modeled by the function. Since this form of simulation involves variable symbols rather than constant inputs as in traditional simulation, we often refer to such simplification as *simplification by symbolic simulation*. An obvious but useful insight is that if a term involves a fixed number of compositions of the state transition function of a machine, then symbolic simulation can be used to simplify the term. We will use this insight fruitfully when we move on to more complicated systems and programs in the next part.

3.3.2 Encapsulation Principle

A function symbol introduced using the definitional principle is *total*. On the other hand, the *encapsulation principle* affords extension of the current theory by introducing new function symbols axiomatized only to satisfy certain desired properties, without specifying the return value of the function for every input. For example, using this principle, one can extend GZ introducing a unary function symbol *natural* such that the sole associated axiom is *natp*(*natural*(n)). That is, *natural*(n) is axiomatized only to return a natural number. An axiom introduced via the encapsulation principle is called a *constraint* and the function symbols introduced are referred to as a *constrained functions*. Given a theory $\mathcal{T}$, ACL2 stipulates that the introduction of a collection of function symbols $f_1, \ldots, f_k$ with axioms $\mathcal{C}_1, \ldots, \mathcal{C}_l$ is admissible via the encapsulation principle if and only if the following conditions are satisfied.

- The function symbols to be introduced are not in the current theory $\mathcal{T}$.
- It is possible to extend $\mathcal{T}$ with function symbols $f_{1w}, \ldots, f_{kw}$ so that the formulas $\mathcal{C}_{1w}, \ldots, \mathcal{C}_{lw}$ are theorems, where $\mathcal{C}_{iw}$ is obtained from $\mathcal{C}_i$ by replacing every occurrence

of the function symbols $f_1, \ldots, f_k$ respectively with $f_{1w}, \ldots, f_{kw}$.

The functions $f_{1w}, \ldots, f_{kw}$ are often referred to as *local witnesses*, and their existence guarantees that the extended theory after introducing the constrained functions is consistent. To introduce function symbols by encapsulation, the user must furnish such witnesses. For instance, to introduce *natural* above, one can provide as local witness the constant function that always returns 0.

Since the only properties prescribed by the encapsulation are the constraints, any theorem that is provable about a constrained function is also provable about another function that satisfies all the constraints. This is stipulated by a derived inference rule called *functional instantiation* as follows. Let Φ be a formula in some theory $\mathcal{T}$ which refers to the constrained function symbols $f_1, \ldots, f_k$ that have been introduced in $\mathcal{T}$ with constraints $\mathcal{C}_1, \ldots, \mathcal{C}_l$. Let $g_1, \ldots, g_k$ be functions in $\mathcal{T}$ such that $\mathcal{C}_{1g}, \ldots, \mathcal{C}_{lg}$ are theorems, where $\mathcal{C}_{ig}$ is obtained from $\mathcal{C}_i$ by consistently replacing $f_1, \ldots, f_k$ with $g_1, \ldots, g_k$. Then functional instantiation says that if Φ is provable in $\mathcal{T}$ then one can infer Φ_g, where Φ_g is obtained by replacing $f_1, \ldots, f_k$ in Φ with $g_1, \ldots, g_k$. For instance, since the constant function 10 satisfies the constraints associated with *natural*, if *foo*(*natural*(n)) is a theorem then so is *foo*(10). We call a constrained function symbol *uninterpreted* if there is no associated constraint. Given a formula Φ containing an uninterpreted function symbol f, functional instantiation thus allows us to infer the formula Φ_g that replaces occurrences of f with *any* other function g in Φ.

The encapsulation principle, along with functional instantiation as a rule of inference, provides a limited form of second order reasoning to ACL2 and enables us to circumvent some of the limitations in its expressive power.

3.3.3 Defchoose Principle

The final extension principle we consider here is the *defchoose principle* which allows the introduction of function symbols axiomatized using quantified formulas. The discussion on quantifiers in the context of ACL2 sometimes comes as a surprise even to some experienced ACL2 users. The reason for the surprise is something we have already mentioned, that is, ACL2 is a *quantifier-free* logic. Since the use of quantification is going to be important for much of our work, we take time here to understand the connection between quantification and ACL2.

Assume that we have a formula $\Phi(x, y)$ containing two free variables x and y, and we want to write a formula which is equivalent to what we would have written in first order logic as $\exists y : \Phi(x, y)$, that is, the formula is valid if and only if there exists a y such that $\Phi(x, y)$ is valid. To write such a formula, we first introduce a unary function symbol f with the only axiom being the following:

- $\Phi(x, y) \Rightarrow \Phi(x, f(x))$

Then the formula we seek is $\Phi(x, f(x))$. The formula is provable if and only if there exists a y such that $\Phi(x, y)$ is provable. Here, f is called the Skolem witness of the existential quantifier and the axiom above is called the *choice axiom* for Φ. Notice that the new incarnation of the formula, namely $\Phi(x, f(x))$ is quantifier-free in so far as its syntax goes.

The defchoose principle in ACL2 simply allows the extension of a theory $\mathcal{T}$ by such Skolem witnesses. The syntax of defchoose is a little complicated and we will not discuss it here. Instead, we will use the notation of quantified first order logic. That is, we will appeal to this principle to say that we extend $\mathcal{T}$ by introducing predicates such as p below:

$$p(x_1, \ldots, x_n) \triangleq \exists y : \Phi$$

Here, Φ must be a formula expressible in $\mathcal{T}$, and $\nu(\Phi)$ must be equal to the set $\{x_1, \ldots, x_n, y\}$. When the predicate p is admissible in the current theory, we will assume that the extended theory has introduced a new n-ary function symbol wit_p as the Skolem witness. Of course we can also introduce universal quantified predicates by exploiting the duality between existential and universal quantification. When we write $\forall x : \Phi$ we use it as an abbreviation for $\neg(\exists x : \neg\Phi)$.

3.4 Bibliographic Notes

ACL2 is an industrial-strength successor of the Boyer-Moore theorem prover Nqthm. Nqthm has been developed by Boyer and Moore and grew out of the Edinburgh Pure Lisp Theorem Prover. The Nqthm logic and theorem prover are described in several books [BM79, BM88a, BM97]. Kaufmann introduced interactive enhancements to Nqthm [BKM95]. Nqthm supported a home-grown dialect of pure Lisp. In contrast, ACL2 was designed to support Common Lisp. ACL2 has been developed by Kaufmann and Moore, with important early contributions from Boyer. Two books [KMM00b, KMM00a] have been written on the ACL2 theorem prover in addition to about 5MB of hypertext documentation available from the ACL2 home page [KMb]. Our description of the ACL2 logic is based on two foundational papers written by Kaufmann and Moore [KM97, KM01]. In addition, several papers describe the many heuristics and implementation details of the theorem prover [KM94, Moo01, BM02, HKM03], and its applications on real-world systems.

Well-founded induction is a part of folklore in mathematical logic. Ordinals form the basis of Cantor's set theory [Can95, Can97, Can52]. The general theory of ordinal notations was initiated by Church and Kleene [CK37], which is recounted

by Rogers [Rog87, § 11]. Both ACL2 and Nqthm have used ordinals to prove well-foundedness of recursive definitions and inductive proofs. The representation of ordinals currently used by ACL2 and described here, is popularly known as *Cantor Normal Form*. This representation was introduced in ACL2 by Manolios and Vroon [MV04a], replacing the older and less succinct one. Our illustration in Figure 3.2 was taken from the documentation on ordinal representation in the current version of ACL2 [KMa]. Manolios and Vroon also provide efficient algorithms for arithmetic on ordinals [MV03].

Both Nqthm and ACL2 have been used in constructing several non-trivial mechanically checked proofs. Using Nqthm, Shankar formalized and proved Gödel's Incompleteness Theorem [Sha94], Russinoff proved Gauss' Law of quadratic reciprocity [Rus92], and Kunen verified a version of the Ramsey's theorem [Kun95]. Notable among the computing systems formally verified by Nqthm is the so-called CLI stack [BHMY89] consisting of the gate-level design of a microprocessor together with the operational semantics of a machine-code instruction set architecture [Hun94], the operational semantics of a relocatable, stack-based machine language [Moo96], the operational semantics of two high-level languages [You88, Fla92], some application programs [Wil93], and an operating system [Bev87]. Another significant success with Nqthm is the verification by Yu [BY96] of 21 out of 22 subroutines of the Berkeley C string library. Indeed, the exacting demands of these large verification projects influenced the development of ACL2 as the industrial-scale successor of Nqthm. ACL2 has been used to prove correctness of several commercial microprocessor systems, such as the correspondence of the pipelined microcode of the Motorola CAP digital signal processor [BH99, BKM96] with its ISA, the IEEE compliance of the floating

point division algorithm of the AMD K5$^{\text{TM}}$ processor [MLK98] and RTL implementations of floating point operations of Athlon$^{\text{TM}}$ [Rus98, RF00] and Opteron$^{\text{TM}}$ processors, modeling and verification of the Rockwell Collins JEM1 processor which is the first Java Virtual Machine implemented in silicon [Gre98], and an analysis of the FIPS-140 level 4 certification of the IBM 4578 secure co-processor [SA98]. Currently, a detailed operational semantics of the Java Virtual Machine is being modeled with ACL2 [LM03], which includes about 700 pages of formal definitions, and several JVM bytecode programs have been proven correct for this model [LM04].

Part II

Sequential Program Verification

Chapter 4

Sequential Programs

In this part, we will study deterministic sequential programs and investigate how we can provide automation in reasoning about their correctness. In this chapter we will discuss models of sequential programs, formalize the statement of correctness that we want to prove, and present the standard deductive approaches to derive such a correctness statement. We will then discuss some deficiencies in the standard approaches. The issues we raise in this chapter will be expanded upon and addressed in Chapters 5 and 6.

4.1 Modeling Sequential Programs

We model sequential programs using operational semantics. This allows us to talk about program executions in the mathematical logic of ACL2. We have seen an example of an operational model already in page 53. We now discuss how we do that in general for sequential programs.

Recall from Chapter 2 that an operational model of a program is formalized

by defining how the machine executes the program and stipulates how the system transitions from one state to the next. For our purpose, the machine state is represented as a tuple of values of all machine variables (or *components*). Assume for simplicity that the program is represented by a list of instructions. Let $\mathit{pc}(s)$ and $\mathit{prog}(s)$ be two functions that, given a state s, respectively return the values of two special components of s, namely the program counter and the program being executed. These two functions fix the "next instruction" that is poised to be executed at state s, namely the instruction in $\mathit{prog}(s)$ that is pointed to by $\mathit{pc}(s)$; in terms of functions in GZ (appropriately augmented by defining functions pc and prog), this instruction is given by the function instr below:

$$\mathit{instr}(s) \triangleq \mathit{nth}(\mathit{pc}(s), \mathit{prog}(s))$$

To formalize the notion of state transition, let us first define a binary function effect. Given an instruction I and a state s, $\mathit{effect}(s, I)$ returns the state s' obtained by executing the instruction I from state s. For example, if I is the instruction `LOAD` then its effect might be to push the contents of some specific variable on the stack and increase the program counter by some specific amount.

We can now define our state transition function step as follows, such that for any state s, $\mathit{step}(s)$ returns the state of the machine after executing one instruction.

$$\mathit{step}(s) \triangleq \mathit{effect}(s, \mathit{instr}(s))$$

It should be noted that the representation of a program as a *list* of instructions and the representation of a machine state as a *list* of components in our description above is merely for the purpose of illustration. Different machines are formalized in different ways; for example, the states might be modeled as an array or association list instead of a list. In what follows, the actual formal representation of the machine states or the actual definition of step is mostly irrelevant. What we assume is that

there is *some* formal representation of the states in the formal theory, and given a state s, $\mathit{step}(s)$ can be interpreted to return the state of the machine after executing one instruction from s. This can always be done as long as we are concerned with reasoning about deterministic sequential programs.

It will be convenient for us to define a new function *run* as follows to return that state of the machine after n *step*s from s.

$$\mathit{run}(s, n) \triangleq \begin{cases} s & \mathit{if}\ \mathit{zp}(n) \\ \mathit{run}(\mathit{step}(s), n-1) & \text{otherwise} \end{cases}$$

Function *run* has some nice algebraic properties which we will make use of. For instance, the following is an important lemma that says that the state reached by running for $(m+n)$ *step*s from a state s is the same as running for m *step*s first, and then for n additional *step*s. This lemma is easy to prove by induction based on $\mathit{run}(s, n)$.

Lemma 1 $\mathit{natp}(m) \wedge \mathit{natp}(n) \Rightarrow \mathit{run}(s, m+n) = \mathit{run}(\mathit{run}(s, m), n)$

Terminating states are characterized by a special unary predicate *halting*, which is defined as:

$$\mathit{halting}(s) \triangleq (\mathit{step}(s) = s)$$

That is, the execution of the machine from a state s satisfying *halting* yields a no-op. Many assembly language programs provide an explicit instruction called `HALT` whose execution leaves the machine at the same state; using such languages, programs are written to terminate with the `HALT` instruction, to achieve this effect.

What do we want to prove about such programs? As we talked about in Chapter 2, there are two notions of correctness, namely *partial* and *total*. First, we will assume that two predicates *pre* and *post* have been defined so that *pre* holds

for the states satisfying the precondition of the program and *post* holds for states satisfying the postcondition of the program. For instance, for a sorting program *pre* might say that some machine variable l contains a list of numbers and *post* might say that some (possibly the same) machine variable contains a list l' that is an ordered permutation of l. The two notions of correctness are then formalized as below.

Partial Correctness

Partial correctness involves showing that if, starting from a *pre* state, the machine ever reaches a *halting* state, then *post* holds for that state. Nothing is claimed if the machine does not reach a *halting* state. This can be formalized by the following formula.

$$\mathit{pre}(s) \wedge \mathit{halting}(\mathit{run}(s,n)) \Rightarrow \mathit{post}(\mathit{run}(s,n))$$

Total Correctness

Total correctness involves showing, in addition to partial correctness, the *termination condition* which states that the machine, starting from a state satisfying the precondition, eventually halts. Termination can be formalized as follows.

$$\mathit{pre}(s) \Rightarrow (\exists n : \mathit{halting}(\mathit{run}(s,n)))$$

4.2 Proof Styles

Given an operational model of a program defined by *step*, and the predicates *pre* and *post*, we want to use theorem proving to prove the (total or partial) correctness theorems above. How do we go about actually doing it? There are two popular approaches, namely the use of *step invariants*,[1] and *clock functions*.

[1]Step invariants are also often referred to as *inductive invariants*. We do not use this term for sequential programs, since we want to reserve its use to apply to an analogous concept for reactive

4.2.1 Step Invariants

In the *step invariants* approach, one defines a new unary function inv so that the following three formulas can be proven as theorems:

I1: $\mathit{pre}(s) \Rightarrow \mathit{inv}(s)$

I2: $\mathit{inv}(s) \Rightarrow \mathit{inv}(\mathit{step}(s))$

I3: $\mathit{inv}(s) \wedge \mathit{halting}(s) \Rightarrow \mathit{post}(s)$

The function inv is often referred to as a *step invariant*. It is easy to construct a proof of partial correctness in a formal theory if one has proven **I1-I3** above as theorems. First we prove the following lemma.

Lemma 2 $\mathit{inv}(s) \Rightarrow \mathit{inv}(\mathit{run}(s, n))$

The lemma says that for every state s satisfying inv and for every n, $\mathit{run}(s, n)$ also satisfies inv. The lemma can be proved by induction based on the term $\mathit{run}(s, n)$. The proof of partial correctness then follows from **I1** and **I3**.

For total correctness, one defines, in addition to inv above, a unary function r (called the *ranking function*) so that the following two formulas are theorems (in addition to **I1-I3**):

I4: $\mathit{inv}(s) \Rightarrow \mathit{o\text{-}p}_{\prec}(r(s))$

I5: $\mathit{inv}(s) \wedge \neg \mathit{halting}(s) \Rightarrow r(\mathit{step}(s)) \prec r(s)$

Here, the structure defined by $\langle \mathit{o\text{-}p}_{\prec}, \prec \rangle$ is assumed to be well-founded. Total correctness can now be proved from these conditions. To do so, we need only to show how termination follows from **I1-I5**. Assume for contradiction that termination is

systems.

not valid, that is, the machine does not reach a *halting* state from some state $\hat{s}$ satisfying *pre*. By **I2**, each state in the sequence $\langle \hat{s}, \mathit{step}(\hat{s}), \mathit{step}(\mathit{step}(\hat{s})) \ldots \rangle$ satisfies *inv*. Since we assume that none of the states in this sequence satisfies *halting*, by **I5**, we now have an infinitely descending chain, namely $\langle \ldots \prec r(\mathit{step}(\mathit{step}(\hat{s}))) \prec r(\mathit{step}(\hat{s})) \prec r(\hat{s}) \rangle$. This contradicts the well-foundedness of $\langle \mathit{o\text{-}p}_{\prec}, \prec \rangle$.

4.2.2 Clock Functions

A direct approach to proving total correctness is the use of *clock functions*. Roughly, the idea is to define a function that maps every state s satisfying *pre*, to a natural number that specifies an upper bound on the number of *step*s required to reach a *halting* state from s. Formally, to prove total correctness, one defines a unary function *clock* so that the following formulas are theorems:

TC1: $\mathit{pre}(s) \Rightarrow \mathit{halting}(\mathit{run}(s, \mathit{clock}(s)))$

TC2: $\mathit{pre}(s) \Rightarrow \mathit{post}(\mathit{run}(s, \mathit{clock}(s)))$

Total correctness now follows from **TC1** and **TC2**. Termination is obvious, since, by **TC1**, for every state s satisfying *pre*, there exists an n, namely $\mathit{clock}(s)$, such that $\mathit{run}(s, n)$ is halting. To prove correctness, we need the following additional lemma that says that running from a *halting* state does not change the state.

Lemma 3 $\mathit{halting}(s) \Rightarrow \mathit{run}(s, n) = s$

Thus, the state $\mathit{run}(s, \mathit{clock}(s))$ uniquely specifies the *halting* state reachable from s. By **TC2**, this state also satisfies *post*, showing correctness.

For specifying partial correctness, one weakens **TC1** and **TC2** to **PC1** and **PC2** below, so that $\mathit{run}(s, \mathit{clock}(s))$ is required to satisfy *halting* and *post* only if a *halting* state is reachable from s.

PC1: $\mathit{pre}(s) \wedge \mathit{halting}(\mathit{run}(s,n)) \Rightarrow \mathit{halting}(\mathit{run}(s,\mathit{clock}(s)))$

PC2: $\mathit{pre}(s) \wedge \mathit{halting}(\mathit{run}(s,n)) \Rightarrow \mathit{post}(\mathit{run}(s,\mathit{clock}(s)))$

Partial correctness theorems follow from **PC1** and **PC2** exactly using the arguments above.

4.3 Discussions

Given our presentation above, it should be clear that the use of operational semantics and a mathematical logic allow us to specify a clear statement of correctness as a succinct formula in the logic. Contrast this with our description of axiomatic semantics and VCG in Chapter 2. There the correctness statement was specified as a predicate transformation rather than a state transformation and Hoare axioms were necessary to state it. Further, in general, the statement would be encoded inside the formula manipulation process of the VCG itself.

In spite of its clarity, it is generally believed that operational semantics are more difficult to use in terms of conducting the actual verification. The use of step invariants and clock functions constitute the basic theorem proving strategies for reasoning about sequential programs modeled using operational semantics. Let us try to understand the difficulty of applying each method using as illustration the simple one-loop program we showed in Figure 2.1 in Chapter 2 (page 28). Recall that the program consists of a single loop which is executed 10 times and the variable `X` is incremented by 1 in each iteration, while `Y` is decremented. It should be clear from the discussion above that the program can be easily modeled operationally. Assume that given a state s, functions $\mathit{pc}(s)$, $X(s)$, and $Y(s)$ return the values of the program counter, variable `X`, and variable `Y` respectively. Also assume that *prog-loaded* is a predicate that holds for a state s if and only if the program shown

$$
\textit{inv-aux}(s) \triangleq \begin{cases} \texttt{T} & \textit{if } \textit{pc}(s) = 1 \\ X(s) = 0 & \textit{if } \textit{pc}(s) = 2 \\ X(s) + Y(s) = 10 \wedge \textit{natp}(Y(s)) & \textit{if } \textit{pc}(s) = 3 \\ X(s) + Y(s) = 10 \wedge \textit{natp}(Y(s)) \wedge Y(s) > 0 & \textit{if } \textit{pc}(s) = 4 \\ X(s) + Y(s) = 11 \wedge \textit{natp}(Y(s)) \wedge Y(s) > 0 & \textit{if } \textit{pc}(s) = 5 \\ X(s) + Y(s) = 10 \wedge \textit{natp}(Y(s)) & \textit{if } \textit{pc}(s) = 6 \\ X(s) = 10 & \text{otherwise} \end{cases}
$$

$$
\textit{inv}(s) \triangleq \textit{inv-aux}(s) \wedge \textit{prog-loaded}(s)
$$

Figure 4.1: Step Invariant for the One-Loop Program

has been loaded in s starting from location 1. The precondition and postcondition for this program are given by the following functions:

$$
\textit{pre}(s) \triangleq (\textit{pc}(s) = 1) \wedge \textit{prog-loaded}(s)
$$

$$
\textit{post}(s) \triangleq (X(s) = 10)
$$

The predicate *prog-loaded* is a standard "frame condition" that we need to add as a conjunct in the precondition (and in any other assertions) to make sure that we limit ourselves to states of the machine in which the right program is being executed.

How do the two approaches cope with the task of proving the correctness of this simple program? A step invariant is shown in Figure 4.1. It is not important to understand the function in detail, but some aspects of the definition should be obvious immediately. First, the definition involves a collection of "assertions" attached to *every* value of the program counter (or, equivalently, every state reached by the machine during the execution). The key reason for this is the strong requirement imposed by the proof obligation **I2**, namely that if any state s satisfies *inv* then *step*(s) must do so too. Consider a state $\hat{s}$ so that *inv* asserts $A_{\textit{step}}(\textit{step}(\hat{s}))$. Then *inv*, to satisfy **I2**, must also assert $A(\hat{s})$ such that $A(\hat{s}) \Rightarrow A_{\textit{step}}(\textit{step}(\hat{s}))$. This requirement complicates the definition of a step invariant and forces one to think

$$\mathit{lpc}(s) \triangleq \begin{cases} 0 & \mathit{if}\ \mathit{zp}(Y(s)) \lor \neg\mathit{prog\text{-}loaded}(s) \\ 4 + \mathit{lpc}(\mathit{run}(s, 4)) & \text{otherwise} \end{cases}$$

$$\mathit{clock}(s) \triangleq 2 + \mathit{lpc}(s) + 1$$

Figure 4.2: Clock Function for the One-Loop Program

about the "right" assertion to be attached to every reachable state. On the other hand, once an appropriate invariant has been defined, proving **I2** is trivial. The proof does not require any inductive argument on the length of the execution and usually follows by successively proving for each value p of the program counter that the assertions attached to p imply the assertions attached to the program counter p' of the state obtained by executing the instruction associated with p.

Notice that we have only talked about the issues involved in defining *inv*. Of course, for total correctness, we also need a ranking function r. For our simple one-loop program, it is not difficult to define r so that we can prove **I1**-**I5**. We omit the definition here, but merely observe our comments regarding the complications involved in defining *inv* have exact parallels with the definition of r. In particular, **I5** forces us to attach a ranking function to every value of the program counter in order to show that the rank decreases (according to the well-founded relation we choose) at every step.

How does the clock function approach work? A clock function is shown for this program in Figure 4.2. A quick look at the definition should indicate that the definition closely mimics the actual loop structure of the program. In particular, consider the function *lpc* (which stands for "loop clock"). If the machine is in a state s where the program counter has the value 3, then $\mathit{lpc}(s)$ merely counts the number of steps before the loop is exited. Notice that no assertion is involved that characterizes the state reached by the program at every program counter value. It

$$
\begin{array}{l}
\quad \mathit{natp}(Y(s) \wedge \mathit{prog\text{-}loaded}(s) \wedge (\mathit{pc}(s) = 3) \\
\Rightarrow \\
\quad \mathit{run}(s, \mathit{lpc}(s)) = \mathit{upd}(s, X(s) + \mathit{lpc}(s), Y(s) - \mathit{lpc}(s))
\end{array}
$$

Figure 4.3: A Key Lemma for the One-loop Program

seems, therefore, that coming up with a clock function is significantly easier than coming up with a step invariant. However, the "devil" in this approach lies in the details. First, to admit the definition of *lpc* under the definitional principle, we must prove that the sequence of recursive calls terminate. To show this, we must be able to define a measure m so that following formulas are theorems.

- $\mathit{o\text{-}p}_{\prec}(m(s))$
- $\neg \mathit{zp}(Y(s)) \Rightarrow m(\mathit{run}(s, 4)) \prec m(s)$

But this can be achieved only if the loop itself terminates! Indeed, the function seems to precisely capture the time complexity of the program. Note that in this case the definition can be shown to be admissible by choosing the measure $m(s) \triangleq \mathit{nfix}(Y(s))$.

Once the appropriate clock function is defined, we can try to prove the total correctness theorem. To do so, we must prove a theorem that characterizes the loop itself. One possibility is shown in Figure 4.3. Here $\mathit{upd}(s, a, b)$ be the state obtained by assigning the value a to component `X` and the value b to component `Y` respectively in state s. The formula can be proven as a theorem by induction based on the term $\mathit{lpc}(s)$. **TC1** and **TC2** now follow from this theorem and lemma 1.

The discussion above suggests that step invariants and clock functions have orthogonal advantages and disadvantages as verification strategies. Step invariants involve attaching assertions (and ranking functions) to *all* reachable states in a way that every execution of the program must successively satisfy the assertions at

each *step*. Clock functions involve defining functions that *certain* states stipulate how many *step*s are remaining before the program terminates. Step invariants (and ranking functions) are more difficult to come up with, but once they are defined the proof obligations are simple, and in practice automatic. Clock functions are easier to define, but there are several non-trivial theorems that the user must prove, often resorting to induction. Depending on the nature of the program to be verified one method would be more natural or easier to use than the other. Nevertheless, it should be clear that both approaches involve considerable human expertise. We therefore need to figure out how we can derive the correctness theorems with more automation. To do this, we identify two key "deficiencies" of program verification using step invariants and clock functions, which we call the problem of *composition* and the problem of *over-specification*. We discuss how the extent of manual effort involved in the two approaches relate to these problems, and how we can attempt to solve them.

4.3.1 Composition

The correctness theorems we described characterize an entire program. In practice, we often want to verify a program component, for instance a subroutine. The proof styles (and indeed, the correctness statements) as we described above, are not suitable for application to individual program components. The disturbing element in the correctness characterization stems from the fact that postconditions are attached only to *halting* states. The definition of *halting* specifies program termination in a very strong sense; as lemma 3 shows, running from a *halting* state for *any* number of *step*s must leave the machine at the same state. This is an effective char-

acterization of a terminating program. The theory of computation and the Turing machine model are based on this view. But it does not provide a characterization of "exit from a program component". After exiting the execution of a subroutine, a program does not halt, but merely returns the control to the calling routine. We must generalize the correctness theorems (and proof styles) so that we can attach postconditions to the exitpoints of components like subroutines.

Suppose for the moment that we have actually achieved this generalization of the correctness theorems so that postconditions can be attached to exitpoints of components, and assume that the two proof styles are generalized to reason about subroutines in some way as well. (We will see such a way in the next chapter in Section 5.2.) A brief reflection suggests that we want more. Step invariants and clock functions are very different approaches, as we saw from the illustration of the simple example, and one or the other is probably more natural to come up with for a given component. We want to be able to verify individual subroutines using the style that is most natural for that subroutine, and then be able to *compose* the results to the correctness theorems for an entire program when necessary. As an example of a situation where this might be useful, assume that we have implemented a collection of subroutines to manipulate a Binary Search Tree (BST) data structure. Assume that two of the subroutines are (i) CREATE() that creates (initializes) an empty BST, and (ii) INSERT-N(els, B), that inserts a list els of n elements to an already existing BST B. We might want to prove for each of these subroutines that the resulting structure is indeed a Binary Search Tree. The verification of CREATE is probably easier to do using clock functions. Initialization routines often involve a constant number of *steps* (that is, are loop-free) and hence we can define the *clock*

by just counting the *step*s without resorting to induction; but coming up with a step invariant usually involves a very clear understanding of the operational details of each instruction. On the other hand, it is probably easier to use step invariants to verify INSERT-N, by proving that the Binary Search Tree structure is preserved after every single insertion of the elements of *els*. But given these two proofs, how do we formally tie them together into a proof of correctness of a program that first calls CREATE and then INSERT-N? We want to be able to reuse the proofs of components in some way, at the same time allowing the user the freedom to verify a component using the proof style that is best for the component alone.

This seems to be difficult to do in general, since the two proof styles are so different in appearance. As we remarked, one is about attaching assertions while the other is about attaching functions that count *step*s. Is it possible to view a clock function proof as a step invariant one and vice versa? Are the two styles really the same at some level, or fundamentally different? If they are really the same, for instance if it is possible to turn a step invariant proof of a component into a clock proof and vice versa, then we might have a uniform proof framework in which we can study how to compose the proofs of subroutines automatically.

In Chapter 5, we provide answers to these questions. We show that clock functions and step invariants are indeed equivalent despite appearances, in the sense that a proof in one style can be mechanically translated to a proof in the other. We will also show how each framework can be generalized to talk about proofs of program components, and discuss how we can compose the proofs of individual components to the proof of their composition.

4.3.2 Over-specification

Both the step invariant and clock function proofs of our simple program seem to require a substantial amount of "unnecessary" manual work. For instance, consider the step invariant proof. We needed to attach assertions to every value of the program counter. Contrast this situation with what is enjoyed by someone who uses axiomatic semantics and assertions. As we saw in Chapter 2, the user was required to attach assertions only to certain cutpoints which correspond to the entry and exit of the basic blocks of the program. The Hoare axioms could be used to derive (using a VCG) the verification conditions which implied the postcondition at the program exit. The clock functions approach suffers from a similar defect. We placed assertions only at "cutpoints" — the theorem above characterizing the loop can be thought of as specifying some kind of loop invariant — but we also had to define a *clock* and needed to show that the function terminates. The termination argument was required for *logical* reasons, namely to satisfy the definitional principle, and apparently forced us to prove termination of the loop even when only a partial correctness theorem was desired. Of course, we can get the automation of a VCG by simply implementing one and verifying it with respect to the operational semantics. But as we remarked before, that needs substantial manual effort and the verification needs to be done for every single programming language that we want to deal with.

We show how to circumvent this limitation in Chapter 6. We show that it is possible to get the correctness theorem as desired by the operational semantics, by attaching assertions to the user-chosen cutpoints, as one would do if one were using assertional reasoning. From these assertions the verification conditions can be generated by symbolic simulation of the operational model. Furthermore, by

proving certain theorems, we can make the theorem prover itself act as a symbolic simulator. Thus we can get both the clarity and conciseness of operational semantics and the benefits of automation provided by VCG without requiring the construction, let alone verification, of a VCG.

4.4 Summary

We have shown how sequential programs are modeled using operational semantics, what correctness theorems we want to prove about them, and how deductive methods are used to derive such theorems. We have also identified some deficiencies of the traditional verification approaches.

4.5 Bibliographic Notes

McCarthy [McC62] introduced operational semantics. The operational approach for reasoning about programs has since been extensively used in the theorem proving community. It has been particularly popular in the Boyer-Moore theorem provers, namely Nqthm and ACL2. All code proofs that we are aware of in Nqthm and ACL2, including the systems mentioned in the bibliographic notes of Chapter 3, have involved operational semantics. Liu and Moore's model of the JVM [LM03] is possibly one of the largest formalized operational models, with about 700 pages of formal definitions. Operational models are also popular among other theorem provers. Significant projects involving operational semantics in HOL include Norrish's formalization of the semantics of C [Nor98], and Strecker's models of Java and the JVM [Str02]. In PVS, operational semantics have been used to model UML

state charts [HR04].

Proofs of operational models have involved either step invariants and clock functions, or the modeling and verification of a VCG to discharge the assertional proof obligations. See the bibliographic notes for Chapters 5 and 6 for detailed references on such proofs.

Chapter 5

Mixing Step Invariant and Clock Function Proofs

In this chapter, we will see how we can mix step invariant and clock function proofs for sequential programs modeled using an operational semantics. In other words, we want to be able to use the step invariant approach for one procedure, and the clock functions approach for the other, and then somehow use these individual verification results to prove the correctness of a program that (say) sequentially calls the two procedures.

To do this, we will first show that the two proof styles are equivalent. That will let us think of a step invartiant proof as a clock proof and vice versa. We will then generalize the verification framework so that we can reason about program components, and finally, we will see how the equivalence lets us reuse the proofs of the different components to a proof of their composition.

5.1 Proof Style Equivalence

To prove that the two proof styles are equivalent, we will show how, given theorems that satisfy the proof obligations for one style, we can mechanically produce a collection of theorems that satisfy the obligations of the other style. The equivalence holds independently of the actual definitions of the operational semantics or the precondition and postcondition involved in the proof obligation. In ACL2, we achieve this via the encapsulation principle; that is, we model the "step invariant style" for total (resp., partial) correctness by encapsulated functions *step*, *pre*, *post*, *inv*, and r axiomatized to satisfy obligations **I1-I5** (resp., **I1-I3**). Similarly we model the "clock function style" by encapsulated functions *step*, *pre*, *post*, and *clock*, axiomatized to satisfy obligations **TC1** and **TC2** (resp., **PC1** and **PC2**). Given an encapsulated model of one proof style we will derive the proof obligations for the other.

5.1.1 Step Invariants to Clock Functions

Suppose we are given a step invariant proof of total correctness of an operational semantics, that is, we are given an operational semantics modeled as a function *step*, precondition and postcondition given by functions *pre* and *post*, and two functions *inv* and r so that the formulas **I1-I5** are theorems. How do we define a *clock* so that **TC1** and **TC2** are theorems? The following definition "works".

$$\mathit{clock}(s) \triangleq \begin{cases} s & \mathit{if\ halting}(s) \vee \neg \mathit{inv}(s) \\ 1 + \mathit{clock}(\mathit{step}(s)) & \text{otherwise} \end{cases}$$

The crucial observation is that the function *clock* is admissible via the definitional Principle given **I1-I5**. In particular, **I4** and **I5** serve as measure conjectures to show

that the sequence of recursive calls terminates. Given the definition of *clock*, we can now show that **TC1** and **TC2** are theorems. The following lemma comes in handy.

Lemma 4 $\mathit{inv}(s) \Rightarrow \mathit{halting}(\mathit{run}(s, \mathit{clock}(s)))$

Lemma 4 says that for any state s satisfying *inv*, $\mathit{run}(s, \mathit{clock}(s))$ must be a *halting* state; it can be proved by induction based on the function *clock*. **TC1** now follows from **I1**.

To prove **TC2**, note that from Lemma 2 (page 66) we know that if s satisfies *inv* then $\mathit{run}(s, n)$ satisfies *inv* too, for all n. In particular, therefore, $\mathit{run}(s, \mathit{clock}(s))$ must satisfy *inv*. Thus, by **I1** and lemma 4, if s satisfies *pre* then $\mathit{run}(s, \mathit{clock}(s))$ satisfies both *inv* and *halting*. **TC2** now follows from **I3**.

The argument above critically depends on our ability to define the function *clock*. This is easy to do in case of total correctness, where the well-foundedness arguments for the step invariant proof provides the measure arguments allowing us to define the function *clock* using the definitional principle. The situation is a bit subtle for partial correctness, where there may be no such measure. To define *clock* for partial correctness, we invoke the defchoose principle as follows:

$$h(s) \triangleq \exists n : \mathit{halting}(\mathit{run}(s, n))$$

$$\mathit{clock}(s) \triangleq \mathit{wit}_h(s)$$

Thus, given a state s, if there exists some n such that $\mathit{run}(s, n)$ is *halting*, then the state $\mathit{run}(s, \mathit{clock}(s))$ must be *halting* too. Notice that **PC1** requires that if *some* *halting* state is reachable from a state s satisfying *pre*, then $\mathit{run}(s, \mathit{clock}(s))$ must be a *halting* state; **PC1** therefore follows directly from this definition of *clock*. Finally, **PC2** can be derived by exactly the same argument as we used to derive **TC2** for total correctness.

5.1.2 Clock Functions to Step Invariants

To obtain a step invariant proof from clock functions, we define the function `inv` expressing the following property: A state s satisfies *inv* if and only if s is reachable from some state satisfying *pre*. We can express this property by the following quantified predicate:

$$\mathit{inv}(s) \triangleq \exists \langle s_0, n \rangle : (\mathit{pre}(s_0) \wedge (s = \mathit{run}(s_0, n)))$$

Given this definition of *inv*, **I1** and **I2** are trivial. Further, **PC2** (resp., **TC2**) guarantee that if a *halting* state is reachable from a state s satisfying *pre*, then there is at least one such *halting* state, namely the state $\mathit{run}(s, \mathit{clock}(s))$ must also satisfy *post*. However, if indeed a *halting* state is reachable from s, then by lemma 3 (page 67) the halting state must be unique. Thus every halting state reachable from s must be equal to $\mathit{run}(s, \mathit{clock}(s))$; **I3** is therefore valid.

For total correctness **I4** and **I5** require that we must also show that we can define a ranking function r. We define r by determining the number of *step*s to reach a *halting* state:

$$\mathit{r\text{-}aux}(s, i, c) \triangleq \begin{cases} \mathit{nfix}(i) & \mathit{if}\ \mathit{halting}(s) \vee \neg \mathit{natp}(i) \vee \neg \mathit{natp}(c) \vee (i > c) \\ \mathit{r\text{-}aux}(\mathit{step}(s), i+1, c) & \text{otherwise} \end{cases}$$

$$r(s) \triangleq \mathit{r\text{-}aux}(s, 0, \mathit{clock}(s))$$

The exact reason why we define r (and more specifically *r-aux*) this way has got to do with the fact that it must be admissible via the definitional principle. The measures we need to use for the admission is not germane to our discussion; suffice it to say that it is admissible. What is relevant to us here is that r returns a natural number, and does count the minimum number of *step*s from s to a *halting* state. Thus for any state s reachable from some *pre* state s_0, if s is not *halting*, then $r(\mathit{step}(s)) < r(s)$

(and in fact $r(\mathit{step}(s))$ is equal to $r(s) - 1$). Since $\langle \mathit{natp}, < \rangle$ defines a well-founded structure, it follows that **I4** and **I5** are valid for *inv* and r as defined.

5.2 Generalized Proof Styles

The equivalence shown in the previous section provides "mental satisfaction" that the step invariant and clock functions approach are equivalent. But they do not as yet provide a way of reasoning about different program components using different approaches. As we remarked in Chapter 4 (page 72), we need to somehow replace the function *halting* in the framework with the concept of returning from a subroutine.

First, let us assume that the precondition *pre* is defined so that if s satisfies *pre* then s must be poised to execute the procedure of interest. For example if the range of program counter values for the procedure is $\{\pi_0, \pi_1, \ldots, \pi_k\}$, with π_0 being the starting value, then we simply conjoin the term $(\mathit{pc}(s) = \pi_0)$ in the definition of $\mathit{pre}(s)$ so that $\mathit{pre}(s) \Rightarrow (\mathit{pc}(s) = \pi_0)$ is a theorem. We can also define a unary function *exit* so that $\mathit{exit}(s)$ returns `T` if and only if the program control has exited the procedure. Again this is easy to do, by just characterizing the program counter values. For instance in the above example of program counters, we can define *exit* as follows.

$$\mathit{exit}(s) \triangleq (\mathit{pc}(s) \notin \{\pi_0, \pi_1, \ldots, \pi_k\})$$

Here we assume that the function "$\notin$" has been appropriately defined to be interpreted as the negation of set membership.

Given the discussion above, a tempting suggestion might be to simply change the proof obligations for the two approaches by replacing the function *halting* with *exit*. Thus, for example, we will replace **TC2** with the following proof obligation:

$$\mathit{pre}(s) \Rightarrow \mathit{exit}(\mathit{run}(s, \mathit{clock}(s)))$$

A slight reflection will indicate however that this naive modification of proof obligations does not suffice. The problem is that the proof obligations do not require the entries and exits of a subroutine to match up. Consider a program consisting of procedures `A` and `B`, which alternately goes on calling `A` and `B` in succession. Consider reasoning about the correctness of `A`. We should then define *exit* to characterize the return of program control from `A` and attach the postcondition to the *exit* states. Unfortunately, given a *pre* state s, a number of (in this case infinitely many) *exit* states are reachable from s. Hence the proof obligations do not prevent us from specifying a *clock* that counts the number of *step*s from a *pre* state to the *second* *exit* state in the execution sequence of the program.

The observations above suggest that the proof obligations need to be slightly modified so that the postcondition is attached to the *first* *exit* state reachable from a *pre* state. This is easy for the clock functions approach. Here are the "generalized" proof obligations for total correctness.

GTC1: $\mathit{pre}(s) \Rightarrow \mathit{exit}(\mathit{run}(s, \mathit{clock}(s)))$

GTC2: $\mathit{pre}(s) \Rightarrow \mathit{post}(\mathit{run}(s, \mathit{clock}(s)))$

GTC3: $\mathit{natp}(\mathit{clock}(s))$

GTC4: $\mathit{natp}(n) \wedge \mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow (n \geq \mathit{clock}(s))$

Obligations **GTC1** and **GTC2** are merely rewording of **TC1** and **TC2** respectively using *exit* instead of *halting* states. **GTC3** and **GTC4** "make sure" that the function *clock* counts the minimal number of *step*s to the first *exit* state. As usual, for partial correctness we will weaken the obligations by the additional hypothesis $\mathit{exit}(\mathit{run}(s, n))$. The partial correctness obligations are shown below.

GPC1: $\mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{exit}(\mathit{run}(s, \mathit{clock}(s)))$

GPC2: $\mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{post}(\mathit{run}(s, \mathit{clock}(s)))$

GPC3: $\mathit{pre}(s) \Rightarrow \neg \mathit{exit}(s)$

GPC4: $\mathit{exit}(\mathit{run}(s.n)) \Rightarrow \mathit{natp}(\mathit{clock}(s))$

GPC5: $\mathit{natp}(n) \wedge \mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s,n)) \Rightarrow (n \geq \mathit{clock}(s))$

How do we generalize the step invariant approach to specify correctness of program components? The issues with step invariants (not surprisingly) involve the strength of obligation **I2**. Notice that **I2** requires us to come up with a function *inv* which "persists" along every *step* of the execution, irrespective of whether the control is in the program component of interest or not. We therefore weaken **I2** requiring it only to persist when the program control is in the procedure of interest, that is, until the first *exit* state is reached. The modified step invariant obligations are shown below. The obligations **GI1**-**GI3** are for partial correctness, and **GI4** and **GI5** are additionally required for total correctness.

GI1: $\mathit{pre}(s) \Rightarrow \mathit{inv}(s)$

GI2: $\mathit{inv}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{inv}(\mathit{step}(s))$

GI3: $\mathit{inv}(s) \wedge \mathit{exit}(s) \Rightarrow \mathit{post}(s)$

GI4: $\mathit{o\text{-}p}_{\prec}(r(s))$

GI5: $\mathit{inv}(s) \wedge \neg \mathit{exit}(s) \Rightarrow r(\mathit{step}(s)) \prec r(s)$

Are our proofs of equivalence generalizable too? The answer is "yes". The proof of equivalence for the generalized framework follows exactly the same reasoning as the original one, that is, we transform a step invariant proof to a clock function proof by defining a *clock* that counts the number of *step*s to the first *exit* state, and we transform clock functions proofs to step invariants by defining an *inv* that posits that s is reachable from some state satisfying *pre*. Of course the exact definitions are a little more subtle; for example $\mathit{inv}(s)$ must posit not only that s is reachable

from some state s_0 satisfying *pre* but also that there is no *exit* state in the path between s_0 and s. But the crux of the proof is exactly the same.

The skeptical reader might ask if the idea of attaching postconditions to the first *exit* state is general enough. In particular, can the framework be used to reason about recursive procedures? A recursive procedure `A` might call itself several times, and the returns occur in the opposite order of calls. That is, the last call of `A` returns first. Hence if *exit* merely stipulates that the values of the program counter is associated with the return of the procedure `A`, then by attaching the postcondition to the first exit states we are presumably matching the wrong calls!

The objection is indeed legitimate if *exit* were only stipulated to be a function of the program counter. In fact that is the reason why we allow *exit* to be a function of the entire machine state. Our framework imposes no restriction on the form of *exit* other than that it be an admissible function. With this generalization, we can easily reason about recursive and iterative procedures. We will encounter a recursive procedure and reason about it in Chapter 6. To match up the calls and returns of recursive procedures, we simply define *exit* so as to characterize not only the program counter values but also the stack of return addresses.

We have surveyed several proofs of operational semantics in ACL2, including proofs about the JVM model in ACL2 [Moo99b, Moo03c, LM03]. For all the non-trivial programs, the verification could be decomposed into proofs of components, and reasoning about each component could be accomplished by our framework.

5.3 Verifying Program Components

Given the generalized reasoning approach for program components, we now turn to the problem of verifying composition of components. For simplicity we here consider only sequential composition. That is, assume that we have proved total (partial) correctness of two components A and B, and consider a program that sequentially executes the two components. How do we mechanically produce a proof of correctness of the program? Note that other more non-trivial compositions can be built out of sequential compositions.

Since the step invariant and clock functions approaches are shown to be equivalent we can assume without loss of generality that the proof of each component has been carried out using the clock functions method. Let $\mathit{pre}_{\mathtt{A}}$, $\mathit{post}_{\mathtt{A}}$, $\mathit{exit}_{\mathtt{A}}$, and $\mathit{clock}_{\mathtt{A}}$ be the precondition, postcondition, exit point characterization, and clock function for the component A, and $\mathit{pre}_{\mathtt{B}}$, $\mathit{post}_{\mathtt{B}}$, $\mathit{exit}_{\mathtt{B}}$, and $\mathit{clock}_{\mathtt{B}}$ be the corresponding functions for component B in the clock function proof. We need the following two additional proof obligations that stipulates that the program executes B after executing A, and the $\mathit{pre}_{\mathtt{A}}$ states are not $\mathit{exit}_{\mathtt{B}}$ states.

Comp1: $\mathit{exit}_{\mathtt{A}}(s) \wedge \mathit{post}_{\mathtt{A}} \Rightarrow \mathit{pre}_{\mathtt{B}}(s)$

Comp2: $\mathit{pre}_{\mathtt{A}}(s) \Rightarrow \neg \mathit{exit}_{\mathtt{B}}(s)$

We now define the following *clock* to count the number of *step*s from a state satisfying $\mathit{pre}_{\mathtt{A}}$ to a state satisfying $\mathit{post}_{\mathtt{B}}$.

$$\mathit{clock}(s) \triangleq \mathit{clock}_{\mathtt{A}}(s) + \mathit{clock}_{\mathtt{B}}(\mathit{run}(s, \mathit{clock}_{\mathtt{A}}(s)))$$

We can now prove that running for $\mathit{clock}(s)$ number of *step*s from a state s satisfying $\mathit{pre}_{\mathtt{A}}$ will reach an $\mathit{exit}_{\mathtt{B}}$ state satisfying *post*. Using lemma 1 that we proved about composition of *run*s (page 64), we can prove the following theorems that characterize

properties of total correctness of the composition.

$$\mathit{pre}_{\mathtt{A}}(s) \Rightarrow \mathit{exit}_{\mathtt{B}}(\mathit{run}(s, \mathit{clock}(s)))$$

$$\mathit{pre}_{\mathtt{A}}(s) \Rightarrow \mathit{post}_{\mathtt{B}}(\mathit{run}(s, \mathit{clock}(s)))$$

Similar theorems can be proved for partial correctness if every component has been proven partially correct. The partial correctness theorems additionally contain hypotheses saying that an $\mathtt{exit}_{\mathtt{A}}$ state is reachable from s and an $\mathit{exit}_{\mathtt{B}}$ state is reachable from $\mathit{run}(s, \mathit{clock}_{\mathtt{A}}(s))$.

So have we achieved composition? The answer is “almost”. Recall that to prove the correctness of the composition, we must prove all the obligations stipulated in **GTC1-GTC4** (resp., **GPC1-GPC4**). The key problem is the “minimality obligation” stipulated as **GTC4** (resp., **GTC4**). The problems are simple but draconian. Assume that from some state s satisfying $\mathit{pre}_{\mathtt{A}}$ it is possible first to “jump” to a state s' satisfying $\mathit{exit}_{\mathtt{B}}$ and then to “jump back” to continue execution until some $\mathit{exit}_{\mathtt{A}}$ is reached. Presumably an $\mathit{exit}_{\mathtt{B}}$ state might not also be an $\mathit{exit}_{\mathtt{A}}$ state and thus the minimality obligation for clocks is violated. It should be clear that the postcondition cannot be asserted for s' even though we have proved that each of A and B is individually correct using total (resp., partial) correctness conditions.

There are several solutions out of this draconian possibility. The solution we employ is to “expand” the notion of *exit* states, so that any state p which executes a program instruction outside the component of interest satisfies *exit*. This is easy to achieve in practice by characterizing the *exit* states to stipulate that the program counter is outside the range of interest.[1] With this extension on the notion of *exit*

[1] The suggestion for this simple solution was given to the author by John Matthews in a private communication on February 21, 2005. The author acknowledges the contribution with gratitude.

states, we can compose proofs of program components mechanically by generating a "composite" clock.

5.4 Mechanically Switching Proof Styles

In mechanically deriving the equivalence of the different proof styles we have used the encapsulation principle to model each style. That is, we modeled a style by stipulating the existence of functions *pre*, *post*, etc. constrained to satisfy the obligations of that style, and proved that the obligations of the other style can be derived from these constraints. Because of the use of encapsulation, we can now functionally instantiate the derivation of the mechanical transformation to transform a step invariant proof of a concrete program fragment to a clock function proof and vice versa. Indeed, the mechanical transformation can be completely automated in ACL2 by macros.

We have not talked about macros or any other programming features in our overview of ACL2 in Chapter 3. We will do so briefly now. In Lisp and ACL2 (and in other programming languages), macros provide ways of transforming expressions. In ACL2, we can use macros that expand to functions definitions, formulas, theorems, etc. For instance we can write a macro that takes a function symbol f, a list of arguments $x_1 \ldots x_n$ and a term τ and generates a definition $f(x_1, \ldots, x_n) \triangleq \tau$ (as long as such a definition is admissible). Macros are suitable as shorthands to program different proof strategies in ACL2.

How do we use macros to automate the transformation of a step invariant proof to a clock function? Assume that the concrete operational semantics are given by the function step_c, and the precondition, postcondition, step invariant, exit

point characterization, and ranking function used in a step invariant proof of total correctness of the component are pre_c, post_c, inv_c, exit_c, and r_c respectively. Suppose we now want to generate a clock function proof of the same component mechanically. We can of course mechanically generate the function clock_c to be used as the clock function. We then must prove the clock function proof obligations **GTC1-GTC5**. Consider the proof of **GTC1** for this program. We must prove the formula:

$$\mathit{pre}_c(s) \Rightarrow \mathit{exit}_c(\mathit{run}_c(s, \mathit{clock}_c(s)))$$

To prove this, we will appeal to the generic version of this theorem by instantiating the generic functions *pre*, *post*, etc., with the concrete versions, namely pre_c, post_c, etc., respectively. To successfully carry out the functional instantiation, the constraints on the generic functions must be satisfied by the concrete ones. But these constraints are merely the obligations (in this case) for a step invariant proof of total correctness which have been already proven for the concrete ones!

Our macros, which are available with the ACL2 distribution, simply automate this functional instantiation. The actual implementations of the macros are more elaborate, for example to ensure that the proof of the concrete instantiation is always automatic, but the key approach is to generate concrete theorems from the constrained ones by functional instantiation of the generic proofs.

5.5 Summary and Comments

We have shown that the two proof styles, namely step invariants and clock functions, are logically equivalent in that proofs in one can be mechanically transformed to proofs in the other. We have also shown that the two styles can be generalized uniformly to reason about program components, and proofs of individual components

can be composed. The results are not mathematically deep; careful formalization of the question essentially leads to the answer. Nevertheless, the question has been asked informally too often, since it was (erroneously) believed that clock function proofs require reasoning about efficiency of the program in addition to correctness, while step invariant proofs do not. We have shown that such informal beliefs are flawed. In addition, our proofs and transformation tools free the user of theorem proving from strict adherence of a single style to prove each program component. As we discussed in Chapter 4, some style might be more natural or easier to apply for reasoning about a particular component, notwithstanding the logical equivalence of the methods. The user now can choose the style most suitable for the component of interest.

We should note that the proof styles are applicable to operational semantics alone. If programs are modeled via other semantics, for example axiomatic semantics, then the styles and hence the transformation tools are not useful. In Chapter 6, we will see how to mimic the assertion-based reasoning used in axiomatic semantics within the operational domain. In particular, we will see how we can derive clock function proofs based on assertional reasoning. But the presence of operational semantics is imperative for all the results, and indeed for being able to treat programs as formal objects inside a classical mathematical logic. It should also be noted that while we can compose proofs of individual components to automatically derive the proof of correctness of a complete program, as we compose a large number of component proofs the actual definition of the composite clock (or, as mechanically translated, the step invariant and ranking function) becomes more complicated. We are not usually concerned with the efficiency with which the clock function is eval-

uated; it generally serves only to establish that there exists a suitable number of steps and we only care about how to define the function within the logic and how to reason about it. But if the efficiency of evaluating clock functions becomes our concern then we will have to look for "simplification" of the clock function. For reference, Golden [private communication] uses certain theorem proving techniques to simplify such clocks.

There is one other related observation that is worth making. One might wonder, although the proof styles are equivalent, how much stronger are they than the actual correctness obligation we discussed in Chapter 4? In other words, are step invariants and clock functions just ways to prove a (maybe) stronger property than the correctness theorems we desired? Recent results by Kaufmann [private communication] show that this is not so. In fact Kaufmann shows that it is possible to mechanically generate a clock function (and hence a step invariant) from the correctness theorem as we formalized. Thus step invariants and clock functions are ways of merely seeing the correctness theorems in a different light, in a way that is more amenable to mechanical proof, and the equivalences we established can be seen as ways of formally composing the correctness theorems of different program components.

It is also important to note that our ability to prove the equivalence theorems and hence, the mechanical transformation tools, depends crucially on the use of quantification, that is, the defchoose principle. The defchoose principle, and more specifically, the use of quantification and the so-called "non-constructive" arguments are often overlooked while doing theorem proving, especially in a quantifier-free theorem prover like ACL2, where the emphasis is more on defining recursive functions

and proving theorems about them by induction. Indeed, it is the author's contention that the traditional misunderstanding of the expressiveness of clock function proofs in ACL2 and their relation with step invariants has been mainly due to the disinclination of most ACL2 users to think in terms of quantification. We think it is not possible to derive the equivalence between the two styles in a logic without first order quantifiers, even though proofs in both styles can be carried out in such a logic. While recursive functions and inductive proofs play to the strength of the theorem prover, we have found many situations in which thinking in non-constructive terms yields simple theorems and proof obligations. In this dissertation we will see several instances of the use of quantification, particularly in Part III when we reason about reactive systems.

5.6 Bibliographic Notes

Step invariants have been widely used in the formal verification community to reason about programs. Clock functions have been used relatively less, but is probably more common in reasoning about operational semantics. Since operational semantics are advocated by Boyer-Moore style theorem proving, clock functions have found applications in both Nqthm and ACL2. All the proofs in the verification of the different components of the CLI stack [Hun94, Bev87, You88, Moo96], Yu's proofs of Berkeley C-string library [Yu92, BY96], and proofs of JVM bytecodes [Moo99b, Moo03c, LM04] involve clock functions. Relatively few researchers outside this community have used clock functions, though there have been some proofs done with PVS [Wil97]. The reason is possibly that relatively few researchers outside the Boyer-Moore community have done proofs of large programs based on

operational semantics. Grievances and concerns were, however, typically expressed about clock functions in private communications and conference question-answer sessions, and there was a nagging feeling that the method was somehow different from step invariants. The absence of published analysis of clock functions and the presence of this "nagging feeling" have been confirmed by both a comprehensive literature search and discussion with authors of other theorem provers.

The results described here have been accomplished in collaboration with J Strother Moore, and a shorter account has been published in a previous paper [RM04]. The presentation here uses some of the text from that paper with permission from Moore.

Chapter 6

Operational Semantics and Assertional Reasoning

In the last chapter, we saw how to remove the deficiency of lack of compositionality in theorem proving approaches to reason about sequential programs modeled using operational semantics. The work in this chapter aims to remove the deficiency we identified, namely over-specification. In particular, we want to be able to derive both total and partial correctness proofs of operational semantics by requiring the user to attach no more assertions than what the user of assertional reasoning and axiomatic semantics would do. But we want to achieve this effect without implementing and verifying a VCG.

How do we do it? We again resort to the expressive power of theorem proving. Expressiveness afforded by quantification allowed us to mix different proof styles. Expressiveness afforded by our ability to admit tail-recursive partial functions will let us incorporate assertional reasoning. In particular, we will define a tail-recursive

function that counts the number of *step*s from one cutpoint to the next. We will show how we can prove certain theorems about this function which enable us to derive the proof obligations involved in assertional reasoning by symbolic simulation. Further, it will be possible to use a theorem prover itself for symbolic simulation with no necessity for any external VCG.

6.1 Cutpoints, Assertions, and VCG Guarantees

To describe how we achieve the above, we first need to understand how assertional reasoning actually works, and the relationship of the VCG guarantees with operational semantics. We will then understand how we can mimic the workings of a VCG via symbolic simulation of the operational model.

Recall from our illustration in Chapter 4 that a step invariant proof involves attaching assertions to all values of the program counter (or equivalently, with all states reachable by the machine while executing the program). In using assertions, instead, we attach assertions to certain specific states which are often referred to as *cutpoints*. The cutpoints are simply states which are poised to execute the entry and exit of the basic blocks of the program, such as entry points for loops and procedure calls and returns. Such states are typically characterized by the value of the program counter, although additional state components such as the configuration of the return address stack, etc., are relevant for recursive procedures. As we showed in Chapter 2, the cutpoints for our one-loop program in Figure 2.1 are given by the program counter values 1 (entry to the program), 3 (entry to the loop), and 7 (exit from the program).

Let us abstract the details of the specific cutpoints for a program and assume

that we can define a unary predicate *cutpoint* so that $\mathit{cutpoint}(s)$ returns `T` if s is a cutpoint and `NIL` otherwise. So for our example, the function will be defined as follows:

$$\mathit{cutpoint}(s) \triangleq (\mathit{pc}(s) = 1) \vee (\mathit{pc}(s) = 3) \vee (\mathit{pc}(s) = 7)$$

To apply assertional reasoning, we must now attach assertions to the cutpoints. The concept of attaching assertions can be formalized by a unary function *assertion*. Using this function we can succinctly represent the assertions for our simple program example as follows:

$$\mathit{assertion}(s) \triangleq \begin{cases} \mathit{prog\text{-}loaded}(s) & \textit{if } \mathit{pc}(s) = 1 \\ \mathit{prog\text{-}loaded}(s) \wedge (X(s) + Y(s) = 10) & \textit{if } \mathit{pc}(s) = 3 \\ X(s) = 10 & \text{otherwise} \end{cases}$$

Note that other than the addition of the predicate *prog-loaded*, the assertions are exactly the same as the ones we showed in Figure 2.1 for the axiomatic semantics.

Assume now that we have a formal model of an operational semantics given by a function *step*, and we have defined two functions *cutpoint* and *assertion* as above that allow us to attach assertions to cutpoints. To achieve assertional reasoning using symbolic simulation, we will define a function *csteps* that “counts” the number of *step*s from a state to its nearest following *cutpoint*.

$$\mathit{csteps\text{-}aux}(s, i) \triangleq \begin{cases} i & \textit{if } \mathit{cutpoint}(s) \\ \mathit{csteps\text{-}aux}(\mathit{step}(s), i + 1) & \text{otherwise} \end{cases}$$

$$\mathit{csteps}(s) \triangleq \mathit{csteps\text{-}aux}(s, 0)$$

Of course the crucial question is why the function *csteps-aux* would be admissible. To admit it under the definitional principle we would need to show that some measure decreases along the recursive call, which will be (as we discussed) equivalent to showing that the program terminates when initiated from all states, indeed even states that do not satisfy the precondition.

The answer comes from recent work by Manolios and Moore [MM03]. Briefly, what they show is that any function axiomatized by a tail-recursive equation is admissible in ACL2. That is, assume that we wish to extend a theory $\mathcal{T}$ by introducing an axiom as follows:

$$f(x) = \begin{cases} \alpha(x) & \textit{if } \beta(x) \\ f(\gamma(x)) & \text{otherwise} \end{cases}$$

Here α, β, and γ are assumed to be any terms expressible in $\mathcal{T}$. Then it is possible to find a witness for f using the defchoose principle, that satisfies the above axiom. Thus we can extend $\mathcal{T}$ with the above axiom.

As a consequence of this result, and with the observation that *csteps-aux* is tail-recursive, we can introduce this function. Notice that if s is a state such that no *cutpoint* is reachable from s, then the return value of *csteps*(s) is not specified by the defining axiom.

We can now formalize the concept of the "next *cutpoint* reachable from a state". To do this, we first fix a "dummy state" *dummy*() such that if there is a state which is not a *cutpoint*, then *dummy*() is not a cutpoint. This is easy to do using the choice principle as follows:

$$\textit{D-exists}() \triangleq \exists s : \neg \textit{cutpoint}(s)$$

$$\textit{dummy}() \triangleq \textit{wit}_{\textit{D-exists}}()$$

We can now define the function *nextc* below such that for any state s, *nextc*(s) returns the closest cutpoint following s.

$$\textit{nextc}(s) \triangleq \begin{cases} \textit{run}(s, \textit{csteps}(s)) & \textit{if cutpoint}(\textit{run}(s, \textit{csteps}(s))) \\ \textit{dummy}() & \text{otherwise} \end{cases}$$

Given functions *cutpoint*, *assertion*, and *nextc*, the formulas **AR1**-**AR4** below are formal renditions of the VCG guarantees. In particular, **AR4** stipulates that if

some $\mathit{cutpoint}$ s that is not an exit state satisfies $\mathit{assertion}$, then the next $\mathit{cutpoint}$ encountered in an execution from s must also satisfy $\mathit{assertion}$.

AR1: $\mathit{pre}(s) \Rightarrow \mathit{cutpoint}(s) \wedge \mathit{assertion}(s)$

AR2: $\mathit{exit}(s) \Rightarrow \mathit{cutpoint}(s)$

AR3: $\mathit{exit}(s) \wedge \mathit{assertion}(s) \Rightarrow \mathit{post}(s)$

AR4: $\mathit{cutpoint}(s) \wedge \mathit{assertion}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{assertion}(\mathit{nextc}(\mathit{step}(s)))$

Before proceeding further, let us reassure ourselves that we can prove partial correctness given **AR1-AR4**. We show partial correctness by defining a clock, and showing the proof obligations **GPC1-GPC4**. The following definition will do the job:

$$\mathit{exitsteps}(s,i) \triangleq \begin{cases} i & \text{if } \mathit{exit}(s) \\ \mathit{exitsteps}(\mathit{step}(s), i+1) & \text{otherwise} \end{cases}$$

$$\mathit{clock}(s) \triangleq \mathit{exitsteps}(s, 0)$$

Again, notice that by using tail-recursion, we have saved ourselves the requirement of proving termination of the recursive axiom. It should be clear from the definition of clock that if *some* exit state is reachable from a state s, then $\mathit{clock}(s)$ returns the first such exit state. We provide a proof sketch below for **GCP1-GPC4**, which is abridged and paraphrased from the mechanical derivation.

Proof sketch of **GPC1-GPC4***:* Note that if some exit state is reachable from s, then $\mathit{clock}(s)$ returns the number of steps from s to the first reachable exit state. Hence the proof of **GPC1**, **GCP3**, and **GPC4** are trivial. To prove **GPC2**, note that by **AR4**, if some $\mathit{cutpoint}$ s satisfies $\mathit{assertion}$, then every $\mathit{cutpoint}$ reachable from s satisfies $\mathit{assertion}$ up to (and including) the first exit state. Since by **AR1**,

a *pre* state satisfies *assertion*, it follows that the first *exit* state reachable from a *pre* state satisfies *assertion*. **GPC2** now follows from **AR3**. ∎

The proof above shows that if indeed we can define functions *cutpoint* and *assertion* so that **AR1-AR4** are theorems, then the partial correctness follows. How about total correctness? To obtain total correctness theorems we must also specify a ranking function. Recall from our discussion of step invariants that we attached invariants and ranking functions to all values of the program counter (or equivalently, all reachable states). In assertional reasoning we are attempting to gain advantage over the step invariant approach by attaching whatever we need to attach only to *cutpoint*s. With this view, *assertion*s take the position of the step invariants. When designing a ranking function r, we will only require it to decrease (according to some well-founded relation) along cutpoints. The obligations **AR5**, **AR6**, and **AR7** below are the formalizations of termination condition in assertional reasoning.

AR5: $\mathit{o\text{-}p}_{\prec}(r(s))$

AR6: $\mathit{cutpoint}(s) \wedge \mathit{assertion}(s) \wedge \neg \mathit{exit}(s) \Rightarrow r(\mathit{nextc}(\mathit{step}(s))) \prec r(s)$

AR7: $\mathit{cutpoint}(s) \wedge \mathit{assertion}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{cutpoint}(\mathit{nextc}(\mathit{step}(s)))$

AR5 stipulates that "$\prec$" is well-founded, and **AR6** stipulates that the rank decreases along consecutive *cutpoint*s as long as an *exit* state is not reached. The two conditions therefore together attach well-founded ranks to the *cutpoint*s. The interesting new obligation is **AR7**, which says that if s is a *cutpoint* and not an *exit* state, then *nextc*(*step*(s)) is actually a *cutpoint*. Why do we need this obligation? Consider a hypothetical system that has only one *pre* state s and no *cutpoint* is reachable from *next*(s). Given our definitions, *nextc*(*step*(s)) must return *dummy*() which is not a *cutpoint*. (If *dummy*() were a *cutpoint* then by definition of *dummy*() every state must be a *cutpoint*, including *step*(s).) However, suppose *dummy*() just

happens to satisfy *assertion*, some *exit* state e is reachable from *dummy*(), and the ranking function decreases from s to *dummy*() and along all *cutpoint* states from *dummy*() to p. In that case our hypothetical system satisfies **AR1**-**AR6**, although the system should not satisfy total correctness since no *exit* state is reachable from the only *pre* state s. Notice that this obligation is not necessary for partial correctness since partial correctness provides non-trivial guarantees only for *pre* states from which some *exit* state is reachable.

The above discussion merely shows that coming up with a collection of proof obligations for assertional reasoning based on operational semantics can be subtle, and it is imperative to have a formal proof system to check that the proof obligations indeed do their job. Here is an abridged proof sketch paraphrased from the mechanized version showing that **AR1**-**AR6** do imply **GTC1**-**GTC4**.

Proof sketch of **GTC1**-**GTC4***:* Since we have proven partial correctness, it suffices to show that for every state s satisfying *pre*, there exists an *exit* state reachable from s. By **AR1** and **AR7**, we can show that for every non-*exit cutpoint* p reachable from s, there is a subsequently reachable *cutpoint* p'. But, by well-foundedness, **AR5**, and **AR6** eventually one of these cutpoints must be an *exit* state. ∎

6.2 VCG Guarantees and Symbolic Simulation

We have formalized the notion of VCG guarantees inside an operational semantics via proof rules. But the reader might wonder how we will actually prove the obligations **AR1**-**AR4** (resp., **AR1**-**AR6**) without a VCG with any automation, given an operational semantics. In this section we show how we achieve this. First we prove the following two theorems, which we will call *symbolic simulation rules*.

SSR1: $\neg \mathit{cutpoint}(s) \Rightarrow \mathit{nextc}(s) = \mathit{nextc}(\mathit{step}(s))$

SSR2: $\mathit{cutpoint}(s) \Rightarrow \mathit{nextc}(s) = s$

The theorems above are trivial by the definition of *nextc*. Let us now think about them as rewrite rules with the equalities oriented from left to right. What happens? Suppose we encounter the term $\mathit{nextc}(p)$ where p is a state such that $\mathit{step}(\mathit{step}(p))$ is the closest *cutpoint* reachable from p. Assuming that for every state s we encounter we can ascertain whether s is a *cutpoint* or not, then **SSR1** and **SSR2** will let us simplify $\mathit{nextc}(p)$ to $\mathit{step}(\mathit{step}(p))$. Given any state p from which some *cutpoint* q is reachable in a constant number of *step*s (and q is the nearest *cutpoint* from p), **SSR1** and **SSR2** thus let us simplify $\mathit{nextc}(p)$ to q. Since this is done by expansion of the state transition function of the operational model, by our discussion on page 55 it is simply symbolic simulation. Notice now that the proof obligations for assertional reasoning that involved crawling over the program (or equivalently, mentioned *step* in our formalization of the VCG guarantees), namely **AR4**, **AR6**, and **AR7**, all involve application of the function *nextc* on some argument! The discussion above therefore shows that if whenever the proof obligations are required to be discharged for any state p such that some *cutpoint* is reachable from p in a constant number of *step*s, then **SSR1** and **SSR2** can be used to prove these obligations by simplifying *nextc*. The rules in fact do just the "operational semantics equivalent" of crawling over the program, namely, repeated expansion and simplification of the *step* function. The rules thus exactly mimic the VCG reasoning without requiring the implementation of a VCG.

It should be noted that the symbolic simulation rules can do the job with a caveat: for every *cutpoint* p, the next *cutpoint* must be reachable in a constant

number of *step*s; this condition is satisfied if the start and end of all basic blocks in the program are classified as *cutpoints*. Otherwise simplification based on the above rules will possibly fall into an infinite loop. This is exactly the behavior that is expected in the corresponding situation by the user of a VCG.

Note that the proofs of **GPC1**-**GPC4** (resp., **GTC1**-**TGC4**) and the symbolic simulation rules **SSR1** and **SSR2** do not require the actual definitions of *step*, *pre*, *post*, *cutpoint*, and *exit*, other than the obligations **AR1**-**AR4** (resp., **AR1**-**AR7**). We carry out the mechanical derivation using the encapsulation principle.[1] That is, we constrain functions *step*, *pre*, *post*, etc., to satisfy **AR1**-**AR4** (resp., **AR1**-**AR7**), and show that the correctness theorems can be derived from the constraints. As we saw in the previous chapter, use of encapsulation affords the possibility of creation of macros that functionally instantiate the generic proofs for concrete system descriptions. We have created such a macro in ACL2, which will be distributed with the ACL2 distribution. The macro constructs partial (resp., total) correctness proofs of operationally modeled sequential programs using the following recipe:

1. Mechanically generate the functions *csteps*, *nextc*,*clock*, etc., for the concrete model and functionally instantiate theorems **SSR1** and **SSR2** for the generated functions.
2. Use the symbolic simulation rules and prove concrete versions of **AR1**-**AR4** (resp., **AR1**-**AR7**).
3. Functionally instantiate the generic derivation of correctness theorems **GPC1**-**GPC4** (resp., **GTC1**-**GTC4**) to complete the correctness proof.

[1]Although we describe the proofs in this dissertation in terms of the ACL2 logic, most of our techniques are portable to other theorem provers. In particular, the derivation of assertional reasoning methods for operational semantics has been formalized by John Matthews in the Isabelle theorem prover.

```
100  pushsi 1     *start*
102  dup
103  dup
104  pop 20                      fib0 := 1;
106  pop 21                      fib1 := 1;
108  sub                         n := max(n-1,0);
109  dup          *loop*
110  jumpz 127                   if n == 0, goto *done*;
112  pushs 20
113  dup
115  pushs 21
117  add
118  pop 20                      fib0 := fib0 + fib1;
120  pop 21                      fib1 := fib0 (old value);
122  pushsi 1
124  sub                         n := max(n-1,0);
125  jump 109                    goto *loop*;
127  pushs 20     *done*
129  add                         return fib0 + n;
130  halt         *halt*
```

Figure 6.1: TINY Assembly Code Computing Fibonacci

6.3 Examples

We now apply the method outlined above to verify two illustrative programs on two different machine models. The actual operational details of the machines are irrelevant to our discussion, and the detailed definition of the different functions specifying the semantics of the instructions in the models are omitted for brevity. Instead, we describe the actions of the instructions in high-level pseudo-code as comments. We chose these machines merely since their operations had been previously formalized in ACL2 and the formal models were accessible to us.

6.3.1 An Iterative Program: Fibonacci on the TINY Machine

Consider the iterative assembly language program shown in Figure 6.1 to generate

the k-th Fibonacci number. The k-th Fibonacci number is defined recursively as follows:

$$\mathit{fib}(k) \triangleq \begin{cases} 1 & \mathit{if}\ \mathit{zp}(k) \vee (k = 1) \\ \mathit{fib}(k-1) + \mathit{fib}(k-2) & \text{otherwise} \end{cases}$$

Our program runs on TINY [GWH00], a stack-based operational machine model with 32-bit word size. TINY has been developed at Rockwell Collins as an example of an analyzable, high-speed simulator. The program is a compilation of the standard iterative implementation to compute the Fibonacci sequence. In Figure 6.1, the program counter values for the loaded program are shown to the left of each instruction, and pseudo-code for the high-level operations is shown at the extreme right of the corresponding rows. The two most recently computed values of the Fibonacci sequence are stored in memory addresses `20` and `21`, and the loop counter `n` is maintained on the stack. Each loop iteration puts the sum of these numbers in address `20`, and moves the old value of `20` to `21`. Since TINY performs 32-bit integer arithmetic, given a number k the program computes the low-order 32 bits of $\mathit{fib}(k)$. We designate the program counters for the cutpoints by symbols `*start*`, `*loop*`, `*done*`, and `*halt*` for clarity. These values correspond to states in which the program is poised to execute initiation, loop test, loop exit, and program termination respectively. The functions *pre*, *post*, and *exit* for this model are as follows.

$$\begin{aligned} \mathit{pre}(k,s) &\triangleq (\mathit{pc}(s) = *\mathtt{start}*) \wedge (\mathit{tos}(s) = k) \wedge \mathit{natp}(k) \wedge \mathit{fib\text{-}loaded}(s) \\ \mathit{post}(k,s) &\triangleq (\mathit{tos}(s) = \mathit{fix}(\mathit{fib}(k))) \\ \mathit{exit}(s) &\triangleq (\mathit{pc}(s) = *\mathtt{halt}*) \end{aligned}$$

Here $\mathit{tos}(s)$ returns the top of stack at state s, and $\mathit{fix}(n)$ returns the low-order 32 bits of n. A state s satisfies *fib-loaded* if the program in Figure 6.1 is loaded in the memory starting at location `*start*`. Predicates *pre* and *post* specify the

Program Counter	Assertions
`*start*`	$(\mathit{tos}(s) = k) \wedge \mathit{natp}(k) \wedge \mathit{fib\text{-}loaded}(s)$
`*loop*`	$(\mathit{nth}(20, \mathit{mem}(s)) = \mathit{fix}(\mathit{fib}(k - \mathit{tos}(s)))) \wedge (\mathit{tos}(s) \leq k) \wedge$ $(\mathit{nth}(21, \mathit{mem}(s)) = \mathit{fix}(\mathit{fib}(k - \mathit{tos}(s) - 1))) \wedge \mathit{fib\text{-}loaded}(s)$ $\wedge \mathit{natp}(\mathit{tos}(s)) \wedge \mathit{natp}(k)$
`*done*`	$(\mathit{nth}(20, \mathit{mem}(s)) = \mathit{fix}(\mathit{fib}(k))) \wedge (\mathit{tos}(s) = 0) \wedge \mathit{fib\text{-}loaded}(s)$
`*halt*`	$(\mathit{tos}(s) = \mathit{fix}(\mathit{fib}(k)))$

Figure 6.2: Assertions for the Fibonacci Program on TINY

classical correctness conditions for a Fibonacci program: *pre* specifies that a 32-bit non-negative integer k is at the top of stack at program initiation, and *post* specifies that upon termination, $\mathit{fix}(\mathit{fib}(k))$ is at the top of the stack.

The assertions associated with the different cutpoints are shown in Figure 6.2. The user of assertional reasoning will find them fairly traditional. The key assertion is the loop invariant which specifies that the two most recently computed numbers stored at addresses `20` and `21` are $\mathit{fix}(\mathit{fib}(k - \mathtt{n}))$ and $\mathit{fix}(\mathit{fib}(k - \mathtt{n} - 1))$ respectively, where `n` is the loop count stored at the top of the stack when the control reaches the loop test.

For total correctness, we also need to specify a ranking function. The ranking function r we use maps the cutpoints to the set of ordinals below ϵ_0. Note that for this program it is possible to specify a ranking function that maps cutpoints to natural numbers; ordinals are used merely for illustration. Function r is defined as:

$$r(s) \triangleq \begin{cases} 0 & \mathit{if\ exit}(s) \\ (\omega \cdot_o \mathit{tos}(s)) +_o |\mathtt{*halt*} - \mathit{pc}(s)| & \text{otherwise} \end{cases}$$

Here $\cdot_o$ and $+_o$ are functions that axiomatize the ordinal multiplication and addition operations respectively. Informally, r can be viewed as a lexicographic ordering of the loop count and the difference between the location `*halt*` and $\mathit{pc}(s)$.

In verifying this program we provided only the functions *pre*, *post*, *exit*, r,

```
class Factorial {
  public static int fact (int n) {
   if (n > 0) return n*fact(n-1);
   else return 1;
  }
}
```

Figure 6.3: Java Program for Computing Factorial

and *assertion* (the last one encoded as a collection of cases based on Figure 6.2) other than of course libraries of previously proved theorems to reason about functions involved in modeling TINY. Symbolic simulation based on our approach does the rest, generating exactly the verification conditions as required by assertional reasoning. For instance, one of the verification conditions is the following:

$$(\mathit{nth}(20, \mathit{mem}(s)) = \mathit{fix}(\mathit{fib}(k))) \wedge (\mathit{tos}(s) = 0) \wedge \mathit{fib\text{-}loaded}(s) \Rightarrow \mathit{tos}(\mathit{push}(20, s)) = \mathit{fix}(\mathit{fib}(k))$$

The condition stipulates that if *assertion* holds at `*done*` then it also holds at `*halt*`. Any VCG for TINY, if implemented, will produce a corresponding condition. We achieve the same effect by symbolic simulation of the semantics.

6.3.2 A Recursive Program: Factorial on the JVM

We now briefly sketch the application of our method to verify JVM byte-codes for the Java factorial method `fact` shown in Figure 6.3. The machine model we use is M5 [Moo03c], which has been developed at the University of Texas to formally reason about JVM byte-codes. M5 provides operational semantics for a significant fragment of the JVM in ACL2. It specifies 138 byte-codes, and supports features like invocation of static, special, and virtual methods, inheritance rules for method resolution, multithreading, and synchronization via monitors. The byte-codes for `fact`, shown in Figure 6.4, are produced by disassembling output of `javac`, and can

```
Method int fact (int)
0  ILOAD_0                                      *start*
1  IFLE 12                                              if (n<=0) goto *done*
4  ILOAD_0
5  ILOAD_0
6  ICONST_1
7  ISUB
8  INVOKESTATIC #4 <Method int fact (int)>              x:= fact(n-1)
11 IMUL                                                 x:= n*x
12 IRETURN                                      *ret*   return x
13 ICONST_1                                     *done*
14 IRETURN                                      *base*  return 1
```

Figure 6.4: M5 Byte-code for the Factorial Method

be loaded on to M5.

The factorial method is recursive. For recursive methods, the characterization of cutpoints must take into account not only the program counter but also the "depth" of recursive invocations. On the JVM, an invocation involves recording the return address in the current call frame of the executing thread and pushing a new call frame with the invoked method on the call stack. The formulas showing the precondition, postcondition, assertions etc., for the `fact` method are large; we content ourselves here with the English description below.

- The precondition specifies that some thread `th` is poised to execute the instructions of the `fact` method invoked with some 32-bit integer argument n, the call stack of `th` has height h, and the `pc` is at location labeled `*start*`.
- A state is a *cutpoint* if either **(i)** the call stack of `th` has height less than h (that is, the initial invocation has been completed), or **(ii)** the `pc` is in one of the locations labeled `*start*`, `*ret*`, or `*base*` (that is, the program is about to initiate execution of, or return from, the current invocation).
- A state s is an *exit* state if the call stack of `th` has height less than h.

- The postcondition specifies that $\mathit{fix}(\mathit{mfact}(n))$ is on the top of the operand stack of `th`, where *mfact* is the standard recursive definition of factorial we showed in page 51.
- Let the height of the call stack for `th` at some cutpoint be h'. If $h' < h$, the assertions specify merely that the postcondition holds. Otherwise, let $i := (n - h' + h)$. We assert that **(i)** the top i frames in the call stack correspond to successive invocations of `fact`, **(ii)** the return addresses are properly recorded on all the frames (other than the frame being executed), and **(iii)** if the `pc` is at location `*ret*` or `*base*` (that is, poised to return from the current frame), then $\mathit{fix}(\mathit{mfact}(i))$ is about to be returned.

The height of the call stack merely tracks the "recursion depth" of the execution. Further, since the assertions involve characterization of the different call frames in the call stack of the executing thread, one might be inclined to think that specification of assertions for recursive programs require careful consideration of the operational details of the JVM. In fact, that is not the case. The key insight is to recognize that the next instruction on a method invocation is *not* the following instruction on the byte stream of the caller but the first instruction of the callee. The instruction following the invocation is executed immediately after return from the callee. Hence one only needs to ensure that **(i)** the assertions at invocation can determine the execution of the callee, and **(ii)** the assertions at return can determine the subsequent execution of the caller. However, for a recursive program the caller and the callee are "copies" of the same method, and so the assertions must be a symmetric characterization of all call frames invoked in the recursive call. For the factorial program, the characterization is merely that in the i-th recursive call to `fact`, the system computes $\mathit{fix}(\mathit{mfact}(i))$, and this value is returned from the callee to the caller on return.

For termination, our ranking function maps each cutpoint s to an ordinal

representing the lexicographic pair consisting of the invocation argument for the current call frame and the height of the call stack at s. Note that along the successive recursive invocations of `fact`, the argument of the recursive calls decreases, while along the successive returns, the depth of the call stack decreases.

6.4 Comparison with Related Approaches

Assertional reasoning has been used widely with theorem proving. There have been a number of substantial projects focused on implementing a verified VCG to obtain the fruits of assertional reasoning with operational semantics. This has been applied, for instance, in the work of Mehta [MN03] to reason about pointers in Isabelle, and Norrish [Nor98] to reason about C programs in HOL. The bibliographic notes in Section 6.6 list some of these approaches.

Two recent research results in ACL2 that are most related to our approach are the work of Moore [Moo03b] for reasoning about partial correctness of operational models, and the work of Matthews and Vroon [MV04b] for reasoning about termination. To the knowledge of the author, these are the only previous approaches which incorporate assertional reasoning in general-purpose theorem proving, without requiring a verified VCG. Indeed, the research reported here is a culmination of the joint desires of the author and the authors of these methods to devise a uniform framework for reasoning about operational semantics using assertions. In this section, we therefore look carefully at these previous approaches, and understand the difference between them and the work presented here.

Moore's method is geared towards deriving a partial correctness proof. He makes use of tail-recursion to define a step invariant *inv*. In our terminology, the

definition can be stated as follows:

$$\mathit{inv}(s) \triangleq \begin{cases} \mathit{assertion}(s) & \mathit{if\ cutpoint}(s) \\ \mathit{inv}(\mathit{step}(s)) & \text{otherwise} \end{cases}$$

The method then attempts to prove that *inv* is a step invariant. However, instead of defining separate symbolic simulation rules as we did, it uses the definition of *inv* itself. How does this work? Consider a *cutpoint* s that satisfies *assertion*. By the definition of *inv* above, s must satisfy *inv*. If s is not an *exit* state then to prove the step invariance of *inv* we must prove $\mathit{inv}(\mathit{step}(s))$. If $\mathit{step}(s)$ is not a *cutpoint*, then the definition of *inv* merely says that $\mathit{inv}(\mathit{step}(s)) = \mathit{inv}(\mathit{step}(\mathit{step}(s)))$. Applying the definition repeatedly will lead us simply to a term containing repeated composition of *step*s until we reach a *cutpoint*. Say $\mathit{step}(\mathit{step}(s))$ is a *cutpoint*. Then the expansion above leads us to the following proof obligation:

$$\mathit{cutpoint}(s) \wedge \mathit{assertion}(s) \Rightarrow \mathit{assertion}(\mathit{step}(\mathit{step}(s)))$$

Notice that this is simply our obligation **AR4** for this case. The approach can be used to prove partial correctness, when (as in our case and in any other situation related to VCG based reasoning) for any *cutpoint* s, the next subsequent cutpoint s' is a constant number of *step*s away. However, this approach is not applicable for proving total correctness. Why? Notice that the symbolic simulation rules in this approach are "merely" by-products for showing that *inv* is a step invariant. In particular, there is no symbolic simulation rule to determine the value of the ranking function at s', given the ranking function at s. We overcome this limitation by constructing rules that compute s' directly.

The method of Matthews and Vroon [MV04b] specifies symbolic simulation rules similar to ours, based on tail-recursive clock functions, which are then used in termination proofs. Our work differs in the formalization of assertions and cutpoints

in the reasoning framework. Matthews and Vroon define a single function *at-cutpoint* to characterize the cutpoints together with their assertions. This is applicable for termination proofs but cannot be used for proving partial correctness. The problem is that conflating assertions with cutpoints causes function *nextc* to "skip past" cutpoints that do not satisfy their corresponding assertion, on their way to one that does. However, one of those skipped cutpoints could be an *exit* state, and so the postcondition can not be inferred. Thus partial correctness becomes unprovable. Characterization of the cutpoints must be disentangled from the assertions in order to verify partial and total correctness in a unified framework.

6.5 Summary

We have presented an approach based on symbolic simulation to incorporate assertional reasoning to prove correctness of sequential programs modeled using operational semantics. No VCG is required, and manual construction of a step invariant or clock function is not necessary. Instead the user annotates certain cutpoints of the program with assertions. Symbolic simulation is used then to derive the verification conditions as in fact a VCG would do, but directly from the operational model. Our method thus achieves the clarity and concreteness of operational semantics together with the ease of verification as afforded by assertional methods.

It is possible to think, as some researchers have done, that our symbolic simulation rules **SSR1** and **SSR2** *define* a VCG. But if so, then it differs from a traditional VCG in several ways. First, it is based on states rather than formulas. Second, it is trivial compared to practical VCGs since it is intended to leverage on the formal definition of the operational semantics inside the logic. By generating the

verification conditions and discharging them on a case-by-case basis using symbolic simulation, we provide a means of incorporating the VCG reasoning directly inside the formal proof system without requiring any extra-logical tool.

6.6 Bibliographic Notes

The notion of assertions was made explicit in a classic paper by Floyd [Flo67], although the idea of attaching assertions to program points appears much earlier, for example in the work of Goldstein and von Neumann [GvN61], and Turing [Tur49]. Program logics were introduced by Hoare [Hoa69] to provide a formal basis for assertional reasoning. Dijkstra [Dij75] introduced weakest preconditions and guarded commands for the same purpose. King [Kin69] wrote the first mechanized VCG. Implementations of VCGs abound in the program verification literature. Some of the recent substantial projects involving complicated VCG constructions include ESC/Java [DLNS98], proof carrying code [Nec98], SLAM [BR01], etc. The VCGs constructed in all these projects have been built as extra-logical programs which generated verification conditions. The reliability of the VCG implementation is usually assumed, in addition to the soundness of the theorem prover used in discharging the proof obligations.

In theorem proving, VCGs have been defined and verified to render the assertional reasoning formal in the logic of the theorem prover. Gloess [Glo99] uses a verified VCG to derive the correctness of code proofs of an imperative language in PVS. Homeier and Martin [HM95] verify VCGs for an imperative language in HOL. Assertional methods have been applied, using a verified VCG, to reason about pointers in Isabelle [MN03], and C programs in HOL [Nor98].

Moore [Moo03b] uses symbolic simulation to achieve the same effect as a VCG inside an operational semantics in the context of partial correctness proofs. Matthews and Vroon [MV04b] present a related approach to reason about termination. A comparison of the method presented here with these two approaches appears in Section 6.4.

The work reported in this chapter has been done in collaboration with John Matthews, J Strother Moore, and Daron Vroon. The description in this chapter has been adapted by the author from a previous write-up on this work [MMRV05], with permission from the other co-authors.

Part III

Verification of Reactive Systems

Chapter 7

Reactive Systems

In the last part, we developed a generic verification methodology for reasoning about sequential programs. Our framework succeeded in marrying the clarity and conciseness of operational semantics with the automation afforded by assertional reasoning in a single unified deductive framework for reasoning about such systems. But sequential programs form only a small fragment of critical computing systems. Many interesting systems, such as operating systems, microprocessors, banking systems, traffic controllers, etc., are what are called *reactive systems*. In this part, we develop verification methodologies for reasoning about such systems.

To develop such methodologies, we will do exactly what we did for sequential programs. Namely, we will formalize the correctness statement we want to use, and then define and prove several proof rules to facilitate the verification of systems. In case of sequential programs, our formalization of correctness was the characterization of final (*halting* or *exit*) states of the machine in terms of postcondition, and the proof rules involved codification of the VCG process, proof rules allowing

transformation of different styles, and composition. That statement is inadequate for reactive systems. We will formalize a different correctness statement for reactive systems based on refinements, and we will derive a deductive recipe for proving such correctness statement.

Why do we need a different correctness statement for reactive systems than sequential programs? Sequential programs are characterized by *finite* executions. Executions start from some initial (or *pre*) state, and after a sequence of steps, the system reaches a final state. The postcondition is asserted only if a final state is reached. This view of computation forms the basis of recursive function theory. In contrast, reactive systems are characterized by non-terminating or infinite executions. Executions start from some initial state and then continue for ever, while possibly receiving stimulus from some external environment. We cannot characterize the system by properties of its *halting* states since there is no *halting* state. Hence a verification methodology for reasoning about such systems must account for behaviors of infinite computations.

One way to talk about executions of reactive systems is to relate them with the executions of a "high-level model" called the *specification system* ($\mathcal{S}$ for short). To distinguish $\mathcal{S}$ from the system we are interested in verifying, let us call the latter the *implementation system* ($\mathcal{I}$ for short). The (infinite) executions of $\mathcal{S}$ define the desirable (infinite) executions of $\mathcal{I}$. Verification then entails a formal proof showing that every execution of $\mathcal{I}$ can be appropriately viewed as some execution of $\mathcal{S}$.

The above approach requires that we define some notion of correspondence that can be used to relate the execution of $\mathcal{I}$ with those of $\mathcal{S}$. The notion must be such that it allows for specifications capturing the user's intuitive idea of the desired

behavior of the implementation. What kind of notion is appropriate? Our choice, in effect, can be informally described as: "For every execution of $\mathcal{I}$ there exists an execution of $\mathcal{S}$ that has the same visible behavior up to finite stuttering." This notion is well-studied and is referred to as *stuttering trace containment* [Lam83b, AL91]. We will see how we can formalize this notion effectively in ACL2, and how it affords definition of intuitive specifications.

7.1 Modeling Reactive Systems

We will model a reactive system M by three functions, namely $M.\mathit{init}$, $M.\mathit{next}$, and $M.\mathit{label}$, which can be interpreted as follows.

- $M.\mathit{init}$ is a 0-ary function that returns the initial state of M.
- Given a state s and an input i, $M.\mathit{next}(s, i)$ returns the next state of M.
- For any state s, $M.\mathit{label}(s)$ returns the observation (or the visible component) of the system in state s.

Notice that there is one major difference between our models of reactive systems and those of sequential programs. Our next state function *step* in sequential program models are unary, that is, a function of only the current state. This was appropriate since most sequential programs (and certainly all that we reasoned about) are *deterministic*. However, non-determinism is one of the intrinsic features of a reactive system. The second argument i of $M.\mathit{next}$ represents both the external stimulus, as well as the non-deterministic choice of the system. One way to think of it is to view the external environment as "selecting" which of the possible next states the system should transit to.

The model above is closely related to the Kripke Structures that we discussed briefly in Chapter 2. Indeed, representing systems by a tuple of "initial state", "transition" and "labels", is taken from the Kripke Structure formalisms. There are two key differences between our models and Kripke Structures. First, we model the state transition as a *function* instead of a *relation*. Secondly, we do not provide an explicit characterization of the set of states of the system, that is, a state can be any object in the ACL2 universe. Nevertheless, it is easy to see that the two formalizations are equivalent. For instance, given a state transition function *next*, a transition relation can be defined as:

$$R(p, q) \triangleq (\exists i : (\mathit{next}(p, i) = q))$$

As in the case of sequential programs, we will want to talk about the state of the system after n transitions. Since the execution of a reactive system depends on the stimulus it receives from the environment, the state reached after n transitions depends on the sequence of stimuli the system receives on the way. Let *stimulus* be a unary function such that $\mathit{stimulus}(k)$ may be interpreted to be the stimulus received by a system M at time k. We can then define the function $M.\mathit{exec}[\mathit{stimulus}]$ below (the reactive counterpart of the function *run* we defined for sequential programs) which returns the state reached by the system M after n transitions given the input sequence specified by *stimulus*.

$$M.\mathit{exec}[\mathit{stimulus}](n) \triangleq \begin{cases} M.\mathit{init}() & \mathit{if}\ \mathit{zp}(n) \\ M.\mathit{next}(M.\mathit{exec}[stimulus](n-1), \mathit{stimulus}(n-1)) & \text{otherwise} \end{cases}$$

Notice that we have chosen a weird name for the function; in particular, one that has the name of another function in square brackets. We follow this convention to remind ourselves that the function $M.\mathit{exec}[\mathit{stimulus}]$ depends on the definition of the function *stimulus*. In this sense, the above definition should be considered to be a "definition schema". Given two different functions $\mathit{stimulus}_1$ and $\mathit{stimulus}_2$ defining two different

input stimuli sequences, the above scheme provides two different definitions, namely $M.\mathit{exec}[\mathit{stimulus}_1]$ and $M.\mathit{exec}[\mathit{stimulus}_2]$. If the logic were more expressive, that is, admitted the use of functions as parameters of other functions, then it would have been possible to provide one single definitional equation by defining $M.\mathit{exec}(\mathit{env}, n)$ by a closed-form axiom. But the logic of ACL2 is first order, and hence functions cannot be used as arguments of other functions.

There is one other useful (but trivial) observation which we will make use of to formalize the notion of correctness for reactive systems. This observation deals with the relation between infinite sequences and functions. Recall that the correctness of reactive systems must be described in terms of infinite executions, that is, infinite sequences of states. Let $\pi \doteq \langle \pi^0, \pi^1, \ldots \rangle$ be an infinite sequence. Then we can model π as a function f_π over natural numbers such that $f_\pi(i)$ returns π^i. Conversely, any function f over natural numbers can be thought of as an infinite sequence π with π^i being $f(i)$. With this view, we can think of the function *stimulus* as an infinite sequence of inputs, and the the function $M.\mathit{exec}[\mathit{stimulus}]$ will be thought of as specifying an infinite execution of system M, namely the execution that occurs if *stimulus* is the sequence of inputs presented to the system.

7.2 Stuttering Trace Containment

We will now formalize the notion of stuttering trace containment. For simplicity, let us for the moment disregard stuttering and attempt to simply formalize trace containment. Informally, we want to say that a system $\mathcal{S}$ is related to $\mathcal{I}$ by trace containment if and only if for every execution σ of $\mathcal{I}$ there exists an execution π of $\mathcal{S}$ with the same (infinite) sequence of *label*s. Notice that the notion requires

quantification over infinite sequences since it talks about "for all executions of $\mathcal{I}$" and "there exists an execution of $\mathcal{S}$", and as we saw above, an execution is a function over naturals. To formally capture the idea of quantification over functions, we will make use of the encapsulation principle. Let *ustim* be an uninterpreted unary function representing an arbitrary input stimulus sequence. Thus we can think of $M.\mathit{exec}[\mathit{ustim}]$ as specifying *some arbitrary* execution of system M. Then the formal rendition of the statement above is as follows. We will say that $\mathcal{S}$ is a trace containment of $\mathcal{I}$ if and only if there exists some unary function *stim* such that the following is a theorem.

TC: $\mathcal{I}.\mathit{label}(\mathcal{I}.\mathit{exec}[\mathit{ustim}](n)) = \mathcal{S}.\mathit{label}(\mathcal{S}.\mathit{exec}[\mathit{stim}](n))$

Once the user has defined the appropriate function $\mathcal{S}.\mathit{stimulus}$, the characterization above, then, reduces to a first order obligation which can be proved with ACL2.

The characterization did not talk about stuttering. We now rectify this. To do this, we will think of the external environment as providing, in addition to the *stimulus*, a new sequence that controls stuttering. More precisely, a unary function *ctr* will be called a *stuttering controller* if it always returns an ordinal:[1]

E1: $\mathit{o\text{-}p}(\mathit{ctr}(n))$

Given functions *ctr* and *stimulus*, we now formalize the notion of a stuttering trace by defining the function $M.\mathit{trace}[\mathit{stimulus}, \mathit{ctr}]$ in Figure 7.1. By this definition, a stuttering trace of M is simply an execution of M in which some states are repeated a finite number of times. We now see why we needed the condition **E1**. This condition, together with the definition of $\mathit{stutter}[ctr]$ guarantees that stuttering is

[1]Throughout this chapter, we will make use of well-foundedness to formalize several arguments. We use the well-founded structure of ordinals as a basis for our formalization, since it is the only well-founded set axiomatized in ACL2. However, one can replace the ordinals with other well-founded structures to obtain the same overall results.

$$\mathit{stutter}[ctr](n) \triangleq (\mathit{ctr}(n) \prec_o \mathit{ctr}(n-1))$$

$$M.\mathit{trace}[\mathit{stimulus}, \mathit{ctri}](n) \triangleq \begin{cases} M.\mathit{init}() & \mathit{if}\ \mathit{zp}(n) \\ M.\mathit{trace}[\mathit{stimulus}, \mathit{ctr}](n-1) & \mathit{if}\ \mathit{stutter}[\mathit{ctr}](n) \\ M.\mathit{next}(M.\mathit{trace}[\mathit{stimulus}, \mathit{ctr}](n-1), \\ \quad \mathit{stimulus}(n-1)) & \text{otherwise} \end{cases}$$

Figure 7.1: Definition of a Stuttering Trace

finite. We insist that the stuttering be finite since we want a stuttering trace to reflect both the safety and progress properties of the corresponding execution. We will return to the significance of finiteness of stuttering in Section 7.4.

We now formalize the notion of stuttering trace containment. Let *tstim* and *tctr* be functions such that (1) *tstim* is uninterpreted, and (2) *tctr* is constrained to be a stuttering controller. We will say that $\mathcal{I}$ is a *stuttering refinement* of $\mathcal{S}$ if and only if there are unary functions *stim* and *ctr* such that *ctr* is a stuttering controller and the following condition is satisfied.

STC: $\mathcal{I}.\mathit{label}(\mathcal{I}.\mathit{trace}[\mathit{tstim}, \mathit{tctr}](n)) = \mathcal{S}.\mathit{label}(\mathcal{S}.\mathit{trace}[\mathit{stim}, \mathit{ctr}](n))$

We write $(\mathcal{S} \rhd \mathcal{I})$ to mean that $\mathcal{I}$ is a stuttering refinement of $\mathcal{S}$. We will also refer to the system $\mathcal{S}$ as a *stuttering abstraction* of $\mathcal{I}$. For a given implementation system $\mathcal{I}$ and a specification system $\mathcal{S}$, our notion of correctness, now, is merely to show $(\mathcal{S} \rhd \mathcal{I})$.

7.3 Fairness Constraints

As a final point in our formalization of notions of correctness, we will consider the issue of *fairness* and discuss how fairness constraints can be integrated effectively

with stuttering trace containment.

Why do we need fairness? Note that the notion of stuttering trace containment stipulates that for *every* *trace* of $\mathcal{I}$ there exists a *trace* of $\mathcal{S}$ with the same *label*s up to finite stuttering. However, there are situations, arising particularly in asynchronous distributed systems, when we are not interested in *every* *trace* (or computation path) of $\mathcal{I}$ but only in certain *fair* *trace*s. For instance, consider a multiprocess system in which processes request resources from an arbiter. To reason about such a system one is usually interested in only those executions in which, for instance, the arbiter is eventually scheduled to make the decision about granting a requested resource. But this means that we must constrain the executions of the implementation which we are interested in.

As should be clear from the definition schema for *trace* in Figure 7.1, different *trace*s correspond to different *stimulus* sequences. To consider only fair traces, we will constrain the *stimulus* to satisfy certain "fairness requirements". Informally, we would like the environments to select stimuli in such a way that every candidate stimulus is selected infinitely often. Naively, we might want to state this requirement by constraining *stimulus* as follows:

$$\mathit{natp}(n) \Rightarrow (\exists m : \mathit{natp}(m) \wedge (m > n) \wedge \neg \mathit{stutter}[\mathit{ctr}](m) \wedge (\mathit{stimulus}(m) = i))$$

The obligation specifies that for any time n and any input i, there is a "future time" m when i is selected by the *stimulus*. Notice that the condition $\neg \mathit{stutter}[ctr](m)$ is required to make sure that the system actually "uses" the *stimulus* at time m and does not simply bypass it via stuttering. Nevertheless, one can show that it is not possible to define any function *stimulus* to satisfy the above requirement. Why? Notice that we have put no constraints on what the candidate input i can be. Thus, it can be any object in the ACL2 universe. The universe, however, is not closed.

That is, there is no axiom in GZ that says that the universe consists of only the five types of objects we talked about in Chapter 3, that is, numbers, strings, characters, etc. Hence there are models of the ACL2 universe which contain an uncountable number of objects. Thus according to the obligation, we must be able to select for any n, any member of this uncountable set within a finite time after n.

To alleviate this problem, we restrict the legal inputs to be only one of the "good objects", that is, one of the five types of objects we discussed in Chapter 3.[2] This can be done easily. We define a predicate *good-object* such that it holds for x if and only if x is one of the recognized objects. Notice that although it is a restriction on the inputs, it is not a restriction in the practical sense since we are possibly not interested in the behavior of our systems on inputs which are not *good-object*s. Thus, we will say that a pair of functions *stimulus* and *ctr* is a *fair* input selector if and only if the following constraints are satisfied (in addition to **E1** above):

E2: $\mathit{good\text{-}object}(\mathit{stimulus}(n))$

E3: $\mathit{natp}(n) \wedge \mathit{good\text{-}object}(i) \Rightarrow$

$$(\exists m : \mathit{natp}(m) \wedge (m > n) \wedge \neg \mathit{stutter}[\mathit{ctr}](m) \wedge (\mathit{stimulus}(m) = i))$$

Is it possible to define fair input selectors? The affirmative answer comes from Sumners [Sum03]. To do so, Sumners first shows that there is a mapping between *good-objects* and natural numbers. That is, he defines functions *n-to-g* and *g-to-n* be two functions such that the following are theorems.

1. $\mathit{natp}(\mathit{g\text{-}to\text{-}n}(x))$
2. $\mathit{good\text{-}object}(\mathit{n\text{-}to\text{-}g}(x))$

[2] The authors of ACL2 are considering extending GZ with an axiom positing the enumerability of the ACL2 universe, that is, the existence of a bijection between all ACL2 objects and the natural numbers. The restriction of the legal inputs to only constitute good objects for the purpose of formalizing fairness will not be necessary once such an axiom is added.

3. $\mathit{good\text{-}object}(x) \Rightarrow \mathit{n\text{-}to\text{-}g}(\mathit{g\text{-}to\text{-}n}(x)) = x$

Sumners then first defines a input function *natenv* which can be interpreted as a fair selector of natural numbers. That is, the following is a theorem:

$$\mathit{natp}(n) \wedge \mathit{natp}(i) \Rightarrow (\exists m : \mathit{natp}(m) \wedge (m > n) \wedge (\mathit{natenv}(m) = i))$$

This means that for all natural numbers i and n if $\mathit{natenv}(n)$ is not equal to i, then there exists a natural number $m > n$ such that $\mathit{natenv}(n)$ is equal to i. The function *natenv* is defined using a state machine. At any instant, the state machine has a fixed upper bound, and it counts down from the upper bound at every step. When the count-down reaches 0, the upper bound is incremented by 1 and the counter reset to the new upper bound. Since any natural number i becomes eventually less than this ever-increasing upper bound, each natural number must be eventually selected in a finite number of steps from every instant n.

Defining a fair input selector for all *good-object*s is now easy. First, we define a function $\mathit{nostutter}[\mathit{ctr}]$ that selects the "non-stuttering" points from the stuttering controller.

$$\mathit{ns\text{-}aux}[\mathit{ctr}](n, l) \triangleq \begin{cases} 0 & \textit{if } \mathit{zp}(n) \wedge \neg \mathit{stutter}[\mathit{ctr}](l) \\ 1 + \mathit{ns\text{-}aux}[\mathit{ctr}](n, l+1) & \textit{if } \mathit{stutter}[\mathit{ctr}](l) \\ 1 + \mathit{ns\text{-}aux}[\mathit{ctr}](n-1, l+1) & \text{otherwise} \end{cases}$$

$$\mathit{nostutter}[\mathit{ctr}](n) \triangleq \mathit{ns\text{-}aux}[\mathit{ctr}](n, 0)$$

Notice that the condition **E1** guarantees that the recursion in the definitional equation of *ns-aux* above is well-founded, and hence it is admissible. Let us define a function *dummy*() constrained to return a *good-object*. We now define the fair stimulus below:

$$\mathit{e\text{-}ns}[\mathit{ctr}](n) \triangleq (\exists k : \mathit{nostutter}(k) = n)$$

$$\mathit{fstimulus}[\mathit{ctr}](n) \triangleq \begin{cases} \mathit{n\text{-}to\text{-}g}(\mathit{wit}_{\mathit{e\text{-}ns}[\mathit{ctr}]}(n)) & \mathit{if}\ \mathit{e\text{-}ns}[\mathit{ctr}](n) \\ \mathit{dummy}() & \text{otherwise} \end{cases}$$

The function $\mathit{fstimulus}[\mathit{ctr}]$ returns *good-object*s at the different non-stuttering points, and *dummy*() otherwise. It is possible, though not trivial, to prove that this function does indeed satisfy **E1-E3**.

Given that there exists at least one fair selector, how do we modify the notion of stuttering refinement to incorporate fairness? Let *cfstim* and *cfctr* be unary functions constrained to satisfy **E1-E3**. That is, we think of *cfsim* as a constrained fair input stimulus, and *cfctr* as the corresponding stuttering selector. We then say that $\mathcal{I}$ is a stuttering refinement of $\mathcal{S}$ *under fairness assumption* (written $(\mathcal{S} \rhd_{\mathcal{F}} \mathcal{I})$) if one can define a (not necessarily fair) input selectors *stim* and *ctr* such that the following is a theorem.

FSTCA: $\mathcal{I}.\mathit{label}(\mathcal{I}.\mathit{trace}[\mathit{cfstim}, \mathit{cfctr}](n)) = \mathcal{S}.\mathit{label}(\mathcal{S}.\mathit{trace}[\mathit{stim}, \mathit{ctr}](n))$

The proof obligation for **FSTCA** guarantees that every fair trace of $\mathcal{I}$ is a trace of $\mathcal{S}$. In certain cases (as we will see), one wants to define $\mathcal{S}$ to ensure that each fair *trace* of $\mathcal{I}$ corresponds to some *fair trace* of $\mathcal{S}$. We can then say that $\mathcal{I}$ is a refinement of $\mathcal{S}$ under *fairness requirement*, written $(\mathcal{S} \rhd^{F} \mathcal{I})$. Formally, given the trace $\mathcal{I}.\mathit{trace}[\mathit{cfstim}, \mathit{cfctr}]$, the proof obligation for $(\mathcal{S} \rhd^{F} \mathcal{I})$ is to find some *fair* selector functions *fstim* and *fctr* such that the following is a theorem.

FSTCR: $\mathcal{I}.\mathit{label}(\mathcal{I}.\mathit{trace}[\mathit{cfstim}, \mathit{cfctr}](n)) = \mathcal{S}.\mathit{label}(\mathcal{S}.\mathit{trace}[\mathit{fstim}, \mathit{fctr}](n))$

Notice that $(\mathcal{S} \rhd^{F} \mathcal{I})$ does not necessarily imply $(\mathcal{S} \rhd \mathcal{I})$ since the former imposes no restriction on the "unfair" *trace*s.

How do our notions of fairness compare with those in other formalisms? The differences in the underlying logics make it difficult to provide a quantitative comparison. Fairness has always played an important role for proving progress

properties of reactive systems. Temporal logics like CTL and LTL provide a similar notion of fairness, where paths through a Kripke Structure are defined to be fair if they satisfy a specified fairness condition infinitely often. In this terminology, a fairness constraint is given by a subset of the states of the Kripke Structure. If the number of states is finite, then our formalism for fairness is merely a special case of such fairness constraints. Fairness also plays a crucial role in logics that are designed specifically to reason about reactive concurrent programs. One such popular logic is Unity [CM90]. The Unity logic is more expressive than propositional temporal logics, and we briefly review how our formalizations can be viewed in terms of the corresponding constructs of Unity.

In Unity, the transitions of a reactive system are modeled by providing a collection of "guarded actions". A guard is a predicate on the system state; a transition from a state s consists of non-deterministically selecting one of the actions whose guard evaluates to true om s and performing the update to s as prescribed by the action. Unity provides three notions of fairness, namely *minimal progress*, *weak fairness*, and *strong fairness*. Minimal progress says that *some action* is selected at every transition. Weak fairness says that if the guard of an action always evaluates to true then it is selected infinitely often. Strong fairness says that if the guard of an action evaluates to true infinitely often then it is selected infinitely often. To relate our notions with the notions of Unity we view the stimulus provided by the environment as prescribing the non-deterministic choice the system must make to select among the possible transitions. The restriction of the inputs to *good-objects* then can be thought to specify that the guards for the corresponding actions evaluate to true at every state and the guards for the other actions always evaluate to false.

With this view our models of reactive systems can be thought of as restricted versions of Unity models with a very simplified set of guards for each action.

With the above view, how do the different fairness notions of Unity translate to our world? Minimal progress is not very interesting in this context; it merely says that some *good-object* is always selected. Our notion of fairness constraints can be thought to be akin to weak fairness; every *good-object* prescribes a guard that always evaluates to true, and the constraint specifies that every such action must be selected infinitely often. Sumners [Sum03] shows that it is possible to formalize a notion corresponding to strong fairness in ACL2 as well. However, for the reactive systems we have modeled and reasoned about, we did not need strong fairness and hence we refrain from discussing it in this dissertation.

We end the discussion on fairness by pointing out one deficiency in our formalization.[3] In this dissertation, we typically use fairness as a way of *abstraction*, that is, hiding the *trace*s of the implementation that are not of interest to us. Our formalization is sufficient for the use of fairness this way. However, another problem that is frequently encountered in the verification of large-scale systems is *composition*. In compositional reasoning it is customary to reason about a specific component of a system by treating the remaining components as its "environment". A specific form of compositional reasoning, namely *assume-guarantee reasoning* [MQS00] affords the possibility of proving the correctness of each component separately under the assumption that the corresponding environment behaves properly, and then compose these individual proofs to a proof of the entire system. But consider what happens when we can show that a component $\mathcal{I}_c$ of the implementation $\mathcal{I}$ is a refinement of a

[3]This deficiency was pointed out to the author by John Matthews in a private conversation on October 20, 2005. The author is grateful to him for the contribution.

component $\mathcal{S}_c$ of the specification $\mathcal{S}$ under fairness assumptions. Then the obligation that the environment behaves appropriately is tantamount to showing that it selects every *good-object* infinitely often. But this is a severe restriction, and is often not satisfied by the other components of the system $\mathcal{I}$. On the other hand, it is possible that we can prove that $\mathcal{I}_c$ is a refinement of $\mathcal{S}_c$ under less restrictive assumptions on the environment (say an assumption that the environment selects inputs from numbers 1 to 10 infinitely often), which is satisfied by the other components of $\mathcal{I}$. This suggests that for the purpose of facilitating compositionality we might want a notion of fairness that is more flexible.

This limitation is indeed genuine and it is important to look for ways of modifying the notion of fairness to allow for it. While the current work does not address this, we believe that it is possible to extend our work to accomodate this possibility. In particular, a "nice" aspect of the construction of the fair selector for *good-object*s is that any subset of *good-object*s is selected infinitely often. Thus it is possible to modify the conditions **E2** and **E3** by replacing the *good-object*s with any other arbitrary (countable) set. With this modification we believe that it will be possible to use compositional reasoning with fairness.

7.4 Discussions

We have now formalized our statement of correctness of a reactive system implementations as a notion of refinement with respect to a specification system. Although we needed a number of definitions for doing it in ACL2, the notion itself is simple and easily understood. In the next two chapters, we will design proof rules to facilitate proofs of such refinements and see how such rules help automate verification of a

number of systems of different types. In this section, we compare our correctness statement with other formalisms for reasoning about reactive systems.

Aside from showing execution correspondence with a simpler system, the other method used for reasoning about a reactive implementation is specifying its desired property as a temporal logic formula. The relation between stuttering-invariant specifications and temporal logic is well-known; here we merely state the main result for completeness.

To talk about temporal logic properties, we must think about our models in terms of Kripke Structures. We will also have to assume that the range of the *label*s of the different models is a fixed (though not necessarily finite) set of atomic propositions $\mathcal{AP}$ The following proposition can be found in textbooks on temporal logic [CGP00], but has been restated in terms of our terminology.

Proposition 1 *Let ψ be an LTL formula such that (1) the atomic propositions mentioned in ψ are from $\mathcal{AP}$, and (2) ψ does not have any occurrence of the operator* X*. If a system $\mathcal{S}$ satisfies ψ, and $(\mathcal{S} \rhd \mathcal{I})$ then $\mathcal{I}$ satisfies ψ.*

Proof sketch: We prove this by induction on the structure of ψ and noting, from the semantics of LTL (page 21) that all the operators other than X are insensitive to finite stuttering. ■

Note that since trace containment only guarantees that every *trace* of $\mathcal{I}$ corresponds to some *trace* of $\mathcal{S}$ one cannot infer that an (LTL$\backslash\mathsf{X}$) formula ψ is satisfied by $\mathcal{S}$ if we know that $\mathcal{I}$ satisfies ψ.

As an aside, note that we are calling the statement above a "proposition" rather than a "lemma" or "theorem" as is customary. One of the facets of formal verification and in particular theorem proving is that we must distinguish between

theorems proved mechanically in the logic of a theorem prover and careful mathematical arguments. In this dissertation, we will always reserve the terms *lemma* and *theorem* for the former purpose. When we talk about mathematical arguments we refer to them as *propositions* and *claims*.

The connection suggests that if we write the desired property of a reactive system as an (LTL$\setminus$X) formula ψ then we can check whether a property holds for $\mathcal{I}$ by checking if it holds for $\mathcal{S}$. In particular, if $\mathcal{S}$ is finite state, then we can check if ψ holds in $\mathcal{I}$ by applying model checking on $\mathcal{S}$. Doing this has been one of the key methods for integrating model checking and theorem proving. Indeed, using ACL2, Manolios, Namjoshi, and Sumners [MNS99] do exactly this to verify the Alternating Bit Protocol [BSW69]. In Chapter 16, we will formalize the semantics of (propositional) LTL in ACL2 and use model checking to check LTL formulas for finite state systems.

Given the connection, one enticing thought is to *always* use temporal logic formulas to state the desired property of a reactive system of interest, and use stuttering refinement merely as a proof rule to transfer the verification of the property from the implementation system $\mathcal{I}$ to the more abstract system $\mathcal{S}$. Temporal logic specifications have a number of advantages [Lam83a], for example permitting description of safety and liveness properties directly as formula. Nevertheless we do not do this and prefer to think of the desired properties of $\mathcal{I}$ as encoded by the executions of the specification system $\mathcal{S}$. There are several reasons for this choice. Note that the logic of ACL2, on which our formalism is based, is first order. To talk about LTL formulas, we must encode the semantics of LTL as first order formulas in ACL2. As we will see in Chapter 16, this is non-trivial, and in fact the

natural semantics of LTL cannot be defined in ACL2. This limitation can be circumvented in a more expressive proof system such as HOL or PVS [CP99]. But it should be noted that one of the goals for using theorem proving is to be able to prove properties of *parameterized* systems. Indeed, in all the concurrent protocols we verify in Chapter 8, the number of processes is left unbounded. As we discussed in Chapter 2 specification of parameterized systems using temporal logic requires an expressive language for atomic propositions, such as quantification over process indices. Even when semantic embeddings are possible for such expressive temporal logic formulas, the specification in terms of such semantics is obscure and complicated, and significant familiarity in formal logic is necessary to decide if it captures the intended behaviors of the implementation. The problem is particularly acute in an industrial setting where often the most relevant reviewers of a specification are the designers of the implementation who might not have sufficient training in formal logic. Further, model checking is typically not applicable when reasoning about parameterized systems. Parameterized model checking is undecidable in general [AK86], and decidable solutions are known only under restrictions on the class of systems and properties [EK00], which might not be viable in practice.

On the other hand, as noted by several researchers [Lam83b, AL91], specification systems based on stuttering-invariant refinements typically capture the intuitive idea of the implementation designer regarding the desired behavior of the implementation. This is no coincidence. Reactive systems in practice are often elaborations of simpler protocols. The elaborations are designed to achieve desired execution efficiency, refine atomicity, match a given architecture, and so on. This simpler protocol provides a succinct description of the intended behaviors of the

implementation. Furthermore, the use of a *system* as a means of specification affords design of specifications and implementations in the same language, and hence makes the specifications suitable for review by implementation designers. Note also that refinements are used anyhow for decomposition of the verification problem even when the specifications are defined in terms of temporal logic. We feel that refinements are suitable artifacts for use in both specification and the proof process, and reserve temporal logic specifications for finite state systems where model checking is applicable.

Why do we insist that stuttering be finite? We do so since we wish to reason both about safety and progress properties of the implementation. To understand the point clearly consider an arbitrary specification system $\mathcal{S}$. If we did not want stuttering to be finite then the following trivial implementation system $\mathcal{I}$ would be considered a refinement of $\mathcal{S}$:

- $\mathcal{I}.\mathit{init}() \triangleq \mathcal{S}.\mathit{init}()$
- $\mathcal{I}.\mathit{next}(s, i) \triangleq s$

That is, the system $\mathcal{I}$ here always "loops" at the initial state. Notice that for every *trace* of $\mathcal{I}$ there *is* a *trace* of $\mathcal{S}$ (namely the one that always stutters) that has the same *label*. However, if $\mathcal{S}$ satisfies a progress property (that is, a property of the form something good eventually happens), we clearly cannot deduce that property for $\mathcal{I}$. One of the important requisites of a notion of correspondence is that once it has been established between an implementation and a specification we should be able to deduce the properties of the implementation that we are interested in by proving the corresponding properties for the specification. To allow us to do so for progress properties, we restrict the notion of correspondence so that such properties are preserved in the specification.

We end this discussion by comparing of our notion of correctness with another related notion, called Well-founded Equivalence Bisimulations (WEBs) that has been successfully used in ACL2 to reason about reactive systems [MNS99]. The proof rules for WEBs were shown to guarantee stuttering bisimulation between two systems. Our work is a direct consequence of the research with WEBs; for instance in Chapter 8, we will derive single-step proof rules for guaranteeing trace containment, which are analogous to the WEB rules. Nevertheless there are important differences between the two notions. First, WEBs use bisimulation, which is a branching time notion of correspondence. In this notion, executions are viewed, not as an infinite sequence of states, but as infinite trees. Further, given two systems $\mathcal{S}$ and $\mathcal{I}$, WEB (and in general bisimulation) proofs involve showing that for each execution of $\mathcal{S}$ there is a corresponding execution of $\mathcal{I}$ and vice versa, while we require only that the executions of $\mathcal{I}$ can be appropriately viewed as executions of $\mathcal{S}$. While bisimulation proofs show stronger correspondence, we have found that allowing the more abstract system to have more execution affords the definition of more succinct definition of the specification.[4] It should be noted here that one of the motivations for using a branching time notion of correspondence is that in the special case when $\mathcal{I}$ and $\mathcal{S}$ have a finite number of states, one can design polynomial time algorithm to check if there exists a branching time correspondence between the two systems, while checking the correspondence for the linear time notions that we used is PSPACE-complete. Manolios [Man01] implements such algorithms to complete the finite proof of the Alternating Bit Protocol. Since, at least for present, we are interested in formalizing a notion of correctness rather than designing efficient algorithms, we prefer trace

[4]Manolios [Man03] achieves this one-sided abstraction for branching time, by introducing proof rules for stuttering simulation.

containment as the definition is more intuitive. As an aside, many of the proof rules we present in Chapters 8 and 9 do preserve a branching time correspondence. Indeed, in Chapter 9, we will talk about *simulation correspondence*, a branching-time notion of correspondence that is weaker than bisimulation, in connection with reasoning about proof rules for pipelined microprocessor verification.

7.5 Summary

We have shown a formalization of stuttering trace containment as a notion of correspondence between two systems modeled at different levels of abstraction. We have also discussed how fairness notions can be integrated in the framework effectively as environmental constraints. As we will see, the use of this notion affords intuitive specification of a wide class of reactive systems.

It should be clear from the above presentation that the first order aspect of ACl2's logic provides some limit to what can be succinctly specified. Although the notion of stuttering trace containment is simple and well-studied, formalizing it in the logic takes some work. Nevertheless, at least for what we have done so far, the limitation is not much more than a passing annoyance. While the formalization in ACL2 is more complicated than what would have been possible in a theorem prover for higher-order logic (that allows functions to be arguments of other functions), it is not difficult to inspect our formalization and satisfy oneself that it does indeed capture the traditional notion. Further, in the next two chapters, we will prove several *proof rules* or *reduction theorems* which ought to increase confidence in the accuracy of our notion. Such proof rules reduce the actual process of verification of a reactive system to a first order problem for which the logic of ACL2 is effective.

However, the limitation of the logic will be an object of serious concern when we discuss model checking later in the dissertation.

7.6 Bibliographic Notes

The notions of trace containment, trace equivalence, and trace congruence are discussed in a well-cited paper by Pnueli [Pnu85]. The corresponding branching time notions of simulation and bisimulation are due to Milner and Park [Mil90, Par81]. Stuttering bisimulation was introduced by Browne, Clarke and Grumberg [BCG88].

Lamport [Lam83b] argued in favor of defining specifications that are invariant up to stuttering. Abadi and Lamport [AL91] presented several important properties of stuttering trace containment. Several researchers have since worked on building and extending theories for reasoning about reactive systems using stuttering-invariant specifications [EdR93, Hes02, Att99, JPR99]. Namjoshi [Nam97] presents sound and complete proof rules for symmetric stuttering bisimulation. Manolios, Namjoshi, and Sumners [MNS99] introduce a related notion, called well-founded equivalence bisimulation (WEB for short) and show how to reduce proofs of WEBs to single-step theorems. A brief comparison of our notion with theirs appears in Section 7.4.

Fairness has been dealt with extensively in the context of model checking [CGP00], and also forms an important component of logics like Unity [CM90] and TLA [Lam94] that are intended to reason about concurrent programs. Our models of fairness are based on the notion of unconditional fairness in the work of Sumners [Sum03]. We briefly compared our models of fairness with Unity in Section 7.3.

Many recent researches have involved formalization of the metatheory of abstraction in a theorem prover. The proof rules of Unity have been formalized in Nqthm [Gol90] and Isabelle [Pau00]. Chou [CP99] formalizes partial order reductions in HOL. Manolios, Namjoshi, and Sumners [MNS99] formalize the proof rules for WEBs in ACL2.

The results described in this chapter and the next are collaborative work with Rob Sumners.

Chapter 8

Verifying Concurrent Protocols Using Refinements

In the last chapter, we formalized (fair) stuttering trace containment in ACL2. In this chapter, we will use it as a notion of correctness to verify several reactive concurrent protocols. Concurrent (or multiprocess) protocols form some of the most important reactive systems, and are also arguably some of the most difficult computing systems to formally verify. The reason for this difficulty is well-known. Systems implementing concurrent protocols involve composition of a number of processes performing independent computations with synchronization points often far between. The number of possible states that such a system can reach is much more than that reached by a system executing sequential programs. Further, the non-determinism induced by independent computations of the processes makes it very difficult to detect or diagnose an error.

We use stuttering refinements to verify concurrent protocols. Irrespective of

the protocol being verified, we use similar notion of correctness, namely defining a specification system and showing one of $(\mathcal{S} \rhd \mathcal{I})$, $(\mathcal{S} \rhd_{\mathcal{F}} \mathcal{I})$, or $(\mathcal{S} \rhd^{F} \mathcal{I})$. However, while the statement of correctness given by this correlation is simple, it is cumbersome to prove the statement directly for a practical system. To facilitate such proofs, we define and prove a collection of *reduction theorems* as proof rules. We sample some of these theorems in this chapter. The theorems themselves are not novel, and are known in some form in the formal verification literature. Nevertheless, by formalizing them in ACL2 we can carefully orchestrate their application to produce refinement proofs of a large class of concurrent protocols. In Section 8.4 we will see applications of the reduction theorems on system examples. However, it should be understood that the theorems are by no means exhaustive; we only discuss those that we have found useful in our work on verification of concurrent programs in practice. The use of theorem proving and a formalized notion of correctness allows the user to augment them in a sound manner as necessary.

It is to be noted here that the reduction theorems we show are mostly second order formulas, and as such, cannot be directly written in ACL2. To formalize them in ACL2, we make heavy use of the encapsulation principle, as indeed, we did to formalize the notion of refinements. For succinctness and clarity, we write them here as higher order statements, and skip the details of how they are proved in ACL2. However, we stress that this is a matter of presentation; the proof rules have been effectively formalized in ACL2, although not in closed form.

8.1 Reduction via Stepwise Refinement

The first observation about the notion of stuttering refinements is that it is transitive. This observation is formalized by the following trivial set of proof rules.

SR1: Derive $(\mathcal{S} \rhd \mathcal{I})$ from $(\mathcal{S} \rhd \mathcal{I}_1)$ and $(\mathcal{I}_1 \rhd \mathcal{I})$

SR2: Derive $(\mathcal{S} \rhd_{\mathcal{F}} \mathcal{I})$ from either (1) $(\mathcal{S} \rhd \mathcal{I}_1)$ and $(\mathcal{I}_1 \rhd_{\mathcal{F}} \mathcal{I})$, or (2) $(\mathcal{S} \rhd_{\mathcal{F}} \mathcal{I}_1)$ and $(\mathcal{I}_1 \rhd \mathcal{I})$

SR3: Derive $(\mathcal{S} \rhd^{F} \mathcal{I})$ from $(\mathcal{S} \rhd^{F} \mathcal{I}_1)$ and $(\mathcal{I}_1 \rhd^{F} \mathcal{I})$

The system $\mathcal{I}_1$ is often referred to as an *intermediate model*, and the use of **SR1**, **SR2**, and **SR3** are referred to as *stepwise refinement*. Application of these proof rules allows us to introduce a series of intermediate models at different levels of abstraction starting from the implementation $\mathcal{I}$ and leading up to the specification $\mathcal{S}$. We then show refinements between every pair of consecutive models in this "refinement chain" and finally functionally instantiate **SR1** (resp., **SR2** and **SR3**), to derive the correspondence between $\mathcal{S}$ and $\mathcal{I}$.

8.2 Reduction to Single-step Theorems

The proof rules **SR1-SR3** show how to decompose a refinement proof into a sequence of refinements. We now focus on trying to decompose a refinement proof into a collection of proof obligations each involving a single transition of the two systems being compared.

Given two systems $\mathcal{S}$ and $\mathcal{I}$, we will say that $\mathcal{I}$ is a *well-founded refinement* of $\mathcal{S}$, written $(\mathcal{S} \trianglerighteq \mathcal{I})$ if and only if there exist functions *inv*, *skip*, *rep*, *rank*, and *pick* such that the following formulas are theorems.

SST1: $\mathit{inv}(s) \Rightarrow \mathcal{I}.\mathit{label}(s) = \mathcal{S}.\mathit{label}(\mathit{rep}(s))$

SST2: $\mathit{inv}(s) \wedge \mathit{skip}(s,i) \Rightarrow \mathit{rep}(\mathcal{I}.\mathit{next}(s,i)) = \mathit{rep}(s)$

SST3: $\mathit{inv}(s) \wedge \neg\mathit{skip}(s,i) \Rightarrow \mathit{rep}(\mathcal{I}.\mathit{next}(s,i)) = \mathcal{S}.\mathit{next}(\mathit{rep}(s), \mathit{pick}(s,i))$

SST4: $\mathit{inv}(\mathcal{I}.\mathit{init}())$

SST5: $\mathit{inv}(s) \Rightarrow \mathit{inv}(\mathcal{I}.\mathit{next}(s,i))$

SST6: $\mathit{o\text{-}p}(\mathit{rank}(s))$

SST7: $\mathit{inv}(s) \wedge \mathit{skip}(s,i) \Rightarrow \mathit{rank}(\mathcal{I}.\mathit{next}(s,i)) \prec_o \mathit{rank}(s)$

Although well-founded refinements comprise of several conditions, they are actually easy to interpret. Informally, **SST1-SST7** guarantee the following correspondence between $\mathcal{S}$ and $\mathcal{I}$: "For every *execution* of $\mathcal{I}$ there is a *trace* of $\mathcal{S}$ with the same label up to finite stuttering." Given a state s of $\mathcal{I}$, $\mathit{rep}(s)$ may be thought of as returning a state of $\mathcal{S}$ that has the same label as s. With this view, $\mathit{rep}(s)$ is also referred to as the *representative* of s.[1] The key conditions are **SST2** and **SST4** which say that given a transition of $\mathcal{I}$, $\mathcal{S}$ either has a "matching transition" or it stutters. The choice of $\mathcal{S}$ is governed by the predicate *skip*; if $\mathit{skip}(s,i)$ holds then **SST3** guarantees that $\mathcal{S}$ has a transition (namely by choosing the input $\mathit{pick}(s,i)$) that matches the transition of $\mathcal{I}$ from state s on input i, otherwise **SST3** guarantees that $\mathcal{S}$ can stutter. The finiteness of stuttering is guaranteed by conditions **SST6** and **SST7**. If $\mathit{skip}(s,i)$ does not hold, then *rank* decreases, where *rank* is guaranteed (by **SST7**) to return an ordinal. The conditions **SST4**, **SST5** guarantee that the *inv* is an *invariant*, that is, it holds for every reachable state of $\mathcal{I}$. This allows us to assume $\mathit{inv}(s)$ in the hypothesis of the other conditions.

The reader should note that the conditions for well-founded refinements allow only $\mathcal{S}$ to stutter, and not $\mathcal{I}$, although the definition of stuttering trace containment

[1] The function *rep* is often called the *abstraction function* in the literature on abstract interpretation [CC77] and is referred to as α. We prefer the more verbose name *rep* in this dissertation.

allowed both. We have not yet found it necessary to use two-sided stuttering for verification of actual systems. Since $\mathcal{S}$ is the more abstract system, we want it to stutter so that several transitions of $\mathcal{I}$ can correspond to one transition of $\mathcal{S}$. However, it is possible to suitably modify the characterization above in order to allow for two-sided stuttering.

The relation between well-founded refinements and **STC** is summarized by the following proof rule.

SST: Derive $(\mathcal{S} \rhd \mathcal{I})$ from $(\mathcal{S} \unrhd \mathcal{I})$

This proof rule is essentially a formalization of a corresponding rule that has been defined for well-founded bisimulations [Nam97, MNS99]. In particular, to prove it, we show that if for every execution of $\mathcal{I}$ there is a matching *trace* of $\mathcal{S}$ then there must be a matching *trace* of $\mathcal{S}$ for every *trace* of $\mathcal{I}$. The proof then follows by showing that the conditions **SST1**-**SST7** guarantee that for every execution of $\mathcal{I}$ there is a matching *trace* of $\mathcal{S}$.

The notion of well-founded refinements is useful since it reduces the proof of **STC** to *local reasoning*. Notice that none of the proof obligations **SST1**-**SST7** requires reasoning about more than one transition of the two systems $\mathcal{I}$ and $\mathcal{S}$.

It should be clarified, however, that the notion of well-founded refinements is *stronger* than that of **STC** in a technical sense. Well-founded refinements actually guarantee that $\mathcal{S}$ is a *simulation* of $\mathcal{I}$ up to finite stuttering. Simulation is a branching time notion of correspondence. As we mentioned briefly in the last section, in branching time notions the executions of a system are conceptualized as infinite trees instead of infinite sequences of states as we have done in our formalizations. It is well known that the notion of simulation is stronger than trace containment in the

sense that there are systems $\mathcal{S}$ and $\mathcal{I}$ such that $\mathcal{S}$ is related to $\mathcal{I}$ by trace containment but not by simulation [CGP00]. In such cases we will not be able to use the proof rule **SST** to derive **STC**. However, in practice we do not find this restriction prohibitive. Indeed, in the next chapter, we will use the notion of simulation itself directly as a proof rule in order to reason about pipelined machines.

So far, we have only talked about single-step theorems for showing **STC**; but we did not talk about how fairness constraints can be integrated with well-founded refinements. We do that now. First, we define a "single-step fair selector" as follows. We define two functions *sfselect* and *sfmeasure*, such that **SF1-SF3** below are theorems.

SF1: $\textit{good-object}(\textit{sfselect}(n))$

SF2: $\textit{o-p}(\textit{sfmeasure}(n, i))$

SF3: $\textit{good-object}(i) \wedge (\textit{sfselect}(n) \neq i) \Rightarrow \textit{sfmeasure}(n+1, i) \prec_o \textit{sfmeasure}(n, i)$

It is easy to see that the conditions **SF1-SF3** are equivalent to **E2** and **E3** we talked about in the last chapter. By *equivalent*, we mean the following. If one can define a function *fstim* that satisfies **E2** and **E3** then one can also define the functions *sfselect*, *sfmeasure*, and *sfstep* satisfying **SF1-SF3**. This equivalence is easy to formalize and prove in ACL2. Thus, by our description in the previous chapter, we know that there is at least one pair of functions that satisfies the single-step fairness conditions.

Nevertheless, even with the single-step selector, it is cumbersome to integrate fairness with well-founded refinements. Why? Even with the single-step selector, we are essentially talking about the input produced by the selector at different *times*. Notice, however, that one of the features of the conditions **SST1-SST7** is that we never have to talk about times but merely about transitions from certain states.

Thus we need to relinquish the elegance of single-step theorems somewhat in order to reason about fairness.

Fortunately, the loss is not that great. To understand the reason, we first note that fairness is necessary in practice to guarantee that progress is being made. The notion of progress is encoded in stuttering trace containment by the condition that stuttering needs to be finite. In the context of well-founded refinements, this is translated into conditions **SST6** and **SST7**; these specify that whenever $\mathit{skip}(s, i)$ holds, then $\mathit{rank}(\mathcal{I}.\mathit{next}(s, i))$ must be less (according to $\prec_o$) than $\mathit{rank}(s)$. We thus want to somehow modify these two conditions to exploit the fairness constraints.

The modified conditions are shown below. Thus we replace the function *rank* in **SST6** and **SST7** with the function *trank* which is stipulated only to be a function of current time rather than the current state. In the conditions below, *stim* is assumed to be an uninterpreted function and *fselect* and *fmeasure* are functions constrained to satisfy only **SF1-SF3** above.

FSST6: $\mathit{o\text{-}p}(\mathit{trank}(n))$

FSST7: $\mathit{inv}(\mathcal{I}.\mathit{exec}[\mathit{stim}](n)) \wedge \mathit{skip}(\mathcal{I}.\mathit{exec}[\mathit{stim}](n), \mathit{sfselect}(n)) \Rightarrow \mathit{trank}(n+1) \prec_o \mathit{trank}(n)$

We will call $\mathcal{I}$ a *fair well-founded refinement* of $\mathcal{S}$ (written $(\mathcal{S} \trianglerighteq_{\mathcal{F}} \mathcal{S})$) if one can define *inv*, *skip*, *pick*, and *trank* so that **SST1-SST5** and **FSST6** and **FSST7** are theorems.

The conditions **FSST6** and **FSST7** guarantee that the input selection for $\mathcal{I}$ is fair, but provides no guarantee on the inputs *pick*ed for $\mathcal{S}$. For this purpose, we need another function *srank*, and have the following proof obligations in addition to those for $(\mathcal{S} \trianglerighteq_{\mathcal{F}} \mathcal{I})$ above.

FR1: $\mathit{o\text{-}p}(\mathit{srank}(n, i))$

FR2: $\mathit{inv}(\mathcal{I}.\mathit{exec}[\mathit{stim}](n)) \wedge \mathit{good\text{-}object}(i) \wedge (\mathit{pick}(\mathcal{I}.\mathit{exec}(n), \mathit{fselect}(n)) \neq i) \Rightarrow$

$$\mathit{srank}(\mathit{fclk}(n+1), i) \prec_o \mathit{srank}(n, i)$$

Here the function *fclk* is defined as:

$$\mathit{fclk}(n) \triangleq \begin{cases} n & \mathit{if}\ \neg \mathit{inv}(\mathcal{I}.\mathit{exec}[\mathit{stim}](n)) \vee \mathit{skip}(\mathcal{I}.\mathit{exec}[\mathit{stim}](n), \mathit{fselect}(n)) \\ 1 + \mathit{fclk}(n+1) & \text{otherwise} \end{cases}$$

The function *fclk* determines the next time after n when $\mathcal{S}$ has to make a transition matching $\mathcal{I}$. The definition is admissible by using *trank* as a measure. The conditions **FR1** and **FR2** guarantee that the input used by $\mathcal{S}$ to match a transition of $\mathcal{I}$ satisfies the fairness requirements. Notice that the conditions guarantee that *srank* at "non-stuttering" points, thus ensuring that the fair inputs are not bypassed by $\mathcal{S}$. The following two proof rules summarize these observations.

FSST1: Derive $(\mathcal{S} \rhd_{\mathcal{F}} \mathcal{I})$ from $(\mathcal{S} \unrhd_{\mathcal{F}} \mathcal{I})$

FSST2: Derive $(\mathcal{S} \rhd^{F} \mathcal{I})$ from $(\mathcal{S} \unrhd_{\mathcal{F}} \mathcal{I})$ and **FR1-FR2**.

Note that the conditions to guarantee fairness are admittedly more cumbersome than their "non-fair" counterparts since they talk about entire *trace*s rather than proof obligations involving a state and its successor. This is necessary to integrate fairness as we discussed above. Nevertheless, the proofs of these obligations for a particular system usually involve reasoning about single steps of $\mathcal{I}$. The reason is easy to see. Since *stim* is an uninterpreted function, we do not know anything about $\mathcal{I}.\mathit{exec}[\mathit{stim}](n)$ other than that it returns some reachable state of $\mathcal{I}$. In general, we need to define *trank* as a function of $\mathcal{I}.\mathit{exec}[\mathit{stim}]$ to prove the obligation **FSST8**. However, the crucial observation now is that we can use *sfmeasure* in defining *trank* and *srank* above, and thus can use the conditions **SF2** and **SF3** to prove **FSST8**. We will see an example of how this is used in Section 8.4 when we discuss the correctness of a Bakery implementation.

We end this description of single-step theorems with a brief note on *invariants*. The reader looking at conditions **SST4** and **SST5** must have been reminded of step invariants that we discussed in the last part. In the context of reactive systems, we will call the predicate *inv* an *inductive invariant* of a system $\mathcal{I}$ if and only if it satisfies conditions **SST4** and **SST5**. These two conditions, of course, guarantee that *inv* holds for every reachable states of $\mathcal{I}$ and hence can be assumed as a hypothesis in each of the remaining conditions. The new terminology *inductive invariant* is used in place of *step invariants* to remind ourselves that we are now considering reactive systems. The difference in definition of course, is clearly shown by the fact that in the "persistence condition" **SST5**, we want that if *inv* holds for s then it must hold for the *next* state from s *irrespective of the input stimulus.* For step invariants this was not required since there was a unique next state from s. Nevertheless, the problem of defining inductive invariants for reactive systems is exactly analogous to the problem of defining step invariants that we talked about in Chapter 4. Namely, how should we define a predicate *inv* such that the following two objectives are met:

1. *inv* persists along every transition of the system, and

2. assuming *inv* as a hypothesis we can prove the obligations **SST1-SST3**, **SST6**, and **SST7** (and other conditions in case we are interested in fairness).

In practice, we decouple these two objectives as follows. We first define a predicate *good* so that we can prove the obligations of (fair) well-founded refinements other than **SST4** and **SST5** by replacing *inv* with *good.* As we will see when we consider system examples, coming up with the predicate *good* is not very difficult. We then have the following additional proof obligations:

RI1: $\mathit{inv}(\mathcal{I}.\mathit{init}())$

RI2: $\mathit{inv}(s) \Rightarrow \mathit{inv}(\mathcal{I}.\mathit{next}(s, i))$

RI3: $\mathit{inv}(s) \Rightarrow \mathit{good}(s)$

Clearly if we can do this then it follows that $(\mathcal{S} \trianglerighteq \mathcal{I})$. But this only delays the problem. Given *good* how would we define *inv* so that **RI1-RI3** are theorems? In case of the analogous problem for sequential programs, we saw in Chapter 6 how we can "get away" with defining a step invariant. We instead defined a partial clock function and a collection of symbolic simulation rules. Unfortunately, the non-determinism of reactive systems do not allow us to effectively port the approach. However, we will study the problem more closely in the next part and come up with an analogous solution. For now, we will consider the definition of *inv* strengthening *good* as above to be a manual process and see where that leads.

8.3 Equivalences and Auxiliary Variables

If $(\mathcal{S} \rhd \mathcal{I})$ and $(\mathcal{I} \rhd \mathcal{S})$ both hold then we say that $\mathcal{S}$ is *equivalent* to $\mathcal{I}$ (up to stuttering). We write $(\mathcal{S} \Diamond \mathcal{I})$ to mean that $\mathcal{S}$ are $\mathcal{I}$ are equivalent. Similarly, we say that $\mathcal{S}$ and $\mathcal{I}$ are *equivalent on fair executions* (written $(\mathcal{S} \Diamond^F \mathcal{I})$) if and only if $(\mathcal{S} \rhd^F \mathcal{I})$ and $(\mathcal{I} \rhd^F \mathcal{S})$. Of course it is trivial to note that if $(\mathcal{S} \Diamond \mathcal{I})$ (resp., $(\mathcal{S} \Diamond^F \mathcal{I})$), then $(\mathcal{S} \rhd \mathcal{I})$ (resp., $(\mathcal{S} \rhd^F \mathcal{I})$). Nevertheless, there are certain situations in which it is easier to prove equivalence than refinements.

Recall that one of the conditions for well-founded refinements (namely **SST3**) required the existence of a function *pick*. Given s and i, $\mathit{pick}(s, i)$ returned the matching input for the specification system $\mathcal{S}$ corresponding to a non-stuttering transition of $\mathcal{I}$ from state s on input i. Let us call $\mathcal{I}$ an *oblivious* well-founded refinement of $\mathcal{S}$, written $(\mathcal{S} \trianglerighteq_o \mathcal{I})$ if the following two conditions hold:

1. $(\mathcal{S} \trianglerighteq \mathcal{I})$

2. the function *pick* involved in the proof of (1) is the identity function on its second argument, that is, $\mathit{pick}(s, i) = i$ is a theorem.

Oblivious refinements guarantee that $(\mathcal{S} \diamond \mathcal{I})$. The proof is slightly non-trivial, and requires showing that under condition 2, given a *trace* of $\mathcal{S}$ one can construct a *trace* for $\mathcal{I}$. In addition, we can also use oblivious refinements with fairness requirements. We will say that $\mathcal{I}$ is an oblivious well-founded refinement of $\mathcal{I}$ *with fairness requirements*, written $(\mathcal{S} \trianglerighteq_o^F \mathcal{I})$ if and only if (1) $(\mathcal{S} \trianglerighteq_o \mathcal{I})$ and (2) the proof obligations **FR1** and **FR2** above are satisfied. In such cases, it is possible to show that for each fair *trace* of $\mathcal{S}$ there is a fair *trace* of $\mathcal{I}$ and vice versa. The consequences of oblivious refinements are summarized by the following two proof rules:

OR: Derive $(\mathcal{S} \diamond \mathcal{I})$ from $(\mathcal{S} \trianglerighteq_o \mathcal{I})$

ORF: Derive $(\mathcal{S} \diamond^F \mathcal{I})$ from $(\mathcal{S} \trianglerighteq_o^F \mathcal{I})$

Why is oblivious refinement useful? Consider a simple example due to Abadi and Lamport [AL91]. System $\mathcal{I}$ is a 1-bit digital clock, and system $\mathcal{S}$ is a 3-bit clock, and the *label* of a state in each system is simply the low-order bit. Clearly, $\mathcal{I}$ implements $\mathcal{S}$ since it has the same behavior as $\mathcal{S}$ up to stuttering. However, it is easy to see that we cannot show $(\mathcal{S} \trianglerighteq \mathcal{I})$; no mapping *rep* can define the state of a 3-bit clock as a function of 1-bit. However, it is easy, indeed trivial, to show $(\mathcal{I} \trianglerighteq_o \mathcal{S})$. Given a state s of $\mathcal{S}$, (that is, a configuration of a 3-bit clock), we simply need the representative mapping *rep* to project the low-order bit of s. Then we can use **OR** to show the desired result.

In general, we use equivalences to add *auxiliary variables*. Suppose we want to show $(\mathcal{S} \rhd \mathcal{I})$ (resp., $(\mathcal{S} \rhd^F \mathcal{I})$). We will often find it convenient to construct an intermediate system $\mathcal{I}^+$ as follows. A state $s+$ of $\mathcal{I}^+$ has all the components that a state s of $\mathcal{I}$ has, but in addition, has some more components. Further, the

components that are common to $\mathcal{I}$ and $\mathcal{I}^+$ are updated in $\mathcal{I}^+$ along any transition in exactly the same way as they are in $\mathcal{I}$, and the *label* of a state in $\mathcal{I}$ is the same as the label of a state in $\mathcal{I}^+$. Why are the extra variables used then? The answer is that they are used often to keep track of the history of execution which might be lost in state s. For instance the system $\mathcal{I}$, in reaching from $\mathcal{I}.\mathit{init}()$ to s might have made certain control decisions and encountered certain states. These decisions and choices are of course "lost" in state s, but might nevertheless be important to show that an execution of $\mathcal{I}$ is matched by some *trace* of $\mathcal{S}$. Then we can use $\mathcal{I}^+$ to explicitly store such decisions in some additional state component, and use them in the proof of correspondence between $\mathcal{S}$ and $\mathcal{I}^+$. However, in doing so, we have had to change the implementation $\mathcal{I}$. To apply transitivity we must now show $(\mathcal{I}^+ \rhd \mathcal{I})$ (resp., $(\mathcal{I}^+ \rhd^F \mathcal{I})$). But this might be difficult for exactly the reason that it was difficult to show that a 1-bit clock was a refinement of 3-bit clock, namely that $\mathcal{I}^+$ has more state components than $\mathcal{I}$ and it is often impossible to determine a representative mapping from the states of $\mathcal{I}$ to the states of $\mathcal{I}^+$. But as in that example, we can easily show $(\mathcal{I} \unrhd_o \mathcal{I}^+)$. Namely we define a mapping from the states of $\mathcal{I}^+$ to the states of $\mathcal{I}$ so that only the components common to both $\mathcal{I}$ and $\mathcal{I}^+$ are preserved. Further, in this case, we can prove oblivious refinements without stuttering. That is, we can prove $(\mathcal{I} \unrhd_o \mathcal{I}^+)$ by choosing the predicate $\mathit{skip}(s, i)$ to be identically `NIL`. By doing so, we note that the fairness requirements for oblivious refinements, namely **FR1-FR2**, become trivial. Thus by **OR** and **ORF** we note that when auxiliary variables are added to $\mathcal{I}$ to obtain a system $\mathcal{I}^+$, then both $(\mathcal{I}^+ \diamond \mathcal{I})$ and $(\mathcal{I}^+ \diamond^F \mathcal{I})$ hold. Thus for a refinement proof we are now allowed to freely add auxiliary variables to the implementation system in order to show that

the implementation is a (fair) refinement of the specification.

Note that the use of auxiliary variables is often considered a standard and trivial step in proving correspondences. Thus it might seem a little odd that we are going through so much trouble in justifying their use in our framework. But we believe that a theorem stating the correctness of a system must show a clear relation between the implementation and a specification, and any proof step, however trivial, should be formally "explained" as a proof rule.

8.4 Examples

We now show how stuttering refinement and the proof rules we have formalized can be used to reason about concurrent protocols. For illustration, we consider three example systems, namely a simple ESI cache coherence protocol, an model of the Bakery Algorithm, and a Concurrent deque implementation. Although the three systems are very different, we will see that we can use the notion of refinements to define very simple and intuitive specifications of the systems, and the same proof rules can be applied in each case to decompose the verification problem. We omit several other systems that have been verified using the same approach, which include a synchronous leader election protocol on a ring, and the progress property of the JVM byte-codes for a monitor-based synchronization protocol.

8.4.1 An ESI Cache Coherence Protocol

As a "warm-up", we will consider verifying a very simple system implementing a cache coherence protocol based on **ESI**. In this protocol, a number of *client* processes communicate with a single controller process to access memory blocks (or cache

lines). Cache lines consist of addressable data. A client can *read* the data from an address if its cache contains the corresponding line. A client acquires a cache line by sending a *fill* request to the controller; such requests are tagged for **E**xclusive or **S**hared access. A client with shared access can only *load* data in the cache line. A client with exclusive access can also *store* data. The controller can request a client to **I**nvalidate or *flush* a cache line and if the line was exclusive then its contents are copied back to memory.

The system consists of four components, which are described as follows.

- A 1-dimensional array called `mem`, that is indexed by cache lines. For a cache line `c`, `mem[c]` is a record with two fields, namely address and data. We assume that for any address `a`, there is a unique cache line which contains `a` in the address field. For simplicity, we also assume that the function *cline* takes an address and returns the cache line for the address. The data corresponding to some address `a` is the content of `mem[c].a.data` where `c` is *cline*(`a`).
- A 1-dimensional array `valid`. For each cache line `c`, `valid[c]` contains a set of process indices that have (**S**hared or **E**xclusive) access to `c`.
- A 1-dimensional array `excl`. For each cache line `c`, `excl[c]` contains a set of process indices that have **E**xclusive access to `c`.
- A 2-dimensional array `cache`, which is indexed by process index and cache line. For any process index `p` and any cache line `c`, `cache[p][c]` returns the local copy of the contents of cache line `c` in the cache of `p`.

Figure 8.1 describes in pseudo-code how each of these components are updated at every transition. The external stimulus `i` received for any transition is interpreted as a record of four fields `i.proc`, `i.op`, `i.addr`, and `i.data`. Here `i.proc` gives the index of the process that makes the transition, and `i.op` stipulates the operation

```
a := i.addr
c := cline(a)
p := i.proc
if (i.op == "flush") ∧ (p ∈ excl[c])
   mem[c] := cache[p][c]
```

```
a := i.addr
c := cline(a)
p := i.proc
if (i.op == "flush") ∧ (p ∈ excl[c])
   valid[c] := valid[c] \ p
else if (i.op == "fills") ∧ (excl[c] == ∅)
        valid[c] := valid[c] ∪ {p}
     else if (i.op == "fille") ∧ (valid[c] == ∅)
             valid[c] := valid[c] ∪ {p}
```

```
a := i.addr
c := cline(a)
p := i.proc
if (i.op == "flush") ∧ (p ∈ excl[c])
   excl[c] := excl[c] \ p
else if (i.op == "fille") ∧ (valid[c] == ∅)
        excl[c] := excl[c] ∪ {p}
```

```
a := i.addr
c := cline(a)
p := i.proc
if (i.op == "fills") ∧ (excl[c] == ∅)
   cache[p][c] := mem[c]
else if (i.op == "fille") ∧ (valid[c] == ∅)
        cache[p][c] := mem[c]
     else if (i.op == "store")
             cache[p][c].a.data := i.data
```

Figure 8.1: A Model of the **ESI** Cache Coherence Protocol.

performed by `i.proc`. This operation can be one of `"store"`, `"fille"`, `"fills"`, and `"flush"`. These correspond to writing to the local cache of a process, requesting a shared access to a cache line, requesting an exclusive access to a cache line, and writing back the contents of a local cache line of a process to the main memory. The system responds to these operations in the obvious way. For example, if the operation is a `"flush"` of a cache line `c`, then the process `p` is removed from `valid` and `excl`, and the contents of `c` in the local cache of `p` are copied to the memory.

It should be clear that the protocol can be effectively modeled in ACL2 as a reactive system. Let us call the system **esi**. Any state s of **esi** is a tuple of the four components above. The transition function **esi**.*next* is defined to formalize the updates to each component. The initial state is defined so that the sets `valid` and `excl` are empty for each cache line.

How would we write a specification for this system? In the specification we do not want to think of processes or cache, nor the sets `valid` and `excl`. Informally, we want to think of the implementation as a refinement of a simple memory whose contents are updated atomically at every `"store"`. This system, which we call **mem**, has a state transition function **mem**.*next* which is shown in pseudo-code in Figure 8.2. The *label* of a state of **mem**, as for the implementation **esi**, returns the memory component. The goal of our verification, then, is to show (**mem** $\rhd$ **esi**).

We can verify this simple system by showing (**mem** $\unrhd_o$ **esi**). Furthermore, we do not need stuttering for this example. That is, we will define *skip*$(s, i) \triangleq$ `NIL`. This means that the obligations **SST2**, **SST6**, and **SST7** are vacuous. We define the representative function *rep* as follows to construct a state s' of **mem** from a state s of **esi**.

- For each cache line `c`, if `valid[c]` is empty then `mem[c]` in s' is the same as `mem[c]`

```
a := i.addr
c := cline(a)
if (i.op == "store")
   mem[c].a.data := i.data
```

Figure 8.2: Pseudo-code for State Transition of System **mem**

in s. Otherwise, let p be an arbitrary but fixed process in `valid[c]`. Then `mem[c]` in s' is the same as `cache[p][c]`.

Finally, we need to come up with a predicate *inv* so that we can prove **SST1-SST5** to be theorems. We will follow the decoupling approach we discussed in Section 8.2. That is, we first define a predicate *good* so that we can prove **SST1-SST3** as theorems by replacing *inv* with *good*, and then prove **RI1-RI3**.

What should the predicate *good* be? It should be one that guarantees cache coherence. Cache coherence can be stated more or less directly as follows.

- Let s' be *rep*(s). Then in s for any cache line `c` and any process p such that `valid[c]` contains p, `cache[p][c]` in s is the same as `mem[c]` in s', otherwise `mem[c]` in s is the same as `mem[c]` in s'

The predicate *good* is defined so that it holds for a state s if the above condition is satisfied. Notice that we have almost bypassed the verification problem in the proofs of **SST2** and **SST3**. For example, the definition of *good*, itself, guarantees that the *label* of s is the same as the *label* of *rep*(s) for any **esi** state s. However, the complexity of the problem, as much as it exists in this system, is reflected when we want to define *inv* so that **RI1-RI3** are theorems. Predicate *inv* must imply *good* and also must "persist" along every transition. The predicate *inv* is a conjunction of the following conditions.

1. For any cache line `c` in state s, `excl[c]` is a subset of `valid[c]`.

2. For any cache line `c` in state s, `excl[c]` is either empty or a singleton.

3. For any cache line `c` in state s, if `excl[c]` is empty then the value of `cache[`p`][c]` is the same for each member p of `valid[c]`.

4. For each cache line `c` in state s, if `excl[c]` is a singleton and contains the process index p, then `valid[c]` must be equal to $\{p\}$.

Conditions 3 and 4 guarantee that *inv*(s) implies *good*(s). The remaining conditions are required so that *inv* is strong enough to persist along every transition. Notice that the definition of *inv* forms the crux of the verification effort which guarantees cache coherence. Once this predicate is defined, it is easy to show **RI1-RI3**, and hence complete the proof. One should also note that even for this trivial example some creativity is necessary in coming up with the definition of *inv*. It is probably fair to say that the complexity of a refinement proof principally depends on the complexity involved in defining this predicate. We will understand this more clearly as we move on to more involved concurrent protocols.

8.4.2 An Implementation of the Bakery Algorithm

Our second example is an implementation of the Bakery algorithm [Lam74]. This algorithm is one of the most well-studied solutions to the mutual exclusion problem for asynchronous multiprocess systems. The algorithm is based upon one commonly used in bakeries, in which a customer receives a number upon entering the store. The number allocated to a customer is higher than all the allotted numbers, and the holder of the lowest number is the next one to be served. This simple idea is commonly implemented by providing two local variables, a Boolean variable `choosing`

and an integer variable `pos`, for each process. Every process can read the private copy of these variables of every other process; however, a process can only modify the values of its own local copy. The variable `choosing` indicates if the process is involved in picking its number, and the value of `pos` is the number received by the process. Since two processes can possibly receive the same number, ties are broken by giving priority to the process with the smaller index.

We refer to our model of the implementation of this protocol as the system **bakery**. Figure 8.3 shows in pseudo-code the program that the process p having index `j` executes in **bakery**. The numbers to the left of the program instructions are the program counter values. A state of the system comprises of a vector `procs` of *local states* of all the processes and the value of the shared variable `max`. Given the vector `procs`, *keys*(`procs`) returns the list of indices of all the participating processes. The local state of process with index `j` is `procs[j]`. For each `j`, `procs[j]` consists of the value of the program counter, and the variables `choosing`, `temp`, and `pos`. The predicate $<_l$ is defined as follows. Given natural numbers a and b, and process indices c and d, and an irreflexive total order $<<$ on process indices, $(a, c) <_l (b, d)$ holds if and only if either $a < b$ or $a = b$ and $c << d$.

Notice that the implementation, in fact, depends on lower-level synchronization primitives, namely the atomicity of "compare-and-swap" or `cas` instruction.[2] However, the model is actually motivated by a microarchitectural implementation of the protocol. Further, the implementation is optimized and generalized in two aspects. We optimize the allotment of number or `pos` to a process, by keeping track

[2]The instruction `cas` is provided in many microarchitectures for synchronization purposes. The instruction takes three arguments, namely `var`, `old`, and `new`, where `var` is a shared variable and `old` and `new` are local to a process. The effect of executing the instruction is to swap the values of `var` and `new`, if the original value of `var` was equal to `old`.

```
Procedure Bakery
   1   choosing := T
   2   temp := max
   3   pos := temp + 1
   4   cas(max, temp, pos)
   5   choosing := nil
   6   indices := keys(procs)
   7   if indices = nil
           goto 11
       else
           current := indices.first
       endif
   8   curr := procs[current]
   9   if choosing[curr] == T
           goto 9
  10   if (pos[curr] ≠ nil) and
           ((pos[curr], current) <_l (pos[p],j))
           goto 10
       else
           indices := indices.rest; go to 7
  11   ⟨ critical section ⟩
  12   pos := nil
  13   ⟨ non-critical section ⟩
  14   goto 1
```

Figure 8.3: The Bakery Program Executed by Process p with Index j.

of the maximum number already allotted in the shared variable `max`. The variable can be read by the processes, and updated using the compare-and-swap instruction as specified in line 4. In addition, the process indices are not constrained to be natural numbers, nor are there a fixed number of processes. In other words, in our model, processes can "join" the algorithm at any time.

It should be clear from our description that the workings of the algorithm can be effectively defined as a state transition function **bakery**.*next*. The "input parameter" i of the state transition function is used to choose the index of the process which transits. That is, given a state s, **bakery**.*next*(s, i) returns the state that the system reaches when the process with index i executes one instruction from state s.

Finally, to complete our description of the **bakery** system, we must describe the *label* of a state. To do so, we will define a function *bmap* that maps the local state of a process to a string as follows.

- If the program counter of process p has value in the range `2-10` then *bmap*(p) returns `"wait"`.
- If the program counter of process p has value `11` then *bmap*(p) returns `"critical"`. Note that the pc value `11` corresponds to the critical section.
- Otherwise *bmap*(p) returns `"idle"`.

The *label* of a state s is then a vector indexed by process indices, so that the j-th element of the vector contains *bmap*(`procs[j]`) in s.

How about a specification for this system? Our specification system **spec** is simply a vector of processes, each of which is a simple state machine moving from local state `"idle"` to `"wait"` to `"critical"` and back to `"idle"`. Given a **spec** state s and a process index i, the *next* state of **spec** updates s as follows.

1. If the process p of index i has state `"critical"` it becomes `"idle"`.

2. If the process p of index i has state `"wait"` and no other process has state `"critical"` then it transits to state `"critical"`.

3. Otherwise the process p of index i has a local state of `"wait"`.

The *label* of a state s of **spec** is simply a vector of the local states of the processes in state s. It is easy to see that **spec** indeed does capture the intended behavior of **bakery**. Our mechanical proofs then show (**spec** $\rhd_{\mathcal{F}}$ **bakery**).

Before proceeding further, let us first discuss why we need fairness in the verification of **bakery**. A look at the **spec** system suggests that it has two properties, namely (1) at most one process has local state `"critical"` at any state, and (2) if in some state s some `wait`ing process is selected to transit and no process has state `"critical"` then the waiting process has state `"critical"` in the next state. A consequence of this definition is that in any execution of **bakery** which matches some execution of **spec** up to stuttering must have the following two properties.

Mutual Exclusion: If a process p is in the critical section then no other process q is in the critical section.

Progress: If in a state s there is at least one `wait`ing process and no process is in the critical section and only `wait`ing processes are selected after state s in the execution, then some process must eventually enter the critical section.

Both of these properties are desirable for a mutual exclusion protocol. Unfortunately, the **progress** property, however, does not hold for every execution of **bakery**. Consider the following scenario. Assume that the system is in a state s where no process is in the critical section, and processes p and q wish to enter. Let p first set its `choosing` to `T` and assume that it never again makes a transition. Then q sets `choosing` to `T`, obtains a value `pos`, and finally reaches the program point given by

the program counter value `9`. At this point, q `wait`s for every other process that had already set `choosing` to `T` to pick their `pos` and reset `choosing`. Thus, in our scenario, q indefinitely `wait`s for p and never has the opportunity to proceed. In fact, as long as p does not make a transition, *no* other process can proceed to the critical section, with each attempting process looping and `wait`ing for p to proceed.

This fact about **bakery** is sometimes overlooked even in rigorous analyses of the algorithm, since one is usually more interested in the **mutual exclusion** property rather than **progress**. However, while it does not hold in general, it does hold in fair executions of **bakery** since fairness guarantees that p is eventually selected to make progress.

To verify **bakery**, we will first define a new system **bakery**$^+$ that adds two auxiliary shared variables `bucket` and `queue`. Note that since these are only auxiliary variables, by our discussions in Section 8.3 we can justify (**spec**$\rhd_{\mathcal{F}}$**bakery**) by showing (**spec** $\rhd_{\mathcal{F}}$ **bakery**$^+$). These two variables are updated as follows:

- When a process sets `choosing` to `T`, it inserts its index into the `bucket`. The contents of `bucket` are always left sorted in ascending order according to <<. Recall that << is an irreflexive total order on process indices.

- If a process with program counter `4` successfully executes the `cas` instruction by updating `max` then the contents of `bucket` are appended to the end of `queue`. By *successfully executing* we mean that the value of `max` is actually incremented by the operation. Note that by the semantics of `cas`, nothing happens by executing `cas(max, temp, pos)` if the value of `max` is not equal to `temp` at the point of its execution. Such an execution of `cas` is not deemed successful.

- A process is dequeued when it moves from program counter value `10` to `11`.

We can now understand the intuition behind the `bucket` and `queue`. The `queue`

is intended to reflect the well-known assumption about the Bakery algorithm that the processes enter critical section "on a first-come first-served basis". However, two processes can have the same value of `pos` if the update of `pos` by one process is preceded by the reading of `pos` by another, and such ties are broken by process indices. The `bucket` at any state maintains a list of processes that all read the "current value" of `max`, and keeps the list sorted according to the process indices. Some process in the list is always the first to successfully increment `max`, and this causes the `bucket` to be flushed to the `queue`. The `queue` therefore always maintains the order in which processes enter the critical section.

How do we prove ($\mathbf{spec} \trianglerighteq_{\mathcal{F}} \mathbf{bakery}^{+}$)? To do so, we must define functions *rep*, *skip*, *rank*, and *inv*. As in other cases, we will define *good* to let us prove the single-step obligations for well-founded refinements with fairness assumptions, and think about *inv* later. The functions *rep*, *skip*, and *good* are defined as follows.

- Given a state s, *rep*(s) is simply the vector which maps the process indices in s to their *label*s.
- The predicate *skip*(s, i) is `NIL` if the process index i at state s has program counter value 1, 10, or 11, or outside the range `1-14` (in which case it will be assumed to join the protocol in the next state by setting the pc to `1`).
- The predicate *good* posits
    1. For each process index `j` that if the program counter of process `j` has value `7` and it is about to enter the critical section (that is, its local copy of `indices` is `nil`), then it must be the head of the queue
    2. A process in `queue` has the program counter values corresponding to `"wait"` in *label*.

As can be noticed from the descriptions above, these definitions are not very com-

plicated. The definition of *rank*, unfortunately, is more subtle. The reason for the subtlety is something we already discussed, namely that we need to make sure that if p is `waiting` for q to pick its `pos` then q must be able to progress. The *rank* we came up with is principally a lexicographic product of the following:

1. The number of processes having program counter value 1 if `queue` is empty.
2. The program counter of the process p at the head of `queue`.
3. Fairness measure on the index of the process p at the head of `queue`.
4. Fairness measure on the process indexed by `curr` in p if the program counter of p is in line 9 and if `curr` has its `choosing` set.

We are not aware of a simpler definition of *rank* that can demonstrate progress of **bakery**. While the definition is subtle, however, the complication is an inherent feature of the algorithm. The fairness condition 3 is required to show that every process which wants a `pos` eventually gets it, and the program counter and fairness measure of the process p are necessary to show that the system makes progress towards letting p in the critical section when it selects the index of p (thus decreasing the program counter), or some other process (thus decreasing p's fairness measure).

Of course, the chief complexity of verification is again in the definition and proof of an inductive invariant *inv*. The definition of *inv* principally involves the following considerations.

1. All the processes in the `bucket` have the same value of `temp`.
2. The `queue` has the processes arranged in layers, each layer having the same value of `temp` and sorted according to their indices.
3. Two consecutive layers in the `queue` have the difference in `temp` of exactly 1 for the corresponding processes.

4. The value of `pos` (after it is set) is exactly 1 more than the value of `temp`.

5. For every process past line 3 and before line 12, the value of `pos` is set, otherwise it is `NIL`.

The formal definitions of these conditions are somewhat complex, and further auxiliary conditions are necessary to show that their conjunction is indeed an inductive invariant. The formal definition contains about 30 predicates whose conjunction is shown to be an inductive invariant.

8.4.3 A Concurrent Deque Implementation

The final concurrent protocol that we present is an implementation of a concurrent deque.[3] A *deque* stands for "double-ended queue"; it is a data structure that stores a sequence of elements and supports insertion and removal of items at either end. We refer to the two ends of a deque as *top* and *bottom* respectively. The system we analyze contains a shared deque implemented as an array `deque` laid out in the memory. The deque is manipulated by different processes. However, the system is restricted as follows:

1. Items are never inserted at the top of the deque.

2. There is a designated process called the *owner*, which is permitted to insert and remove items from the bottom of the deque; all other processes are called *thieves* and are permitted only to remove items from the top of the deque.

[3]The work with stuttering refinements and its use in the verification of concurrent protocols is based on joint collaboration of the author with Rob Sumners. The mechanical proof of the concurrent deque is work by Sumners, and has been published elsewhere [Sum00, Sum05]. We describe this proof here with his permission since the proof is illustrative of the complexity induced in reasoning about subtle concurrent systems. Of course the presentation here based solely on the author's understanding of the problem and the details, and consequently the author is responsible for any errors in the description.

```
void pushBottom (Item item)
   1  load localBot := bot
   2  store deq[localBot] := item
   3  localBot := localBot + 1
   4  store bot := localBot
```

```
Item popTop()
   1  load oldAge := age
   2  load localBot := bot
   3  if localBot ≤ oldAge.top
   4      return NIL
   5  load item := deq[oldAge.top]
   6  newAge := oldAge
   7  newAge.top := newAge.top + 1
   8  cas (age, oldAge, newAge)
   9  if oldAge = newAge
  10      return item
  11  return NIL
```

```
Item popBottom()
   1  load localBot := bot
   2  if localBot = 0
   3      return NIL
   4  localBot := localBot - 1
   5  store bot := localBot
   6  load item := deq[localBot]
   7  load oldAge := age
   8  if localBot > oldAge.top
   9      return item
  10  store bot := 0
  11  newAge.top := 0
  12  newAge.tag := oldAge.tag + 1
  13  if localBot = oldAge.top
  14      cas (age, oldAge, newAge)
  15      if oldAge = newAge
  16          return item
  17  store age := newAge
  18  return NIL
```

Figure 8.4: Methods for the Concurrent Deque Implementation

Figure 8.4 shows a collection of programs implementing this restricted deque. The procedures `pushBottom`, `popBottom`, and `popTop` accomplish insertion of an item at the bottom, removal of an item from the bottom, and removal of an item from the top respectively. This implementation is due to Arora, Blumofe, and Plaxton, and arises in the context of *work-stealing algorithms* [ABP01]. In that context, the deque is used to store jobs. A designated process, namely the owner, spawns the jobs and stores them in a deque by executing `pushBottom`, and executes `popBottom` to remove the jobs from the bottom of the deque to execute them. Other processes, namely *thieves* "steal" jobs from the top and interleave their execution with the execution of the owner.

Since the system involves multiple processes manipulating a shared object, one has to deal with contention and race conditions. In this case, the contention is among the thieves attempting to pop an item from the deque, or between a thief and the owner when the deque contains a single item. The implementation, although only involving about 40 lines of code as shown, is quite subtle. The reason for the complexity is that the implementation is *non-blocking*, that is, when a process removes a job from the deque it does not need to wait for any other process. Analysis of the efficiency of the work stealing algorithm depends critically on the non-blocking nature of this implementation.

A rigorous, though not mechanical, proof of the concurrent deque implementation has been done before [BPR99]. This proof showed that any interleaving execution of the methods shown in Figure 8.4 could be transformed into a synchronous execution in which every process invokes the entire program atomi-

cally.[4] This transformation is demonstrated by permuting different sequences of program steps, termed *bursts*, of different processes until the resulting execution is synchronous. The permutations are presented through a series of 16 congruences, beginning with the identity permutation and ending with a relation that ties every execution with a synchronous one. For instance, consider two bursts Π_1 and Π_2 executed by two different processes, that only involve updates to the local variables. Then we can commute the executions of Π_1 and Π_2 without affecting the results.

The hand proof above involved a large collection of cases and subtle arguments were required for many of the individual cases. Thus it makes sense to do a mechanical proof of the system using a notion of correctness that clearly connects the implementation and the specification. This is done by formalizing the implementation as a reactive system in ACL2 (which we term **cdeq**) defining a specification of the system **cspec** where the "abstract deque" is modeled as a simple list and insertion and removal of items are atomic. Proof of correctness, then, is tantamount to showing (**cspec** $\rhd$ **cdeq**).

How complicated is the refinement proof? The proof was done by defining three intermediate systems **cdeq**$^+$, **icdeq**, and **icdeq**$^+$, and showing the following chain of refinements:

$$(\mathbf{cspec} \trianglerighteq \mathbf{icdeq}^+ \diamond \mathbf{icdeq} \diamond \mathbf{cdeq}^+ \diamond \mathbf{cdeq})$$

Let us understand what the intermediate models accomplish. This will clarify how one should use a chain of refinements to decompose a complex verification problem. We look at the **icdeq** system first since it is more interesting. In **cdeq**, the deque is represented as an array laid out in the memory. In fact the deque is specified to be

[4]Strictly speaking, in the *synchronous* execution, several `popTop` executions could happen at exactly the same point with exactly one of them succeeding.

the portion of the memory between two indices pointed to by two shared variables `age.top` and `bottom`, where `bottom` represents the bottom of the deque (actually the index of the memory cell succeeding the bottom of the deque) and `age.top` represents the top. When a thief process pops an item from the top of the deque, it increments the pointer `age.top`, while an insertion causes increment in the `bottom` pointer. (Thus `bottom` is "above" the `top`.) On the other hand, in **cspec**, the deque is represented as a simple list and processes remove items from either end of the list. Also, insertion and removal of items are atomic. The goal of **icdeq** is to allow a list representation of the deque but allow the owner and thief transitions to be more fine-grained than **cspec**. What do the owner and thief transitions in **icdeq** look like? Essentially, any sequence of local steps of a process is "collapsed" to a single transition. A local step of a process is one in which no update is made to a shared variable. For instance, consider the sequence of actions taken by a thief (executing `popTop`) as it makes the sequence of transitions passing through pc values `6` $\rightarrow$ `7` $\rightarrow$ `8`. None of the transitions involved changes any of the shared variables in the system, namely `age`, `bot`, or `deque`. In **icdeq**, then, this sequence is replaced with a single transition that changes the thief pc value from `6` to `8` and the update made to the local state of the thief in making this transition in **icdeq** is a composition of the updates prescribed in **cdeq** for this sequence of transitions. The system **icdeq** also simplifies the representation of `age` in certain ways. This latter simplification is closely connected with the need for the model **cdeq**$^{+}$ and we will discuss it in that context.

How does one prove that our implementation is indeed a refinement of **icdeq**? This is proved by using well-founded refinements but with the input selection function *pick* that is identity on its second argument. As we discussed in Section 8.3,

this is tantamount to proving stuttering equivalence. To do this, we must of course define functions *rep*, *rank*, *skip*, and *inv* so that we can prove the obligations for well-founded refinements. This is done as follows.

- For an implementation state s and input i (specifying the index of the process poised to make a transition from s), $\textit{skip}(s, i)$ holds if and only if the transition of process i corresponds to a local step. For instance if the value of the program counter of i in s is 7 and i is a thief process, then $\textit{skip}(s, i)$ holds.

- The function *rep* is defined component-wise. That is, for each process i in a state s of the implementation, $\textit{rep}(s)$ "builds" a process state in **icdeq** corresponding to process i. This involves computation of the "right" program counter values for **icdeq**. For instance consider the example above where i is a thief with pc 7. How do we map process i to a process in **icdeq**? Since for every transition of i after encountering pc 6 corresponds to a stuttering in **icdeq**, we will build a process state with the pc having the value 6, and let $\textit{rep}(s)$ map process i to this state. In addition, $\textit{rep}(s)$ also maps the configuration of the shared variables of s to the representation prescribed in **icdeq**. Thus, for instance, the configuration of the array-based deque is mapped to the corresponding list representation.

- The *rank* function is simple. Given a state s, we determine, for each process i, how many transitions i needs to take before it reaches a state with program counter that is "visible" to **icdeq**. Call this the *process rank* of i. For instance if i is the thief above, then its process rank is i, since this is the number of transitions left before it reaches the pc value of 8, which is visible to **icdeq**. The *rank* of s then is the sum of the ranks of the individual processes. Clearly this is a natural number (and hence an ordinal) and decreases for every stuttering transition.

- How do we define *inv*? We decompose it into three components, namely predicates on the owner, predicates on the thieves, and predicates on the shared variables. For a process (the owner or a thief) in an "invisible" local state, we need to posit that it

has made the correct composition of updates since the last visible state. For a process in a visible local state, we define predicates relating the process variables with the shared ones.

The reader would note that in the above we have used the term "implementation" loosely. Our refinement chain shows that we relate **icdeq** not with **cdeq** but actually with $\mathbf{cdeq}^+$. The system $\mathbf{cdeq}^+$ plays the same role as $\mathbf{bakery}^+$ did for our Bakery implementation. That is, it is really the system **cdeq** augmented with some auxiliary variables to keep track of the history of execution. Therefore, the proof of $(\mathbf{cdeq}^+ \Diamond \mathbf{cdeq})$ is trivial by our earlier observations. Nevertheless, we briefly discuss the role of $\mathbf{cdeq}^+$ in the chain of refinements for pedagogical reasons. There is an analogous role played by $\mathbf{icdeq}^+$ which is the system **icdeq** augmented with some auxiliary variables. We omit the description of $\mathbf{icdeq}^+$.

The reason for having $\mathbf{cdeq}^+$ stems principally from the fact that we want to have a simpler representation of the shared variables in **icdeq** than that in **cdeq**. For example we represent the deque as a list in **icdeq** rather than an array laid out in the memory. We also want to simplify the "job" done by the shared variable `age`. This variable does several duties in **cdeq**. It has two components `age.top` and `age.tag`. The component `age.top`, together with `bottom`, determines the layout of the deque in the memory. The role of `age.tag` is more interesting. When the owner detects that the deque is empty while performing a `popBottom`, it resets both `bottom` and `age.top` to `0`. This is performed by execution of the `cas` instruction at pc `14`. However, to ensure that a thief that might have read a "stale" `top` value of `0` earlier does not attempt to remove an item from the empty deque after reset, the owner increments the value of `age.tag`. This value is monotonically increasing and

therefore would not match with the value that the thief would have read.

Since `age` tracks so many things, it is desirable to "abstract" it. In **icdeq**, `age` is replaced by a simple counter that is incremented every time an item is removed from the (list-based) deque. Unfortunately it is difficult to determine a consistent value of the counter from the value of `age` at an arbitrary **cdeq** state s. This is solved by defining **cdeq**$^+$ in which the counter we want is an auxiliary variable that is incremented every time `age.tag` is updated. The system **cdeq**$^+$ adds some other auxiliary variables to facilitate proof of correspondence with **icdeq**.

What did we achieve from the intermediate models? The goal of the system **icdeq** is to hide local transitions and provide a simpler representation of the data so as to facilitate thinking about the central concepts and subtleties behind workings of the system. This illustrates how refinements can afford compositionality of verification. The **icdeq** system was not defined arbitrarily but with the objective of hiding some specific details (in this case the complexity of `age` and memory layout of the deque) in mind. In general for verifying a subtle system, we decide on decomposing the refinement problem into a chain with every "link" representing some specific details that are hidden.

The crux of the verification is to show (**cspec** $\trianglerighteq$ **icdeq**$^+$). Here we must reason about how every step of the **icdeq** system updates the process states and the shared variables. The complexity manifests itself in the definition of the invariant involved. The size of the invariant is baffling. As a measure of the complexity, we merely mention that one fragment of the proof that the invariant is indeed inductive involved 1663 subgoals! Nevertheless, the definition is created more or less in the same process as we saw for the other systems, namely coming up with an

appropriate predicate *good* that allows us to prove the single-step theorems, followed by iteratively strengthening *good* until one obtains a predicate that persists along every transition. Thus for example, we start with some predicate on the owner state and notice, by symbolic expansion of the transition function, that in order for it to hold one step from s, it is necessary that some other predicate on the owner (or a thief) must hold at s. The definition of the inductive invariant could be thought of as determining a "fixpoint" over this analysis. This intuition will be invaluable to us in the next part when we design procedures to automate the discovery and verification of invariants.

8.5 Summary

We have shown how refinement proofs can be used to reason about concurrent protocols. To facilitate reasoning about such systems, we have developed a collection of reduction theorems as proof rules. We proved a number of concurrent protocols in ACL2 by proving a chain of refinements using these proof rules.

The proofs of different concurrent protocols using the same notion of refinements shows the robustness of the notion as a means of specification of concurrent protocols. The three systems we discussed in the previous section are all very different. Nevertheless, the same notion of correspondence could be used to define intuitive specifications of all three systems. Refinements with stuttering have been used and proposed in many papers so that systems at different levels of abstraction can be compared [AL91, Lam83b, MNS99, Man03]. But we believe that the work reported here provides the first instance of their effective formalization in a theorem prover to the point that they allowed intuitive specification of realistic systems in

the logic of the theorem prover. In the next chapter we will see that a different class of systems, namely pipelined machines, can also be verified in the same framework.

Admittedly, as we commented several times already, the first order nature of ACL2 makes the process of formalization difficult. The reduction theorems are not in closed form, and their formal proofs are sometimes subtle. Nevertheless, formalizing the notion of correctness itself in the theorem prover allowed us to design effective proof rules that could be adapted to different types of systems. Indeed, many of the rules were developed or modified when the existing rules were found inadequate for a particular problem. In the next chapter, we will slightly modify the single-step theorems in order to be able to reduce flushing proofs of pipelined machines to refinement proofs. Theorem proving, in general, requires manual expertise. Thus when using theorem proving one should use this expertise judiciously. One way in which theorem provers can be effective is in formalizing a verification framework itself with its rules and decision procedures which can then be applied to automate proofs of individual systems. We have now seen formalization of a deductive framework using theorem proving. Later in this dissertation we will see how decision procedures can be formalized too.

We should admit that the limiting problem in deriving refinement proofs is in the complexity of defining inductive invariants. Inductive invariants are both tedious and complicated to derive manually. Thus we should look at some ways of automating their discovery. We will investigate how we can do that in the next part.

8.6 Bibliographic Notes

Proving the correctness of reactive concurrent protocols has been the focus of much of recent research in formal verification. Among model checking approaches, there has been work with the Java pathfinder [VHB+03] for checking assertions in concurrent Java programs. Model checking has been used to check concurrent programs written in TLA [LMTY02]. Among work with theorem proving, Jackson shows how to verify a garbage collector algorithm using PVS [Jac98], and Russinoff verifies a garbage collector using Nqthm [Rus94]. In ACL2, Moore [Moo99a] describes a way of reasoning about non-blocking concurrent algorithms. Moore and Porter [MP02] also report the proof of JVM program implementing a multi-threaded Java class using ACL2.

Abstraction techniques have also been applied to effectively reason about concurrent programs. McMillan [McM98] uses built-in abstraction and symmetry reduction techniques to reduce infinite state systems to finite states and verify them using model checking. The safety (mutual exclusion) property of the bakery algorithm was verified using this method [MQS00]. Regular model checking [BJNT00] has been used to verify unbounded state systems with model checking by allowing rich assertion languages. Emerson and Kahlon [EK03] also show how to verify unbounded state snoopy cache protocols, by showing a reduction from the general parameterized problem to a collection of finite instances.

Chapter 9

Pipelined Machines

In the previous chapter we saw the application of stuttering refinement to reason about concurrent protocols. In this chapter, we will see how the same notion of correctness can be used to reason about about pipelined microprocessors.

Microprocessors are modeled at several levels of abstraction. At the highest level is the *instruction set architecture* (**isa**). The **isa** is usually modeled as a non-pipelined machine which executes instructions atomically one at a time. This model is useful for the programmer writing programs for the microprocessor. A more detailed view of the microprocessor execution is given by the *microarchitectural model* (**ma**). This model reflects pipelining, instruction and data caches, and other design optimizations. *Microprocessor verification* means a formal proof of correspondence between the executions of **ma** and those of **isa**.

Both **ma** and **isa** can be modeled as reactive systems. We can define the *label* of a state to be the programmer-visible components of the state. Using such models, we can directly apply stuttering trace containment as a notion of correctness

for microprocessor verification. Then, verifying a microprocessor is tantamount to proving ($\mathbf{isa} \rhd \mathbf{ma}$).

9.1 Simulation Correspondence, Pipelines, and Flushing Proofs

In prior work on verification of *non-pipelined* microprocessors [Coh87, Hun94, HB92], the correspondence shown can be easily seen to be the same as trace containment. More precisely, such verification have used *simulation correspondence*. Simulation correspondence can be informally described as follows. One defines a predicate *sim* as a relation between the states of **ma** and **isa** with the following properties:

- $\mathit{sim}(\mathbf{ma}.\mathit{init}(), \mathbf{isa}.\mathit{init}())$
- If ma and isa are two states such that $\mathit{sim}(ma, isa)$ holds, then
    1. $\mathbf{ma}.\mathit{label}(ma) = \mathbf{isa}.\mathit{label}(isa)$
    2. for all i there exists some i' such that $\mathit{sim}(\mathbf{ma}.\mathit{next}(ma, i), \mathbf{isa}.\mathit{next}(isa, i'))$ holds.

The predicate *sim* is referred to as a *simulation relation*. It should be clear that the two conditions above imply that for every execution of **ma** there is a matching execution of **isa** having the same corresponding *label*s. More precisely, assume that *mstimulus* is an arbitrary function so that $\mathit{mstimulus}(n)$ is the stimulus provided to **ma** at time n. Then it is easy to show that the above conditions guarantee that there exists some function *istimulus* so that the following is a theorem.

- $\mathbf{ma}.\mathit{label}(\mathbf{ma}.\mathit{exec}[\mathit{mstimulus}](n)) = \mathbf{isa}.\mathit{label}(\mathbf{isa}.\mathit{exec}[istimulus](n))$

This is simply trace containment, and in fact there is no necessity for stuttering. Indeed, it is known that simulation is a stronger characterization than trace containment, in that if we can prove simulation then we can prove trace containment but

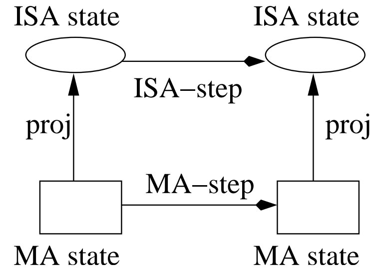

Figure 9.1: Pictorial Representation of Simulation Proofs Using Projection

not necessarily vice versa. In microprocessor verification, one often uses a specific simulation relation that is based on so-called "projection functions". In this approach, one defines a function *proj* such that, given a **ma** state ma, $\mathit{proj}(ma)$ returns the corresponding **isa** state. Using the terminology we introduced in the previous chapter, we can think of *proj* as the *representative function* *rep*. The function is called *projection* since it "projects" the programmer-visible components of an **ma** state to the **isa**. The theorem proved to show the correctness of **ma** can then be written as:

$$\mathit{proj}(\mathbf{ma}.\mathit{next}(ma, i)) = \mathbf{isa}.\mathit{next}(\mathit{proj}(ma, \mathit{pick}(ma, i)))$$

The theorem is shown more often as a diagram such as in Figure 9.1. This can be cast as a simulation proof by defining the simulation relation to be:

$$\mathit{sim}(ma, isa) \triangleq (\mathit{proj}(ma) = isa)$$

All this can be effectively used for microarchitectures that are not pipelined. For pipelined microarchitectures, however, one cannot use simple simulation correspondence. Why? In a pipelined **ma**, when one instruction completes execution others have already been partially executed. Thus, it is difficult to come up with a simulation relation such that the properties 1 and 2 above hold. This problem is often referred to as the *latency problem* [SB90, BT90]. As a result of this problem, a large

number of correspondence notions have been developed to reason about pipelined machines. As features like interrupts and out-of-order instruction execution have been modeled and reasoned about, notions of correctness have had to be modified and extended to account for these features. Consequently, the correspondence theorems have become complicated, difficult to understand, and even controversial [ACDJ01]. Furthermore, the lack of uniformity has made composition of proofs of different components of a modern processor cumbersome and difficult.

The work in this chapter shows that such complicated notions of correctness are not necessary to verify pipelined machines. Indeed, stuttering trace containment is a sufficient and useful correctness criterion that can be applied to most pipelines. This particular observation was made by Manolios [Man00a]. Manolios showed that one can use WEBs to reason about pipelined machines. WEBs are stronger notions of correctness than stuttering trace containment and we briefly compared the two notions in Chapter 7. Thus Manolios' results guarantee that one can reason about pipelines using stuttering trace containment as well. Nevertheless, our proofs have one important repercussion. Using our approach, it is now possible to understand most of the notions of correctness used in pipeline verification as merely proof rules for proving trace containment. In particular, we show that most of the so-called "flushing proofs" of pipelines can be mechanically translated into refinement proofs.

What are flushing proofs? The notion was introduced by Burch and Dill in 1994 [BD94] as an approach to compare **ma** states with **isa** states, where the **ma** now is a pipelined machine. The notion is shown pictorially in Figure 9.2. To construct an **isa** state from an **ma** state, we simply flush the pipeline, that is, complete all partially executed executions in the pipeline without introducing any

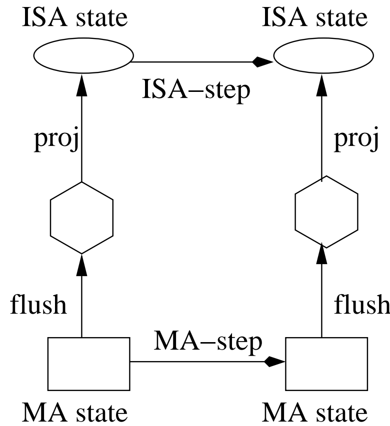

Figure 9.2: Pictorial Representation of Flushing Proofs

new instruction. We then project the programmer-visible components of this flushed state to create the **isa** state. Then the notion of correctness says that **ma** is correct with respect to the **isa**, if whenever flushing and projecting an **ma** state ma yields the state isa, it must be the case that for every possible next state ma' in **ma** from ma there must be a state isa' in **isa** such that (1) isa' can be reached from isa in one **isa** step, and (2) flushing and projecting from ma' yields isa'.

An advantage of the flushing method in reasoning about pipelines is that many pipelined machines contain an explicit `flush` signal. Using this signal, one can use the pipeline itself to generate the flushed state from a micro architectural state ma. If such a signal does not exist, however, then in a logic one can define a function *flush* that symbolically flushes the pipeline. It should be noted that the diagram shown in Figure 9.2, unlike the diagram in Figure 9.1, does not render itself directly to a proof of simulation correspondence or trace containment. This is because we cannot treat the composition of *flush* and *proj* as a representative function

(unlike *proj* alone), since the *label* of a state obtained by flushing a state *ma* might be very different from that of *ma* itself, unless the flusing operation is defined as a component of the state label. Indeed, as shown by Manolios [Man00a], certain types of flushing diagrams are flawed in that trivial, obviously incorrect machines satisfy such notion of correctness.

We must clarify here that we do not take any position one way or the other on whether one should use flushing diagrams to reason about pipelined machines. Indeed, in Chapter 14 we will verify a reasonably complicated pipeline implementation and establish that the microarchitectural implementation is a refinement of the instruction set architecture directly using the proof rules we developed in the last chapter. Other approaches for showing refinements have been used in reasoning about pipelined machines. For example, Manolios has developed a proof strategy based on what he calls *commmitment approach*. Our goal here is to simply point out that flushing proofs of pipelines, when appropriate, can be formally viewed as a way of deriving a refinement theorem given our notion of correspondence.

9.2 Reducing Flushing Proofs to Refinements

How do we reduce flushing proofs to trace containment? Here we give an informal overview. We will see the approach work with a concrete example in Section 9.4 and discuss ways of adapting the approach to advanced pipelines in Section 9.5.

For our informal overview, assume that the pipeline in **ma** involves in-order instruction execution and completion, no interrupts, and also assume that only the instruction being completed can affect the programmer-visible components of a state. Consider some state *ma* of **ma**, where a specific instruction i_1 is poised

to complete. Presumably, then, before reaching ma, **ma** must have encountered some state ma_1 in which i_1 was poised to enter the pipeline. Call ma_1 the *witnessing state* of ma. Assuming that instructions are completed in order, all and only the incomplete instructions at state ma_1 (meaning, all instructions before i_1) must complete before the pipeline can reach state ma starting from ma_1. Now consider the state ma_{1F} obtained by flushing ma_1. The flushing operation also completes all incomplete instructions without fetching any new instructions; that is, ma_{1F} has all instructions before i_1 completed, and is poised to fetch i_1. If only completed instructions affect the visible behavior of the machine, then this suggests that ma_{1F} and ma must have the same programmer-visible components.[1]

Based on the above intuition, we define a relation *psim* to correlate **ma** states with **isa** states in the following manner: ma is related to isa if and only if there exists a witnessing state ma_{1F} such that isa is the projection of the visible components of ma_{1F}. We will show that *psim* is a simulation relation between **ma** and **isa** states as follows. Recall that projection preserves the visible components of a state. From the arguments above, whenever states ma and isa are related by $psim$ they must have the same *label*s. Thus to establish that *psim* is a simulation relation, we only need to show that if ma is related to isa and ma' and isa' are states obtained by 1-step execution from ma and isa respectively, then ma' and isa' are related by *psim*. The approach to show this is shown pictorially in Fig. 9.3. Roughly, we show that if ma and isa are related, and ma_1 is the witnessing state for ma, then one can construct a witnessing state for ma' by running the pipeline for a sufficient number of steps from ma_1. In particular, ignoring the possibility of stalls or bubbles in the

[1]Note that ma and ma_{1F} would possibly have different values of the program counter. This is normally addressed by excluding the program counter from the *label*s of a state.

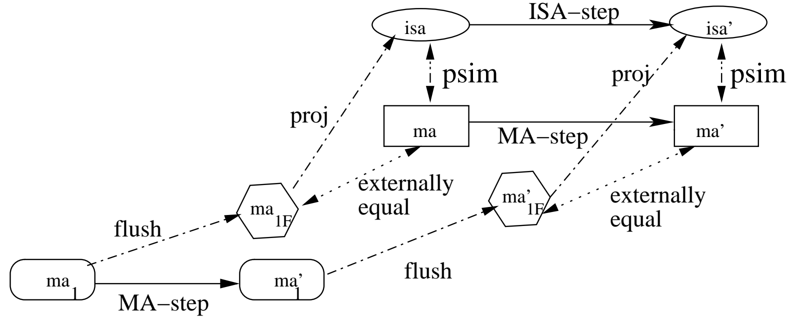

Figure 9.3: Using Flushing to Obtain a Refinement Theorem

pipeline, the state following ma_1, which is ma'_1, is a witnessing state of ma', by the following argument. Execution of the pipeline for one transition from ma completes i_1 and the next instruction, i_2, *after* i_1 in program order is poised to complete in state ma'. The witnessing state for ma' thus should be poised to initiate i_2 in the pipeline. And indeed, execution for one cycle from ma_1 leaves the machine in a state in which i_2 is poised to enter the pipeline. We will make this argument more precise in the context of an example in Section 9.4. Finally, the correspondence between ma'_1 and isa' follows by noting that **ma** states ma_1, ma'_1, and ISA states isa and isa' satisfy the flushing diagram of Figure 9.2.

The reader should note that the above proof approach depends on our determining the witnessing state ma_1 given ma. However, since ma_1 is a state that occurs in the "past" of ma, ma might not retain sufficient information for *computing* ma_1. In general, to compute witnessing states, one needs to define an intermediate machine to keep track of the history of execution of the different instructions in the pipeline. However, an interesting observation in this regard is that the witnessing state need not be constructively computed. Rather, we can simply define a predi-

cate specifying "some witnessing state exists". Skolemization of the predicate then produces a witnessing state.

In the presence of stalls, a state ma might not have *any* instruction poised to complete. Even for pipelines without stalls, no instruction completes for several cycles after the initiation of the machine. Correspondence in the presence of such bubbles is achieved by allowing finite stutter. In other words, if ma is related to isa, and ma has no instruction to complete, then ma' is related to isa instead of isa'. Since stuttering is finite, we can thus show correspondence between pipelined **ma** with finite stalls and **isa**.

9.3 A New Proof Rule

Since we want to use simulation correspondence with stuttering for reasoning about pipelined machines, let us first introduce this notion as a proof rule. We do so as follows. We say that $\mathcal{I}$ is a stuttering *simulation refinement* of $\mathcal{S}$, written $(\mathcal{S} \trianglerighteq_s \mathcal{I})$, if and only if there exist functions *psim*, *commit*, and *rank* such that the following are theorems:

1. $\mathit{psim}(\mathcal{I}.\mathit{init}(), \mathcal{S}.\mathit{init}())$
2. $\mathit{psim}(s, s') \Rightarrow \mathcal{I}.\mathit{label}(s) = \mathcal{S}.\mathit{label}(s')$
3. $\exists j : \mathit{psim}(s, s') \wedge \mathit{commit}(s.i) \Rightarrow \mathit{psim}(\mathcal{I}.\mathit{next}(s, i), \mathcal{S}.\mathit{next}(s', j))$
4. $\mathit{psim}(s, s') \wedge \neg\mathit{commit}(s, i) \Rightarrow \mathit{psim}(\mathcal{I}.\mathit{next}(s, i), s')$.
5. $\mathit{o\text{-}p}(\mathit{rank}(s))$
6. $\mathit{psim}(s, s') \wedge \neg\mathit{commit}(s, i) \Rightarrow \mathit{rank}(\mathcal{I}.\mathit{next}(s, i)) \prec \mathit{rank}(s)$

One should note that the notion of simulation refinements is a slight generalization of the well-founded refinements that we talked about in the last chapter. Indeed, by

specifying $\mathit{psim}(s, s') \triangleq (\mathit{inv}(s) \wedge \mathit{rep}(s) = s')$, and $\mathit{commit}(s, i) \triangleq \neg(\mathit{skip}(s, i))$ we can easily show that if $(\mathcal{S} \trianglerighteq \mathcal{I})$ then $(\mathcal{S} \trianglerighteq_s \mathcal{I})$. For turning flushing proofs of pipelined machines to refinement proofs, we find it convenient to prove ($\mathbf{isa} \trianglerighteq_s \mathbf{ma}$) rather than ($\mathbf{isa} \trianglerighteq \mathbf{ma}$) since we plan to use quantification and Skolemization to create the witnessing states as we discussed above, and such Skolem witnesses might not be unique, thus limiting our ability to be able to define the representative function *rep* as required for showing well-founded refinements. Nevertheless, it is not difficult to prove that simulation refinements imply stuttering trace containment as well. That is, the following proof rule can be formalized and verified in ACL2 (although not in closed form as indeed in the case of the other proof rules we talked about in the last chapter).

Simulation: Derive $(\mathcal{S} \triangleright \mathcal{I})$ from $(\mathcal{S} \trianglerighteq_s \mathcal{I})$

The proof of this rule follows the same concepts as the proof of well-founded refinements, namely to show that for every execution of $\mathcal{I}$ there is a *trace* of $\mathcal{S}$ with the same visible behavior up to stuttering. We do not talk about fairness here, since we have not needed it for reasoning about pipelines.

The reader might be troubled by our sudden introduction of a new proof rule. After all, in the last chapter we have devised quite a few rules already. Although undecidability of the underlying logic implies that we might have to extend the set sometimes, it is imperative that we have a robust and stable repertoire of rules for reasoning about a wide variety of systems in practice. In particular, we definitely do not want to add proof rules every time we want to verify a new reactive system. However, we point out here that we have introduced the **Simulation** rule *not* for verifying pipelined system implementations but rather to reason about certain styles of *proofs* about pipelines. Indeed, we believe that the collection of proof rules we

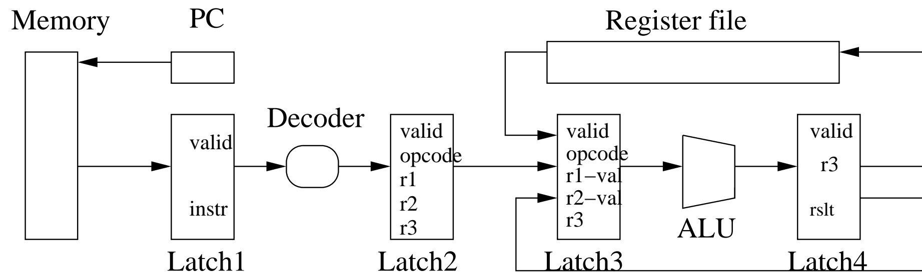

Figure 9.4: A Simple 5-stage Pipeline

developed in the last chapter are quite adequate in practice for reasoning about microarchitectures. In Chapter 14 when we verify a complex pipelined microprocessor design, we will require no other proof rules.

Incidentally, one of the advantages of using a universal and general notion of correspondence between two systems and formalizing the notion itself in the logic of a theorem prover is that we *can* do what we have just done, namely create and generalize proof rules when appropriate. In contrast to "ad hoc" refinement rules, such formalized proof rules give us the confidence in computing systems verified using them up to our trust in the notion of correctness and its formalization in the theorem prover.

9.4 Example

Let us now apply our new proof rule and the idea of witnessing function on a simple but illustrative pipeline example. The machine is the simple deterministic 5-stage pipeline shown in Figure 9.4. The pipeline consists of *fetch*, *decode*, *operand-fetch*, *execute*, and *write-back* stages. It has a decoder, an ALU, a register file, and 4

latches to store intermediate computations. Instructions are fetched from the memory location pointed to by the program counter (PC) and loaded into the first latch. The instructions then proceed in sequence through the different pipeline stages in program order until they complete. Every latch has a `valid` bit to indicate if the latch is non-empty. We use a 3-address instruction format consisting of an opcode, two source registers, and a target register. While the machine allows only in-order execution, data forwarding is implemented from latch 4 to latch 3. Nevertheless, the pipeline can still have a 1-cycle stall: If the instruction i_2 at latch 2 has as one of its source registers the target of the instruction i_3 at latch 3, then i_2 cannot proceed to latch 3 in the next state when i_3 completes its ALU operation.

The executions of the pipeline above can be defined as a *deterministic* reactive system **ma**. By *deterministic* we mean that **ma**.*next* is a function only of the current state. We assume that **ma**.*init*() has *some* program loaded in the memory, and *some* initial configuration of the register file, but an empty pipeline. Since we are interested in the updates of the register file, we let **ma**.*label* preserve the register file of a state. Finally, **isa** merely executes the instruction pointed to by PC atomically; that is, it fetches the correct operands, applies the corresponding ALU operation, updates the register file, and increments PC in one atomic transition.

Notice that we have left the actual instructions unspecified. Our proof approach does not require complete specification of the instructions but merely the constraint that the ISA performs analogous atomic update for each instruction. In ACL2, we use constrained functions for specifying the updates applied to instructions at every stage of the pipeline.

Our goal now is to show that ($\mathbf{isa} \trianglerighteq_s \mathbf{ma}$). We define *commit* so that

$\mathit{commit}(ma)$ is T if and only if latch 4 is `valid` in ma. Notice that whenever an **ma** state ma satisfies *commit*, some instruction is completed at the *next transition*. It will be convenient to define functions characterizing partial updates of an instruction in the pipeline. Consequently, we define four functions next_1, next_2, next_3, and next_4, where $\mathit{next}_i(ma, inst)$ "runs" instruction *inst* for the first i stages of the pipe and updates the i-th latch. For example, $\mathit{next}_2(ma, inst)$ updates latch 2 with the decoded value of the instruction. In addition, we define two functions, *flush* and *stalled*. Given a pipeline state ma, $\mathit{flush}(ma)$ returns the flushed pipeline state ma_F, by executing the machine for sufficient number of cycles without fetching any new instruction. The predicate *stalled* holds in a state ma if and only if both latches 2 and 3 are `valid` in ma, and the destination for the instruction at latch 3 is one of the sources of the instruction at latch 2. Notice that executing a pipeline from a *stalled* state does not allow a new instruction to enter the pipeline. Finally the function $\mathit{proj}(ma)$ projects the PC, memory, and register file of ma to the ISA.

Lemma 5 is a formal rendition of the flushing diagram for pipelines with stalls and can be easily proved using symbolic simulation.

Lemma 5 *For each pipeline state ma,*

$$\mathit{proj}(\mathit{flush}(\mathbf{ma}.\mathit{next}(ma))) = \begin{cases} \mathbf{isa}.\mathit{next}(\mathit{proj}(\mathit{flush}(ma))) & \mathit{if}\ \neg\mathit{stalled}(ma) \\ \mathit{proj}(\mathit{flush}(ma)) & \mathit{otherwise} \end{cases}$$

We now define *witnessing states*. Given states ma_1 and ma, ma_1 is a *witnessing state* of ma, recognized by the predicate $\mathit{witness}(ma_1, ma)$, if (1) ma_1 is not *stalled*, and (2) ma can be derived from ma_1 by the following procedure:

1. *Flush* ma_1 to get the state ma_{1F}.

2. Apply the following update to ma_{1F} for $i = 4, 3, 2, 1$:

   - If latch i of ma is `valid` then apply next_i with the instruction pointed to by the PC in ma_{1F}, correspondingly update latch i of ma_{1F}, and advance PC; otherwise do nothing.

We now define predicate *psim* as follows:

$$\mathit{psim}(ma, isa) \triangleq (\exists ma_1 : (\mathit{witness}(ma_1, ma) \wedge (isa = \mathit{proj}(\mathit{flush}(ma_1)))))$$

Now we show how the predicate *psim* can be used to prove $(\mathbf{isa} \trianglerighteq_s \mathbf{ma})$. First, the function *rank* satisfying conditions 5 and 6 can be defined by simply noticing that for any state ma, *some* instruction always advances. Hence if i is the maximum number in MA such that latch i is `valid`, then the quantity $(5 - i)$ always returns a natural number (and hence an ordinal) that decreases at every transition. (We take i to be 0 if ma has no `valid` latch.) Also, $\mathit{witness}(\mathbf{ma}.\mathit{init}(), \mathbf{ma}.\mathit{init}())$ holds, implying condition 1. Further, condition 2 is trivial from definition of *witness*. We therefore focus here on only conditions 3 and 4. We first consider the following lemma:

Lemma 6 *For all ma, ma_1 such that $\mathit{witness}(ma_1, ma)$:*

- $\neg \mathit{commit}(ma) \Rightarrow \mathit{witness}(ma_1, \mathbf{ma}.\mathit{next}(ma))$
- $\mathit{commit}(ma) \wedge \mathit{stalled}(\mathbf{ma}.\mathit{next}(ma_1)) \Rightarrow$
  $\mathit{witness}(\mathbf{ma}.\mathit{next}(\mathbf{ma}.\mathit{next}(ma_1)), \mathbf{ma}.\mathit{next}(\mathit{ma}))$
- $\mathit{commit}(ma) \wedge \neg \mathit{stalled}(\mathbf{ma}.\mathit{next}(ma_1)) \Rightarrow \mathit{witness}(\mathbf{ma}.\mathit{next}(ma_1), \mathbf{ma}.\mathit{next}(ma))$

The lemma merely states that if ma is not a *commit* state, then stepping from ma preserves the witnessing state, and otherwise the witnessing state for $\mathbf{ma}.\mathit{next}(ma)$ is given by the "next non-stalled state" after ma_1. The lemma can be proved by

symbolic simulation on **ma**. We can now prove the main technical lemma that guarantees conditions 3 and 4 for simulation refinement.

Lemma 7 *For all ma and isa, such that* $\mathit{psim}(ma, isa)$*:*

1. $\neg \mathit{commit}(ma) \Rightarrow \mathit{psim}(\mathbf{ma}.\mathit{next}(ma), isa)$

2. $\mathit{commit}(ma) \Rightarrow \mathit{psim}(\mathbf{ma}.\mathit{next}(ma), \mathbf{isa}.\mathit{funcnext}(isa))$

Proof sketch: Let ma_1 be the Skolem witness of $\mathit{psim}(ma, isa)$. Case 1 follows from lemma 6, since $\mathit{witness}(ma_1, \mathbf{ma}.\mathit{next}(ma))$ holds. For case 2, we consider only the situation $\neg \mathit{stalled}(\mathbf{ma}.\mathit{next}(ma_1))$ since the other situation is analogous. But by lemma 5 and definition of *witness*, $\mathit{proj}(\mathit{flush}(\mathbf{ma}.\mathit{next}(ma_1))) = \mathbf{isa}.\mathit{next}(isa)$. The result now follows from lemma 6. ■

The reader might think, given our definition of *witness* and the lemmas we needed to prove simulation refinement, that using this notion requires much more manual effort than a simple flushing proof. In practice, however, the function *witness* can be defined more or less mechanically from the structure of the pipeline. We will briefly discuss how to define witnessing states and prove refinement for advanced pipelines in the next section. Furthermore, as in the case of a flushing proof, all the lemmas above are provable by symbolic simulation of the pipeline. By doing these proofs we can now compose proofs of pipelines with other proofs on **isa** by noting that the notion of trace containment is hierarchically composible.

We end this section with a brief comparison between our approach and that of Manolios [Man00a]. Manolios shows that certain types of flushing proofs are flawed, and uses WEBs to prove the correctness of pipelines. We focus on a comparison with this work since unlike other related work, this approach uses a uniform notion of

correctness that is applicable to both pipelined and non-pipelined machines. Indeed our use of stuttering simulation is a direct consequence of this work. The basic difference is in the techniques used to define the correspondence relation to relate **ma** states with **isa** states. While we define a quantified predicate to posit the existence of a witnessing state, Manolios defines a refinement map from the states of **ma** to states of **isa** as follows: Point the PC to the next instruction to complete and invalidate all the instructions in the pipe.[2] He calls this approach the *commitment rule*. Notice immediately that the method requires that we have to keep track of the PC of each intermediate instruction. Also, as the different pipeline stages update the instruction, one needs some invariant specifying the transformations on each instruction at each stage. Short of computing such invariants manually based on the structure of the pipeline and the functionality of the different instructions, we believe that any generic approach to determine invariants will reduce his approach to ours, namely defining a quantified predicate to posit the existence of a witnessing state.

What about the flaws with flushing proofs that were discovered by Manolios? One of the problems he points out is that a trivial machine that does not do anything satisfies flushing proofs. Unfortunately since we are using simulations, this problem remains with our approach. More precisely, simulation refinement (and trace containment) guarantee that for every execution of the implementation there is an execution of the specification that has the same observation up to finite stutter. Thus if we consider a trivial machine that has no non-stuttering execution, our notion of

[2]Manolios also describes a "flushing proof" and shows that flushing can relate inconsistent MA states to ISA states. But our application of flush is different from his in that we flush ma_1 rather than the current state ma. In particular, our approach does not involve a refinement map constituting the *flush* of **ma**.

correctness can always be used to prove that it is a refinement of *any* system. Nevertheless, in practice, this is not a difficulty with our notion of correspondence since it is usually easy to prove that the implementation is non-trivial. Indeed, in recent work [Man03], Manolios concedes that stuttering simulation, though undoubtedly weaker than WEBs is a reasonable notion of correctness of reactive systems. The same argument goes to trace containment. He, however, points out two other problems with flushing proofs, which are both avoided in our approach. The first is that it is possible for **ma** to deadlock. We do not have this problem since flushing proofs are applied in our approach to prove trace containment using witnessing states. The predicate *witness* cannot be to be admissible in ACL2 when the machine deadlocks. We will see one instance of this in the next section when we show how to define *witness* in the presence of multiple-cycle stalls. The second concern with flushing is that it is possible to relate inconsistent **ma** states with consistent **isa** states. He shows that as follows. He defines a function *rep* mapping **ma** states to **isa** states as $\mathit{rep}(ma) \triangleq \mathit{proj}(\mathit{flush}(ma))$, and uses this function to prove WEB. However, notice that this function does *not* preserve the *label*s, and thus does not satisfy our notion of correspondence. By flushing from a different state, namely ma_1, we avoid the problem of inconsistency. Indeed, the formalization of trace containment guarantees that such inconsistency does not arise.

9.5 Advanced Features

Our example above is illustrative, but trivial. The pipeline was a simple straight-line pipe with no interrupts, exceptions, and out-of-order executions. Can we use stuttering trace containment to reason about pipelines with such features and can

we turn the derivation of some form of flushing diagrams into refinement proofs? We now explore these questions by considering some of these features.

9.5.1 Stalls

The pipeline in Fig. 9.4 allowed single-cycle stalls. Pipelines in practice can have stalls ranging over multiple cycles. If the stall is finite, it is easy to use stuttering simulation to reason about such pipelines. Stalls affect lemma 6 since given a witnessing state ma_1 of ma, the witnessing state for $\mathbf{ma}.\mathit{next}(ma)$ is given by the "next non-stalled state" after ma_1. But such a state is can be determined by executing **ma** for $(\mathit{clk}(\mathbf{ma}.\mathit{next}(ma_1)) + 1)$ steps from ma_1, where the function clk (defined below) merely counts the number of steps to reach the first non-stalled state. Finiteness of stalls guarantees that the function terminates.

$$\mathit{clk}(s) \triangleq \begin{cases} 0 & \mathit{if}\ \neg \mathit{stalled}(s) \\ 1 + \mathit{clk}(\mathbf{ma}.\mathit{next}(s)) & \text{otherwise} \end{cases}$$

9.5.2 Interrupts

Modern pipelines allow interrupts and exceptions. To effectively reason about interrupts, we model both MA and **isa** as non-deterministic machines, where the "input argument" is used to decide whether the machine is to be interrupted at the current step or proceeds with normal execution. Recall that our notion of correspondence can relate non-deterministic machines. Servicing the interrupt might involve an update of the visible components of the state. In the **isa**, we assume that the interrupt is serviced in one atomic step, while it might take several cycles in **ma**.

We specify the witnessing states for an interruptible **ma** state as follows. ma_1 is a witnessing state of ma if either (1) ma is not within any interrupt and ma_1 initiates the instruction next to be completed in ma, or (2) ma is within

some interrupt and ma_1 initiates the corresponding interrupt. Then *commit* holds if either the current transition returns from some interrupt service or completes an instruction. Assuming that pipeline executions are not interleaved with the interrupt processing, we can then show ($\mathbf{isa} \trianglerighteq_s \mathbf{ma}$) for such non-deterministic machines. We should note here that we have not analyzed machines with nested interrupts yet. But we believe that the methodology can be extended for nested interrupts by the witnessing state specifying the initiation of the most recent interrupt in the nest.

9.5.3 Out-of-order Execution

The use of witnessing states can handle pipelines with out-of-order instruction execution as long as the instructions are initiated to the pipeline and completed in program order. For a pipeline state ma, we determine the instruction i_1 that is next to be completed, by merely simulating the machine forward from ma. We then specify ma_1 to be a witnessing state of ma if i_1 is initiated into the pipeline at ma_1. Notice that since instructions are initiated and completed in-order, any instruction before i_1 in program order must have been already initiated in the pipeline in state ma_1. Since flushing merely involves executing the pipeline without initiating any instruction, flushing from state ma_1 will therefore produce the state ma_{1F} with the same visible behavior as state ma.

9.5.4 Out-of-order and Multiple Instruction Completion

Some modern pipelines allow completion of multiple instructions at the same clock cycle, and out-of-order completion of instructions. Such features cannot be directly handled by stuttering trace containment if **isa** is chosen to be the machine that

sequentially executes instructions one at a time. In this section, we outline the problem and discuss a possible approach. We admit, however, that we have not attempted to apply the outlined approach in the verification of actual systems and thus our comments here are merely speculative.

Consider a pipeline state *ma* poised to complete two instructions i_1 and i_2 at the same cycle. Assume that i_1 updates register r_1 and i_2 updates register r_2. Thus the visible behavior of the pipeline will show simultaneous updates of the two registers. The **isa**, however, can only update one register at any clock cycle. Thus, there can be *no* **isa** state *isa* with the properties that (1) *ma* and *isa* have the same *label*s, and (2) executing both machines from *ma* and *isa* results in states that have the same *label*s, even with possible stutter. In other words, there can be no (stuttering) simulation relation relating *ma* and *isa*. The arguments are applicable to out-of-order completion as well, where the updates corresponding to the two instructions are "swapped" in the pipelined machine.

One might think that the issues above arise only in superscalar architectures. Unfortunately, that is not true. Even with pipelines that are essentially straight lines this issue can arise when the pipeline is involved in the update of both the register file and the memory. The reason is that these two updates usually occur on two different pipeline stages. Thus it is possible for a memory update instruction to update the memory before or simulataneously with a *previous* instruction updating the register file. If both the memory and the register file are in the *label* of the two systems then the situation is exactly the same as the scenario described above. Indeed, we will see in some detail how this scenario causes problems with refinement proofs in the context of a more elaborate example in Chapter 14.

Since multiple and out-of-order completion affect the visible behavior of the pipelined machine, we need to modify the execution of the ISA to show correspondence. We propose the following approach: The ISA, instead of executing single instructions atomically, non-deterministically selects a *burst* of instructions. Each *burst* consists of a set of instruction sequences with instructions in *different* sequences not having any data dependency. The ISA then executes each sequence in a burst, choosing the order of the sequences non-deterministically, and then selects the next burst after completing all sequences. Notice that our "original" **isa** can be derived from such an ISA by letting the bursts be singleton sets of one instruction.

9.6 Summary

We have shown how one can turn flushing proofs of pipelined microarchitectures to proofs of stuttering trace containment using simulation refinement. The method requires construction of a predicate determining the witnessing state for **ma**, and Skolemization of the predicate gives us the simulation relation. The proofs can be conducted by symbolic simulation of the pipeline.

It should be noted that we have not done anything to simplify the flushing proofs themselves. We have only shown that if one can prove certain types of flushing theorems then it is possible to construct a (stuttering) refinement proof of the pipelined machine more or less mechanically. Nevertheless, this allows us to use a generic notion of correctness to reason about pipelined machines while still benefiting from the scalability often afforded by flushing. In particular, the use of witnessing states and the extra proof obligations introduced to show simulation guarantee that some deficiencies of flushing approaches can be alleviated without

much effort. Furthermore, witnessing states can be used to reason about many of the features of modern pipelines.

We will reiterate here that one of the key reasons for the applicability of our methods is the use of quantification. As in the previous part, we saw extensive use of quantification and Skolemization in this part both in reasoning about different proof rules, and, in particular, in the case of pipelines for defining simulation relations. One of the places in which quantification is really useful is the so-called "backward simulation". Notice that the witnessing states occur in the past and thus one must be simulating the machine backwards somehow to determine such states. A constructive approach for finding such a state would require keeping track of the history of execution. But we manage to simply define a predicate that specifies when ma_1 might be considered a witnessing state of **ma**. The definition of the predicate, then, requires only "forward simulation" from ma_1. Quantification then does the rest, by allowing us to posit that some such ma_1 exists.

9.7 Bibliographic Notes

Reasoning about pipelines is an active area of research. Aagaard *et al.* [ACDJ01] provide an excellent survey and comparison of the different techniques. Some of the early studies have used *skewed abstraction functions* [SB90, BT90] to map the states of the pipeline at different moments to a single **isa** state. Burch and Dill [BD94] introduced the idea of *flushing* to reason about pipelines. This approach has since been widely used for the verification of pipelined machines. Brock and Hunt use this notion to verify the pipeline unit of the Motorola CAP DSP [BH99]. Sawada uses a variant, called *flush point alignment* to verify a 3-stage pipelined machine [Saw00].

The same approach has been used by Sawada and Hunt [SH97, SH98, SH02] to verify a very complicated pipelined microprocessor with exceptions, stalls, and interrupts. The key to this approach has been the use of an intermediate data structure called MAETT, that is used for keeping track of the history of execution of the different instructions through the pipeline. Hosabettu *et al.* [HGS00] use another variant variant of the Burch and Dill approach, namely *flush point refinement*, to verify a deterministic out-of-order machine using *completion functions*. Recently techniques have been proposed for combining completion functions with equivalence checking to verify complicated pipelines [ACJTH04]. Both flush point alignment and flush point refinement require construction of complicated invariants to demonstrate correlation between the pipelined machine and the **isa**.

There have also been compositional model checking approaches to reason about pipelines. For example, Jhala and McMillan [JM01] use symmetry, temporal case-splitting, and data-type reduction to verify out-of-order pipelines. While more automatic than theorem proving, applicability of the method in practice requires the user to explicitly decompose the proof into manageable pieces to alleviate state explosion. Further, it relies on symmetry assumptions which are often violated by the heterogeneity of modern pipelines. Finally, there has been work on using a combination of decision procedures and theorem proving to verify modern pipelines [BGV99, LB03], whereby decision procedures have been used with lightweight theorem proving to verify state invariants.

Manolios [Man00a] shows logical problems with the Burch and Dill criterion as a notion of correctness. He uses WEBs to reason about variants of Sawada's 3-stage pipeline. This work, to our knowledge, is the first attempt in reasoning

about pipelines using a general-purpose correctness notion expressing both safety and liveness properties. We provided a brief comparison of our approach with that of Manolios in Section 9.4. Manolios and Srinivasan [MS04, MS05] present ways of automating WEB proofs of pipelines by translating them to formulas in UCLID [BLS02].

The work reported in this chapter is joint research with Warren A. Hunt Jr., and the presentation is adapted with his permission from a previous paper on the subject [RH04].

Part IV

Invariant Proving

Chapter 10

Invariant Proving

In the last part, we saw how a notion of refinements can be used to reason about reactive systems using theorem proving. A hurdle in application of stuttering refinements, as we saw in the examples in Chapter 8, was in the definition and proof of appropriate inductive invariants. In this part, we will therefore explore ways of facilitating invariant proofs.

A unary predicate *good* is said to be an invariant for a reactive system if and only if it holds for every reachable state of the system. More precisely, let $\mathcal{I}$ be a reactive system, and *stimulus* be a fixed uninterpreted unary function specifying a sequence of inputs to system $\mathcal{I}$. Then *good* is an *invariant* if and only if $\mathit{good}(\mathcal{I}.\mathit{exec}[\mathit{stimulus}](n))$ is a theorem. Throughout this part, we will write $\mathcal{I}.\mathit{exec}(n)$ as a shorthand for $\mathcal{I}.\mathit{exec}[\mathit{stimulus}](n)$. The goal of *invariant proving* is to show that the predicate *good* is an invariant.

Let us first understand the relations between the definition of invariant as shown above with our discussions on invariants in Chapter 8. There we said that one could prove that $\mathcal{I}$ is a well-founded refinement of $\mathcal{S}$ if one could prove the single-

step obligations **SST1-SST3**, **SST6**, and **SST7** in page 139, (and the corresponding fairness obligations) by replacing *inv* with predicate *good* if we can then prove that the following formulas are theorems.

RI1: $\mathit{inv}(\mathcal{I}.\mathit{init}())$

RI2: $\mathit{inv}(s) \Rightarrow \mathit{inv}(\mathcal{I}.\mathit{next}(s, i))$

RI3: $\mathit{inv}(s) \Rightarrow \mathit{good}(s)$

It is easy to see that *good* can be proved to be an invariant (based on the definitions above) if **RI1-RI3** hold. Then *inv* is called an *inductive invariant* strengthening *good*. In general, we can use *good* instead of *inv* in the conditions for well-founded refinements if *good* is an invariant. The proof of invariance of a predicate *good* by exhibiting an inductive invariant strengthening it is often referred to as an *inductive invariant proof*.

Devising techniques for proving invariants of reactive systems, of course, has been of interest to the formal verification community for a long time independent of our framework. Proofs of *safety properties*, that is, those that say that the "system does not do anything bad" [Lam77], essentially amount to the proof of an invariant. Even for proving liveness properties, that is, those that say that "the system eventually does something good", one often requires some auxiliary invariance conditions. Our way of verifying reactive systems, by showing correspondence based on stuttering trace containment with a simple specification, can be recognized as an approach for showing both safety and liveness properties together. It is therefore not surprising that invariants form a big stumbling block for the efficacy of our methods.

Proving invariants is known to be a difficult problem. The theorem proving

approach to doing this is what we have been talking all along, namely finding an inductive invariant proof of the invariance of the predicate concerned. The difficulty of this approach should by now be obvious to the reader. Inductive invariants also suffer from the problem of scalability. In particular, the definition of *inv* is brittle. Consider for example the **esi** system. The description of the system as we showed in Chapter 8 is of course wholly impractical. A more practical implementation might be one in which reading of a cache line is not atomic but proceeds for a number of transitions. This and other elaborate implementations of **esi** exist, and in each case the predicate *good* as we showed is still an invariant; after all, it was just saying that the caches are coherent. But the inductive invariant *inv* changes drastically. For instance if the reading of a cache line is not atomic then *inv* must keep track, at each state, how much of the reading has been done, which portions of the block can be locked and unlocked, and so on. What this implies is that as the system design evolves the inductive invariant proofs might require extensive modification to keep up with the design changes.

In the case where the system of interest has a finite number of states, however, there is another approach possible. We can just check algorithmically if all the reachable states of the system do satisfy *good*. Reachability analysis forms the *model checking* approach for proving invariants of finite-state systems. In LTL, an invariant is typically written as a property of the form $\mathsf{G}\,good$. Of course here we have to think of the system as a Kripke Structure and assume that the set $\mathcal{AP}$ of atomic propositions involved are expressive enough so that *good* can be expressed as a Boolean combination of the elements of this set. When reachability analysis succeeds, we therefore say that the formula $\mathsf{G}good$ holds for the system. The

model checking approach, when it is applicable, of course, is completely automatic with the additional advantage of being able to find a counterexample when the check fails. However, as is usual in the context of model checking, the problem lies with *state explosion*. While many abstraction techniques have been developed to improve the scalability of this approach, practical application of such techniques needs restrictions on the kind of systems that they can handle and properties that can be proved as invariants.

In this part, we will explore an approach to bridge the gap of automation between theorem proving and model checking for the proof of invariants, *without* limiting the kind of system that can be analyzed. The goal of our method is to derive abstractions of system implementations using *term rewriting*. The abstractions that we produce are so-called *predicate abstractions* [GS97], which let us reduce the invariant proving problem on the original system to a reachability analysis of a finite graph. Rewriting is guided by a collection of *rewrite rules*. The rewrite rules are taken from theorems proven by a theorem prover.

The reader might ask at this point why we need a separate procedure for proving invariants of reactive systems. After all, the problem with inductive invariants is not new to us. We faced the same problem with *step invariants* for sequential programs in Part II. There we managed to avoid the problem of defining inductive invariants by doing symbolic simulation given assertions at certain cutpoints. Why can we not try to extend that approach for reactive systems? Our short answer is that we have so far not found an effective way of achieving such extensions. The reason is principally in the non-deterministic nature of reactive systems. The reason that symbolic simulations "worked" for sequential programs is that the *cutpoint*s

were relatively few. Recall that we had to manually add *assertion*s to *cutpoint*s for our symbolic simulation to work. If for example every state (and hence every value of the program counter) corresponded to a *cutpoint* then the approach would have done us little good. Yet in reactive systems that is exactly the scenario we face. For example consider a multiprocess system where each process is executing the same sequential program over and over again. If there were only one process then we would have chosen the loop tests and entry and exit points of procedures as *cutpoint*s. But what do we choose for multiprocess systems? For every state of the system, there are several possible ways the control can reach that state, namely via different interleaving execution of the different processes. The "best" we can hope for by symbolic simulation is to collapse the steps of the processes where the only updates are on local state components of the process concerned. But this still leaves us in most cases with an inordinate number of *cutpoint*s at which we need to attach assertions. Of course there has been significant work on the use of VCGs for non-deterministic programs, but we could not adapt them suitably for our work [DLNS98]. Thus we want to explore more aggressive approaches for automating invariant proofs.

To understand our method, we will first briefly review what predicate abstractions are and how generating them leads to invariant proofs. We will then outline our approach and see how it benefits from the use of both theorem proving and model checking. The ideas discussed in this chapter will be expanded upon in the next chapter.

10.1 Predicate Abstractions

The concept of *predicate abstractions* was proposed by Graf and Saidi [GS97], although the basic approach is an instance of the idea of *abstract interpretation* introduced by Cousot and Cousot [CC77]. Suppose we have a system $\mathcal{I}$ that we want to analyze. In the predicate abstractions approach, first a finite number of predicates is defined over the states of $\mathcal{I}$. These predicates are then used to define a finite-state system $\mathcal{A}$ that can serve as an abstraction of $\mathcal{I}$.

How does this all work? Assume that we are given a set $\mathcal{P} \doteq \{P_0, \ldots, P_{m-1}\}$ of predicates on the states of a system $\mathcal{I}$. Suppose then that we can construct a collection of functions N_j that stipulate how each predicate is "updated" in system $\mathcal{I}$. That is, suppose we can construct N_j such that the following is a theorem for $j \in \{0, \ldots, m-1\}$.

A1: $N_j(P_0(s), \ldots, P_{m-1}(s), i_0, \ldots, i_l) \Rightarrow P_j(\mathcal{I}.\mathit{next}(s, i))$

Then we can construct a finite-state system $\mathcal{A}$ as follows.

- A state of $\mathcal{A}$ consists of m components (or an m-tuple). Each component can have the value `T` or `NIL`.
- The m-tuple $\langle b_0, \ldots, b_m \rangle$ is the state $\mathcal{A}.\mathit{init}()$ if and only if $b_j = P_j(\mathcal{I}.\mathit{init}())$.
- Given a state $\overline{s} \doteq \langle a_0, \ldots, a_{m-1} \rangle$ of $\mathcal{A}$, and an input $\overline{i} \doteq \langle i_0, \ldots, i_l \rangle$, the j-th component of $\mathcal{A}.\mathit{next}(\overline{s}, \overline{i})$ is given by $N_j(a_0, \ldots, a_{m-1}, i_0, \ldots, i_l) \Leftrightarrow$ `T`.
- The *label* of a state $\overline{s}$ is the set of components in $\overline{s}$ that have the value `T`.

The system $\mathcal{A}$ is referred to as a *predicate abstraction* of $\mathcal{I}$.

What has predicate abstraction got to do with invariant proofs? Assume that the predicate *good* which we want to establish to be an invariant on system $\mathcal{I}$ is one of the predicates in $\mathcal{P}$. In fact, without loss of generality, assume $P_0 \doteq \mathit{good}$. It is easy

to prove that *good* is an invariant of $\mathcal{I}$ if in every state p in $\mathcal{A}$ reachable from *A.init*(), the 0-th component of p is `T`. But since $\mathcal{A}$ has only a finite number of states, this question can now be answered by reachability analysis! Thus proofs of invariance of unbounded state systems can be reduced by predicate abstraction to the model checking of a finite state system. If the number of predicates in $\mathcal{P}$ is small, or at least the number of reachable abstract states is small, then predicate abstraction provides a viable approach to invariant proving. Notice however that the results from such a check can only be *conservative* in the sense that if the reachability analysis succeeds then *good* is an invariant but if it fails then nothing can be formally claimed about the invariance of *good*. Indeed, one can choose $\mathcal{P} \doteq \{good\}$ and $N_0(s) \triangleq \texttt{NIL}$ to create a "bogus" abstract system $\mathcal{A}_B$ which is a predicate abstraction of any system $\mathcal{I}$ according to the above conditions but is useless for proving invariance of *good*.

As should be understood from above, the basic idea of predicate abstraction is very simple. There are two chief questions to be answered in order to successfully use predicate abstractions in practice. These are the following:

Discovery How do we effectively obtain the set of relevant predicates $\mathcal{P}$?

Abstraction Given $\mathcal{P}$, how do we construct the abstract transition function?

Let us take the **abstraction** step first. Thus suppose one has a set $\mathcal{P} \doteq \{P_0, \ldots, P_{m-1}\}$ of m predicates. Then one considers all possible 2^m evaluations of these predicates. For each pair of evaluations $b \doteq \langle b_0, \ldots, b_{m-1}\rangle$ and $b' \doteq \langle b'_0, \ldots, b'_{m-1}\rangle$, one then asks the following question. If for some state s of the implementation $\mathcal{I}$ the predicates in $\mathcal{P}$ have the values as specified by b, then is there some i such that in $\mathcal{I}.next(s, i)$ the predicates have the values induced by b'? If the answer is yes, then one can add b' as one of the successors of b in $\mathcal{A}$. In practice, the answer is typically obtained

by theorem proving [GS97, SS99, FQ02, DDP99]. Some other techniques have been recently devised to make use of techniques based on Boolean satisfiability checking. In particular, an approach due to Lahiri and Bryant [LBC03] that has been employed with the UCLID verification tool represents the abstract transition functions and state space symbolically using BDDs and efficiently computes the successors of sets of states using efficient algorithms for Boolean satisfiability checking.

Predicate **discovery** is a more thorny issue. Many techniques for predicate discovery follow a paradigm often known as the *abstraction refinement* paradigm. That is, one starts with a small set of predicates, initially possibly only containing the proposed invariant *good*. One then computes an **abstraction** based on this small set, and applies model checking on the abstract system. If the model checking fails, one attempts to augment the set of predicates based on the counterexample returned. This general approach has been applied in different forms in practice [DD02, CGJ$^+$00] for generating effective predicate abstractions incrementally. Another promising approach, developed by Namjoshi and Khurshan [NK00] involves *syntactic transformations* to generate the predicates iteratively on the fly. In this approach, one starts with an initial pool of predicates, and creates a "Boolean program" in which the predicates are represented as variables. The updates to these variables are specified by a computation of the weakest precondition of the corresponding predicate over each program action. This computation might produce new predicates which are then added to the pool to be examined in the next iteration. Our approach to predicate abstractions is based on this method and we will provide fuller comparison with it when discussing our procedure in the next chapter.

10.2 Discussions

Given the success and popularity of predicate abstractions, it makes sense to ask if we can make use of such work with ACL2 in our framework to automate the discovery and proof of invariants. Our applications, however, are different from most other domains where the method has been applied. In most of the other work, the predicates were either simple propositions on the states of the implementation, or built out of such propositions using simple arithmetic operations. While recent work by Lahiri and Bryant [LB04b] and Qadeer and Flanagan [FQ02] allow some generality, for example allowing quantified predicates over some *index variables*, the language for representing predicates has always been restricted. This is natural, since the focus has been on designing techniques for automating as much of the predicate discovery as possible. While the work of Graf and Saidi [GS97] and other related methods used theorem proving as a component for predicate abstraction, this component was used typically for the **abstraction** rather than the **discovery** phase.

However, we do want to preserve the ability of defining predicates which can be arbitrary recursive functions admissible in ACL2. This is important since we want predicate abstractions to mesh with the strengths of theorem proving, namely expressive logic and consequent succinctness of definitions. We also want an approach that can be configured to reason about *different* reactive systems that can be formally modeled with ACL2. As a consequence of these goals, we must rely on theorem proving approaches for predicate discovery rather than design procedures that work with restricted models and properties.

How do we achieve such goals? While the predicates (and indeed, systems

that we define) are modeled using arbitrary recursive functions in theorem proving, one usually makes disciplined use of these functions and often applies them over and over again in building different systems. For example, we have used functions defining operations on sets and finite records to model all the systems in Chapter 8. The user of a theorem prover usually writes libraries of lemmas and theorems that help in simplifying the terms that arise during a proof. These lemmas are often generic facts about the different functions used in the definition of the system and its properties. Thus if two different systems are modeled with the same functions, the same library of lemmas can be used to reason about them. In designing predicate abstractions, we wish to leverage these generic lemmas to "mine" the useful predicates.

Our approach is to use term rewriting on the next state function of the implementation to determine the appropriate predicates. The rewriting is governed by *rewrite rules* which are taken from theorems proven in ACL2. The user can control and extend the rewriting by proving additional theorems. By depending on rules rather than on built-in heuristics, the same procedure can be used for proving invariants of many different systems by simply supplying different rules. Of course, writing theorems about functions defining a reactive system and its properties involves careful understanding of the functions involved. But as we will see, most of the rules that we need are generic theorems, already available in a deductive setting. While in some cases some "system specific" rules are necessary, the concepts behind such rules can be usually reused for similar systems. We will see this in reasoning about two different cache coherence protocols in the next chapter.

10.3 An Illustrative Example

So what is our procedure? We will provide a technical description in the next chapter, but here we present a trivial but illustrative example. Consider a reactive system consisting of two components `C0` and `C1`, which initially both contain 0 and are updated at each transition as follows.

- If the external stimulus i is `NIL` then `C0` gets the previous value of `C1`; otherwise `C0` is unchanged.
- If i is `NIL` then `C1` is assigned the value `42`; otherwise `C1` is unchanged.

This system can be easily modeled as a reactive system in ACL2. Assume that $\mathit{C0}(s)$ returns the value of the component `C0` in state s and $\mathit{C1}(s)$ returns the value of the component `C1`. Given this system, consider proving that the component `C0` always contains a natural number. This means that we must prove that the predicate $\mathit{natp}(\mathit{C0}(s))$ is an invariant. It should be clear, however, that since the value of `C0` sometimes depends on the previous value of `C1`, $\mathit{natp}(\mathit{C0}(s))$ is not an inductive invariant. An appropriate inductive invariant for this system is given by:

$$\mathit{inv}(s) \triangleq \mathit{natp}(\mathit{C0}(s)) \wedge \mathit{natp}(\mathit{C1}(s))$$

Thus to obtain an inductive invariant we need to "discover" this new predicate $\mathit{natp}(\mathit{C1}(s))$.

We will now see how our method handles this simple system. For technical reasons, our procedure works with functions of "time" rather than states and inputs. However, it is easy to obtain from any system, a collection of equations that are theorems describing the behavior of the system at time n. For a system $\mathcal{I}$, and a state component `C`, assume that $\mathit{C}(s)$ returns the value of `C` in state s. Then we will define $\widehat{\mathit{C}}(n) \triangleq \mathit{C}(\mathcal{I}.\mathit{exec}[\mathit{stimulus}](n))$. Recall that $\mathit{stimulus}$ is a fixed uninterpreted

1. $\widehat{C0}(0) = 0$

2. $\widehat{C1}(0) = 0$

3. $\widehat{C0}(n+1) = \begin{cases} \widehat{C0}(n) & \textit{if } \neg \textit{stimulus}(n) \\ \widehat{C1}(n) & \text{otherwise} \end{cases}$

4. $\widehat{C1}(n+1) = \begin{cases} 42 & \textit{if } \neg \textit{stimulus}(n) \\ \widehat{C1}(n) & \text{otherwise} \end{cases}$

Figure 10.1: Equations showing the transitions of the Two Component System

function stipulating an arbitrary infinite sequence of inputs to the system $\mathcal{I}$. With these conventions, equations **1-4** in Figure 10.1 shows the equations which stipulate the behavior of our example system as a function of time. Writing $P_0 \doteq \textit{natp}(\widehat{C0}(n))$, we note that the invariance problem above is equivalent to proving that P_0 is an invariant. For the purpose of our procedure, we will assume that the predicates we are dealing with all contain one variable n.

Let us now see how rewriting can discover predicates. Consider rewriting $\textit{natp}(\widehat{C0}(n+1))$ using equation **3** along with the following equation **5** which is a generic theorem about *natp* and *if*.

5. $\textit{natp}(\textit{if}(x, y, z)) = \textit{if}(x, \textit{natp}(y), \textit{natp}(z))$

For the purpose of rewriting, we will always assume that the equations are oriented from left to right. Since all the equations used in rewriting are theorems, it follows that if rewriting a term τ produces a term τ' then $\tau = \tau'$ is a theorem.[1] It is easy to see that rewriting $\textit{natp}(\widehat{C0}(n+1))$ yields the following term.

$$\textbf{T0}: \quad \textit{if}(\textit{stimulus}(n+1), \textit{natp}(\widehat{C0}(n)), \textit{natp}(\widehat{C1}(n)))$$

[1] In a strict sense what we can derive is $\tau \Leftrightarrow \tau'$. This distinction is not important for our discussion since we are interested here in predicates.

We will treat this term **T0** as a Boolean combination of terms $\mathit{stimulus}(n+1)$, $\mathit{natp}(\widehat{\mathit{C1}}(n))$, and $\mathit{natp}(\widehat{\mathit{C0}}(n))$. Let us decide to abstract the term $\mathit{stimulus}(n+1)$ and explore the term $P_1 \doteq \mathit{natp}(\widehat{\mathit{C1}}(n))$ as a new predicate. By *exploring*, we mean that we will replace n by $(n+1)$ in the term and apply rewriting. We will come back to our reasons for abstracting $\mathit{stimulus}(n+1)$ later when we discuss the details of our procedure. But for now, note that rewriting $\mathit{natp}(\widehat{\mathit{C1}}(n+1))$ using equations **4** and **5**, along with the computed fact $\mathit{natp}(42) = \mathtt{T}$ then yields the following term:

$$\textbf{T1}: \quad \mathit{if}(\mathit{stimulus}(n+1), \mathit{natp}(\widehat{\mathit{C1}}(n)), \mathtt{T})$$

We can treat the terms **T0** and **T1** as specifying how the predicates P_0 and P_1 are "updated" at every instant. We can now create our finite-state abstract system $\mathcal{A}$ very simply as follows.

- The states of the system are pairs of Booleans. Thus the system has 4 states.
- The initial state is the pair $\langle \mathtt{T}, \mathtt{T} \rangle$, given by the evaluation of the predicates P_0 and P_1 at time 0, that is, the values of $\mathit{natp}(\widehat{\mathit{C0}}(0))$ and $\mathit{natp}(\widehat{\mathit{C1}}(0))$.
- The updates to the components of a state are given by the terms **T0** and **T1** respectively. That is, suppose the system is in state $p \doteq \langle b_0, b_1 \rangle$. For any Boolean input i, the value of the 0-th component in $\mathcal{A}.\mathit{next}(p, i)$ will be given by the value of the (ground) term $\mathit{if}(i, b_0, b_1)$.

The system is shown pictorially in Figure 10.2. By our previous discussions, it is easy to see that $\mathcal{A}$ is a predicate abstraction of our system. We can now prove our invariant by checking that the 0-th component of every reachable state is $\mathtt{T}$.

10.4 Summary

We have shown how the problem of proving invariants for reactive systems can be formally reduced to predicate abstraction. We have also suggested how predicate

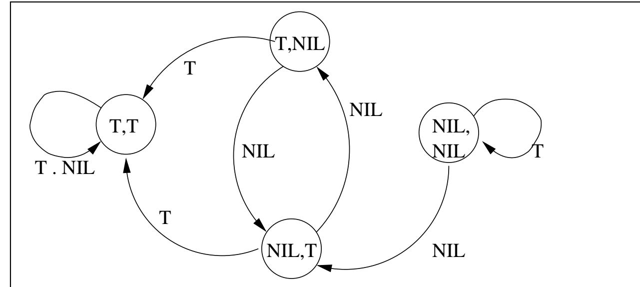

Figure 10.2: Finite-state Abstraction of the Two Component System

abstractions can be performed in a deductive reasoning system using term rewriting.

Given this motivation, several questions arise. Is it useful to do predicate abstraction and discovery for the kind of invariants we want to prove? Does it reduce manual effort substantially? Does it scale up to the complexity of large systems? In the next two chapters, we will answer all these questions in the affirmative by designing a procedure and demonstrating its use in proving invariants of reasonably complex systems.

10.5 Bibliographic Notes

Predicate abstractions have been the focus of a lot of recent research. The idea was suggested by Graf and Saidi [GS97], and it forms an instance of the theory of *abstract interpretation* [CC77]. Many verification tools for both software and hardware have built predicate abstraction and discovery methods. Notable among software verification tools are SLAM [BR01], BLAST [HJMS02], and ESC/Java [DLNS98]. Abstraction techniques in both SLAM and BLAST use Boolean programs, which are formed by replacing the control predicates of the program with Boolean vari-

ables [BMMR01], and these predicates are then iteratively refined. UCLID [BLS02] has developed predicate abstraction mechanisms where the abstract states and transitions are represented symbolically, to take advantage of algorithms for Boolean satisfiability checking [LBC03, LB04a]. Predicate discovery has involved refinement-based techniques based on model checking counterexamples [DD02, CGJ$^+$00] and syntactic manipulation of the concrete program [NK00, LBBO01].

The results described in this part are based on collaborative work with Rob Sumners [SR04, SR05]. The results also appear as part of the Ph.D. dissertation of Sumners [Sum05].

Chapter 11

Predicate Abstraction via Rewriting

The example in the last chapter, though trivial, illustrated the key ingredients involved in our approach to predicate abstraction and discovery. We start with a set of predicates to explore, initially only the predicate P_0 that we want to prove to be an invariant. For each predicate P to be explored we consider the term P/σ where σ is the substitution $[n \rightarrow (n+1)]$, and rewrite this term to some term P'. We then inspect the subterms of P' to find new predicates which we decide to explore or abstract, continuing this process until we reach a closure over the predicates to be explored. The abstract system is obtained whose state components correspond to the predicates we explored and the inputs correspond to predicates which we abstract. We then employ reachability analysis on this abstract system to check if the predicate P_0 is an invariant.

In this chapter, we will present some technical details of the procedure, show

why it is sound, and discuss some of the design choices we made in implementing it and some of its capabilities that afford user control and automation. We will then see how the procedure can be used effectively in proving invariants of some reactive systems.

Our abstraction generation constitutes the following two major pieces

REWRT: A term rewriter for simplifying terms given a collection of rewrite rules.

CHOP: An algorithm for finding relevant predicates from such rewritten terms.

Procedure REWRT is a simple conditional term rewriter. It takes a term τ and a theory $\mathcal{T}$, and produces a term τ^* as follows. Any formula in $\mathcal{T}$ of the form $\gamma \Rightarrow \alpha = \beta$ where α, β, and γ are terms, is treated as a *rewrite rule*.[1] The rule is *applicable* to term τ if there is a substitution σ such that $(\gamma/\sigma \Leftrightarrow \mathtt{T})$ is a theorem, and α/σ is syntactically equal to τ. Then β/σ is referred to as the *result* of the rewriting. A *rewriter* applies the rules in $\mathcal{C}$ to τ until no rule is applicable. The result is then said to be in *normal form.* Rewriting is in general a non-deterministic process. We implement REWRT principally as an inside-out, ordered rewriter. By *inside-out*, we mean that the arguments of a term are rewritten before the term. By *ordered*, we mean that the rules in $\mathcal{T}$ are applied in a pre-determined total order. The theory $\mathcal{T}$, as always, is assumed to be an extension of the ACL2 ground zero theory GZ. In particular, $\mathcal{T}$ should contain axioms defining the reactive system of interest and its properties.

It should be clear that since all the rewrite rules used by REWRT are either axioms or theorems in theory $\mathcal{T}$, if rewriting τ produces τ^*, then $(\tau = \tau^*)$ is a theorem in $\mathcal{T}$. This is the only critical property of REWRT that will be relevant

[1] The formulas of the form $\alpha = \beta$ are treated as rewrite rules $\mathtt{T} \Rightarrow \alpha = \beta$.

to us in what follows. The choice of inside-out rewriting and the order in which rules are applied by REWRT are influenced in part by the success of similar choices made in the implementation of the ACL2 theorem prover itself. In particular, our choice allows us to use theorems and lemmas that have been formulated for term simplification by the ACL2 simplifier. While these choices do affect how the theory $\mathcal{T}$ should be constructed in order for REWRT to be effective in generating the relevant predicates, we ignore them as "implementation details" in this presentation, along with other heuristics that have been implemented to make the rewriting efficient.

There is one aspect of the implementation of REWRT however that we briefly mention here, since it will directly concern the predicate generation process. REWRT gives special treatment to two function symbols *hide* and *force*. These are unary functions axiomatized in GZ to be identity functions as follows.

$$\mathit{hide}(x) = x$$

$$\mathit{force}(x) = x$$

But REWRT gives them special treatment in the sense that it ignores any term of the form $\mathit{hide}(\tau)$ or $\mathit{force}(\tau)$ and any of their subterms. This is done to allow the user to control the generation of predicates and we will understand their application when we discuss user-guided abstraction facilities in our procedure.

The second piece in the process of abstraction generation is the procedure CHOP. The procedure is described in Figure 11.1. Here $\emptyset$ is the empty set. CHOP takes a term (assumed to have a single variable n) and returns two sets of predicates, namely the set $\mathcal{E}$ of *exploration predicates* which are used for the creation of the abstract system, and the set $\mathcal{U}$ of *abstraction predicates*. In our example, CHOP, given the term **T0**, classified $\mathit{natp}(\widehat{C1}(n))$ as an exploration predicate and $\mathit{stimulus}(n+1)$

If τ is of the form *if*(τ_1, τ_2, τ_3)
$\langle \mathcal{E}_1, \mathcal{U}_1 \rangle := \text{CHOP}(\tau_1)$
$\langle \mathcal{E}_2, \mathcal{U}_2 \rangle := \text{CHOP}(\tau_2)$
$\langle \mathcal{E}_3, \mathcal{U}_3 \rangle := \text{CHOP}(\tau_3)$
Return $\langle \mathcal{E}_1 \cup \mathcal{E}_2 \cup \mathcal{E}_3, \mathcal{U}_1 \cup \mathcal{U}_2 \cup \mathcal{U}_3 \rangle$
Else If $(n+1)$ or *hide* occurs in τ then
Return $\langle \emptyset, \{\tau\} \rangle$
Else Return $\langle \{\tau\}, \emptyset \rangle$

Figure 11.1: Chopping a Term τ

as an abstraction predicate. The basis for the classification is very simple. If the term τ given to CHOP is of the form *if*(τ_1, τ_2, τ_3) then it recursively CHOPs each of these component terms. Otherwise it characterizes τ itself as an abstraction or exploration predicate as follows. If τ contains $(n+1)$ or the application of *hide* in any subterm then it is classified for abstraction, otherwise for exploration.

The reason for choosing subterms containing $(n+1)$ for abstraction should be clear from our example. We intend to apply CHOP on the result of rewriting $P' \doteq P/[n \rightarrow (n+1)]$ where P is one of the exploration predicates. The predicates are formulas describing properties of the different state components of a system. The value of a component at time $(n+1)$ can depend on (1) the values of components at time n, and (2) the *stimulus* received at time $(n+1)$. Thus any term containing an occurrence of $(n+1)$ in the result of rewriting P' must have been "contributed" by terms descibing the external stimulus. Since the invariants are required to hold irrespective of the value of this stimulus, it is a reasonable heuristic to abstract such terms. The reason for using *hide* for abstraction, as for treating it specially for rewriting, is to provide user control as will be clarified below.

Initially $old := \{P_0\}$; $new_s := \emptyset$; $new_i := \emptyset$
Repeat
1. Choose some predicate $P \in old$
2. REWRT $P/[n \to n+1]$ to obtain term P^*
3. $\langle \mathcal{E}, \mathcal{U} \rangle :=$ CHOP(P^*)
4. $old := (old \setminus \{P\}) \cup \mathcal{E}$;
5. $new_s := new_s \cup \{P\}$;
6. $new_i := new_i \cup \mathcal{U}$

Until $old == \emptyset$
Return $\langle new_s, new_i \rangle$

Figure 11.2: Procedure for Generation of Predicates

Given REWRT and CHOP, our procedure for predicate discovery is described in Figure 11.2. Given a proposed invariant P_0, it returns two sets new_s and new_i. The set new_s contains the predicates that are explored, and new_i contains those that are abstracted. The procedure at any time keeps track of the predicates to be explored in the variable *old*. At any iteration, it chooses a predicate P from *old*, rewrites the term $P/[n \to (n+1)]$, finds the new predicates by CHOPping the term P' so produced, and augments the set *old* with new exploration predicates so produced, until it reaches a closure.

With the predicates generated, it is now easy to construct an abstract system $\mathcal{A}$. It will have one *state variable* for every predicate in new_s, and one *input variable* for every predicate in new_i. Let us assume that the set of state variables is $\{v_0, \ldots, v_{m-1}\}$ and that of input variables are $\{i_0, \ldots, i_l\}$. Let us represent the predicate associated with a variable v by P_v. By the description of REWRT and CHOP above, it should be clear that P_v is a term containing a single variable n. Without loss of generality, assume that the predicate P_0 which we set out to prove

as invariant, is P_{v_0}. We can think of the states of our system as m-tuples where for each m-tuple a, every component a_j is either T or NIL. The initial state of the system is the m-tuple obtained by the term $P_v/[n \rightarrow 0]$ for each state variable v.

To define the transitions of the system note from our description in Figure 11.2 that for each state variable v, the term $P_v/[n \rightarrow (n+1)]$ is rewritten in step 2 to produce the term P'_v. We construct a term N_v describing the transition of v by "massaging" term P'_v as follows. If a subterm of P'_v does not have any occurrence of *if* then by our description of CHOP it must be one of the predicates in new_s or new_i. The term N_v is now obtained by replacing all such subterms with the corresponding variables. Since the only function symbol in the term N_v is *if*, we can treat the term as a Boolean combination of the state and input variables in the abstract system. Thus, given two abstract states (or m-tuples) a and b, b is a *successor* of a in $\mathcal{A}$ if and only if there exists a Boolean substitution σ such that the following holds.

1. σ associates a Boolean (T or NIL) to every state and input variable.
2. If b_j has the value $x \in \{\texttt{T}, \texttt{NIL}\}$, then σ associates x to the variable v_j.
3. $(N_{v_j}/\sigma \Leftrightarrow b_j)$ is a theorem for each $j \in \{0, \ldots, m-1\}$

By conditions 1 and 2, the term $(N_{v_j}/\sigma \Leftrightarrow b_j)$ is a ground term and the only function symbol that might occur is *if*. Thus the theoremhood in condition 3 can be established by simple evaluation.

Before proceeding further, let us first prove that we can check invariance of the predicate P_0 by showing that in every reachable state in $\mathcal{A}$, the variable v_0 has the value T. To do so, we will show how to construct a term I with a single variable n such that the following three formulas are theorems in $\mathcal{T}$.

- $I/[n \rightarrow 0]$

- $I \Rightarrow P_0$
- $I \Rightarrow I/[n \rightarrow (n+1)]$

We can think of I as an inductive invariant, although we have represented it as a function of "time" instead of "state". How do we construct I? For an m-tuple a, we first construct a term $M_a \doteq \bigwedge_{i=0}^{m-1}(P_{v_j} \Leftrightarrow a_j)$. Call M_a the *minterm* of a. Let $nbrs_a$ denote the set of successors of a. Then the following proposition specifies the relation between the minterms of an abstract state and the minterms of its successors.

Proposition 2 *Let a be any m-tuple. Then the formula*

$$M_a \Rightarrow (\bigvee_{b \in nbrs_a} M_b)/[n \rightarrow (n+1)]$$

is a theorem in $\mathcal{T}$.

Proof sketch: Since all the rules used in REWRT are axioms or theorems, it follows that if the application of REWRT on P in step 2 of Figure 11.2 produces P' then $(P/[n \rightarrow (n+1)]) \Leftrightarrow P'$ is a theorem. Further note that the minterm M_a returns `T` if and only if the value of every predicate P_{v_j} is the same as a_j. The proposition now follows from the definition of successors. ∎

Let R_a be the set of abstract states reachable from a. The following proposition shows that we can use the minterms for the states in R_a to construct I.

Proposition 3 *Let a be an abstract state. Then the formula*

$$(\bigvee_{q \in R_a} M_q) \Rightarrow (\bigvee_{q \in R_a} M_q)/[n \rightarrow (n+1)]$$

is a theorem.

Proof: Since R_a is the set of all reachable states from a, for any state $q \in R_a$, $R_q \subseteq R_a$, and $nbrs_a \subseteq R_a$. The proof now follows from proposition 2. ∎

From proposition 3, it follows that we can construct I above as follows. Let $\hat{a}$ be the initial state of $\mathcal{A}$. Then $I \doteq (\bigvee_{q \in R_{\hat{a}}} M_q)$. Indeed, from our description in the last chapter, it is easy to see that the abstract system $\mathcal{A}$ forms a predicate abstraction. The differences between $\mathcal{A}$ and a traditional predicate abstraction is cosmetic. Namely, traditionally the state variables in the abstraction are predicates on *states* of the concrete implementation, while in our implementation they are predicates on time.

We conclude this description with a note on convergence. The steps 1-6 in Figure 11.2 need not converge. In practice, we attempt to reach convergence within a user-specified bound. Why not coerce terms on which convergence has not been reached to input variables? We have found that such coercions typically result in coarse abstraction graphs and spurious failures. We prefer to rely on user control and perform such coercions only via user-guided abstractions as we describe below.

11.1 Features and Optimizations

Our method primarily relies on rewrite rules to simplify terms. Even in our trivial example in the last chapter, equation 5 is critical to rewrite P'_0 to $\mathbf{T0}$. Otherwise, the normal form would have been

$$\mathbf{T0}' : \quad \mathit{natp}(\mathit{if}(\mathit{stimulus}(n), \widehat{\mathit{C0}}(n), \widehat{\mathit{C1}}(n)))$$

Then CHOP would have classified it as an abstraction predicate resulting in an abstract system that would produce a spurious failure.

This trivial example illustrates an important aspect of our approach. Equation 5 is a critical but generic "fact" about *natp* and *if*, independent of the system analyzed. Equation 5, known as an *if-lifting* rule, would usually be stored in a library of common rules. While generic rules can normalize most terms, it is important for scalability that the procedure provide control to facilitate generation of manageable abstractions. We now discuss some of these features.

11.1.1 User Guided Abstraction

User-guided abstraction is possibly the most important and powerful feature of our procedure that is relevant in providing user control. This is achieved by the special treatement given to the function *hide*. To see how we can *hide* predicates effectively, consider a system with components `A0`, `A1`, `A2`, etc., where `A0` is specified as follows:

$$\widehat{A0}(0) = 1$$

$$\widehat{A0}(n+1) = \begin{cases} 42 & \textit{if } \textit{natp}(\widehat{A1}(n)) \\ \widehat{A0}(n) & \text{otherwise} \end{cases}$$

Thus `A0` is assigned 42 if the previous value of `A1` is a natural number, and otherwise is unchanged. Consider proving that $P_0 \doteq \textit{natp}(\widehat{A0}(n))$ is an invariant. Our procedure will discover the exploration term $P_1 \doteq \textit{natp}(\widehat{A1}(n))$ and attempt to rewrite $\textit{natp}(\widehat{A1}(n+1))$, thereby possibly exploring other components. But P_1 is irrelevant to the invariance of P_0. This irrelevance can be suggested by the user with the rule:

$$\textit{natp}(\widehat{A1}(n)) = \textit{hide}(\textit{natp}(\widehat{A1}(n)))$$

Since *hide* is logically the identity function, proving the formula above is trivial. But the rule has the effect of "wrapping" a *hide* around $\textit{natp}(A0(n))$; this term is therefore ignored by REWRT and abstracted by CHOP, resulting in a trivial abstract system.

The idea of *hide* can be used not only to abstract predicates but also to *introduce* new predicates. We will see a more serious use of this in Section 11.3

11.1.2 Assume Guarantee Reasoning

The other special function *force* is used for providing limited assume-guarantee capabilities. We mentioned that in trying to prove that P_0 is an invariant REWRT ignores terms of the form $\mathit{force}(P)$. But we did not say what is done with such a term in generating the abstract system. Procedure CHOP replaces $\mathit{force}(P)$ with T. We can think of this process as *assuming* that P is an invariant. Thus for example, in our example of the previous chapter, we could have added the rule:

$$\mathit{natp}(\widehat{C1}(n)) = \mathit{force}(\mathit{natp}(\widehat{C1}(n)))$$

Proving invariance of P_0 with this added rule would have wrapped a *force* around $P_1 \doteq \mathit{natp}(\widehat{C1}(n))$ thereby producing a trivial abstract system with 1 variable instead of 2 as shown in Figure 10.2. To complete the proof, we need to prove that each *force*d predicate, in this case $\mathit{natp}(\widehat{C1}(n))$, is also an invariant. This is done by calling the procedure on each forced predicate recursively. In proving the invariance of P_1 we can assume that P_0 is an invariant. The apparent circularity is resolved by an induction on time. More precisely, when we prove the invariance of P assuming the invariance of Q and vice versa, we are merely proving that P holds for time $(n+1)$ assuming Q at time n. The use of assume-guarantee reasoning is well-known in model checking and the use of *force* provides a way of "simulating" such features via rewriting.

11.2 Reachability Analysis

The abstract system is checked using reachability analysis. *Any* model checker can be integrated with our procedure for that purpose by translating the abstract system into a program understandable by the checker. We have implemented such interfaces for VIS [BHSV+96], SMV [McM93], and NuSMV [CCGR99]. Nevertheless, we also implemented our own "native" reachability analysis procedure which principally performs an on-the-fly breadth first search. While admittedly less efficient than the commercial tools above, it can benefit from tight interaction with the abstraction generation process which is not possible for the other tools. We now present some facets of this simple procedure. As an aside, we note that our native checker has been sufficient for proving invariants of all the systems we discuss in this dissertation, although the external checkers have been often used to confirm the results. Our checker also contains additional features to provide user feedback, such as pruning counterexamples to only report predicates that are relevant to the failures in the reachability check, but we ignore such details here.

One principal way in which our native checker leverages the abstraction procedure is in pruning irrelevant paths in the abstract system during reachability analysis. Recall that user-guided abstraction via *hide* can reduce the number of state variables in the abstract system. However, this can substantially increase the number of input variables. To combat this, the abstraction procedure computes for each abstract state a a set of *representative input valuations*, that is, valuations of the input variables that are relevant in determining $nbrs(p)$. If τ is coerced to an input using *hide*, it contributes to an edge from a only if some $q \in nbrs(a)$ depends on the input variable corresponding to *hide*(τ). In addition, we *filter* exploration of spurious paths by using REWRT to determine provably inconsistent combinations of

exploration and abstraction predicates. For example, assume that for some state variable v, and input variables i_0 and i_1 the predicate P_v, P_{i_0} and P_{i_1} are given by:

$$P_v \doteq (f(n) = g(n))$$

$$P_{i_0} \doteq (f(n) = i(n+1))$$

$$P_{i_1} \doteq (g(n) = i(n+1))$$

Then for an abstract state a such that the variable v is assigned to `NIL`, filtering avoids exploration of edges in which both i_0 and i_1 are mapped to `T`.

11.3 Examples

We now apply our procedure for proving invariants of reactive systems. The examples in this chapter are principally intended to demonstrate the scalability of the method. In the next part, we will apply it on a substantially more complicated reactive system. There are several other reactive systems in which our tool has been applied, which we omit for brevity.

11.3.1 Proving the ESI

As a simple example, let us first try using the procedure on the **ESI** system we presented in Chapter 8. Since that system was small, containing only four state components, it is suitable for experimentation and understanding.

What property do we want to prove as an invariant for this system? We prove the property of cache coherence that was specified as *good* and was sufficient to prove (**mem** $\triangleright$ **esi**) in Chapter 8. In that chapter, we proved *good* to be an invariant by manually constructing an inductive invariant *inv*. We now use our tool

$$in(e, insert(a, s)) = in(e, s) \lor (a = e)$$

$$in(e, drop(a, s)) = in(e, s) \land \neg(a = e)$$

$$get(a, put(b, v, r)) = \begin{cases} v & \textit{if } (a = b) \\ get(a, r) & \text{otherwise} \end{cases}$$

Figure 11.3: Generic Rewrite Rules for Set and Record Operations

to prove *good* directly as an invariant. Of course to do that we need to specify the **esi** system with a collection of equations as a function of time. This is easy to do, and we have 8 equations that correspond to the four state components. Thus $\widehat{excl}(c, n)$ returns the set `excl` for cache line `c` time n, $\widehat{valid}(c, n)$ returns the set `valid`, $\widehat{cache}(p, c, n)$ returns the content of cache line c in process p, etc.

Use of our procedure requires rewrite rules. Since the **esi** has been modeled with sets and records, the rules must be theorems that specify how the operations on such data structures interact. Figure 11.3 shows the generic rules we need. Here $insert(e, s)$ inserts the element e in set s, $drop(e, s)$ returns the set obtained by removing e, $get(a, r)$ returns the value stored in field a in record r, and $put(a, v, r)$ returns the record obtained by storing v in field a of record r. The rules were already available as generic lemmas about these operations in existing ACL2 libraries [KS02], and our procedure merely makes effective use of their existence.

In order to apply the procedure, we need one "system-specific" rule. This theorem is interesting and shows the kind of things that are necessary for effective

application of the procedure. The rule is shown below.

$$
\mathit{in1}(e, s) = \begin{cases} \texttt{NIL} & \mathit{if\ empty}(s) \\ (e = \mathit{choose}(s)) & \mathit{if\ singleton}(s) \\ \mathit{hide}(\mathit{in1}(e, s)) & \text{otherwise} \end{cases}
$$

Here *in1* is defined to be *in* but is expected to be applied to test membership on sets that are expected to be *empty* or *singleton*, and *choose*(s) returns *some* member of set s if s is non-*empty*. Recall that the key insight for proving cache coherence of **esi** is the fact that the set `excl[c]` is always empty or singleton. We "convey" this insight to the procedure by testing membership on $\widehat{\mathit{excl}}(c, n)$ using *in1* rather than *in*. Application of the rule causes terms involving *in1* to be rewritten to introduce a case-split for the cases where the set is empty, singleton, or otherwise, and coerces the third case to an input.

With this rule, our procedure now proves that *good* (restated in terms of time) is indeed an invariant. The abstract system has 9 exploration predicates and 25 abstraction predicates. The search traverses 133 edges exploring 11 abstract states and the proof takes a couple of seconds. Without edge pruning, the search explores 48 abstract states.

The exploration predicates are shown in Figure 11.4. Here $A()$ and $R()$ are uninterpreted functions and designate an arbitrary address and process index respectively, and $D(n)$ returns the last value written in address $A()$. It is not necessary to understand all the predicates, but we only call the attention of the reader to predicate 9. This term is produced by rewriting using the rule about *in1* above. It "tracks" the process whose local cache has the most current value written at address $A()$, without requiring us to preserve all the process indices. In this context it should be remarked that it is exactly this problem of tracking which

1. $\mathit{good}(n)$
2. $\widehat{\mathit{valid}}(\mathit{cline}(A(), n))$
3. $\mathit{in}(R(), \widehat{\mathit{valid}}(\mathit{cline}(A()), n))$
4. $\widehat{\mathit{excl}}(\mathit{cline}(A()), n)$
5. $\mathit{singleton}(\widehat{\mathit{excl}}(\mathit{cline}(A()), n))$
6. $(\mathit{choose}(\widehat{\mathit{excl}}(\mathit{cline}(A()), n)) = R())$
7. $(D(n) = \mathit{get}(A(), \widehat{\mathit{mem}}(\mathit{cline}(A()), n)))$
8. $(D(n) = \mathit{get}(A(), \widehat{\mathit{cache}}(R(), \mathit{cline}(A()), n)))$
9. $(D(n) = \mathit{get}(A(), \widehat{\mathit{cache}}(\mathit{choose}(\widehat{\mathit{excl}}(\mathit{cline}(A()), n)), \mathit{cline}(A()), n)))$

Figure 11.4: State Predicates Discovered for the **ESI** Model

process is relevant at which point that has made it difficult for fully automatic decision procedures to abstract process indices in past work in abstraction, and underlines the importance of using an expressive logic for defining predicates.

11.3.2 German Protocol

It might seem as if we had to do too much work for using our tool on the **esi**. We had to restate the model and properties as functions of time, and even had to write one system-specific rewrite rule. On the other hand, writing an inductive invariant for **esi** was not that complicated. However, the advantage of using our approach becomes pronounced as we consider more and more elaborate and complicated systems. To demonstrate this, we consider a more complex cache system that is based on a protocol by Steven German. In this system, the controller (named *home*), communicates with clients via three channels 1, 2, and 3. Clients make cache requests

(*fill requests*) on channel 1. Home grants cache access (*fill responses*) to clients on channel 2; it also uses channel 2 to send invalidation (*flush*) requests. Clients send flush responses on channel 3, sometimes with data.

The *German protocol* has been studied extensively by the formal verification community [PRZ01, AK86, LB04a]. The original implementation has single-entry channels. In UCLID, *indexed predicates* were used [LB04b] to verify a version in which channels are modeled as unbounded FIFOs. Our system is inspired by the version with unbounded FIFOs. However, since we have not built rules to reason directly about unbounded FIFOs, we modify the protocol to use channels of bounded size, and prove, in addition to coherence, that the imposed channel bounds are never exceeded in our model. As in **esi**, we also model the memory.

Our model is roughly divided into three sets of functions specifying the state of the clients, the *home* controller, and the channels. The state of the clients is defined by the following functions:

- $\widehat{\mathit{cache}}(p, c, n)$ is the content of line c in the cache of client p at time n.
- $\widehat{\mathit{valid}}(c, n)$ is the set of clients having a copy of line c at time n.
- $\widehat{\mathit{excl}}(c, n)$ is the set of clients which have exclusive access of c at time n.

Home maintains a central directory which enables it to "decide" whether it can safely grant exclusive or shared access to a cache line. It also maintains a list of pending invalidate requests it must send, and the state of the memory. The state of *home* is specified by the following functions:

- $\widehat{\mathit{h\text{-}valid}}(c, n)$ is the set of clients which have access to line c at time n.
- $\widehat{\mathit{h\text{-}excl}}(c, n)$ is the client which has exclusive access to line c at time n.
- $\widehat{\mathit{curr\text{-}cmd}}(c, n)$ is the pending request for line c at time n.

- $\widehat{\mathit{curr\text{-}client}}(c, n)$ is the most recent client requesting for line `c` at n.
- $\widehat{\mathit{mem}}(c, n)$ is the value of line `c` in the memory at time n.
- $\widehat{\mathit{invalid}}(c, n)$ is a record mapping client identifiers to the state of a pending invalidate request at time n. It can be "`none pending`", or "`pending and not sent`", or "`invalidate request sent`", or "`invalidate response sent`". This function models part of the state of *home* and part of the state of the channels 2 and 3 (namely, invalidate requests and responses).

Finally, the states of the three channels are specified by the following functions (in addition to *invalid* above):

- $\widehat{\mathit{ch1}}(p, c, n)$ is the requests sent from client `p` for line `c` at time n
- $\widehat{\mathit{ch2\text{-}sh}}(c, n)$ is the set of clients with a shared fill response in channel 2.
- $\widehat{\mathit{ch2\text{-}ex}}(c, n)$ is the set of clients with an exclusive fill response in channel 2.
- $\widehat{\mathit{ch2\text{-}data}}(p, c, n)$ is the data sent to client `p` with fill responses.
- $\widehat{\mathit{ch3\text{-}data}}(p, c, n)$ is the data sent from client `p` with the invalidate responses.

At any transition, one of the following 12 actions is selected to execute nondeterministically: (1) a client sends a shared fill request on channel 1, (2) a client sends an exclusive fill request on channel 1, (3) *home* picks a fill request from channel 1, (4) *home* sends an invalidate request on channel 2, (5) a client sends an invalidate response on channel 3, (6) *home* receives an invalidate response on channel 3, (7) *home* sends an exclusive fill response on channel 2, (8) *home* sends a shared response on channel 2, (9) a client receives a shared fill response from channel 2, (10) a client receives a shared exclusive response from channel 2, (11) a client performs a store, and (12) a client performs a load.

Let us call this system **german**. We prove the same coherence property about **german** that we proved about **esi**. As a separate (and simple) theorem proving exercise, we show that by assuming coherence we can prove (**mem** $\triangleright$ **german**).

The verification of **german** illustrates the utility of our procedure. The system is very different from **esi**, and an inductive invariant, if defined, would be very different and involve extensive manual effort. Nevertheless, little extra "overhead" is involved in proving the coherence property for **german** than for esi. We use the same rewrite rules for set and record operations as showin in Figure 11.3; we also reuse the "system specific" concept of using *in1* to test membership on sets that are empty or singleton. The only extra rule necessary for completing this proof is another rule similar to that for *in1* but for record operations in order to cause a case-split on *invalid*(c, n). With these rules, our procedure can prove coherence along with the bounded channel invariant. The abstract system for coherence is defined by 46 exploration and 117 abstraction predicates. The reachability check explores 7000 abstract states and about $300,000$ edges, and the proof is completed in less than 2 minutes on an 1.8GHz Pentium desktop machine running GNU/Linux. The proof of the bounded channel invariant completed in less time on a smaller abstraction.

11.4 Summary and Comparisons

We have described a deductive procedure for constructing a form of predicate abstractions, and used it to prove invariants of reactive systems. The approach uses term rewriting to simplify formulas that describe updates to predicates along the transitions of a system to discover relevant predicates and create the abstract system. The approach frees the user of theorem proving from the responsibility of manually

defining and maintaining inductive invariants that often need to be modified drastically as the design evolves. Instead the user creates rewrite rules to effectively normalize terms composed of function symbols occurring in the definition of the system and its properties. Since most of the rewrite rules are generic theorems about the functions used in modeling the system and its properties, the approach is reusable for a wide class of systems. Even when system-specific rules are necessary, the concepts behind such rules can be transferred to similar systems. Further, by allowing the user to control the predicates generated via rewrite rules, the method can generate relevant abstractions for complicated systems which are difficult for fully automatic abstraction generation tools. We must mention that in addition to the systems we discussed in this chapter and the microprocessor model we will present in the next part, our tool has been used by Sumners [private communication] to prove the relevant invariants of the concurrent deque of Chapter 8. Given the size and complexity involved of the manual definition of the inductive invariant for this system, we consider the success in this application to be a reasonable indication of the scalability of the method to large systems.

It should be understood that the predicate abstraction tool is not a panacea. As with any deductive approach, it can fail for some systems and reachability analysis then will generate counterexamples. The counterexample is a path through the abstract system which therefore corresponds to a sequence of predicates in the implementation. The user must analyze the counterexample and decide whether it is real or spurious. In the latter case, the user must introduce more rules to guide our procedure to an effective abstraction. This process might not be simple. Our tool does provide some support for focusing the attention of the user in case of a proof

failure, for instance by returning only the predicates relevant to the failure, allowing bounded model checking, and providing some facilities for suggesting effective rewrite rules or restructuring of the system definitions. For instance, the *in1* rule we described above was introduced based on the feedback from the tool. While this has proven sufficient in many cases, we admit that better interfaces and feedback mechanisms are necessary to make the process more palatable.

At a high level, our tool can be likened to the approach suggested by Namjoshi and Kurshan for predicate discovery [NK00]. This method uses syntactic transformations based on computations of weakest precondition of the actions of the implementation to create an abstract system on the predicates. For example, assume consider a system with two variables `x` and `y`, let $X(s)$ and $Y(s)$ return the values of these variables in state s, and let one of the actions be `x := y`. Assume also that one of the exploration predicates is $P_0 \doteq \mathit{natp}(X(s))$. By computation of weakest precondition for the action we can discover the predicate $P_1 \doteq \mathit{natp}(Y(s))$ and "abstract" the action `x := y` to the assignment of the variables representing these predicates. Namjoshi and Khurshan also suggest many of the features we implemented, such as *if-lifting* transformations. Our tool can be easily thought of as a focused implementation of this method with rewriting for syntactic transformations. Conceptually, the novelty in our approach lies in the observation that in a deductive setting one can extend the set of syntactic transformations by adding lemmas and theorems, thus making the approach flexible for a large class of system implementations.

We end this chapter with a remark on the kind of predicates that are explored and produced by our method. Some researchers have expressed concern that

the fact that our predicates only allow one single variable n is too restrictive and not in line with our general claim that we allow expressive languages for defining predicates. There is really no paradox here. While our predicates have a single free variable n, the logic allows us to write expressive recursive functions to model the predicates. If we did not allow arbitrary predicates of "time" but only predicates of the current state, one would have required quantification over process indices to state the cache coherence property for **esi**. Indeed, in related work for example by Lahiri and Bryant [LB04b], that is the motivation for allowing a collection of index variables on which quantification is allowed. But we can write arbitrary functions that can keep track of the relevant details from the history of the execution. In fact, in **esi**, that is exactly what is done by the function D which appears in Figure 11.4. The function keeps track of the last value written at $A()$, where A is a 0-ary uninterpreted function specifying an arbitrary address. The expressiveness afforded by allowing arbitrary definitions for predicates lets us "get away" with a simple and basic rewriting procedure for generating effective abstractions.

11.5 Bibliographic Notes

Term rewriting has a rich history. The reader interested in rewriting is encouraged to read an excellent overview of the field by Baader and Nipkow [BN98]. Rewriting forms has several applications in formal verification and in particular theorem proving. Almost any general-purpose theorem prover implements a rewriter to simplify terms. In addition, the Maude theorem prover [CDE^{+}99] also uses a logic based on rewriting to specify next-state functions of computing systems.

Predicate abstraction has been recently used to simplify invariant proofs.

The bibliographic notes for the last chapter list many papers on the subject. Our implementation of predicate abstractions is closely related to the approach suggested by Namjoshi and Kurshan [NK00]. A comparison of our approach with theirs appears in Section 11.4.

Among reactive systems, verification of cache coherence protocols has received special attention, primarily because of the difficulty in automating such verification. The German Protocol has become one of the benchmark systems to gauge the effectiveness of a verification mechanism. One of the earliest papers reporting a proof of this protocol is by Pnueli, Ruah, and Zuck [PRZ01]. They use a method of *invisible invariants* to simplify the proof of the system. Emerson and Kahlon [EK03] show that verification of parameterized systems implementing certain types of snoopy cache protocols can be reduced to verification of finite instances. They show a reduction of the German protocol to a snoopy system and verify the finite instances of the snoopy system using model checking. Predicate abstractions have been used by Lahiri and Bryant [LBC03, LB04a, LB04b] for verification of this protocol, both for bounded and unbounded channel sizes.

Part V

Verification of RTL Designs

Chapter 12

RTL Systems

The techniques we developed in the last two parts provide a generic deductive approach for showing correspondence between the executions of a reactive system implementation and its abstract specification. We decompose the verification problem by defining a sequence of intermediate models at different levels of abstraction starting from the implementation and leading up to the specification, and derive a refinement theorem for each pair of consecutive models in the sequence; *stepwise refinement* then allow us to "chain" the results of these intermediate verifications. The intermediate refinement theorems are derived by following one of the following two strategies.

1. If the more abstract model in the pair is merely an augmentation of the concrete one with auxiliary variables then we show stuttering equivalence between them via *oblivious refinement*.
2. Otherwise we attempt to show *well-founded refinements* by proving the corresponding single-step proof obligations.

The first case requires little manual effort. In the second, manual effort is involved principally in the definition and proof of an inductive invariant on the states of

concrete model. We reduce the effort involved in the process as follows. We first define some predicate *good* on the states of the concrete model and derive the proof obligations for single-step reduction up to the invariance of *good*. Then, instead of going through the manual process of defining an inductive invariant strengthening *good*, we apply predicate abstractions to prove the invariance of *good* directly.

Is the above approach scalable? We have seen some affirmative evidence in the reactive system implementations that we looked at in the last two parts. Many of the systems in which we applied the methodology were substantially complex, for example the **cdeq**. However, all our examples so far were simplified in one respect: the implementations were modeled at a relatively high level of abstraction, at the so-called *protocol level* or *algorithmic level*. This is an appropriate level of abstraction when we are reasoning about concurrent protocols. But it is not adequate when we are reasoning about a hardware design, for example a component of a microprocessor. Hardware systems are *not* implemented or modeled at the protocol level but at a lower level of abstraction which is often referred to as the register-transfer or RTL level. Our methodology must be robust enough to enable reasoning about systems at the level at which they are implemented. In this part, we therefore explore ways of applying the approach to such designs.

Formal verification of an RTL model of a system is much more difficult and challenging than the verification of a model of the *same* system at the protocol level for a variety of reasons. RTL models are concerned with many low-level details and are often optimized for several disparate goals such as efficient execution, low power overhead, and so on. The intuitive, high-level concepts behind the workings of the different protocols implemented by the system are obscured amidst the plethora of

implementation details, making it difficult to verify them using theorem proving alone. The size and complexity of the systems also make it difficult to apply model checking directly.

In addition to the inherent complexity involved in reasoning about low level models, there is another practical impediment to reasoning about RTL systems which arises from the ambiguity of the *languages* in which they are typically implemented. Let us examine this issue briefly. In order to be able to formally reason about a system, we must first have a formal model of the system. However, most practical systems are not modeled in a formal logic but rather implemented in a programming language. In this dissertation whenever we reasoned about such a system, for example the JVM factorial method in Chapter 6, we used a formal operational model of the language semantics. What should we do about RTL designs? For RTL designs, the typical languages of choice are commercial Hardware Description Languages (HDLs) such as VHDL [Bha92] and Verilog [TM96]. These HDLs are designed to satisfy several disparate goals other than formal verification, namely ease of use, simulation speed, etc. As a result, they are large, unwieldy, and in parts poorly specified [RF00]. While there has been partial success in defining formal operational semantics of commercial HDLs [Gor95, Rus95], such attempts have been limited to very small subsets of the languages that are not suitable for modeling substantially complex RTL systems. Therefore, formal verification of a hardware design written in an HDL has been traditionally restricted to some alternative encoding of the implementation (typically by a human) in some formal language. The utility of such a verification then rests on the assumption that the encoding faithfully reflects the actual implementation.

In this part we address some of the above concerns. In Chapter 13, we attempt to bridge the semantic gap between an HDL implementation of a design and its formal rendition by implementing a translator from a well-defined subset of Verilog to ACL2. The subset of Verilog is carefully chosen to be simple enough so that we can be reasonably confident of the correspondence between the source HDL implementation and the output of the translator, yet rich enough to afford translation of non-trivial designs. In Chapter 14, we demonstrate that with a carefully designed but generic library of rewrite rules and lemmas relating common RTL operations, our methodology can be applied to reason about RTL designs with reasonable automation. To this end, we verify the RTL implementation of a pipelined model of a slightly simplified version of the Y86 processor developed at the Carnegie-Mellon University based on the IA32 Instruction Set Architecture [BO03].[1]

This part is different in spirit from the other parts of the dissertation in a very important sense. In all other chapters, our focus is on *developing* verification techniques and concepts to facilitate reasoning about computing systems; system examples have been used to illustrate the techniques presented. By contrast, in this part, our focus is on showing practicability of the methods we have already developed in the last two parts on a substantially complex system example and our goal to *use* the techniques we have seen so far to combat such complexity.

[1]The author thanks Randal E. Bryant for giving access to the Verilog design of the Y86 processor.

Chapter 13

A Verilog to ACL2 Translator

We have implemented a translator called **V2L2** to translate digital designs implemented in a well-defined subset of Verilog to ACL2. In this chapter, we present **V2L2** and touch upon the design choices we made in its implementation. This also provides us an opportunity to discuss some of the sources of complexity of modern HDLs and the resulting complexity in the verification of hardware designs modeled in such languages.

Before moving into the description of **V2L2**, we wish to clarify that there have been several efforts to formalize RTL designs [RF00, HR05, Rus95, Gor95]. In this context several other translators have been written to translate different subsets of Verilog designs. For instance, a (proprietary) translator has been implemented at AMD, to translate Verilog designs to ACL2. While **V2L2** is different in implementation from other translators, we do not claim any conceptual novelty in either the choice of the particular subset of Verilog we support, nor in the design of the translator implementation. We describe **V2L2** here principally to keep the dissertation

self-contained and present some of the design choices that must be considered for implementing *any* such translator.

13.1 Overview of Verilog

Verilog is a Hardware Description Language for specifying digital systems over a wide range of abstraction levels. It has a large set of language primitives with complex semantics. It supports constructs that can be used to model both the behavioral abstraction of a hardware and its synthesizable netlist-level implementations. A comprehensive or even adequate presentation of the language is beyond the scope of this dissertation; we only provide an informal overview of some of the key features. We focus on the *behavioral* features of Verilog in this review, since they are more complex to translate than the synthesizable ones. To the reader interested in learning the Verilog language, we recommend Thomas and Moorby's book [TM96] for a thorough description.

Consider the Verilog program shown in Figure 13.1. The program implements a 4-bit counter, and shows many of the language features of Verilog. A digital system is modeled using a collection of *modules*. Let us first focus on the module named `m16`. This module implements our counter. A module can have zero or more inputs and outputs. Module `m16` has two outputs `count` and `fifteen`, and one input `clock`. The inputs and outputs to a module are also called *ports* and specified in parentheses after the declaration of the module name as shown. A module can also have local variables; in our case we have declared a variable called `afifteen`. Verilog provides two fundamental data types, namely *register* (called `reg`) and *wire*.[1] A wire is used

[1]Actually wires form only a special type of what are known as *nets*. For our purpose the only

```
module dEdgeFF (q, clock, data)

 output q;
 reg q;
 input clock, data;

 initial
  begin
   q = 0;
  end

 always @(posedge clock)
  begin
   q <= data;
  end

endmodule


module m16 (count, clock, fifteen)

 output [3:0] count;
 output fifteen;
 input clock;

 wire afifteen;

 dEdgeFF
  a(count[0], clock, ~count[0]),
  b(count[1], clock, count[1] ^ count[0]),
  c(count[2], clock, count[2] ^ (count[1] & count[0])),
  d(count[3], clock, count[3] ^ (count[2] & count[1] & count[0]));

 assign afifteen = count[0] & count[1] & count[2] & count[3];
 assign fifteen = afifteen;

endmodule
```

Figure 13.1: A 4-bit Counter Implemented in Verilog

to transmit logic values among the different submodules of a module. A register is used to model state elements of the system that "store" values. A port of a module is assumed to be a wire unless explicitly declared to be a register; all local variables must be explicitly declared. The variables in a Verilog module are also referred to as *signals*. Signals are often classified to be of two types, namely *scalar* and *vector*. For our purpose, a scalar can hold a single bit and a vector is an array of bits. In `m16`, the signals `fifteen`, `clock`, and `afifteen` are scalars while `count` is a bit vector.

How does the module `m16` work? We declare four instances `a`, `b`, `c`, `d` of the module `dEdgeFF`. Module `dEdgeFF` has three ports `q`, `clock`, and `data`, and thus each instance of `dEdgeFF` must have three arguments showing the connections of the signals of `m16` to these ports. Let us look at the instance `a`. In this instance, we connect the `clock` input of `m16` to the `clock` port of `dEdgeFF`, and the bitwise complement of `count[0]` to the `data` input. (The symbol "`~`" stands for bitwise complement.) The output port `q` is connected to `count[0]`. We will look at the design of the `dEdgeFF` module in detail momentarily; for now assume that it implements a standard D flip flop triggered by the rising edge of the `clock` input. Then the effect of the instance `a` is that every rising edge of `clock` causes the value of `count[0]` to toggle, as expected of the least significant bit of a counter. The instances `b`, `c`, and `d` are also fairly standard implementations of bits 1, 2, and 3 of a 4-bit counter. The symbol "`^`" stands for bitwise exclusive OR, and the symbol "`&`" stands for bitwise AND.

The module `m16` shows another important construct of Verilog, namely the `assign` statement. The `assign` statement is also referred to as *continuous assign-*

type of nets we will consider are wires.

ment and is used quite extensively in Verilog to conveniently model combinational logic operations. The statement is reminiscent of the assignment statements of a traditional programming language. The difference, however, is in the way it is evaluated. In a traditional (sequential) programming language we think of the statements in a program as being evaluated in serial order. In contrast, the semantics of `assign` is intended to reflect the parallel nature of combinational circuits. The statement is evaluated as follows. An `assign` statement is *activated* at any time when value of the expression at the right hand side of the statement changes; whenever that happens, the signal at the left hand side is assigned the new value. Thus in the case of `m16`, the first `assign` statement is activated whenever the `count` either becomes or changes from the value `1111`. In other words, the signal `afifteen` at any time has the value `1` if and only if each component of the vector `count` at that time holds `1` (that is, the counter has the value 15). The second `assign` statement is activated every time the value of `afifteen` changes; in effect it merely `output`s the value of `afifteen`. Note that the behavior of the design would have been exactly the same if the two `assign` statements were swapped in order.

Let us now focus on the other module `dEdgeFF`. This module has three ports `q`, `clock`, and `data`, of which the first is an `output` and the other two are `input`s. The variable `q` is also a `reg`, and hence models a storage element. This module shows another feature of Verilog, namely the evaluation of *blocks*. The module has two blocks, an `initial` block and an `always` block. How are these two evaluated? The `initial` block is evaluated at the reset or initial time. Thus initially `q` is assigned to `0`. An `always` block is evaluated at every instant. An `always` block consists of a collection of *procedural* statements which are evaluated in sequential order. An

`always` block often has a *trigger* that is specified right after the word "`always`" and is demarcated by the special symbol "`@`". In that case, the statements inside the block are evaluated only at times when the trigger evaluates to `1`. For our example, the trigger is the expression `(posedge clock)`. When does this expression evaluate to `1`? It does so at a time when the argument `clock` has a rising edge. (There is an analogous construct `(negedge clock)` which evaluates to `1` when `clock` is falling.) Thus in `dEdgeFF` whenever the input `clock` rises, the input `data` is "stored" at the variable `q`. What happens if the `clock` does not rise at some instant? Since `q` is declared to be a `reg`, this means that it simply preserves the previous value of `q`. The module thus implements a clocked (positive) edge-triggered flip flop rather naturally.

We have just illustrated a very tiny subset of the features of Verilog. It should be clear from our discussion however, that many of the language constructs of Verilog have complex execution semantics. Our translator **V2L2** supports the features described here along with many other language constructs, but restricts their application in certain ways in order to facilitate translation to ACL2.

13.2 Deep and Shallow Embeddings

How do we translate Verilog designs to ACL2? Verilog and ACL2 are both programming languages, but it should be abundantly clear that they are languages of very different flavors. Verilog is tailor-made for designing digital systems. The ACL2 programming language is akin to a formal language for specifying recursive functions. Implementing a translator from Verilog to ACL2 is tantamount to defining a semantic embedding of Verilog designs in the logic of ACL2.

There are two well-known approaches to specifying the semantics of a programming language in a formal logic, which are often referred to as *deep embedding* and *shallow embedding* respectively [BGG+92]. In a deep embedding, one specifies semantics for the *language constructs* of the target language by defining an *interpreter* for the language in the formal logic. A program in the language, then, is merely a constant whose meaning is fixed as the result of the interpretation and verifying a particular program is tantamount to verifying the result of its interpretation. On the other hand, in *shallow embedding*, one implements a *translator* for converting programs to functions in the language of the formal logic. Unlike the interpreter in the deep embedding, the translator here is not a component of the formal system.

The two approaches, of course, have their individual pros and cons. Deep embeddings are easier to validate against programs written in the target language because of the semantic closeness between the source program and its formal embedding. Further, deep embedding affords verification of properties of the target language itself; for example, Liu and Moore [LM03] use a deep embedding of the JVM in ACL2 to prove the properties of the JVM byte-code verifier. Deep embeddings have been used in the context of formalizing hardware designs in ACL2 as well. For example, Brock and Hunt [BH97b] define a deep embedding of an HDL called **Dual-Eval** in ACL2. However, if one is interested not in proving properties about the language itself, but in verifying *systems* implemented in the language, the verification is usually more complex and cumbersome than shallow embedding. Verifying a deeply embedded system design entails a two-step process, namely the verification of the properties of the interpreter as well as the interpreted program. On the other

hand, shallow embedding makes the reasoning process easier by dispensing with the interpreter but leaves open the possibility of imprecision in translation. An obvious source of imprecision is that different language constructs are not isomorphic; for example, in ACL2, `NIL` stands for both the empty list and the logical false. Hence if one were to translate ACL2 programs to a logic in which logical false and empty list stand for two different objects, then imprecision would occur in a straightforward translation of `NIL`.

V2L2 uses shallow embedding. Given a Verilog module implementing a digital system, the translator generates a sequence of functions describing the transitions of the system. The functions can be introduced in ACL2 via the extension principles. The translator itself however, although largely implemented in the ACL2 programming language, is outside the purview of the ACL2 logic. Our choice is motivated in part by the fact that we are interested in reasoning about the translated design and not in the translation process, and thus shallow embedding appears to be a "cheaper" alternative. To cope with the semantic imprecision we restrict the subset of Verilog supported by **V2L2** in several ways. Roughly, we restrict the input designs so that (a) we can translate them without any loss of semantic information, and (b) the translated formal system is not substantially more complex than the source Verilog. We will understand the restrictions better as we describe the details of the translation process.

Our choice of shallow embedding for implementing **V2L2**, of course, limits what we can formally claim given a successful verification of an RTL design translated from Verilog to ACL2. Obviously we cannot formally reason about the translator implementation itself, since has not been defined as a legal, formal theory

via extension principles. Thus when we claim to have proved the correctness an RTL design translated using **V2L2**, the claim refers to the model of the system that is the output of the translator and *not* the source Verilog. For instance, in the next chapter, we will verify the correspondence between two RTL implementations of the Y86 processor. In doing so, we only claim the correspondence between the ACL2 models of the implementation. In order to assert the correctness of a Verilog implementation, there must therefore be some onus on the human reviewer to carefully review the input and output to the translator and convince himself that the translation is sound. Nevertheless, we *do* believe that the output of **V2L2** preserves the Verilog semantics for designs in its supported subset. The translator signals error when the input Verilog is not in this subset.

13.3 An RTL Library

Verilog contains several data types including bit vectors, arrays, and (signed and unsigned) integers. The language specifies several operators that manipulate these data types. We have already seen some of the bit vector operators above, such as bitwise conjunction, disjunction, and complementation. The first step in translating Verilog designs to ACL2 is the formalization of such operations.

To this end, we develop a library of formal definitions in ACL2 specifying the semantics of the different Verilog operators. The library, of course, is an independent entity from the implementation of the **V2L2** translator. The translator merely uses the functions defined in the library for translating Verilog expressions. However, since it is the formal definitions of the functions in the library that provide semantics to the output of the translator, we consider the library a central piece of

our implementation.

Figure 13.2 shows some of the bit vector operations formalized in our library, together with the corresponding Verilog expressions wherever appropriate. We represent a bit vector as a Boolean list. That is, the bit vector `1011` will be represented by the list $\langle \mathtt{T}, \mathtt{NIL}, \mathtt{T}, \mathtt{T} \rangle$. With this representation, most of the functions shown are not very hard to define. For instance, the following is the definition of *bv-neg*:

$$\textit{bv-neg}(x) \triangleq \begin{cases} \mathtt{NIL} & \textit{if } \neg \textit{consp}(x) \\ \textit{cons}(\mathtt{T}, \textit{bv-neg}(\textit{cdr}(x))) & \textit{if } \neg \textit{car}(x) \\ \textit{cons}(\mathtt{NIL}, \textit{bv-neg}(\textit{cdr}(x))) & \text{otherwise} \end{cases}$$

Nevertheless, there are still certain "wrinkles" involved in defining RTL operations consistent with Verilog semantics. For instance, consider the trivial matter of representation of a single bit. How should we do it? Given our representation of a bit vector, it might have seemed that we would represent a bit by a single Boolean value, namely `T` for the bit `1` and `NIL` for the bit `0`. However, here is a problem. Most Verilog operators treat a single bit isomorphically with a bit vector of width 1. For instance, bits can be complemented or conjoined exactly like bit vectors using the same operators. If we are to be faithful to the semantics of Verilog, this isomorphism must be manifested in the formal definitions. To achieve this for bit operations, we represent a bit not as a Boolean `T` or `NIL`, but rather as the list $\langle \mathtt{T} \rangle$ or $\langle \mathtt{NIL} \rangle$. Incidentally, this representation also enables us to differentiate between the "empty bit vector" (which arises in the formal definition but has no significance in Verilog) from the single bit `0`. Similar representation issues also arise for other data types.

Our library contains about 60 definitions. Formal definitions have been provided for essentially all Verilog operators on bits and bit vectors. In addition,

Function	Interpretation	Verilog
bvp(x)	Returns `T` if x is a bit vector else `NIL`	
nlv(x)	Returns `T` if x is “all zero” else `NIL`	
unv(x)	Returns `T` if x is “all one” else `NIL`	
width(x)	Returns the width of bit vector x	
bit1()	Constant representing bit `1`	`1`
bit0()	Constant representing bit `0`	`0`
bv-neg(x)	Returns bitwise complement of vector x	`~ x`
bv-and(x, y)	Returns bitwise conjunction of x and y	`x & y`
bv-or(x, y)	Returns bitwise disjunction of x and y	`x | y`
bv-xor(x, y)	Returns bitwise exclusive-OR of x and y	`x ^ y`
bv-eqv(x, y)	Returns bitwise exclusive-NOR of x and y	`x ~^ y`
bitn(i, x)	Returns the i-th bit of vector x	`x[i]`
bits(i, j, x)	Returns the sequence of all bits from index i to j in x	`x[i:j]`
setbitn(i, v, x)	Returns bit vector x with i-th bit set to (bit) v	
bv+(x, y)	Returns 2’s complement sum of bit vectors x and y	`x + y`
bv-(x, y)	Returns 2’s complement difference of x and y	`x - y`
*bv**(x, y)	Returns 2’s complement product of x and y	`x * y`
bv-div(x, y)	Returns 2’s complement quotient of x and y	`x / y`
bv-mod(x, y)	Returns 2’s complement remainder of x and y	`x % y`
l-equal(x, y)	Returns *bit1*() if x and y are equal, else *bit0*()	`x == y`
l-inequal(x, y)	Returns *bit0*() if x and y are equal, else *bit1*()	`x != y`
l-geq(x, y)	Returns *bit1*() if $x \geq y$ (in 2’s complement), else *bit0*()	`x >= y`
l-lt(x, y)	Returns *bit1*() if $x < y$, else *bit0*()	`x < y`
l-not(x)	Returns *bit1*() if both *nlv*(x) holds, else *bit0*()	`!x`
l-and(x, y)	Returns *bit0*() if either *nlv*(x) or *nlv*(y) holds, else *bit1*()	`x && y`
l-or(x, y)	Returns *bit0*() if both *nlv*(x) and *nlv*(y) hold, else *bit1*()	`x || y`
u-or(x)	Returns the unary disjunction of bits of x	`|x`
u-and(x)	Returns the unary conjunction of bits of x	`&x`

Figure 13.2: Functions Representing Bit Vector Operations

we support some operators implementing Verilog's fixed-point arithmetic and 1-dimensional arrays. As above, we take special care to ensure that the formal definitions are conformant with the semantics of the Verilog operators.

Of course, building a library in ACL2 entails more than "just" defining functions. One must also prove lemmas and theorems to facilitate reasoning about those definitions. In our context, we want to use such theorems both for deductive reasoning and as rewrite rules for applying predicate abstractions while proving invariants on RTL systems. Our library contains about 150 such theorems. Since the focus of this chapter is on *modeling* and implementing a translator of Verilog rather than formal reasoning, we refrain from discussing such theorems or their proofs. We will discuss some of the rules developed in the next chapter when we discuss the verification of an RTL processor design that has been translated from Verilog to ACL2 and hence uses the definitions in the library.

We must clarify that our library is not by any means superior to existing libraries [BH97a, FKR+02] already available with the ACL2 theorem prover for reasoning about RTL operations. These previous libraries represent many years' worth of effort by several researchers and contain thousands of lemmas about different facets of RTL operations. In contrast, our library is small and admittedly incomplete. Why did we build our library then, and not simply use one of the existing ones? Since we planned to use the library in connection with translation of Verilog designs, one of our explicit goals has been to define operations with semantics as close to Verilog as possible. This is not true of many of the existing libraries. For example, the Integer Hardware Specification Library of Brock and Hunt [BH97a] represents bit vectors as natural numbers. In that representation, both the bit vec-

tors `000` and `0000` will be represented by the number 0, a state of affairs not in conformance with Verilog. Another reason we built our own library is so that we could experiment with designing rewrite rules and understanding which rules are useful in connection with predicate abstraction for normalizing terms arising commonly in invariant proofs of RTL designs. Experimentation requires that we work with a small, manageable set of definitions and theorems; our library, rather than one of the existing ones, seemed more suitable for the activity.

13.4 Translating Verilog Language Constructs

We can now implement our translator **V2L2** from Verilog to ACL2. The goal of the translator is to generate a sequence of definitions that stipulate the state transitions of its input Verilog design. The sequence is admissible by the extension principles of ACL2 in any extension of a theory $\mathcal{T}$ that contains the definitions provided in the RTL library and the definitions of the records operations which we saw in Chapter 11. The latter definitions are necessary since (as we will see) a state of the system defined by the output of **V2L2** is represented as a record.

As we saw in Section 13.1, a Verilog module definition consists of (1) a *preamble* containing declaration of various signals, and (2) a collection of *statements* stipulating how the signals behave. We describe **V2L2** principally by showing statements in Verilog and their formal translations. Our description is bottom-up. We first describe how simple statements are translated, then show how they are composed, and finally lead up to how we translate entire modules.

We start with the translation of the following trivial Verilog statement:

```
assign x = b[1] & c;
```

Here `x` is assumed to have been declared as a `wire` of type bit (that is, scalar), `b` is a bit vector, and `c` is a bit. The result of our translation is the following definition:

$$\textit{wire-x}(b, c) \triangleq \textit{bv-and}(\textit{bitn}(1, b), c)$$

How do we implement this translation? For a (continuous) assignment statement, we determine the signals involved in the right hand side of the assignment (in this case `b` and `c`). The translated function definition contains these signals as arguments. In order to translate the right hand side of the assignment, we maintain a table of Verilog operators and the name of the corresponding function in our RTL library together with some other book keeping information. The table can be extended by adding more function names and extending the RTL library with definitions corresponding to those names. Then translating an expression in the right hand side of an assignment statement merely involves looking up this table and determining the term that forms the body of the definition.

What should we do if the signal `x` being assigned also occurs in the right hand side of the assignment? For instance, consider the following statement:

```
always @(posedge clock) begin x <= x & b;  end
```

This is possible only if `x` is of type `reg` rather than `wire`. We translate the statement above to the following definition:

$$\textit{reg-x}(x, b) \triangleq \textit{bv-and}(x, b)$$

We have tacitly used some of the restrictions imposed by **V2L2** to the subset of Verilog it supports. We spell these out now. The reason for imposing such restrictions is so that we can accurately translate the source Verilog and determine a formal model without losing semantic equivalence. In our implementation we define a predicate that can "check" if an input Verilog module satisfies all the imposed restrictions.

1. An assignment statement assigning a wire cannot have the same wire in the right hand side of the assignment.

2. The continuous assignment statement `assign` can be applied only to wires.

3. The registers can be updated only using `always` statements (or by instantiation of other modules which we discuss later).

4. Wires cannot be assigned inside an `always` statement.

5. Every register must be clocked, and the same clock must be used for all registers. In addition, the controlling event of the `always` statement (namely `posedge` or `negedge`) must be the same for all registers.

To understand the restrictions, we must explain what the generated functions really "mean". The output of the function returns the value of the signal being assigned, given the value of the input signals. What happens if the signal being assigned is a wire? A wire simply transmits logic from its input to its output. Ignoring gate delays (as we do) we can think of the transmission to be instantaneous. A wire cannot "remember" its previous value. This is imposed by the first restriction, namely that the right hand side of an assignment to a wire cannot have the wire itself. In fact, soon we will talk of an even stronger restriction on how the wires can be laid out. On the other hand, registers *do* remember their previous value. Thus, when we define a function specifying the update to a register, we can have the register signal itself as one of its arguments. The way to think about this is that the function takes the current configuration of the register (and other signals) and returns the updated configuration.

With this view, let us pay some attention to restriction 5, which is, admittedly a severe one. The use of this restriction is manifested in our second example

above, in the fact the variable `clock` does not appear in the translated definition. The restriction says that the source Verilog must implement a *synchronous* digital system. Why do we want this restriction? As we mentioned above, we want to think of a register assignment as the value of its *next configuration* given the value of its current configuration. However, if the updates to registers are not synchronized then it is difficult to determine what the next configuration means. For example one register might be updated in picoseconds and another in nanoseconds. The updated state of the second register, then, might depend upon the intermediate updates to the first. In such cases, it is unclear how we can define functions to translate the update of the second register.

The above restrictions also have the side-effect of precluding some of the ambiguous constructs of Verilog such as "`always @*`". This construct is sometimes used to model a wire that should be evaluated whenever *any* event occurs. We do not know of ways to translate such constructs elegantly. In addition to the above restrictions, we only support non-blocking assignment (denoted by "`<=`") for registers.

Let us now move on to another construct, namely the "if-then-else" statement of Verilog. This construct is important since its translation can span multiple statements in a module. Consider the following statement:

```
if a begin x <= b; end
```

Assume that this is the only statement that assigns `x`. What should we do to the above statement? If `x` is a register, we allow such assignments; in such cases, we translate the statement as follows:

$$\mathit{reg\text{-}x}(x, a, b) \triangleq \begin{cases} x & \mathit{if}\ a \neq \mathit{bit0}() \\ b & \text{otherwise} \end{cases}$$

The function is self-explanatory; if `a` is the bit `0`, we assign `b` to `x`, otherwise leave `x` unchanged. However, we cannot do that if the assignment were to a wire. This imposes the following restriction:

6. If there are conditions controlling assignment to a wire, then the disjunction of the conditions must evaluate to *bit1*(), and the conditions must be mutually exclusive.

This condition is tantamount to tautology checking. We do not implement full tautology checking, but rather a restricted form based on some heuristics.

What do we mean by "disjunction of conditions" above? The following two statements illustrate the point.

```
if a begin x = b; end
if ~a begin x = c; end
```

This sequence of statements *can* be translated by **V2L2** when `x` is a wire since the disjunction of `a` and `ã` *is* equal to *bit1*() (when `a` is declared to be a bit), and the conditions *are* mutually exclusive. We translate them as follows:

$$\mathit{wire\text{-}x}(a, b, c) \triangleq \begin{cases} b & \mathit{if}\ x \neq \mathit{bit0}() \\ c & \text{otherwise} \end{cases}$$

Incidentally, the same definition is produced for the translation of the following statement (as expected):

```
if a begin x = b; end else begin x = c; end
```

The translation of "if-then-else" and its derivative, the `case` statement, can span multiple statements. The latter is actually translated by interpreting it as a chain of "if-then-else" statements. Verilog also has a `casex` statement that allows "don't cares", but we do not support `casex`.

Keeping track of a number of statements for creating a definition is obviously cumbersome. The only situation (other than "if-then-else") in which we support this

is in updates of bit vectors and arrays. We discuss only the update of a bit vector since the situation for arrays is analogous. Consider the following two statements:

```
assign c[0] = a; assign c[1] = b;
```

Here `c` is assumed to be a wire which has been declared as a bit vector of width 2, and `a` and `b` are single bits. We translate the sequence as follows:

$$\mathit{wire\text{-}c}(a, b) \triangleq \mathit{setbitn}(0, a, \mathit{setbitn}(1, b, \mathit{nlv}(2)))$$

For this to satisfy the semantics of Verilog, we need the following additional restriction.

7. If `w` is a bit vector and declared to be a wire, then there must be (a sequence of) statements specifying updates to each component bit in `w`.

Of course, the restriction is applicable only for wires and not registers. For instance the following statement can be handled by the translator where `c` is a register of length 2:

```
always @(posedge clock) begin c[0] <= a; end
```

Just for the sake of completeness, this produces the following definition:

$$\mathit{reg\text{-}c}(c, a) \triangleq \mathit{setbitn}(0, a, c)$$

We now discuss translation of a module. This is the most interesting and challenging aspect of the translator implementation. **V2L2** supports three modes of module translation, which we call *interface translation*, *state translation*, and *timing translation*. We discuss each in turn.

For *interface translation*, we simply extend the approach we have taken so far to "work" with modules. Recall that a module contains some input and output ports, and some internal signals. Interface translation produces one function for

each output and each (internal) register. If a register is also marked as an output, then one function is produced instead of two. Each function takes as argument (a subset of) the set of inputs and internal registers. To understand how the translation works, let us quickly look at the translation of the `dEdgeFF` module we described in Figure 13.1. That module had one output which was also a register. Interface translation of the module produces the following function.

$$\mathsf{dEdgeFF\text{-}q}(data) \triangleq data$$

Notice that the only argument to this function is *data*. This is because the output (and the value stored) changes at every transition. If the assignment were conditional then q would have been an additional argument as well.

Interface translation, though trivial in the above case, might be complicated in general. This is because the value of an output might depend on computation involving a number of internal wires. Notice that internal wires are *not* arguments to the function defining the value of an output.

How do we manage to do it? As the reader might well anticipate by now, this is done by imposing more restrictions on the input Verilog. We discuss the restriction involved in some detail below since it is crucial to the implementation of the translator.

8. The input module should not have a "loop" consisting only of wires. In other words if there is a path in the module from a wire `w` back to itself, then there must be some register in the path.

The restriction, though crucial, is actually a common one in the formal verification community [BH97b, Hun00]. It is often succinctly described as: "The design must not have any combinational loops." The reasons for its commonality are not hard to understand. If the value of a signal `w` at any instant depends on the value of `w` itself

at the *same* instant, then the value held by the signal can be ambiguous. Notice that this restriction subsumes restriction 1. **V2L2** does a light-weight, conservative check for this circularity by checking for combinational cycles in the graph induced by the source Verilog. Since the check is purely syntactic, however, we can "reject" source Verilog where the combinational loop is spurious. For instance, a module with the statement `assign x = 0 & x;` will be rejected by our translator even though the value of `x` can be unambiguously determined to be always `0`.

What has absence of combinational loops got to do with interface translation? A consequence of the restriction is that the value of any signal depends only on the value of registers and inputs to the module. This allows us to define the interface translation for a module as follows. For every output `o`, we topologically sort the signals determining the signals in the module from which there exists a combinational path to `o`. The "leaves" in the output of the sort must, by our restriction, be either registers or inputs. Furthermore, each wire `w` encountered in the process must depend only on wires "below" `w` in the topological ordering. We can then traverse the directed acyclic graph so produced bottom-up, collecting the definition generated for the assignment of each wire encountered. Appropriate composition of these definitions gives the definition for output `o`.

As an aside, it is restriction 8 that allows us to support the continuous assignment statement `assign`. Recall from Section 13.1 that the semantics of `assign` is pretty complicated. It needs to be evaluated every time the expression on the right hand side of the statement changes. In the presence of combinational loops, the evaluation is akin to a fixpoint computation [Saw04]. However, when such loops are disallowed, we can unambiguously specify the value of the variable assigned.

$$\begin{aligned}\textit{reg-count}(count) \triangleq\ & \textit{setbitn}(0, \textit{dEdgeFF-q}(\textit{bv-neg}(\textit{bitn}(0, count))),\\ & \quad \textit{setbitn}(1, \textit{dEdgeFF-q}(\textit{bv-xor}(\textit{bitn}(0, count)),\\ & \qquad \textit{bitn}(1, count)),\\ & \quad \ldots\\ & \quad count)..)\end{aligned}$$

Figure 13.3: Translating Module Instantiation of 4-bit Counter

So far, we have talked about the interface translation of a *module definition*. We now talk about module instantiation. Indeed, the reason we implemented interface translation was so that we could use the same functions defined for a module definition in every instantiation of the module. Let us look at how we can make use of it. Consider the module `m16` in Figure 13.1. It has four instantiations of the `dEdgeFF` module, which sets the different bits of `count`. Figure 13.3 shows a fragment of the formal definition of the translation of this set of instantiations. What is important to note is that the same function *dEdgeFF-q* has been "called" to formalize instantiation of the `dEdgeFF` module.

Interface translations are useful for translating submodules of a main Verilog module, and composing such translations. But the definitions shown so far do not correspond to the formalization of reactive systems that we have talked about in the last two parts. The goal of *state translation* is to produce definitions corresponding to our formalization of reactive systems. Indeed, when we claim to have verified an RTL design implemented in Verilog, the system we have formally proven correct is the one specified by the output of state translation.

So what does the output of state translation look like? We think of the "state" of the reactive system specified by Verilog as a record of all the signals in the module. The output of state translation is a function that specifies the updates

of every field of this record. To understand how it works, consider the function shown in Figure 13.3 showing the update to `count`. Of course, in our record, we have a field for the signal `count`. Let this field be `"count"`. From the body of the definition of *reg-count*, we now create the following term τ:

$$\tau \doteq \mathit{setbitn}(0, \mathit{dEdgeFF\text{-}q}(\mathit{bv\text{-}neg}(\mathit{bitn}(0, (\mathit{get}(\texttt{"count"}, s)))), \ldots, \mathit{get}(\texttt{"count"}, s)))$$

Recall from Chapter 11 that *get* and *set* are functions defining access and update of records respectively. Then the update to the `"count"` field in a state s is specified by the term $\mathit{set}(\texttt{"count"}, \tau, s)$. The state transition function is a composition of these updates over all fields of the record.

All this is simple to do. Nevertheless, the reader might be surprised by our statement that the "state" is represented by a record of *all* signals. Normally, one would think that only the signals that specify storage elements (that is, registers) would be preserved as fields of our state record. Why do we preserve all signals?

The reason we do this is because the other signals act as auxiliary variables, which record the relevant history of computation. Recall that we have proved in Chapter 8 (page 146) that augmenting *any* reactive system model with arbitrary auxiliary variables preserves trace equivalence. Hence we can legitimately reason about such an augmented model and use our reduction theorem to claim the correctness of a system without the irrelevant variables. On the other hand, if we were to do a refinement proof verifying a system we would have augmented it anyhow with some of the variables keeping track of the wire update. Rather than first creating a "minimal" system and then augmenting it manually, we prefer to work with the augmented system directly. However, to make sure that the extra signals do correspond to auxiliary variables, we do not allow the updates to "register fields"

of the record to depend on the values of the wires. We have seen how to do that already, namely via topological sorting of signals. We use the same approach to define the update of every field in the state translation.

As a final point in our description of state translation, we discuss the issue of defining the initial state of the generated reactive system model. We have been avoiding this issue all along, focusing instead on designing function definitions to formalize the updates to the different signals. But a reactive system model must also provide a definition of the initial state. This is slightly subtle issue since a Verilog module might have no specification of initial state. This brings us to the "final" restriction that we discuss for our translator, which is the following:

9. Each module must have either an initial block specifying the initial value of every register, or the user must specify a signal that is used to reset the system. The initial state of the registers in the latter case will be derived by applying the reset signal on the registers.

What do we do about our auxiliary variables? Since they correspond to wires in the design, we should not need to know about their values at the initial "state". In order to specify the value of a wire at the initial state of our model, we generate, for each wire `w`, a 0-ary encapsulated function *unknown-w*() that is constrained to return only the data of the appropriate type. For instance if `w` is declared to be a bit vector of width 32, we specify that *unknown-w*() returns a 32-bit vector.

We end with a brief comment on the final mode of our translation, namely *timing translation.* The name for this mode is a misnomer since this mode has very little to do with Verilog, but rather with the system generated after state translation. Recall from Chapter 10 that our predicate abstraction technology requires that updates to the different components of reactive systems be specified as functions of

"time". Of course we understood that it is easy to transform a "state based" model to one that was based on time. But with large modules we do not want to do this manually. Our translator performs this transformation for us. Thus for each field f in our reactive system model, we specify a function $\hat{f}$ that is a unary function of time. It also generates a collection of lemmas relating this "time based" model with the output of state translation.

13.5 Summary and Comments

We have presented a translator called **V2L2** to translate Verilog designs to ACL2. The translator has been implemented to support a carefully specified subset of Verilog, that can be unambiguously translated to a formal logic. Nevertheless the supported subset is rich enough to model interesting hardware like processor modules, memory, etc. For instance, the processor implementations we will talk about in the next chapter were translated using **V2L2** from a (behavioral) Verilog implementation to a formal ACL2 model. The use of **V2L2** facilitates the process of bridging the gap of faith between Verilog modules that are used in simulation and fabrication of design and their formal models used in reasoning about such designs. Although we implement shallow embedding, we take reasonable care to ensure that the formal definitions generated are consistent with the semantics of Verilog.

V2L2 is work in progress, and is deficient in many respects. No support is provided for many important features of Verilog, for example Verilog functions, facilities for timing analysis, and so on. The procedure only supports a minimal set of Verilog features that we found sufficient to translate all the RTL designs we have experimented with. Indeed, the translator itself was implemented in an extendible

fashion starting with a tiny core which was augmented with more features as they came up in the designs we encountered. We are working on making the translator more robust. Nevertheless, we believe that a better approach to reasoning about RTL designs is to develop a formal semantics of an HDL via deep embedding. Significant progress has been made in that direction by the ACL2 community. For instance, Hunt and Reeber [HR05] develop a formal semantics of an HDL called **DE2** which supports many features of a modern HDL. We believe that when such a formalization becomes mature enough to support some of the richer set of constructs, particularly ones for behavioral RTL specifications, it will provide a better framework for reasoning about RTL designs.

13.6 Bibliographic Notes

Verilog originated at the Automated Integrated Design Systems (later renamed Gateway Design Automation) in 1985. Many books have since been written, describing the language features [TM96, Pal03] and associated simulation and design methodologies [Max04, Ber03]. Verilog and VHDL [Bha92] constitute two of the most widely used HDLs for modeling and implementing commercial digital systems. Verilog is routinely used for microprocessor designs in companies like AMD, Intel, IBM, etc. For instance, currently high-performance high-reliability microprocessor called TRIPS [BKD$^+$04] is being developed in Verilog at the University of Texas at Austin in collaboration with IBM.

Several research projects have concerned themselves to finding a way of reasoning formally reasoning about digital systems implemented in an HDL. To this end, Gordon [Gor95] developed a semantics of a subset of Verilog in the HOL logic,

and Russinoff developed one for a subset of VHDL in Nqthm [Rus95]. Russinoff and Flatau have also developed a translator for translating designs written in a proprietary RTL language designed by AMD to ACL2, and used this translator to verify a floating point multiplier [RF00, Rus00]. Recently, their translator has been "upgraded" to translate a Verilog subset. We believe that our translator is very close to this work, although the supported language subsets are probably different. There has also been work on formalizing an HDL via deep embedding in ACL2. Brock and Hunt [BH97b] introduced such a formalization called **Dual-Eval** and used it to model the FM9001 microprocessor at the netlist level [BHMY89]. FM9001 constituted the formal microprocessor model for the CLI stack verified using Nqthm. Hunt [Hun00] augmented **Dual-Eval** to create the **DE** HDL that was formalized via deep embedding in ACL2. Recently, Hunt and Reeber [HR05] have improved **DE** further, creating an HDL called **DE2** that provides an annotation language, and support for λ-expressions and parameterized evaluations.

The ACL2 community has spent considerable effort in designing libraries of lemmas and theorems for reasoning about RTL designs. Two such libraries are currently distributed with the theorem prover releases. The first, developed by Brock and Hunt, is called the Integer Hardware Specification (IHS) library [BH97a]. This library has been used in the verification of the Motorola CAP DSP processor [BH99] and the floating point operations of the IBM Power 4 microprocessor design [SG02]. Another library [FKR+02], developed in collaboration with AMD, has been used for verifying operations of the AMD Athlon™ processor [Rus98, RF00].

The implementation of **V2L2** uses early work on Verilog parsing by Vinod Vishwanath.

Chapter 14

Verification of a Pipelined RTL Microprocessor

We now have all the pieces necessary for formally reasoning about RTL designs. We will put our verification methodology to test by verifying a reasonably complex RTL implementation of a microprocessor. The microprocessor we verify is a slightly simplified version of the Y86 processor developed at the Carnegie-Mellon University, principally to teach students the different facets of a modern processor design. The processor is described in a book by Bryant and O'Hallaron [BO03]. Although undoubtedly simpler than a modern commercial processor, it has several subtle and complex features like branch prediction, speculative execution, and exception handling. All these features, of course, are implemented at the level of (behavioral) RTL via bit vector manipulations.

In this chapter, we first provide a brief overview of the Y86 instruction set architecture, the different implementations of the processor, and the simplifications

we introduced in the designs we verified (with our reasons for doing so). We then show how our methodology facilitates the verification of the design. Some of the text of the processor description have been adapted from the Bryant and O'Hallaron's book, including Figures 14.1 and 14.2. However, the author is solely responsible for the presentation here.

14.1 The Y86 Processor Design

We start with the instruction set architecture of the Y86. The reader familiar with the architecture of the IA32 processors (for example processors of the Intel Pentium® line) will find this processor fairly familiar, though simplified in many respects. Indeed, the name "Y86" owes its origin to the fact that its design is inspired by the IA32 architectures which are colloquially referred to as the "X86".

The Y86 processor consists of the following state components:

- The *register file* consists of eight program registers. Each register can hold a 32-bit word. The registers are referred to as `%eax`, `%ecx`, `%edx`, `%esi`, `%edi`, `%esp`, and `%ebp`. Register `%esp` is used as a stack pointer by the `push`, `pop`, `call`, and `return` instructions; the rest of the registers have no fixed meanings or values. Each program register has an associated *register identifier*, ranging from 0 to 7.

- The *program counter* (`PC`) is a 32-bit register. It holds the address of the instruction currently being executed.

- The *memory* is conceptually an array of bytes storing both the program and the data.

- There are three single-bit *condition codes* which are referred to as `ZF`, `SF`, and `OF`. They store information about the effect (zero, sign, or overflow) of the most recent arithmetic or logical operation.

The set of instructions of the Y86 processor is largely a subset of the IA32 instruction set. However, the Y86 has a smaller set of instructions, a simpler byte-level encoding of instructions, and fewer addressing modes. For instance it includes only 4-byte integer instructions. Further, some of the IA32 instructions are "split" into multiple instructions in order to simplify the semantics of their execution. For instance, the `movl` instruction of IA32 is split into four instructions `irmovl`, `rrmovl`, `mrmovl`, and `rmmovl`, explicitly indicating the form of the source and destination. That is, the source is either immediate (`i`), register (`r`), or memory (`m`), as designated by the first character of the instruction name, and the destination is either register (`r`) or memory (`m`) as designated by the second character.

There are seven types of instructions in the Y86:

1. The `nop` instruction does not change the state of the processor.
2. There are four different "load-store" type instructions `irmovl`, `rrmovl`, `mrmovl`, and `rmmovl` as described above.
3. There are four integer operations `addl`, `subl`, `andl`, and `xorl`. They operate only on registers, and set the three condition codes when applicable.
4. There are seven branch instructions `jmp`, `jle`, `jl`, `je`, `jne`, `jge`, and `jg`. The branches are taken according to the type of branch and the setting of the condition codes.
5. There is a `call` instruction which pushes the return address on the stack and jumps to the destination address. The `ret` instruction returns from a call.
6. There are two instructions `pushl` and `popl` that push and pop 32-bit data on the stack.
7. The `halt` instruction stops the processor execution.

The instructions are encoded using from one to six bytes. The initial byte of each instruction identifies the instruction type. This byte is split into two 4-bit parts, namely the higher *code* part and lower *function* part. The function values are significant only for cases where a group of related instructions share a common code. For instance, the seven types of branch instructions have the same code, but differ in the function values.

Some instructions, such as `nop`, `halt`, and `ret`, are 1-byte long, but those that require operands are longer. First there is a *register specifier* byte, specifying either at most two registers. These registers are called `rA` and `rB`. They specify the registers used in data sources and destinations, as well as the base register used in address computation, depending on the instruction type. Instructions that require no register operands, such as `jmp` and `call`, do not have a register specifier byte. Those that require only 1 register operand, such as `irmovl` or `pushl`, have the other register specifier bit set to 8. Recall that the register identifiers for the program registers range from 0 to 7.

Some instructions require an additional 4-byte constant word. The word can serve as immediate data for `irmovl`, the displacement for `rmmovl` and `mrmovl` address specifiers, and destination for branches and calls. Integers have little endian encoding; that is, when an instruction is written in disassembled form, the bytes appear in reverse order. In contrast to IA32, the destinations of branches and calls in the Y86 are given as absolute addresses rather than `PC`-relative ones.

14.2 The Y86 Implementation

The above description should make it clear that the Y86 system is a fairly standard 32-bit processor, with some simplifications for pedagogical reasons. How is the processor implemented? Bryant and O'Hallaron provide four different implementations of the processor, which are called **seq**, **seq+**, **pipe-**, and **pipe**. **seq** provides a relatively direct implementation of the Y86 instruction set architecture. The object of our verification is the pipelined processor **pipe**, which provides all the optimizations and pipelining. The processors **seq+** and **pipe-** are intermediate designs developed for didactic reasons, and will not be of interest to us. In the remainder of this section we provide a quick overview of the hardware structures of the **seq** and **pipe** implementations of the Y86.

14.2.1 The seq Implementation

The **seq** processor is a relatively simple-minded implementation of the Y86 instruction set architecture. It executes every instruction in one cycle, and has no pipelining or other optimizations. From the programmer's viewpoint, it is a simple enough machine to study and understand. The hardware structure of **seq** is shown in Figure 14.1. To describe the execution of the processor, we discuss its processing as if organized in stages. Of course since **seq** has no pipelining, the name "stage" is a misnomer in this context. Nevertheless, presenting its execution as a sequence of stages will clarify the overall structure of the design.

Fetch: This stage reads an instruction using the `PC` as the memory address. From the instruction it extracts two 4-bit portions of the instruction specifier byte, referred to as `icode` (instruction code) and `ifun` (instruction function). It also possibly fetches the register specifier byte (giving one or both of the register operand specifiers `rA` and

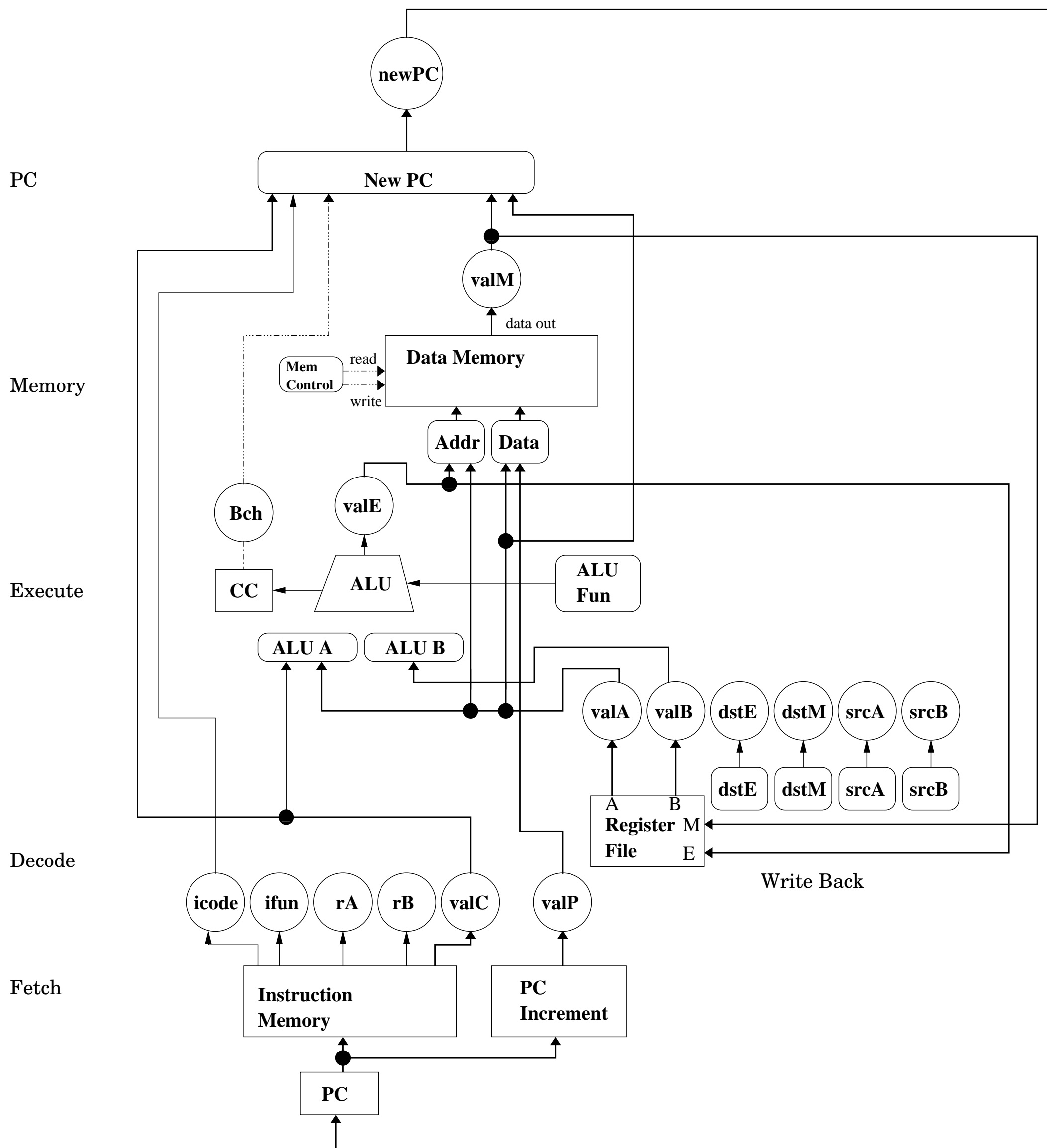

Figure 14.1: Hardware Structure of the **seq** Processor

rB), and the 4-bit constant word valC. It then computes valP, the address of the next instruction in sequential order. This address is equal to the value of the PC plus the length of the fetched instruction.

Decode: This stage reads up to two operands from the register file, giving values valA and/or valB. Typically it reads the registers designated by the instruction fields rA and rB, but for some instructions it reads the register %esp.

Execute: In this stage, the ALU either performs the operation specified by the instruction (according to ifun), or computes the effective address of a memory reference, or increments or decrements the stack pointer. The resulting operation is referred to as valE. In addition, the condition codes are set in this stage if applicable. For a branch instruction, the condition codes are tested to decide if the branch should be taken.

Memory: In this stage data is written back to the memory. Also, data might be read from the memory. In case of the latter, we refer to the data read as valM.

Write Back: In this stage, results are written to the register file when appropriate. Up to two results can be written simultaneously.

PC Update: In this stage the PC is set to the address of the next instruction. Notice that the address of the next instruction might be different from the valP computed in the **Fetch** stage; for example if the instruction is a jmp instruction then the address of the next instruction is the destination of the instruction.

A transition of the **seq** processor constitutes sequential execution of the six stages above. The processor loops infinitely performing one transition per clock cycle until it encounters either a halt instruction or some error condition. The error conditions include attempt to access an invalid memory address or attempt to execute an invalid instruction.

14.2.2 The pipe Implementation

The **seq** processor is a workable microprocessor design implementing the Y86 instruction set architecture, but is not very efficient. The **pipe** processor, on the other hand, is a pipelined implementation of the Y86 instruction set architecture. The hardware structure of **pipe** is shown in Figure 14.2. The stages of **pipe** roughly correspond to the stages in **seq** with one exception. There is *no* **PC Update** stage, and in fact there is no register storing the program counter value for the current instruction. The program counter (referred to as `f-pc`) is computed dynamically based on the state information. To facilitate pipelining, **pipe** contains five latches (or *pipeline registers*) **F**, **D**, **E**, **M**, and **W**, that "sit" between consecutive stages holding the results of partial computation of different instructions in overlapping execution as follows.

F holds the *predicted* value of the program counter for the next instruction. The predicted value is the same as the value `valP` that **seq** computes for instructions other than (conditional) branches. For conditional branch instructions, **pipe** uses an *always taken* branch prediction strategy; thus `F` holds the address at the destination of the branch.

D is between the **Fetch** and **Decode** stages and stores information about the most recently fetched instruction.

E holds information about the most recently decoded instruction and the values read from the register file for processing by the **Execute** stage.

M holds the results of the most recently executed instruction for processing by the **Memory** stage and the information about branch conditions and branch targets for processing conditional jumps.

W is between the **Memory** stage and the feedback paths that supply the computed results to the register file for writing and the return address to the `PC` selection logic.

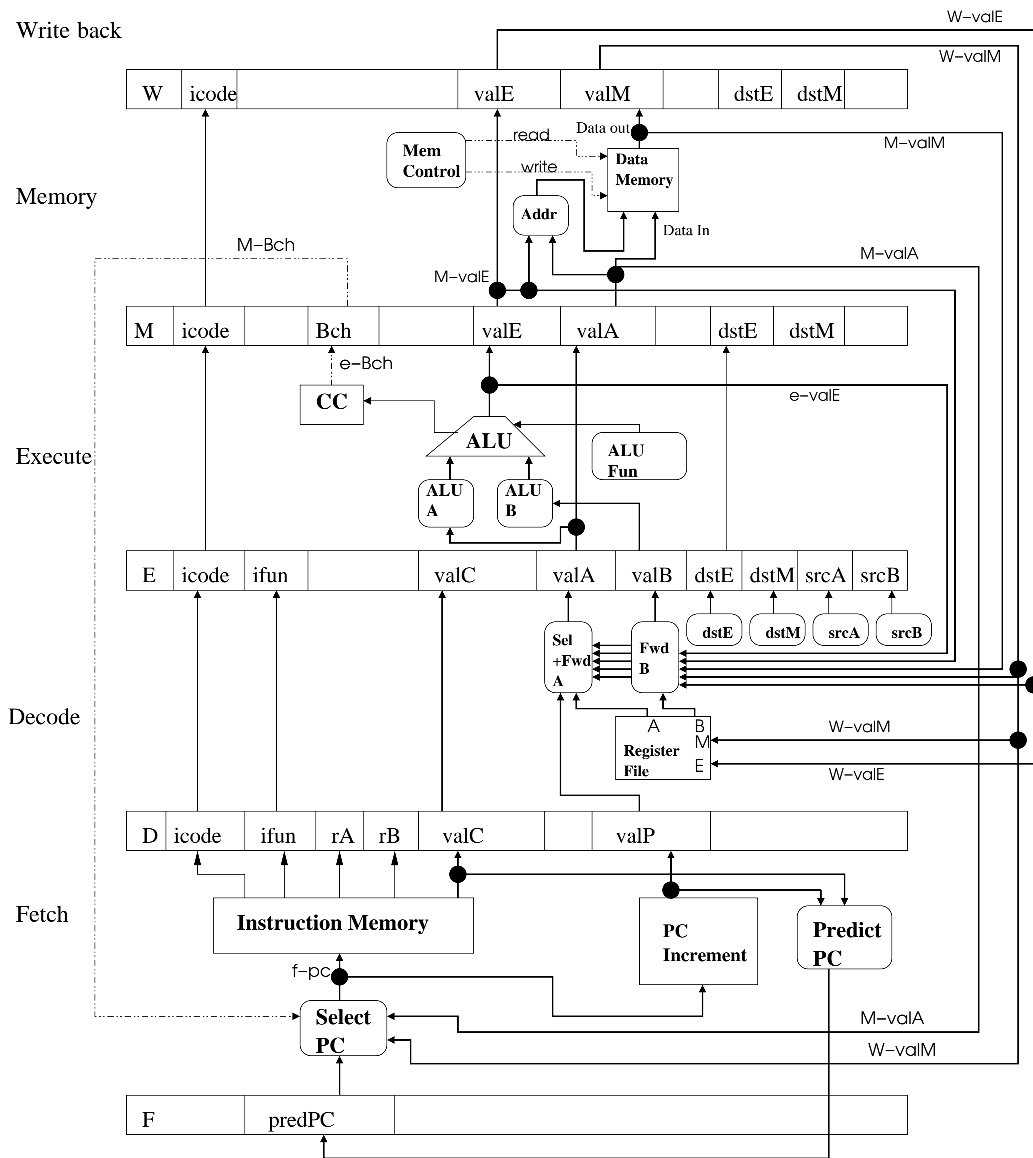

Figure 14.2: Hardware Structure of the **pipe** Processor

The pipelined system, of course, also needs several control mechanisms in order to handle the different pipeline hazards. Such mechanisms are provided in the form of both pipeline stalls and data forwarding. The mechanisms are not new or novel, and are available in some form in most modern pipelines. As is customary, data forwarding is preferred over pipeline stalls whenever possible to resolve dependencies between two incomplete instructions at different stages of the pipeline. Data forwarding is implemented by providing bypass paths from the **M** and **W** latches and the output of the ALU to the **Decode** stage. Thus if the instruction i in the **Decode** stage needs to read a register r, and some instruction in **Execute**, **Memory**, or **Write Back** stage has the same register r as its destination, then the bypass path allows the value about to be written to r to be passed directly to the **Decode** stage for processing of instruction i.

Under normal operations, **pipe** introduces stalls and bubbles for three special cases in which the data dependencies between instructions cannot be resolved by forwarding alone. These cases are the following:

Memory Dependencies: Memory dependencies arise since the memory is read by instructions late in the pipeline. Consider a sequence of two instructions of which the first is `mrmovl` that writes the value of some memory location to a register r, and the second instruction (say i) has register r as one of its source operands. Then the value of the register r must be resolved when i is in the **Decode** stage. But when i is in **Decode**, the `mrmovl` instruction is in the **Execute** stage, and it reads the memory (and hence the value that must be stored into r and read by i) only in its **Memory** stage. Thus, data forwarding cannot be used to resolve this dependency. This is solved in **pipe** by introducing a *bubble* in the **E** latch and holding back instruction i in the **Decode** stage. Forwarding from the output of the data memory to the **Decode** stage allows i to correctly obtain its operand in the next transition.

Procedure Returns: A thorny issue arises in the processing of the `ret` instruction. Recall that `ret` returns from a procedure `call`. The return address is stored in the memory which is indexed by the stack pointer `%esp`. This address thus can be obtained at the **Memory** stage. The **pipe** implementation does not attempt to predict the return address for a `ret` (although this feature is normally present in modern processors), but rather stalls the pipeline for three transitions until the `ret` reaches the **Memory** stage and the return address is resolved. The processing of `ret` actually has an important aspect which will be relevant to us in reasoning about the pipeline. In each of the three cycles as `ret` "moves up" from the **Fetch** to the **Memory** stage, **pipe** actually *does* fetch an incorrect instruction. The way this is dealt with is as follows. At every transition, the logic at the **Fetch** stage reads an instruction from the memory at the address indexed by `f-pc`. The access to the memory cannot be stopped; however, this instruction is immediately "killed" or replaced by a bubble at the **D** latch and hence is never processed.

Mispredicted Branches: The processing of `ret` above shows one example in which an instruction entering the pipeline is subsequently killed and never processed later. Processing mispredicted branches is a more non-trivial illustration of this facet. Recall from above that **pipe** uses the branch prediction strategy of *always taken*; thus instructions in the pipe following a branch instruction are fetched from the target of the branch. If the branch is not taken then all these instructions need to be killed. The important matter to consider here is that these incorrect instructions should not modify any programmer-visible state component before they are killed. At what stage of the pipeline does an instruction first modify a programmer-visible component? The answer is "at the **Execute** stage where the condition codes are set". The decision whether a branch is taken or not is determined when the branch instruction is at the **Execute** stage, (and thus any subsequent instruction has not reached this stage). Hence to process mispredicted branches, **pipe** simply kills the (misfetched)

instructions in the **D** and **E** latches by replacing them with bubbles, and also fetches the correct instruction in the next transition. The latter is achieved by forwarding the branch decision from the **Execute** stage to the logic at the **Fetch** stage that computes the value of the program counter.

We end the description of **pipe** by briefly noting what is meant by *inserting a bubble in a latch*. A bubble is simply a 32-bit number of value 0. What is important to note is that every latch (other than the **F** latch where no bubble is ever inserted) has a "field" which is interpreted as the `icode` of the instruction in the latch (Figure 14.2). This "`icode`" corresponds to the `icode` of the instruction `nop`. Thus once a bubble has been inserted in a latch it advances to the subsequent latches exactly as a `nop` would have done, and thus does not affect any change in the visible state of the machine.

It should be clear that **pipe** embodies many features in a modern pipelined processor, although the design is significantly simpler than a commercial processor implementation. Nevertheless, several aspects of the design, in particular data forwarding and branch prediction, require subtle reasoning. The system is therefore an ideal benchmark for validating the scalability and robustness of the verification methodologies we have been talking about so far in the dissertation.

14.3 Verification Objectives and Simplifications

What should we prove about the Y86? In this case we are fortunate that we have two different implementations, namely **pipe** and **seq**. The **seq** machine is much simpler than **pipe**, can be inspected by the user, and in a very direct sense implements the instruction set architecture of the Y86. On the other hand, **pipe** concerns itself with

low level optimizations aimed at execution efficiency. Thus it is natural to treat **seq** as a specification for **pipe** and show that **pipe** implements the specification faithfully.

Can we prove (**seq** $\triangleright$ **pipe**)? Of course to make this statement formal we need to first model **seq** and **pipe** as reactive systems. This is not difficult to do. We have already discussed what the transitions of the two designs look like. We specify the initial state of each design as the state obtained by applying the `reset` signal. As we did for the simple machine in Chapter 9, we will take the *label* of a state (in both systems) to be the configuration of the register file in the state.

Unfortunately, with the **seq** and **pipe** designs as they stand now, one cannot prove (**seq** $\triangleright$ **pipe**). The reason is both simple and draconian. In both processors, instructions and data share the same memory, leaving them open to hazards due to self-modifying programs which neither processor has any control logic to prevent. The designs simply assume that the programs are not self-modifying. But for verification in a formal logic, all assumptions need to be explicit. Of course just the fact that self-modifying programs can cause erroneous execution does not prevent us from proving that **pipe** is a refinement of **seq**; the problem is that the erroneous executions in the two systems are different. For example, there may be an instruction i that modifies the memory location immediately following i with a new instruction i'. In **seq**, this would mean that i' will be executed immediately after i. In **pipe**, however, the memory will be written only at the **Memory** stage and by that time three subsequent instructions will have already been fetched from the unmodified memory, including the instruction immediately following i. Thus the executions of the two systems will be different.

How do we guard against the possibility of self-modifying code? Our simple approach is to physically split the memory component into two portions, namely the *instruction memory* and *data memory*, by stipulating that the program and the data reside in different components. Note that in the absence of self-modifying programs this is equivalent to just having a single memory with both programs and data.[1] For the rest of this chapter, when we refer to **seq** and **pipe**, we refer to formal models of the corresponding systems with this simplification. We can now prove ($\mathbf{seq} \rhd \mathbf{pipe}$).

14.4 Verification Methodology and Experience

How would we go about with this verification? One enticing possibility might be to use a flushing diagram and then use the results of Chapter 9 to deduce a refinement theorem. However, recall that we did not do anything in that chapter to facilitate the proof of a flushing diagram itself. What we showed was simply that *if given a legitimate proof of a flushing diagram* we can turn it into a proof of refinement. However, with a pipelined processor with all the features that **pipe** has, it is not simple to accomplish a flushing proof in the first place. Rather than attempting that, we prove ($\mathbf{seq} \rhd \mathbf{pipe}$) using the proof rules for verifying refinements that we developed in Chapter 8, using our predicate abstraction tool as necessary in the process.

Our verification strategy follows exactly the outline we discussed in page 236. More precisely, we will show the following chain of refinements:

$$\mathbf{seq} \trianglerighteq \mathbf{pipe}^{+} \Diamond \mathbf{pipe}$$

[1] Indeed, Bryant and O'Hallaron [BO03] implicitly assume this simplification by referring to the two memory components as separate throughout their discussion of the processors. For example, see the block diagrams in Figures 14.1 and 14.2 which are taken directly from their book.

Here **pipe**$^+$ is an augmentation of **pipe** with auxiliary variables to track some history of the computation, as we discuss below. As we mentioned in Chapter 8, this means that the proof of (**pipe**$^+ \diamond$**pipe**) is trivial, and we can concentrate on the proof of (**seq** $\trianglerighteq$ **pipe**$^+$).

Nevertheless, why do we need **pipe**$^+$? The answer in case of the Y86 is illuminating. We need **pipe**$^+$ principally so that we can define, given a state of the pipelined system, the corresponding representative state of **seq**. Recall that in order to show (**seq** $\trianglerighteq$ **pipe**), we must be able to define a function *rep* that maps a pipeline state to a state of the **seq** system. Unfortunately we cannot define such a mapping from the states of **pipe** to the states of **seq** directly. To understand this, consider a **pipe** state $\hat{p}$ where the **W** latch contains an `irmovl` instruction and the **M** latch contains an `rmmovl` instruction. Assume that there is no hazard involved, that is, the destination of `irmovl` is different from the source of `rmmovl`. How would we want to map $\hat{p}$ to a state of **seq**? Intuitively we want the mapped **seq** state $\hat{s}$ to "look" as follows:

- The program counter points to the instruction `irmovl`.
- The configuration of the memory and the register file is the same as that in $\hat{s}$.

Notice immediately that we need to know the value of the program counter of the instruction at the **W** latch to achieve this. The easiest way of doing that is to keep an auxiliary history variable that keeps track of the program counters of the successive issued instructions in the pipeline. Of course, as it stands, it is possible to *compute* the value of the `PC` from the configuration of the pipeline latches but it is cumbersome. However, there is a more subtle reason that *necessitates* the use of history variables, and that has to do with the memory update. To understand this, consider the state $\hat{p}'$ reached by **pipe** after one transition from $\hat{p}$. What has

happened to the **pipe** system? Since `rmmovl` was in the **M** latch at $\hat{p}$ and `irmovl` was in the **W** latch, and since there was no hazard, both the instructions proceed simultaneously, one updating the register file and the other updating the memory. The problem, however, is in mapping the memory configuration of $\hat{p}'$ to form a **seq** state $\hat{s}'$. By analogy with what we wanted for $\hat{p}$ we should now want that the `PC` be pointing to the the `mrmovl` instruction (which is the instruction in the **W** latch and about to be completed in **pipe**). That means that **seq** must be poised to execute `mrmovl`. But the memory in $\hat{p}$ appears as if the `mrmovl` has already been completed! This analysis shows that we cannot simply define the function *rep* so that we can project the `PC`, memory, *and* the register file together to define the **seq** state corresponding to a **pipe** state. Rather, we define **pipe**$^+$ to add auxiliary variables that keep track of the `PC` and the memory configuration necessary to define the *rep* mapping. The history variable for memory, also called `hmem`, holds the same value as the configuration of the memory in **pipe**$^+$ with the exception that the update to this variable occurs at the **Write Back** stage instead of the **Memory** stage.

Incidentally, the analysis above shows why we have preserved *only the register file* in defining the *label* of a state. We could have instead chosen the memory or the program counter, but we *cannot* choose to have all three. If we had done so then the resulting system would not be a refinement of **seq**. We hinted at the problem in Chapter 9 (page 191) in the context of pipelines with out-of-order instruction completion. That is, given a pipeline state *ma* and a matching state *isa* of the instruction set architecture, there would be *no* next state of *isa* that matches a transition from *ma* if the transition causes updates of two different components of the *label* by two different instructions. This fact, of course, is well-known in

the literature on pipelined machine verification; for example, this is the reason why Manolios [Man00a] had to remove the `PC` from the observable components when proving a WEB correspondence between a pipelined machine and its instruction set architecture. In the terminology of the literature [ACDJ01, Aro04, ADJ04] on pipeline verification, our proof of ($\mathbf{seq} \rhd \mathbf{pipe}$) can be referred to as a proof of correspondence showing *synchronization at instruction retirement*, where the observations for the implementation and specification match for the register file. We should note that if **seq** were implemented with *bursts* as we proposed in Chapter 9 instead of single instruction execution, then this problem would not exist.

Let us now consider the proof of ($\mathbf{seq} \unrhd \mathbf{pipe}^+$). To do so, we must define functions *rep*, *skip*, *rank*, and *inv*, and derive the single-step proof obligations as described in page 139. As we did for concurrent protocols in Chapter 8, we will define a predicate *good* instead of an inductive invariant *inv* to do this verification. Subsequently, instead of defining an inductive invariant strengthening *good*, we will demonstrate the invariance of *good* using predicate abstraction.

We now turn to the definitions of *rep*, *skip*, *rank*, and *good* below:

rep: The function *rep* maps a $\mathbf{pipe}^+$ state p to form a **seq** state as follows. The program counter of *rep*(p) is the value of the program counter corresponding to the instruction in the **W** latch of p (as specified by the corresponding history variable), and the memory is the same as the value of `hmem` in p. The register file is simply projected from p.

skip: We want *skip* to hold for a $\mathbf{pipe}^+$ state p if p contains a bubble at the **W** latch that has been introduced earlier in the pipeline by the control logic. We determine this condition as follows. The **W** latch contains a bubble if and only if (a) the `icode` field of the latch specifies a `nop`, and (b) the instruction pointed to by the corresponding

$$dist(p) \triangleq \begin{cases} 4 & \textit{if } latch(p) = \text{"}\mathbf{F}\text{"} \\ 3 & \textit{if } latch(p) = \text{"}\mathbf{D}\text{"} \\ 2 & \textit{if } latch(p) = \text{"}\mathbf{E}\text{"} \\ 1 & \textit{if } latch(p) = \text{"}\mathbf{M}\text{"} \\ 0 & \text{otherwise} \end{cases}$$

$$rank(p) \triangleq 4 - dist(p)$$

Figure 14.3: Definition of *rank* for showing ($\mathbf{seq} \trianglerighteq \mathbf{pipe}^+$)

`PC` is *not* a `nop`. Notice that we are making legitimate use of the fact that we keep track of the `PC` of each instruction in the pipeline.

rank: We need to define *rank* so that it decreases whenever **pipe** makes a transition from a state at which *skip* holds, that is, when we encounter a **W** latch which contains a bubble. But a bubble is inserted in the pipeline in order to (a) resolve a memory dependency, or (b) take care of a mispredicted branch, or (c) process a `ret` instruction. In each case, the "correct" instruction to be executed after the bubble is fetched *before* the bubble reaches the **W** latch. Thus we define *rank* by counting the number of transitions for the correct instruction to move to the **W** latch. The definition is shown in Figure 14.3. Here *latch*(p) is the latch in which the correct instruction resides at state p. Obviously *rank* returns a natural number and hence an ordinal.

good: The predicate *good* posits two things. If *skip* holds for p, then *good* specifies that *latch*(p) holds the correct instruction. Otherwise *good* specifies that (a) the `icode` stored in the **W** latch is obtained by decoding the instruction pointed to by the associated `PC`, (b) the fields `valE` and `valM` contain the result of the execution of the instruction, and (c) the fields `dstE` and `dstM` contain the correct destination address.

With these definitions, it is rather easy to see that one can prove ($\mathbf{seq} \trianglerighteq \mathbf{pipe}^+$) *up to the invariance of good*. Thus we have reduced the proof obligations for the correctness of $\mathbf{pipe}^+$ to an invariant proof.

The reader reading our description of the verification of the Y86 so far might be a little puzzled. We started this part saying that verification of a system modeled at the RTL level is so much more complicated than one modeled at the so-called protocol or algorithmic level. Yet, so far in our description of the verification, we have not talked about any of the complexities induced by the RTL level implementation of the Y86, nor taken any special measure to deal with such complexities. Of course, even at the protocol level there were subtleties in the design, but we have seen such glimpses of such subtleties in the **bakery** or the **cdeq** system as well. Where does the extra complexity of the RTL level design manifest itself?

The complexity manifests itself in the proof of invariance of *good*. In a certain sense, that *is* the crux of the verification. So far in the proof, we have almost bypassed reasoning about the transitions of $\mathbf{pipe}^{+}$ (except for the matter of carefully defining *rank* so that it decreases when $\mathbf{pipe}^{+}$ transits from a *skip* state), by defining *good* appropriately. All the reasoning necessary above required understanding the $\mathbf{pipe}^{+}$ (and therefore, **pipe**) system at a high level. But in order to prove the invariance of *good*, one must show that $\mathbf{pipe}^{+}$ does execute each instruction correctly, handles mispredicted branches in the right manner, and injects bubbles and forwards data along the pipeline at the right times, all this being done via manipulation of bit vectors.

We invoke our predicate abstraction tool to show the invariance of *good*. Of course, to do this, we must have a collection of rewrite rules to reason about the different functions involved in the model of the system and its properties. This is where our library of RTL operations becomes critical. As we mentioned in the last chapter, the library contains about 150 theorems for reasoning about functions

$$\mathit{bvp}(x) \Rightarrow \mathit{bv\text{-}neg}(\mathit{bv\text{-}neg}(x)) = x$$

$$\mathit{width}(\mathit{bv\text{-}neg}(x)) = \mathit{width}(x)$$

$$\mathit{bv\text{-}neg}(\mathit{bv\text{-}or}(x, y)) = \mathit{bv\text{-}and}(\mathit{bv\text{-}neg}(x), \mathit{bv\text{-}neg}(y))$$

$$\mathit{equal}(\mathit{u\text{-}or}(x), \mathit{bit0}()) = \mathit{nlv}(x)$$

$$\mathit{bvp}(x) \Rightarrow \mathit{equal}(\mathit{u\text{-}and}(x), \mathit{bit1}()) = \mathit{unv}(x)$$

$$\mathit{bitn}(i, \mathit{bits}(m, n, x)) = \begin{cases} \mathit{bitn}(m+i, x) & \textit{if } \mathit{natp}(m) \wedge \mathit{natp}(n) \wedge (m \leq n) \wedge (i \leq n - m) \\ \mathit{bit0}() & \text{otherwise} \end{cases}$$

Figure 14.4: Some Theorems about Bit Vector Manipulation

manipulating the different RTL data types. The Figure 14.4 shows some of the theorems about bit vectors from the library which are used as rewrite rules by our tool.

Using our library, our predicate abstraction tool can prove that *good* (actually $\widehat{\mathit{good}}$, which is simply *good* restated as a function of time as required by our tool), is an invariant. The abstract system produced involves 77 exploration and 89 abstraction predicates, and the reachability analysis explores about 8000 nodes and 350000 edges, completing in a little more than 5 minutes on an 1.8GHz Pentium desktop machine running GNU/Linux. The exploration predicates specifying the abstract system mostly involved properties about the pipeline control structure, for example insertion of bubbles in the pipeline in case of a memory operation or a mispredicted branch.

The entire verification starting from the proof of the first theorem relating the *label*s of the two systems to the final invariant proof took the author slightly more than a month. Some of the time was spent in understanding the need for defining $\mathbf{pipe}^{+}$. Most of the verification time involved debugging the definition of *good*. This

resulted from the fact that the author initially made a mistake in its definition such that *good* was not an invariant but allowed us to prove the obligations for refinement. The definition was debugged by using some of the facilities we implemented in the tool for returning (abstract) counterexamples and bounded search. Given the size and complexity of the Y86 design we consider the effort reasonable.

Given that predicate abstractions and rewriting could be used to automate the proof of invariance of *good*, the skeptical reader might ask how difficult it would have been to come up with an inductive invariant strengthening *good* manually. Of course, since doing so requires manual expertise which is a subjective attribute, we cannot provide a precise answer to that question. Nevertheless, the author made an attempt to find a partial answer. We constructed a predicate *good-aux* strengthening *good* so that if *good-aux* holds for a state p then *good* holds for the state p and *skip* does not hold for p, then *good* holds when $\mathbf{pipe}^{+}$ makes a transition from p. Note that of course *good-aux* is *not* an inductive invariant, but only a first step at the attempt to construct one. In particular, we wanted to incorporate all "facts" that we need to know about the **M** latch at state p in order to prove that *good* holds when $\mathbf{pipe}^{+}$ makes a transition from p. (Recall that the predicate *good* talks only about the **W** latch if *skip* does not hold at p.) Figure 14.2 suggests that conceptually the facts about the **M** latch should be simple. After all, no forwarding logic is involved and we need to only say that `valE` stores the right value (in case the `icode` of the instruction in the **M** latch specifies an ALU operation), `valA` has the right memory address (for a memory operation), and `dstE` and `dstM` are the right destination addresses. However, getting this right still turned out to be a tedious and complex matter. Part of the reason is that we had to explore and

understand the control logic specifying how the memory is updated. For instance, although in Figure 14.2 we showed the (data) memory as a single block, it is actually implemented as a collection of eight *memory banks*, each of which is independently `enable`d. The data from the banks emerge in parallel and are then "concatenated" into the byte. The predicate for the **M** latch has to specify how the `enable` signal of each memory bank behaves corresponding to the different instructions in the **M** latch. It is probably fair to say that definition of *good-aux* itself took about as much of the author's time as the definition of all the four other functions taken together. Further, short of manually constructing the entire inductive invariant we have no way of determining if we have incorporated all facts that we need about the **M** latch in the definition, or if we got some of the "facts" wrong. We presume that with the forwarding logic and handling of mispredicted branches that are involved with the other latches, coming up with the necessary facts about them manually would be substantially more complex.

14.5 Summary and Comments

We have proved correspondence between our formal models of the Y86 using predicate abstraction and theorem proving. How large are the models? The formal model of **seq** defined in ACL2 contains 185 function definitions, while **pipe** contains 290. More importantly, the systems contain complex control logics which make reasoning about them challenging. However, as we saw, the complexity of the RTL design is manifest in the definition and proof of invariants. With our methodology, this complexity is managed by predicate abstraction, insulating them from the user. The user, then, can focus principally on the conceptual insights behind the system (as

manifest in the definition of *good* and *rank*) which is the level at which theorem proving is appropriate. Of course it must be admitted that the models are simplistic compared to a commercial processor design. However, the fact that we can automate the invariant proof for such a system does provide substantial evidence in the scalability of our method.

We should note that the reason we could dispatch the invariant proof involved in the verification of the Y86 automatically was that we had carefully crafted our RTL library so that terms representing applications of RTL operators could be effectively normalized. The success of our tool crucially depends on the existence of such libraries. So it makes sense to ask how robust the library is for reasoning about RTL systems. While it is not possible to provide a concrete answer to this question, we believe that with the library in its current form, we can expect reasonable automation in invariant proofs of RTL systems where the complexity is in the *control structure* of the system. As circumstantial evidence on this matter, we point out that most of the rules in our library had been built *before* we had access to the designs of the Y86 processors. However, the library contains few rules about bit vector arithmetic and would be inadequate if used, for example, for the verification of a floating point unit. Indeed, one of the reasons for the success of the library in Y86 verification is that almost no reasoning about arithmetic was necessary. Although the ALU in the two processors do fixed point integer arithmetic, both processors use the same ALU module obviating the need to reason about such ALU operations.

The last sentence above also points out a key matter about formal verification, especially of microarchitectures. All that we have formally proven is that the ACL2 model of **pipe** is a refinement of the ACL2 model of **seq**. It does not

obviously mean that **pipe** is correct. For instance, it is possible that the ALU modules of both systems (which are actually instantiations of the same module) are in fact inaccurate. Given a formal proof it is in fact imperative that the user inspect the specification (in this case the **seq** model) and convince himself that it indeed corresponds to the user's view of the behavior of the system.

We believe our work is the first instance of the use of predicate abstraction and discovery for reasoning about a pipelined system. Indeed, **pipe**$^{+}$ represents one of the largest systems on which predicate abstraction has been applied.

The use of predicate abstraction for invariant discovery is reminiscent of the method of invariant strengthening suggested by Sawada and Hunt [SH99a] in course of verification of a complex pipeline. They specify an initial set of predicates that are necessary to dispatch the other proof obligations for correctness of the system. Then they iteratively strengthen the set of predicates by composing them over the state transition function of the system until they obtain an inductive invariant. Of course the strengthening in their case is performed manually via deductive reasoning. We achieve the same effect for the Y86 using rewriting.

In our work, we have depended on predicate abstraction and deductive reasoning for verifying properties of RTL designs. However, there are have been significant advances in the application of automatic decision procedures for RTL verification, which work with the original VHDL or Verilog designs. It would be interesting to see how they can be integrated with a deductive approach. We will look at decision procedures and the complexities involved in their integration with ACL2 in the next part. Nevertheless, there have been some advances recently in applying such tools with theorem proving. For example, Sawada [Saw04] presents a way of using

VHDL tools for transformation based verification [Bau02] together with ACL2 for reasoning about RTL. The approach is to translate ACL2 functions to a subset of VHDL (instead of the other way round) so that VHDL tools can check equivalence between the formal definitions in ACL2 and their VHDL implementations. This approach provides an interesting dual to our method, and it would be interesting to see if it can be integrated with our approach to further automate the verification process.

14.6 Bibliographic Notes

The Y86 processor has been designed by Bryant and O'Hallaron [BO03] and represents a fairly standard processor design following the IA32 architecture. Processor designs and architecture are dealt with extensively by Hennessey and Patterson [HP02]. Shriver and Smith [SS98] provide a thorough description of the IA32 architecture.

Verification of pipelined microarchitectures is an area of extensive research and the bibliographic notes for Chapter 9 lists many approaches aimed at automating the process. In addition, the bibliographic notes for Chapter 10 lists related approaches to predicate abstraction.

Reasoning about RTL level designs has captured a lot of attention lately, although most of the related work have focused on verification of floating point units. Greer *et al* have verified many RTL units of the Intel Itanium® processor [GHH+02], Sawada and Gamboa for the IBM Power 4 [SG02], and Russinoff and Flatau for the AMD Athlon™ [Rus98, RF00].

Part VI

Formal Integration of Decision Procedures

Chapter 15

Integrating Deductive and Algorithmic Reasoning

We have seen how theorem proving techniques could be combined with reachability analysis to reduce the manual effort involved in invariant proofs. In this part, we will explore the general problem of using theorem proving with decision procedures in a sound and efficient manner.

Theorem proving and decision procedures have orthogonal advantages in scaling up formal verification to solve complex verification problems. Theorem proving affords the use of sophisticated proof techniques to reduce the verification problem to simple manageable pieces. The key deductive methods that the user can employ to produce such decomposition include defining auxiliary functions, introducing insightful generalizations of the problem, and proving key intermediate lemmas. On the other hand, decision procedures contribute largely in automating the proofs when the formula can be expressed in a decidable theory. If we can combine theorem proving with decision procedures, then we can apply the following "strategy"

to effectively exploit the combination:

- Apply theorem proving to decompose the verification problem into proofs of simpler formulas that can be expressed in the decidable theory in which the procedure operates.
- Appeal to the procedure to check if each of these simpler formulas is a theorem.

Indeed, this is what we did in combining predicate abstraction with model checking, for a restricted class of problems, namely invariant proofs. Our method of constructing predicate abstractions was an essentially deductive process where the user controlled the abstractions generated by carefully crafting lemmas which could be used as rewrite rules. But once an abstraction was generated, it reduced the invariant proof to a finite problem which could then be "shipped off" to a model checker.

The appeal of this general strategy above is undeniable. It requires user intervention in a demand-driven fashion only for problems that cannot be solved by decision procedures. On the other hand, state explosion is kept on a "tight leash" by applying decision procedures on sufficiently small pieces. If any of the pieces cannot be be handled by the decision procedure in a reasonable time, in spite of falling in a decidable theory, then theorem proving can step in and reduce it further until they can be handled automatically. In this view, theorem proving and decision procedures can be seen as two opposing ends of a spectrum where user effort is traded for state explosion as one moves from one end to the other.

In this chapter, we will study generic methods for integrating theorem proving with decision procedures. We will consider the following question:

- How can we guarantee that the integration is sound (assuming the soundness of the

theorem prover and the decision procedure concerned) and practically efficient?

Let us consider the soundness issue first. A casual reader might fail to notice at first that there is an issue here at all. Roughly, the issue arises from the incompatibility between the logic of the theorem prover and the theory in which the decision procedure operates. To understand this, let us consider a decision procedure $\mathcal{D}$ that takes three positive rational numbers m, n, and ϵ, and checks if $|\sqrt{m} - n| \leq \epsilon$. Thus it (rightly) passes the check on the triple of number $\langle 2, 1, 1 \rangle$. Suppose now we are in a theory $\mathcal{T}$ and encounter the affirmative answer given by $\mathcal{D}$ on $\langle 2, 1, 1 \rangle$. How do we interpret this answer? We might be tempted to extend $\mathcal{T}$ by introducing a new unary function *sqrt*, so that *sqrt*(x) can be interpreted as the positive square root of x (that is, a formalization of $\sqrt{x}$), and then infer, based on the answer provided by $\mathcal{D}$, that $|\mathit{sqrt}(2) - 1| \leq 1$. Unfortunately, if $\mathcal{T}$ is a legal extension of GZ, then the function *sqrt* cannot be introduced such that the formula $\mathit{sqrt}(2) \times \mathit{sqrt}(2) = 2$ is a theorem. Indeed, the axioms of GZ rule out the existence of irrational numbers and so it is possible to prove that $\neg((x \times x) = 2)$ is a theorem [Gam96, Gam99]. Thus any extension of GZ with a reasonable axiomatization of *sqrt* that lets us interpret the affirmative answer of $\mathcal{D}$ above is inconsistent.

How do we resolve this dilemma? One way might be to simply invoke $\mathcal{D}$ carefully on arguments for which we know how to interpret the answer. For instance, even though we cannot interpret the answer returned from $\mathcal{D}$ on $\langle 2, 1, 1 \rangle$ as per the above discussion, we can attempt to interpret the answer produced for $\langle 1, 1, 0 \rangle$. Thus we can decide that we will invoke the procedure only with the third argument ϵ set to 0, and interpret an affirmative answer to mean $m = n^2$. In some sense, this is what we did when invoking a model checker like SMV or VIS for the purpose

of proving invariants. Model checkers are decision procedures that can be used to check temporal logic properties of finite-state reactive systems. We do not know yet whether we can interpret an affirmative answer provided by the model checker on arbitrary temporal formulas. Indeed, we will see in Chapter 16 that interpreting their answer for arbitrary temporal properties involves complications. But we simply invoked them only for invariant checking, when the result could be interpreted as a successful reachability analysis of a finite graph.

But our current interest is not the integration of a specific decision procedure with theorem proving, but a general formal approach for integration of arbitrary decision procedures. One possibility is the following. We can define the decision procedure itself as a conservative formal theory. After all, a decision procedure is merely a program, and it is possible to code it up in ACL2. But once we define it as a formal theory then we can prove theorems about it. For instance, we can attempt to define a function, say *approx-sqrt* to formalize the decision procedure $\mathcal{D}$ above, and prove the following formula as a theorem about it.

$$(\epsilon = 0) \wedge \textit{approx-sqrt}(m, n, \epsilon) \Rightarrow m = n \times n$$

This theorem, then, can be treated as a *characterization* of the decision procedure in the formal theory. In particular, for two numbers m and n, whenever we wish to deduce whether m is equal to n^2, we can attempt to resolve the question by evaluating the function *approx-sqrt* on m, n, and 0. Notice that a more general theorem characterizing the return value of *approx-sqrt* for arbitrary ϵ based on the informal description above is significantly more complicated.

This, then, is our proposal to integrate decision procedures with theorem proving. We will define a formal theory that specifies the semantics of the decision procedure. Then we will prove theorems that stipulate how we can interpret the

answers produced when the decision procedure is applied to verification problems. We will then use the decision procedure to prove theorems in the decidable fragment of the logic and use the characterization theorems to interpret the answers produced by the procedure.

The approach seems simple as a matter of course. But the problem of coming up with characterization theorems, let alone proving them, is non-trivial. After all, the procedures we are interested in are much more complex than the procedure *approx-sqrt* above. For instance, model checking, one of the procedures that is of interest, requires us to reason about temporal properties of infinite sequences. How feasible is it to apply our approach in practice?

To test such feasibility, we integrate a simple compositional model checking algorithm with ACL2. This algorithm, the issues involved in formally modeling its semantics, and the characterization theorems we prove about it, will be presented in Chapter 16. Our experience indicates that it is possible to do it, although some aspects in defining a formal semantics of model checking are non-trivial. The problems principally stem from certain limitations in the expressiveness of the logic of ACL2, that does not allow us to model the semantics of model checking in the standard way; such limitations make it difficult to formalize and prove some of the standard results about temporal logic. A consequence of our attempt is to expose these limitations and advocate introduction of new axioms to facilitate similar efforts in future. Nevertheless, note that these characterizing theorems are to be proved once and for all for a decision procedure being integrated, and the investment of the manual effort in this exercise is well worth the price if the integration results in substantial automation of proofs.

A more practical objection to the above approach is efficiency. After all, the design of efficient decision procedures in practice is an area of extensive research and modern implementations of model checkers succeed in coping with some of the complexities of modern systems principally because of highly optimized implementations. On the other hand, these implementations are not done with formalization in mind, nor are they written in ACL2. They are often implemented as low-level C programs. While it is surely possible to code them up in ACL2, and even obtain a certain amount of efficiency in execution, it is unlikely that the functions representing a decision procedure as a formal theory will be as efficient as a standard commercial implementation of the same procedure. Also, commercial implementations of decision procedures are being improved continually to cope with the efficiency demands. Thus if we decide to reimplement a procedure in ACL2, then we must incessantly track and implement future improvements to the procedure made by the decision procedure community. This is indeed a viable practical objection, and we discuss the ramifications of this objection in Chapter 17. Notice that our interest in developing the formal semantics of a decision procedure is not because we suspect that its implementation is buggy (although that is indeed possible), but to make sure that our *interpretation* of its affirmative answer inside a formal theory is sound. We discuss how it should be possible to use *external oracles* with a theorem prover, and what kind of soundness guarantees can and should be given when external procedures are integrated with ACL2. This results in some recommendations for improving the implementation of the ACL2 theorem prover to afford the practical use of external oracles.

Chapter 16

A Compositional Model Checking Procedure

Model checking is one of the most widely used verification methods used in the industry today. Model checking at its core is a decision procedure for proving temporal properties of finite-state reactive systems, as we saw in Chapter 2. However, as we mentioned several before, the method is limited in practice by state explosion. Our methods in Parts III and IV were to find theorem proving techniques to find manageable decomposition of the verification problems. As our contributions throughout this dissertation indicate, theorem proving is a general method to achieve this task. Nevertheless, there are certain decision procedures that can achieve substantial decomposition of model checking problems. They include reductions of system state by exploiting symmetry, elimination of redundant variables via cone of influence, decomposition of the proof of a conjunction of temporal formulas to independent proofs of the constituent formulas, assume-guarantee reasoning, and

so on. We will refer to such decision procedures as *compositional model checking procedures*. While less flexible than theorem proving, compositional model checking, if applied appropriately, can often achieve significant simplification in the verification of reactive systems, with the added benefit of substantial automation. It is therefore of advantage to us if we can use them wherever applicable, along with theorem proving.

In this chapter, we will explore how we can integrate compositional model checking with theorem proving. We will consider an extremely simple compositional procedure to study the issues involved. The algorithm is a composition of *conjunctive reduction* and *cone of influence reduction*. Conjunctive reduction is based on the idea that if a temporal formula ψ is a conjunction of several formulas $\psi_1, \ldots, \psi_n$, then checking whether a system M satisfies ψ can be reduced to checking whether M satisfies ψ_i for each i. Cone of influence reduction is based on the idea that if the formula ψ refers to only a subset V of the state variables of M, then it is possible to deduce that M satisfies ψ by checking whether a different (and potentially smaller) system M' satisfies ψ, where the M' is formed by removing from M all the state components that have no effect on the variables in V. The system M' is referred to as the *reduced model* of M with respect to V.

What is our compositional procedure? Given the problem of checking if a system M satisfies ψ, where ψ is a conjunction of several formulas $\psi_1, \ldots, \psi_n$, it first applies conjunctive reduction, reducing the problem to checking if M satisfies ψ_i for each i. It then applies cone of influence reduction for each of these verification problems. That is, for the i-th problem, it reduces the check of whether M satisfies ψ_i to the check of whether M_i satisfies ψ_i, where M_i is the reduced model of M with

respect to the variables in ψ_i.

The procedure as described above is simple, but its integration is illustrative. Recall that to integrate the procedure we must define it as a formal theory in the logic of the theorem prover and prove theorems characterizing the output of the procedure. The remainder of this chapter shows how this can be done, and explores some of the complications involved.

16.1 Formalizing a Compositional Procedure

The compositional procedure described above takes a verification problem and returns a collection of verification problems. A verification problem is a description of a finite state reactive system together with a formula written in some temporal logic. In order to formalize the procedure we must first clarify how a finite state system and temporal formula are presented to the algorithm as input.

16.1.1 Finite State Systems

We have been talking about models of reactive systems for a while in this dissertation. We modeled a system $\mathcal{I}$ as three functions, namely $\mathcal{I}.\mathit{init}()$, $\mathcal{I}.\mathit{next}()$, and $\mathcal{I}.\mathit{label}$. In this chapter, however, we are interested in *algorithms* for reasoning about finite-state systems. Since we intend to model the algorithms as functions in ACL2, we must represent the finite state systems as *objects* that can be manipulated by such functions.

The reader has already seen the basic ingredients for specifying finite-state systems as objects in Part IV. There we talked about an abstract system $\mathcal{A}$ which represented a predicate abstraction of the implementation. The point of interest

here is that $\mathcal{A}$ has a finite number of states. As we saw, we can represent such a system by 3 components.

- A collection of state variables.
- A collection of input variables.
- For each state variable a term specifying how the variable is updated along the transitions of the system. We will call this term the *transition equation* for the variable.
- An initial state.

For our purpose here, we will assume without loss of generality that a variable can take the value `T` or `NIL`. Thus the set of states of the system is the set of all possible Boolean assignments for each state variable. Again for simplicity, we will assume that the transition equation for a variable is a term composed of the (state and input) variables and the Boolean connectives $\wedge$, $\neg$, and $\vee$. Given such a term, we can easily write a function in ACL2 that *interprets* it to determine, for any state and any assignment of the input variables to Booleans, what the next state of the system is.

Since we are interested in model checking, we also need a set AP of atomic propositions. For simplicity, let us assume that AP is simply the set of state variables, and the label of a state s is the list of all variables that are assigned to `T` in s. Most of what we discuss below does not rely on such simplifying assumptions, but they allow us to talk concretely about finite state systems. More precisely, we define a binary predicate *system* so that *system*(M, AP) holds if and only if the following happen:

- AP is the set of state variables in M.
- The transition equation associated with each state variable refers only to the state and input variables and connectives $\wedge$, $\neg$, and $\vee$.

- Every state variable is assigned to either `T` or `NIL` in the initial state.

16.1.2 Temporal Logic formulas

In addition to the description of a finite state system, a compositional algorithm must take a formula written in some temporal logic. Let us fix the temporal logic formalism that we want to use for our algorithms. Our choice for this study is LTL, principally because it is simple and it has been recently growing in popularity for specification of industrial systems. We discussed the syntax of LTL informally in page 20. Recall that an LTL formula is either an atomic proposition or one of $\neg\psi_1$, $\psi_1 \vee \psi_2$, $\psi_1 \wedge \psi_2$, $\mathsf{X}\psi_1$, $\mathsf{G}\psi_1$, $\mathsf{F}\psi_1$, and $\psi_1 \mathsf{U} \psi_2$, where ψ_1 and ψ_2 are LTL formulas. Our formal representation of an LTL formula follows this characterization rather literally as lists. For instance if x is a representation for the formula $\psi_1 \mathsf{U} \psi_2$ then *first*(x) is the representation of ψ_1, *second*(x) returns the symbol `U`, and *third*(x) is the representation of ψ_2. Similarly, if y is the representation of $\mathsf{G}\psi$, then *first*(y) returns the symbol `G`, and *second*(y) is the representation of ψ. Given the characterization of the syntax of LTL, it is easy to define a predicate *formula* such that *formula*(ψ, AP) returns `T` if and only if ψ is an LTL formula corresponding to AP.

16.1.3 Compositional Procedure

We are now ready to formally describe the compositional algorithm we will reason about. The conjunctive reduction algorithm is trivial. It takes a formula ψ and returns a list of formulas $\psi_1, \ldots, \psi_k$ such that $\psi \doteq \psi_1 \wedge \ldots \wedge \psi_k$. The algorithm is formalized by the following recursive function *R-and*:

$$\textit{R-and}(\psi, AP) \triangleq \begin{cases} \textit{list}(\psi) & \textit{if } \neg\textit{andformula}(\psi, AP) \\ \textit{append}(\textit{R-and}(\textit{left}(\psi), AP), \textit{R-and}(\textit{right}(\psi), AP)) & \text{otherwise} \end{cases}$$

Formalizing cone of influence reduction is more involved. Given a finite state system and a formula ψ that refers to a subset V of the state variables of M, cone of influence reduction constitutes first creating a set C of state variables with the following properties:

1. $V \subseteq C$
2. For any $v \in C$, if the transition equation for v in M refers to some state variable v' then v' is in C.
3. C is the minimal set of state variables satisfying 1 and 2 above.

The set C is then called the *cone of influence* of the system M with respect to the formula ψ. We then construct a reduced model M_ψ of M as follows:

- The set of state variables of M' is the set C.
- For each $v \in C$, the transition equation for v in M_ψ is the same as the transition equation of v in M.
- The initial state of M_ψ is obtained by extracting from the initial state of M the valuation of only the variables in C.

We formalize the cone of influence reduction by defining a collection of functions to perform the steps above. We omit the formal definition of the functions here for brevity, but the readers should be able to convince themselves that it can be done given the description above. For the purpose of our discussion, we will assume that we have a binary function *R-cone* such that *R-cone*(ψ, M) returns the reduced model of M with respect to the variables in ψ.

Our compositional algorithm is now defined easily using functions *R-and* and *R-cone*. The algorithm is formalized by the function *Reduce* in Figure 16.1. Given an LTL formula ψ and a finite state system M, it performs the following steps:

$$
\textit{R-1}(fs, M, \mathcal{AP}) \triangleq \begin{cases} \texttt{NIL} & \textit{if } \neg \textit{consp}(fs) \\ \textit{cons}(\textit{problem}(\textit{first}(fs), \textit{R-cone}(M, \textit{first}(fs)), \mathcal{AP}), & \\ \quad \textit{R-1}(\textit{rest}(fs), M, \mathcal{AP})) & \text{otherwise} \end{cases}
$$

$$
\textit{Reduce}(\psi, M, \mathcal{AP}) \triangleq \textit{R-1}(\textit{R-and}(\psi), M, \mathcal{AP})
$$

Figure 16.1: Formalization of the Compositional Model Checking Procedure

1. Apply *R-and* to ψ to create a list *fs* of formulas $\langle \psi_1 \ldots \psi_k \rangle$.
2. For each $\psi_i \in fs$, apply *R-cone* to create a reduced model M_{ψ_i} of M with respect to the variables of ψ_i.
3. Return the collection of verification problems $\langle \psi_i, M_{\psi_i}, AP \rangle$.

Now that we have formalized the compositional procedure, we can ask what its characterization theorem will be. Informally, we want to state the following: "The system M satisfies ψ if and only if for every verification problem $\langle \psi_i, M_{\psi_i}, AP \rangle$ returned by the procedure, M_{ψ_i} satisfies ψ_i with respect to AP." What do we mean by a system M satisfying the formula ψ? The semantics of LTL is specified with respect to infinite paths through a Kripke Structure. It is easy to define a function that can take a finite state system M and returns the Kripke Structure for M. Let us call that function *kripke*. Assume for the moment that we can define a binary predicate *ltlsem* such that $\textit{ltlsem}(\psi, \kappa, AP)$ can be interpreted to return `T` if and only if κ is a Kripke Structure for which ψ holds. We can then define what it means for a finite state system M to satisfy ψ by the binary predicate *satisfies* below:

$$
\textit{satisfies}(\psi, M, AP) \triangleq \textit{ltlsem}(\psi, \textit{kripke}(M), AP)
$$

We can now easily define what it means for a collection of verification problems to be satisfied. We say that a verification problem $prb \doteq \langle \psi, M, AP \rangle$ *passes* if and only

if M *satisfies* ψ. This notion is formalized by the predicate *passes*:

$$\mathit{passes}(prb) \triangleq \mathit{satisfies}(\mathit{formula}(prb), \mathit{sys}(prb), \mathit{ap}(prb))$$

Finally, a collection of verification problems will be said to *pass* if and only if each constituent problem *passes*.

$$\mathit{pass}(prbs) \triangleq \begin{cases} \mathtt{T} & \mathit{if}\ \neg \mathit{consp}(prbs) \\ \mathit{passes}(\mathit{first}(prbs)) \wedge \mathit{pass}(\mathit{rest}(prbs)) & \text{otherwise} \end{cases}$$

With these definitions, we can finally express the correctness of our compositional procedure. The formal statement of correctness is shown below.

Main: $\mathit{system}(M, AP) \wedge \mathit{formula}(\psi, AP) \Rightarrow \mathit{satisfies}(\psi, M, AP) = \mathit{pass}(\mathit{Reduce}(\psi, M, AP))$

16.2 Modeling LTL Semantics

Given the discussion above, we seem to have our work cut out to achieve the goal of reasoning about our procedure. We should define the predicate *ltlsem* to capture the semantics of LTL and then prove the formula labeled **Main** above as a theorem. How do we define *ltlsem*? We have presented the standard semantics of LTL in page 21. The semantics is described in terms of execution paths of Kripke Structures. Thus a naive approach will be to define a binary predicate *pathsem* so that given a path π and an LTL formula ψ $\mathit{pathsem}(\psi, \pi, AP)$ returns `T` if and only if π satisfies ψ with respect to AP based on the recursive characterization we discussed. We should also define a binary predicate *pathp* so that given a Kripke Structure κ and a path π, $\mathit{pathp}(\pi, \kappa)$ returns `T` if π is a path through κ and `nil` otherwise. We can then define *ltlsem* as:

$$\mathit{ltlsem}(\kappa, \psi, AP) \triangleq (\forall \pi : \mathit{pathp}(\pi, \kappa) \Rightarrow \mathit{pathsem}(\psi, \pi, AP))$$

But here we run into an unexpected road-block. How do we define the function *pathsem* above? Presumably the argument π of *pathsem* must be an infinite path,

that is, an infinite sequence of states. How do we model a sequence as a formal object? The standard thing to do is to use lists. Unfortunately, the axioms of the ACL2 ground zero theory GZ rule out the existence of infinite lists. For instance, it is easy to prove the following theorem in GZ:

Theorem 1 $\exists y : \mathit{len}(y) > \mathit{len}(x)$

The theorem says that for any list x, there is some other list y which has a larger length. So there is no infinite list. Indeed, we have found no way of representing infinite paths as formal objects which can be manipulated by functions introduced in ACL2 in any ACL2 theory that is obtained by extending GZ via the extension principles.

Notice that there is no paradox between our current assertion that infinite sequences cannot be represented as objects in ACL2 and our work in the last two parts where we did talk and reason about infinite sequences at a number of places. Whenever we talked about infinite sequences before we modeled them as *functions*. For instance, consider the function *stimulus* that we talked about before. This function could be *interpreted* as an infinite sequence of input stimuli, so that we could talk about $\mathit{stimulus}(n)$ as the state in this sequence at position n. But a function is not an object that can be passed as an argument to another function or otherwise manipulated. As we remarked before, we cannot define a function f that takes *stimulus* as an argument and returns (say) $\mathit{stimulus}(n)$. Indeed, that is the reason why many of our formalizations of the theory of stuttering trace containment and the corresponding proof rules necessitated encapsulation and functional instantiation. But a look at what *pathsem* must "do" indicates that this is exactly what we need if we want to use functions to model infinite sequences! In this case how-

ever, encapsulation cannot be used to do the "trick". Why? Consider the definition schema of $M.\mathit{exec}[\mathit{stimulus}]$ that we talked about in Chapter 7. Although we used the definition as a definition schema, we always reminded ourselves that whenever we use different *stimulus* we actually get different definitions and different functions. For the same reason, if we try to define *pathsem* as a schema so that it uses some infinite sequence of states as an encapsulated function, then whenever we talk about two different paths, we will end up having two different "definitions". But we want to make a statement of the form "For *every* path through the Kripke Structure the formula holds", as a characterizing theorem for a model checking algorithm. This statement then cannot be expressed. Thus we seem to have hit the limits of expressiveness of the logic.

Of course we should note that even if GZ had a simple axiomatization of infinite sequences, it might not have been possible to define *pathsem* directly. To see why, assume for the moment that we have axioms for infinite objects and $\mathit{suffix}(i, \pi)$ returns π_i. Consider the recursive characterization of $\mathsf{F}\psi$ we discussed in page 21 (point 7), which we reproduce below:

7. π satisfies $\mathsf{F}\psi$ if and only if there exists some i, π_i satisfies ψ.

To formalize this, one must be able to use both recursion and quantification. That is, the definition of *pathsem* must be of the following form:

$$\mathit{pathsem}(\psi, \pi, AP) \triangleq \begin{cases} \ldots \\ \exists i : \mathit{pathsem}(\mathit{value}(\psi), \mathit{suffix}(i, \pi), AP) & \mathit{if}\ \mathit{unary}(\psi) \wedge (\mathit{op}(\psi) = \mathsf{F}) \\ \ldots \end{cases}$$

Unfortunately, ACL2 does not allow introduction of recursive functions with quantifiers in the recursive call (for an important logical reason as we will see in Section 16.4). Thus even with axiomatization of infinite sequences it is not obvious

that we can define the semantics of LTL.

What should we do? There are some possible options. One way might be to define the *ltlsem* directly. Why is this feasible even though the definition of *pathsem* is not? Unlike *pathsem*, the predicate *ltlsem* takes as arguments a Kripke Structure κ and an LTL formula ψ, both of which are finitely represented as lists. Nevertheless, we find this approach objectionable for a variety of reasons. First, notice that defining *ltlsem* directly is tantamount to defining a model checking algorithm for LTL. It should be understandable from our informal description of the model checking algorithm in Chapter 2 that such a definition is too complex to be termed "semantics of LTL". Second, our goal for defining the semantics of LTL is to prove theorems about the compositional procedure. Since this is a theorem proving exercise, it is important to our success that our definition of *ltlsem* be as close as possible to the way a person thinks about the semantics of LTL when reasoning about LTL properties of a system. And when one thinks about an LTL property one does not think in terms of the implementation of a model checker but rather in terms of properties of execution paths. We will soon sample some standard proofs of reduction algorithms. If the derivation of the characterization theorems for our simple reduction is significantly more complex than the standard proofs then we will have little hope of scaling up our methodology to integrate more involved reductions like symmetry and assume-guarantee reasoning.

Our "solution" to this obstacle is to model the semantics of LTL in terms of *eventually periodic paths*.[1] An eventually periodic path is simply a path that comprises of a finite *prefix* followed by a finite *cycle* that is repeated forever. Since

[1] We thank Ernie Cohen for suggesting the use of eventually periodic paths to model the semantics of LTL.

both the prefix and the cycle are finite structures, they can be represented as lists of states. It is thus possible to define the predicate *ppathp* so that *ppathp*(π, κ) returns `T` if π is an eventually periodic path through the Kripke Structure κ starting from the initial state. Furthermore, it is possible to define the predicate *ppathsem* using the recursive characterization given in page 21 so that *ppathsem*(ψ, π, AP) returns `T` exactly when π is an eventually periodic path satisfying the formula ψ. We then define the predicate *ltlsem* now by simply quantifying over all eventually periodic paths through the Kripke Structure κ as follows.

$$\textit{ltlsem}(\psi, \kappa, AP) \triangleq (\forall \pi : \textit{ppathp}(\pi, \kappa) \Rightarrow \textit{ppathsem}(\psi, \pi, AP))$$

Given the standard semantics of LTL why is it sufficient to check that all eventually periodic paths satisfy a formula ψ in order to assert that all (infinite) paths do? The result follows from the following proposition. The proposition is a corollary of standard properties of LTL. and is often stated (in an analogous but slightly different form) as the *finite model theorem* [CE81]. We provide an outline of its proof here, for the sake of completeness.

Proposition 4 *For any Kripke Structure $\kappa \doteq (S, R, L, s_0)$ such that the set of states S is finite and any LTL formula ψ, if there exists a path in κ through s_0 that does not satisfy ψ, then there also exists an eventually periodic path in κ through s_0 that does not satisfy ψ.*

Proof sketch: Recall from Chapter 2 that the problem of checking whether ψ satisfies κ can be reduced to checking if the language $\mathcal{L}$ accepted by the Büchi automaton $\mathcal{A}_{\kappa,\psi}$ is empty. Let us assume that κ does not satisfy ψ, and hence there must be an accepting (infinite) sequence ρ for $\mathcal{A}_{\kappa,\psi}$. Since the set of automata states is finite, there is some suffix ρ' of ρ such that every state on it appears infinitely many times.

Thus the states in ρ' are included in a strongly connected component. The component is reachable from the initial state of the automaton and contains an accepting state. Conversely, any strongly connected component that is reachable from the initial state and generates an accepting state generates an accepting sequence.

Thus checking nonemptiness is equivalent to finding a strongly connected component in the automaton. We can restate this equivalence in terms of eventually periodic path. That is, $\mathcal{L}$ is nonempty if and only if there is a reachable accepting state in the automaton with a cycle back to itself. The restatement is legitimate from the following argument. If there is a cycle, then the nodes in the cycle must belong to some strongly connected component. Conversely if there is a strongly connected component containing an accepting state then we can construct a cycle containing an accepting state. Thus if $\mathcal{L}$ is nonempty then there is a counterexample which can be represented as an eventually periodic paths. This eventually periodic path corresponds to a sequence of pairs one component of which is a state of the Kripke Structure κ. Taking this component of the path, we can obtain an eventually periodic path through κ through s_0 that does not satisfy ψ. ∎

We should note that the definition of the function *ppathsem* is not trivial. After all, we have still to cope with the fact that the function needs to be recursive (to follow the characterization of LTL) and therefore the use of quantification in its recursive calls is forbidden. We use a standard work-around for this limitation. Given an eventually periodic path π, let the function *psuffix*(i, π) serve the same purpose as our hypothetical *suffix* above, that is, return the i-th suffix of π. Then, instead of writing $\exists i : \mathit{ppathsem}(\mathit{value}(\psi), \mathit{psuffix}(i, \pi), AP)$ in the recursive call, we write $\mathit{ppathsem}(\mathit{value}(\psi), \mathit{psuffix}(\mathit{witness}(\psi, \pi, AP)), AP)$, where the function *witness*

is defined using mutual recursion with *ppathsem* to explicitly compute the i if it exists. Notice that this “trick” would not have been possible if π were some formal representation of an infinite path as we hypothesized above, since in that case the recursive computation of the index might not have terminated.

16.3 Verification

We now discuss how we prove that the formula labeled **Main** above is a theorem. Our proof plan is to first prove that the conjunctive reduction and the cone of influence reduction are individually correct. How do we specify the correctness of conjunctive reduction? Recall that conjunctive reduction takes a formula ψ and produces a list of formulas. To specify its correctness, we first define the function *probs* below so that given a finite state system M, the atomic propositions AP, and a list of formulas $fs \doteq \langle \psi_1, \ldots, \psi_k \rangle$, $\textit{problems}(fs, M)$ creates a list of verification problems $\langle \psi_i, M, AP \rangle$, one for each ψ_i.

$$\textit{probs}(fs, M, AP) \triangleq \begin{cases} \texttt{NIL} & \textit{if } \neg \textit{consp}(fs) \\ \textit{cons}(\textit{prob}(\textit{first}(fs), M, AP), \textit{probs}(\textit{rest}(fs), M, AP)) & \text{otherwise} \end{cases}$$

Given the definition above, the obligation **Red1** below stipulates the correctness of conjunctive reduction. It merely says that if ψ is an LTL formula and M is a finite state system, then M satisfies ψ if and only if for each ψ_i produced by *R-and*, M satisfies ψ_i.

R1: $\textit{system}(M, AP) \wedge \textit{formula}(\psi, AP) \Rightarrow \textit{satisfy}(\psi, M, AP) = \textit{pass}(\textit{probs}(\textit{R-and}(\psi), M, AP))$

The correctness of the cone of influence reduction can be specified in an analogous manner by the the obligation **R2**, which states that M satisfies ψ if and only if the reduced model of M with respect to the variables in ψ satisfies ψ.

R2: $\textit{system}(M, AP) \wedge \textit{formula}(\psi, AP) \Rightarrow \textit{satisfy}(\psi, M, AP) = \textit{satisfy}(\psi, \textit{R-cone}(\psi, M), AP)$

The proof of the **Main** theorem from **R1** and **R2** is straightforward. Observe that the function *Reduce* replaces the "system" component of each verification problem in *probs*(*R-and*(ψ), M, AP) with the reduced model of M with respect to variables in the corresponding LTL formula. To prove **Main** we note that by **R1** M satisfies ψ if each of the problems in *probs*(*R-and*(ψ), M, AP) passes, and by **R2**, a problem $\langle \psi, M, AP \rangle$ *passes* if and only if the reduced model M_ψ of M with respect to the variables in ψ satisfies ψ.

How do we prove **R1** and **R2**? The proof of **R1** is simple. The following proof sketch follows its mechanical derivation, with some commentaries added for clarity.

Proof Sketch of **R1***:* Recall from the definition of the semantics of LTL (page 21), a path π through a Kripke Structure κ satisfies $\psi_1 \wedge \psi_2$ if and only if π satisfies ψ_1 and π satisfies ψ_2. Our formalization of LTL semantics is in terms of eventually periodic paths, and hence a path π here means a periodic path, but this characterization is preserved by our definition.) By induction over the structure of the formula it therefore follows that if ψ can be decomposed by *R-and* to the collection $\langle \psi_1, \ldots, \psi_k \rangle$ then π must satisfy ψ if and only if it satisfies each ψ_i. Note that κ satisfies ψ if and only if every periodic path through κ satisfies ψ. By above, this means κ satisfies ψ if and only if each periodic path satisfies each ψ_i, that is, κ satisfies each ψ_i. Finally, a finite state system satisfies ψ if and only if *kripke*(M) satisfies ψ. Since *kripke*(M) is a Kripke Structure (by **F1**) it follows that M satisfies ψ if and only if M satisfies each ψ_i. ∎

The crux of the verification, then, is to prove that **R2** is a theorem. How do we prove that? Unfortunately, the definition of the semantics of LTL based on

eventually periodic paths makes this proof complicated. To see the complications involved, let us first review the standard approach to carrying out this proof.

The traditional proof of cone of influence reduction uses a bisimulation argument. Given two Kripke Structures $\kappa \doteq \langle S, R, L, s_0 \rangle$ and $\kappa' \doteq \langle S', R', L', s'_0 \rangle$ on the same set AP of atomic propositions, a predicate B on $S \times S'$ is called a *bisimulation predicate* if and only if the following three conditions hold for any state $s \in S$ and $s' \in S'$ such that $B(s, s')$:

- $L(s)$ is the same as $L(s')$.
- For any $s_1 \in S$ such that $R(s, s_1)$, there exists $s'_1 \in S'$ such that $R'(s', s'_1)$ and $B(s_1, s'_1)$.
- For any $s'_1 \in S'$ such that $R'(s', s'_1)$, there exists $s_1 \in S$ such that $R(s, s_1)$ and $B(s_1, s'_1)$.

Let π and π' be two paths starting from s and s' in κ and κ' respectively. Let us call π and π' *corresponding* if and only if $L(\pi[i])$ is the same as $L'(\pi'[i])$ for each i. The following proposition is a standard result about bisimulations.

Proposition 5 *Let B be a bisimulation predicate and s and s' are two states in κ and κ' respectively such that $B(s, s')$. Then for every path starting from s there is a corresponding path starting from s' and for every path starting from s' there is a corresponding path starting from s.*

Further, by structural induction on the characterization of the semantics of LTL, we can now deduce the following proposition.

Proposition 6 *Let ψ be an LTL formula and let π and π' be corresponding paths. Then π satisfies ψ if and only if π' satisfies ψ.*

Call the Kripke Structures κ and κ' *bisimulation equivalent* if and only if there exists some bisimulation predicate B such that $B(s_0, s_0')$. The following proposition then follows from 5 and 6, and is the crucial result we will use.

Proposition 7 *If κ and κ' are bisimulation equivalent, then for every LTL formula ψ, κ satisfies ψ if and only if κ' satisfies ψ.*

What has all this got to do with cone of influence? If M' is the reduced model of M, then it is easy to define a bisimulation relation on the states of the Kripke Structures of M and M'. Let C be the set of state variables in M'. Then we define the bisimulation predicate B as follows: Two states s and s' to be bisimilar if and only if the variables in C are assigned the same (Boolean) value in both states. It is easy to show that B is indeed a bisimulation predicate and the Kripke Structures for M and M' are therefore bisimulation equivalent. The proof of correctness of cone of influence reduction then follows from proposition 7.

Let us now attempt to turn the argument above into a formal proof. The key is to formalize proposition 5. Recall that in our formalization, π must be an eventually periodic path. Why does this complicate matters? To understand this, let us first see how the traditional argument for correctness of this proposition goes, where π can be an infinite (but not necessarily eventually periodic) path.

Traditional Proof of Proposition 5: Let $B(s, s')$ and let $\pi = p_0 p_1 ...$ be a path starting from $s \doteq p_0$. We construct a corresponding path $\pi' = p_0' p_1' ...$ from p_0' by induction. It is clear that $B(p_0, p_0')$. Assume $B(p_i, p_i')$ for some i. We will show how to choose p_{i+1}'. Since $B(p_i, p_i')$ and $R(p_i, p_{i+1})$, there must be a successor q' of p_i' such that $B(p_{i+1}, q')$. We then choose p_{i+1}' to be q'. Given a path π' from s', the

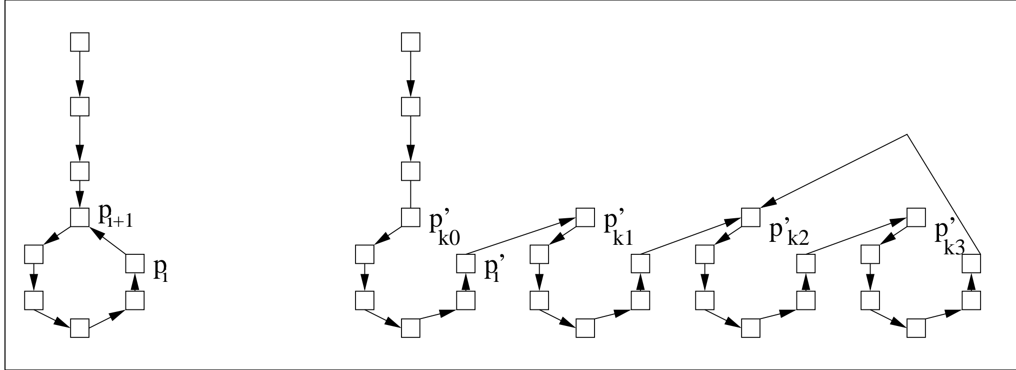

Figure 16.2: A Periodic Path and Its Match

construction of a path π is similar. ■

The argument above is very simple. Indeed, in Chapter 8, when we had the "luxury" of modeling infinite sequences directly as functions, we could formalize a similar argument to formalize the proof rule that relates well-founded refinements to stuttering trace containment.

Let us now attempt to formalize this argument for eventually periodic paths. Thus, given a (periodic) path π, we must now be able to construct the corresponding (periodic) path π' by induction. As in the traditional argument, assume that we have done this for states up to p_i, and that the corresponding state for p_j is p'_j for every j up to i. Now, we need to invoke the bisimulation conditions to determine the corresponding state for state p_{i+1}. Since we know $R(p_i, p_{i+1})$, we therefore know from the that there exists a state p'_{k1} such that s_{i+1} is bisimilar to p'_{k1} and $R(p'_i, p'_{k1})$. We will be therefore tempted to "match" p'_{k1} to s_{i+1}. However, as we illustrate in Figure 16.2, we face a complication since the edge from p_i to p_{i+1} might be an edge back to the beginning of a cycle containing p_i. In that case, p_{i+1} might have already

been matched to some state p'_{k0}, and p_{k0} is not necessarily the same as p'_{k1}. Of course we know that both s'_{k0} and p'_{k1} are bisimilar to p_{i+1}. However, to construct an eventually periodic path, we are required to produce a prefix and a (non-empty) cycle, and the cycle is not completed yet!

Our approach to constructing the periodic path, as shown, is to continue the process of matching, thereby encountering states p'_{k2}, p'_{k3}, and so on, until eventually we reach some state p'_{kl} which is the same as *some* p'_{km} for $m < l$. This must happen, by the pigeon-hole principle, since the number of states in the Kripke Structure κ' is finite. Once this happens we then have our periodic path, which has all states up to p'_{km} as the prefix and the rest in the cycle.

The pigeon-hole argument is possible, though non-trivial, to formalize. The rest of the proof of **R2** follows the traditional structure outlined above, namely by formalization of propositions 6 and 7.

16.4 Discussion

Our descriptions show that it is possible to integrate a compositional model checking procedure by modeling it as a formal theory and proving the characterization theorems. How much effort is it to do such reasoning using a theorem prover? It took the author, then a fairly inexperienced (although not novice) user of ACL2, slightly more than a month to define the procedure and prove all the lemmas up to the **Main** theorem, and most of the time was spent on determining the pigeon-hole argument above. Given our experience we believe that it will be possible, though perhaps not trivial, to integrate more complex compositional procedures.

Our integration, however, has one serious drawback, which makes our work

less satisfactory than what the author would have liked. The semantics of LTL, as formalized, is not the standard formulation. While the equivalence between our formulation and the standard one is a well-known result in model checking, it is less intuitive. As we realized above, this makes the proof of correctness of the reduction procedure complicated. It is important to note that for successfully using theorem proving on a substantially complex problem one must be able to have a simple mental picture of the broad steps in the formal proof. It is therefore crucial that the formal definitions be close to the intuitive notion of the executions of the function that the user has in mind. The complexity in the pigeon-hole argument above principally arises from the fact that it is not inherent in a bisimulation proof but rather an effect of our particular formalization of the semantics of LTL.

Let us therefore understand what the chief obstacles were to define the natural semantics of LTL. There were two: (1) GZ does not have axiomatization of infinite sequences as first class objects, and (2) ACL2 does not permit the use of recursive functions with quantifiers. The second seems to be a more serious obstacle, and hence it is worth our while to understand the reasons for it.

The chief reason for having this restriction is to maintain the property called *conservativity*. Informally, one can think of this property as follows. Suppose we have created a theory $\mathcal{T}$ by introducing several function symbols via the extension principles, and suppose we want to prove a formula Φ as a theorem about these functions. In doing this proof we might have to introduce several other auxiliary functions. We have seen that happen many times already. For example, given two reactive systems $\mathcal{S}$ and $\mathcal{I}$, the theorem $(\mathcal{S} \rhd \mathcal{I})$ involves only function symbols representing the definitions of the two systems along with the definition of appropriate

traces. But we might want to do the proof using well-founded refinements and hence introduce more functions such as *inv*, *skip*, etc. But by introducing such function symbols we extend the theory $\mathcal{T}$ in which we wanted to prove our theorem to a new (extended) theory $\mathcal{T}'$. How do we then know that the formula we wanted to prove is still a theorem in $\mathcal{T}$? This is guaranteed by conservativity. More precisely, a (first order) theory $\mathcal{T}'$ is a *conservative* extension of $\mathcal{T}$ if and only if for any formula Φ that is expressible in $\mathcal{T}$, Φ is (first order) provable in $\mathcal{T}'$ if and only if it is also (first order) provable in $\mathcal{T}$. Kaufmann and Moore [KM01] prove that the extension principles of ACL2 do have this property. Conservativity is a crucial property of ACL2, which, in addition to allowing several structuring mechanisms in the theorem prover, provides the basic arguments for logical consistency of the theories built by the extension principles. The reason for that is as follows. It is well known that a first order theory $\mathcal{T}$ that is strong enough to express arithmetic is consistent if and only if there is at least one formula expressible but not provable in the theory. (Informally this is also written as: "The theory cannot derive *false*.") In ACL2, `NIL` is of course a formula that is expressible in GZ (and any extension). Thus by conservativity, `NIL` is provable in some theory $\mathcal{T}$ if and only if `NIL` is provable in GZ. Thus conservativity reduces the consistency of arbitrary theories to the consistency of GZ.

We now understand that conservativity is an important property of ACL2 that needs to be preserved. To see how it is possible to violate conservativity with recursion and quantification together, let us consider the "definition" of *true-p* in Figure 16.3 which would have been possible had ACL2 allowed it.[2]

[2]This example was furnished by Matt Kaufmann.

$$\textit{true-p}(\Phi, \sigma, dfns) \triangleq \begin{cases} \textit{true-exists-p}(\textit{formals}(\Phi), \textit{body}(\Phi), \sigma, dfns) & \textit{if existsp}(\Phi) \\ \textit{true-forall-p}(\textit{formals}(\Phi), \textit{body}(\phi), \sigma, dfns) & \textit{if forallp}(\Psi) \\ \textit{evaluate-term}(\Phi, \sigma, dfns) & \text{otherwise} \end{cases}$$

$$\textit{true-exists-p}(fs, \Phi, \sigma, dfns) \triangleq (\exists val : \textit{true-p}(\Phi, \textit{map}(fs, val, \sigma, dfns)))$$

$$\textit{true-forall-p}(fs, \Phi, \sigma, dfns) \triangleq (\forall val : \textit{true-p}(\Phi, \textit{map}(fs, val, \sigma, dfns)))$$

Figure 16.3: A Truth Predicate

The predicate *true-p* checks if a formula Φ evaluates to T under some assignment σ to the free variables and a collection of function definitions $dfns$. Here, the predicates *existsp* and *forallp* check if the argument is syntactically of the form of an existential quantification, or universal quantification respectively, and *body* returns the body of a quantified formula. Finally, given a quantifier-free formula Φ, an assignment σ, and an arbitrary collection $dfns$ of definitional equations for the function symbols referred to in Φ, *evaluate* returns the value of Φ under the assignment σ.

It is well-known that given this "definition" one can prove by induction based on the term $\textit{true-p}(\Phi, \sigma, dfns)$ that every formula that is provable is true. Now consider the situation in which $dfns$ is a list of definitions of functions axiomatizing the operations of Peano arithmetic. Then the above discussion indicates that one can prove the consistency of Peano Arithmetic if the formula above is admissible. However, by Gödel's Incompleteness Theorem [Göd31, Göd92], in any theory that is a conservative extension of Peano Arithmetic it is impossible to prove the consistency of Peano Arithmetic. The ACL2 ground zero theory GZ is a conservative extension of Peano Arithmetic. Hence the "definition of *true-p* above is not admissible in any conservative extension of GZ.

The above argument makes it clear that it is not possible to extend the definitional principle of ACL2 to allow quantification in recursive definitions. However, it is still possible to augment the logic in a restricted manner so as to provide a way of introducing the natural semantics of LTL in ACL2. Several such proposal are being considered by the authors of the theorem prover. Here we briefly sketch one of them that involves introduction of functions as first-class objects in the theorem prover, while still retaining the first order character of the logic.[3] Another proposal that has received a lot of attention recently concerns the possibility of integrating ACL2 with the HOL theorem prover.[4]

16.4.1 Function Objects

The proposal is to provide a new extension principle for ACL2, that allows us to introduce a new *function object*. A function object is just like any other object in ACL2, for example integer, string, etc., and different from a *function*. The difference between a function object and a function is to maintain the first order nature of the ACL2 logic. But since these objects are like any other objects, functions can be defined to manipulate such objects. To formalize the notion of function object, the proposal is to axiomatize in GZ two new binary functions *apply* and *rank*. Suppose we want to extend a theory $\mathcal{T}$ with a new function object f that returns a function of a single argument x with the body $(x + 1)$ then the axiom introduced to extend

[3] Although the author strongly supports the proposal for augmenting the ACL2 theorem prover with function objects, it should be made clear that the author claims no credit for the conception of the proposal or in the discussions resulting in its current form. The sketch of the proposal we present here has been worked out by Matt Kaufmann and John Matthews. We merely discuss it to show how it will help in solving the problems we encountered.

[4] HOL is a theorem prover for higher-order logic. The integration is aimed to allow the user to use the expressiveness of HOL together with the automation and fast execution capabilities of ACL2. We believe that this proposal, as the proposal for function objects, will allow us to remove the logical limitations that we faced in our work. However, since the author is unfamiliar with the HOL logic it is difficult for us to qualify how the proposal when implemented should be used in overcoming the limitations.

the theory is as follows (along with the introduction of the function f):

$$\mathit{rank}(x, 0) \Rightarrow \mathit{apply}(f(), \mathit{list}(x)) = x + 1$$

$$\mathit{rank}(f(), 1)$$

The axioms roughly say that *apply*ing the function returned by f on argument x gives us $(x + 1)$. We can think of f as returning a function which, when applied to argument x increments x by 1. Of course in this case the function f itself is 0-ary, but in general it is possible for f to have any number of arguments, thus returning different functions for different arguments. The predicate *rank* is necessary for technical reasons to insure consistency of the extended theory. Since one function object can take other function objects as arguments, it is important for consistency that the extension does not allow a formula like the below to be a theorem in the extended theory.

$$\mathit{apply}(g(), \mathit{list}(x)) = \neg \mathit{apply}(g(), \mathit{list}(x))$$

Notice that the above formula, if it is a theorem, can be easily used with other axioms of GZ to prove `NIL` and hence every formula. Indeed, it is well-known that if functions can take arbitrary other functions as arguments then one can exploit Russell's paradox to get inconsistency. To avoid such issues, the proposal stratifies function objects by using *rank*. More precisely, we think of all the "normal" ACL2 objects to have *rank* 0. This means that if x is, for example, an integer, then $\mathit{rank}(x, 0)$ is a theorem. A function object has a higher rank. The axioms only specify the result of *apply*ing a function object of rank n on objects of rank less than n. When a function object f is introduced by the (proposed) extension principle, its *rank* is axiomatized to be more than the rank of any object occurring in the body of the function returned by f. For simplicity, we ignore the *rank* considerations for the remainder of the discussions. We will also write function objects in traditional notation as λ-expressions, and use $\equiv$ instead of $=$ to remind ourselves that it is

introduced as an axiom in the new principle. Thus the axiom for f above can be written as follows:

$$f() \equiv \lambda x.(x+1)$$

Why is it sound to extend a theory by axioms for function objects? Kaufmann and Matthews [private communication] provide an argument for the consistency of the theory obtained by extending a theory $\mathcal{T}$ with axioms for function objects. The argument shows that there exists a first order structure such that all the objects can be interpreted to be objects in the structure and all the function objects can be interpreted to be functions. Foundational results in logic show that if such a structure — called a model of $\mathcal{T}$ — exists, then $\mathcal{T}$ is consistent.

Can function objects solve the problems that we encountered in modeling the semantics of LTL? Since the proposal is still in its early infancy our response can only be speculative. Our understanding of the proposal seems to give an affirmative answer. Recall that functions can be used to model infinite sequences (or paths). Thus, we can now define a function object that returns infinite paths through a Kripke Structure. What do we need for such paths? We need to be able to quantify predicates over such paths, and have recursion. Quantification, in fact, can also be coded as a function object. For example, suppose for a function object f, we want to write the predicate $\exists f : P(f)$. We can do so by encoding the predicate P as a function object and asking if this predicate is not *equal* to `NIL`. The proposal, for pragmatic reasons, does not allow recursive function objects in its current form. This makes it slightly non-trivial to define the semantics of formulas such as $\mathsf{F}\psi$ over a function object modeling an infinite path. This is done by replacing the recursive application of a function object by a series of non-recursive functions. To see how this can be done to define the semantics of LTL, let us first create a function object *ipath* that creates k "copies" of an infinite path π, each starting with a different

$$\mathit{suff}(\psi, \pi, AP) \triangleq \mathit{ipath}(\mathit{wit}_{\mathit{P\text{-}F}}(\mathit{value}(\psi), \pi, AP), AP)$$

$$\mathit{psemp}(\mathit{sem}, \psi, \pi, AP) \triangleq \begin{cases} \ldots \\ \mathit{psemp}(\mathit{sem}, \mathit{value}(\psi), \mathit{suff}(\psi, \pi, AP) & \mathit{if\ unary}(\psi) \wedge (\mathit{op}(\psi) = \mathsf{F}) \\ \ldots \end{cases}$$

Figure 16.4: Path Semantics in terms of Function Objects

starting point.[5]

$$\mathit{ipath}(k, \pi) \equiv \lambda n.(\begin{cases} \mathit{apply}(\pi, \mathit{list}(k+n)) & \mathit{if\ natp}(n) \\ \texttt{NIL} & \text{otherwise} \end{cases})$$

Notice that we have not introduced π itself as a function object, but merely use it as an argument for *ipath*. This is possible since *apply*, as any other "normal" function, is total.

We now want to write functions that represent formulas such as $\mathsf{F}\psi$ and $\mathsf{G}\psi$. We do this by "normal" predicates using quantifiers. Here is the predicate for $\mathsf{F}\psi$.

$$\mathit{P\text{-}F}(\mathit{sem}, \psi, \pi, AP) \triangleq \exists k : (\mathit{apply}(\mathit{sem}, \mathit{list}(\psi, \mathit{ipath}(k, \pi), AP)))$$

To understand the function above, it is useful to think of the argument *sem* as a function object that specifies for a formula ψ, infinite path π and the set of atomic propositions AP, if the suffix of π starting from k satisfies ψ. Since this is an *object* we can manipulate it using normal ACL2 functions. We can now write a predicate *psemp* more or less naturally to recognize if a function object *sem* indeed specifies the semantics of LTL for an infinite path π. In Figure 16.4 we show the skeleton of such a definition highlighting the case for the temporal operator F. It is instructive to compare the definition of *psemp* with the definitional axiom for *pathsem* that we "wished" in page 307. Notice that we have just avoided the issue of requiring

[5]The definition of the semantics of LTL has been elaborated by the author from an example provided by Matt Kaufmann.

recursion with quantification by first defining a quantified predicate to posit the existence of the relevant suffix and used the Skolem witness for the predicate to define the recursive predicate *psemp*. We can then define the predicate *psem* by simply positing the existence of a function object *sem* such that *psemp* holds.

$$\mathit{psem}(\psi, \pi, AP) \triangleq \exists \mathit{sem} : (\mathit{psemp}(\mathit{sem}, \psi, \pi, AP))$$

Although it is slightly more cumbersome to define the semantics of LTL this way without having recursive function objects, it is surely doable, and the predicate does mimic the natural semantics much more closely than our formalization in terms of eventually periodic paths. In particular, we believe that it will be much simpler to prove the correctness of the reductions based on this formulation when ACL2 is extended with function objects.

It should be noted that with function objects much of the work in the earlier parts of the dissertation would also be simplified and become more elegant. For example, we could then express stuttering trace containment as a closed-form formula in ACL2, and the formalization of our proof rules would be possible without requiring encapsulation for specifying generic input sequences.

16.5 Summary

In this chapter, we have shown how to define and prove characterization theorems for a simple compositional model checking in ACL2. Although the algorithm itself is simple, its verification is nonetheless representative of the issues involved when formally reasoning about model checking. Our work indicates that the approach of defining such algorithms in the logic of a theorem prover, formally verify them, and then applying them as decision procedures for simplifying large verification prob-

lems, is viable. While our proof of the cone of influence reduction is complicated, the complication arises mainly from the limitations in ACL2's expressiveness which forced us to use eventually periodic paths. We must admit that the functions we defined to model the reduction algorithms are not very efficient for execution purposes. Our goal in this chapter has been to investigate if it is indeed practicable to reason about reductions formally in the logic with reasonable human intervention. Our experience with theorem proving suggests that once one can prove theorems characterizing simple implementations of an algorithm it is usually not very difficult to "lift" such theorems to more efficient implementations.

The limiting problem in our work has been posed by the need to reason about eventually periodic paths. Besides making the proofs of reduction theorems complicated, the non-standard definition suffers from another flaw. The equivalence between the standard formulation of LTL semantics and ours is true only when we are talking about finite state systems. Recently there has been work on model checking certain classes of parameterized unbounded state systems [EK00]. It would be useful to integrate such procedures with ACL2. But this is not possible using the current formulation. We therefore consider it important to augment the ACL2 logic with a construct like function objects. Reasoning about infinite sequences is one of the key activities that needs to be performed when verifying (reactive) computing systems. It is possible to do such reasoning in some form even though the logic does not allow infinite sequences as objects. We saw that in Part III when we formalized the notion of stuttering trace containment and proof rules for showing correspondence based on the notion. But our work shows that such formalization is cumbersome and difficult, and does not allow us to state correctness theorems for

systems in closed form. In this context it is important to note that the key advantage of theorem proving over algorithmic decision procedures lies in the expressiveness of the logic. Beyond this, theorem proving admittedly involves more manual effort. To be effective, the logic of a theorem prover therefore must be expressive enough so that the user can formalize the arguments in a manner that is natural to the user. Since one of the strengths of ACL2 is in reasoning about computing systems, we believe it is imperative that it provides some extensions and logical constructs to allow the user to naturally formalize reasoning about infinite sequences.

16.6 Bibliographic Notes

There has been some work in using a theorem prover to model and reason about another proof system or decision procedure. Chou and Peled [CP99] present a proof of correctness of partial order reductions using HOL. A formalization of TLA in HOL was created by von Wright [vW91]. Schneider and Hoffmann [SH99b] present a proof of reduction of LTL to ω-automata in HOL. In ACL2, Manolios [Man00b] models a μ-calculus model checker, and proves that it returns a fixpoint.

The general approach of proving formulas in a theory $\mathcal{T}$ as theorems in some (possibly bigger) theory $\mathcal{T}'$ embedding a proof system for $\mathcal{T}$ in $\mathcal{T}'$ is referred to as *reflection*. Reflection is getting increasingly popular in the theorem proving community, and an excellent paper by Harrison [Har95] provides an extensive survey of the key results in the area. Some of the early uses of embeddings include the work of Weyhrauch [Wey80], which allowed the evaluation of constants in the FOL theorem prover by attaching programs. Metafunctions [BM81] was introduced in the Nqthm theorem prover precisely for the purpose of exploiting reflection. In addition, the

use of verified VCGs with theorem provers can be viewed as the use of reflection to prove theorems about sequential programs by semantic embedding of the axiomatic semantics of the corresponding programming language. The bibliographic notes of Chapter 4 provides some of the references to this approach.

This chapter, and parts of the next, are based on a previous paper written by the author with John Matthews and Mark Tuttle [RMT03], and have been incorporated here with permission from the co-authors. The paper gives more technical details of the proof and the complications involved, which have been omitted for brevity from the current presentation.

Chapter 17

Theorem Proving and External Oracles

In the previous chapter, we showed a way to define the semantics of LTL as a formal theory in ACL2, and formally integrated a compositional procedure. Suppose we are now given a concrete finite state system $\hat{M}$, a concrete collection of atomic propositions $\widehat{AP}$, and a concrete LTL formula $\hat{\psi}$ to check about $\hat{M}$. We can then simply *execute* the function *Reduce* on these concrete inputs to determine the collection of smaller verification problems. The characterization theorem we have proved guarantees that to decide if $\hat{M}$ satisfies ψ it is sufficient to model check these smaller problems. However, we now need to model-check these problems in order to decide whether $\hat{M}$ does satisfy ψ.

How do we do this? Our formalization of the model checker, namely the definition of *ltlsem*, has been defined using quantification, that is, via the defchoose principle. The definition was appropriate as long as our goal was to determine a

formalization of the semantics of LTL that was elegant (or at least as elegant as possible given the limitations on the expressive power of the logic) for the purpose of reasoning. But it is unsuitable for the purpose of *computing* if a finite state system $\hat{M}$ satisfies formula $\hat{\psi}$. Since the definition is non-constructive, it cannot be executed.

One way to resolve this problem, of course, is to define an executable model checker for LTL in ACL2, prove that it is equivalent to *ltlsem* and then use the executable model model checker to check the individual verification problems. Defining an LTL model checker involves a complicated tableau construction as we saw in Chapter 2. Formalizing the construction is possible, although non-trivial, and proving that the construction is equivalent to *ltlsem* possibly requires some formal derivation of properties of Büchi automata. However, from a practical standpoint, such a resolution is unsatisfactory for one crucial reason. A key to the practical success of model checkers, and in general, decision procedures, lies in the efficiency of implementation. Formalizing a decision procedure so as to achieve efficiency comparable to an implementation is at best a substantial undertaking. Further, if we take that route, it implies that every time there is an innovative implementation of a decision procedure is discovered, we must invest considerable effort to "reinvent the wheel" by formalizing the implementation. But what do we gain out of it? One potential benefit is that if we formalize the implementation and prove that it is equivalent to the more natural (and less efficient) one which is "obviously correct" such as *ltlsem* above, then the efficient implementation contains no bugs. But users of formal verification are willing to trust the core implementations of many of the industrial tools. In fact it might be contended that one can bestow about as much

trust to the implementation of a well-used model checker as one would be willing to trust the implementation of a theorem prover like ACL2.

The considerations above suggest that it is important look for ways for integrating external decision procedures (or *oracles*) with ACL2. Notice that integrating an external model checker does not obviate the need for formally defining its semantics as we did using *ltlsem* in the last chapter. This should be done irrespective of whether we use an oracle or choose to formalize an efficient version of the procedure in the logic. When using an oracle, it is the semantics, defined as a formal theory, that tells us what one can conclude *in the logic* from a successful use of the procedure on a verification problem. For example, suppose we use a model checker like Cadence SMV [McM93] to prove some LTL property of a finite state system. Then what we wish to conclude as theorem is that the predicate *ltlsem* holds for the system and the property. We will want to use this theorem to explore other properties of the system considered, possibly using theorem proving or other oracles. However, the use of external oracles does obviate the need for formally defining an efficiently executable version of the decision procedure, the trade-off being that the user must trust that the implementation of the oracle performs as prescribed by its formal semantics.

As an aside, we have already seen some benefits of using external oracles with a theorem prover. Our tool for predicate abstraction in Part IV is a glaring example. There we developed a tool for proving invariance of predicates of reactive systems using rewriting for predicate abstraction. Is it possible to formally verify our tool using ACL2? The answer is a qualified and hesitant “yes”. The tool was implemented using the ACL2 programming language. It is possible to formally

verify the basic *algorithm* in the tool. But verifying the actual implementation that contains several optimizations and heuristics is a substantial investment of effort, which is not viable in practice. And of course the interfaces to SMV and VIS cannot be verified since these tools are not written in ACL2. Eventually we will want to verify our procedure but for now we found it more worth our time to sharpen and optimize the different heuristics and implementations so as to make the tool more efficient. Incidentally, the same development model is followed by the authors of the theorem prover themselves. The ACL2 theorem prover, at its core, can be thought of as a program that proves theorems in the logic we discussed in Chapter 3. The implementation of the theorem prover is among the largest programs written using a functional programming language, implementing several heuristics and procedures. Most of the implementation code is written in the ACL2 programming language itself. In principle, it is possible to verify most of the ACL2 code using a small trusted "kernel" to check such proofs. But formally proving each heuristic and procedure correct is a serious enterprise and practical considerations inhibit such verification although the entire code is carefully inspected and rigorous but informal arguments are often constructed for their correctness. In a certain sense, we can think of each such procedure as an oracle assisting in the proof of theorems written in the logic of the theorem prover.

17.1 Integrating Oracles with ACL2

There are two kinds of oracles that one might integrate with the ACL2 theorem prover. One type is like the predicate abstraction tool. It uses model checking and other procedures to prove a formula as a theorem, but it also makes use of the

theorems and lemmas that the user has proven, that is, the theory in which the tool has been invoked. We will not talked about the theorem prover implementation in this dissertation. But there is one aspect of the implementation that is important for our discussion here. The theorem prover keeps track of the "current theory" and the theorems proved in the theory in (what may be thought of as) a global variable. Let us call this global variable the *state* of the theorem prover. Extending a theory by an axiom, or proving a theorem changes this *state*. Thus, we can think of the predicate abstraction tool as a *state-dependent* oracle. Another form of external oracle whose action can be thought of as efficient function evaluation. Consider for example integrating ACL2 with a model checker in the following way. Whenever we want to check if *ltlsem* holds for a (constant) finite state system and a fixed LTL property, we call this model checker to answer that question. The action of the model checker is in no way related to the current theory, and at least in principle should provide the same response every time it is asked to verify the same LTL property on the same system. These might be called *state-independent* oracles. It should be clear that they form only a special case of state-dependent ones.

The ACL2 implementation does *not* provide a way of integrating either kinds of oracles. Thus any such attempt must involve "hacking" the implementation in some way. Given the size and complexity of the code base, such hacks should be kept to a minimum. It is possible to integrate state-independent model checkers in a way that does not modify the implementation code at all, although it does modify how the theorem prover behaves at run-time. This is done by exploiting the close connection between ACL2 and Lisp. In particular, while evaluating a function on some (constant) arguments, if the theorem prover can ascertain that the value of the function based on the axioms of the logic is the same as the value returned

by Common Lisp, then ACL2 simply uses Lisp's execution engine to evaluate the function. Here is how we can exploit this feature to integrate a state-independent oracle.

1. Define the semantics of the oracle in the logic. Call this function *fsem*.
2. Introduce another function *fsem-hack* in ACL2 with the same arity as *fsem*.
3. Introduce an axiom specifying that *fsem-hack* is logically equal to *fsem*.
4. Introduce another axiom stating that *fsem-hack* is compliant with Common Lisp.

The last two actions are possible since ACL2 does have a construct for extending a theory $\mathcal{T}$ by adding an arbitrary formula expressible in $\mathcal{T}$ as an axiom. This "extension principle" is obviously not conservative, and the user is encouraged not to use it. But it is suitable for our purpose in this case.

Once the four steps above are performed, we can redefine *fsem-hack* in the underlying Lisp as a call to our desired external tool. This is possible since ACL2 (and the underlying Lisp) allows the user to invoke arbitrary operating system commands while programming.

We have integrated Cadence SMV [McM93] with ACL2 using the above approach. Thus we create functions *ltlsem-hack* with different "definitions" in Lisp and ACL2 as above. With the hack, we can use ACL2 to "prove" given a finite-state system and an LTL property if the system does satisfy the property. It uses the definition of *Reduce* to decompose the verification problem into pieces, uses our reduction theorem to show that the original problem *pass*es if each piece does, and uses SMV to verify each piece. The integrated system has been used to verify properties of some finite state systems.

The integration shows how we can "pick and choose" which portions of an

oracle we wish to trust. In our case, while we trust the core model checking engine of SMV, we do not need to rely on its own (unverified) decomposition algorithms. We pay some penalty in performance for that — for example the efficiency of this integrated approach is nowhere near that of the use of SMV alone. This might be improved by a more efficient implementation of *Reduce*, adding more reductions, or deciding which other portions of an external oracle we feel comfortable to trust.

Integrating a state-dependent oracle, however, is more complex. In principle, it is possible to do the same thing that we have done for state-independent ones, namely pass the theorems and axioms in the current theory along with the formula to be proven to the external oracle. But in practice we found the process too inefficient. In addition, when the oracle proves a formula to be a theorem, ACL2 needs to update its *state* to record that fact. In case of the model checker, this problem "worked out" by accident. The formulas that we intended to prove with the oracle were about fixed finite state systems and fixed LTL properties. In other words the formulas did not contain variables. When ACL2 encounters such formulas it attempts to establish their theorem-hood by evaluation; this allowed us to override the evaluation process by directing the theorem prover to execute the redefinition of the functions in Lisp. But in general this is not the case. Furthermore, the program that our state-dependent oracle represents, namely the predicate abstraction tool, is mostly written in the ACL2 programming language itself. We therefore prefer to minimally modify the source of ACL2 allowing the user to call our tool as a hint to the theorem prover.

17.2 External Oracles and Clause Processors

The description of the previous section should make it clear that our integrations are implementation hacks, embarrassingly so. To avoid such hacks it is important that the theorem prover provide some ability to the user for "hooking in" external oracles. We are happy to report that the authors of ACL2 are considering a proposal to add such enhancements.[1] The idea is to allow the user to specify external programs as *clause processors*. The programs might be written in the ACL2 programming language or might be scripts for the operating system. The user is allowed to invoke a clause processor as a hint for proving a lemma or a theorem. The theorem prover keeps track of the clause processors used in the proof. In this case, of course, the theorem prover cannot be "responsible" for the soundness of the formula proven as theorem. One way to view the composite tool is as an implication. That is, if there is a first order structure in which all the axioms of the current theory as well as all the formulas proved by a clause processor are truths then the formula derived is a theorem.

The current proposal also allows the user to incrementally verify some of the clause processors and then use them as *verified clause processors*. Once a clause processor is verified the theorem prover does not need to track its application in the proof of a conjecture. We should note that ACL2 does already have the capability of using user-defined verified formula simplifiers via what are known as *meta theorems* [BM81]. The current proposal, however, is more general than the current capabilities. We believe that the proposal is a practical way of scaling up theorem

[1] The author advocates the use of external oracles and was involved in the motivational discussions for this proposal. But we claim no credit for the details of the proposal itself.

proving. In this way, one writes tools and procedures to assist the theorem prover in proofs in certain domains, uses such tools as (unverified) clause processors, and only after it is established that the tools are indeed useful, verifies them to reduce the overhead and inefficiency involved in tracking their applications. It should be noted that if clause processors are incorporated into ACL2 then it can help decompose the implementation of the theorem prover itself. For example, many heuristics and optimizations now implemented within the source code of ACL2 could then be defined and verified as clause processors. The user then could get the same benefit of automation as ACL2 now provides but would need to trust a much smaller "implementation kernel" than the current code base of the theorem prover.

17.3 Summary

We have discussed some of the issues involved in integrating external oracles with a theorem prover and showed how they can benefit scaling up deductive reasoning. Neither theorem proving nor a decision procedure is sufficient in formally verifying modern reactive systems. It is necessary for the two approaches to cooperate, and for this purpose it is imperative that the different verification tools have a way of communicating with one another. For example, Moore [Moo03a] has issued a grand challenge to the users of formal verification to verify a complete computing system stack from transistors to application programs. Presumably, attacking such a challenge will require collaboration of several tools. A practical approach might be to use decision procedures for solving small but low-level problems in parts of the stack, using theorem proving to compose the results of such verification. However, if ACL2 is now used to embark on the challenge then such collaboration is not possible. We

believe that a solution like the use of clause processors will significantly ameliorate such collaboration problem of ACL2 in future.

17.4 Bibliographic Notes

Using external oracles with a theorem prover has a long history. The Isabelle theorem prover provides a general method of external oracles in lines similar to what we have advocated here [Pau, § 6]. Isabelle's external oracle mechanism has been used to integrate (1) an efficient μ-calculus model checker, as part of a theory for I/O-Automata [MN95], (2) the Stanford Validity Checker as an arithmetic decision procedure for the Duration Calculus [Hei99], and (3) the MONA model checker, as an oracle for deciding formulas expressed in the weak second-order monadic logic of one successor [BF00]. The PROSPER project [DCN$^+$00] is another key project integrating external oracles with the HOL98 theorem prover and attempts to achieve uniform uniform and logically-based coordination between external verification tools. The most recent incarnation of this family of theorem provers, HOL4 [Kr], uses an external oracle interface to decide large Boolean formulas through connections to BDD and SAT-solving libraries [Gor02]. PVS also provides integration with model checkers [RSS95].

We believe that our integration of ACL2 with Cadence SMV represents the first attempt to integrate a decision procedure for proving decidable properties of computing systems with ACL2, although other kinds of external procedures had been integrated before with diverse motivations. The previous research closest to ours is the work of McCune and Shumsky [MS00]. They integrate a resolution/refutation procedure based on the Otter theorem prover [McC97a] with ACL2

to create a composite system called Ivy, which is used to reason about formulas in first order logic. Greve, Wilding, and Hardin [GWH00] also attach efficient C programs to formal operational models of processors for the purpose of efficient simulation.

Since the first publication of our work [RMT03], however there have been independent upsurge in research integrating external oracles with ACL2. Manolios and Srinivasan [MS04, MS05] integrate UCLID with ACL2 using an approach similar to ours by a formal characterization of the semantics of UCLID. Reeber and Hunt [RH05] integrate an external SAT solver with ACL2.

Part VII

Conclusion and Future Directions

Chapter 18

Summary and Conclusion

The focus of this dissertation has been to determine ways for effectively scaling up formal verification for large-scale computing systems using a mixture of theorem proving and decision procedures. The key contributions in the work presented here are the following:

- We developed a compositional approach based on symbolic simulation to apply assertional reasoning for verification of operationally modeled sequential programs.
- We formalized the notion of stuttering trace containment and showed its applicability as a notion of correctness in reasoning about reactive systems. We showed how to use this notion effectively to verify concurrent protocols and pipelined machines.
- We developed an extendible, deductive procedure based on term rewriting to compute predicate abstractions. The procedure allows us to prove invariance of predicates without requiring manual construction of an inductive invariant.
- We explored a general framework for integrating model checking with theorem proving. In particular, we formalized the semantics of LTL in ACL2 and used it to prove characterization theorems for a compositional model checking procedure.

Our approach throughout this dissertation has been to find ways of exploiting the strengths of the different approaches to formal verification without incurring their weaknesses. The strength of theorem proving is two-fold. First, one can express the statement of correctness concisely as a formal statement in a mathematical logic. Secondly, one can control the process of derivation of formulas as theorems. The importance of this second strength often overrides the inevitable lack of automation which is a consequence of the expressiveness of the logic. Most modern systems are currently beyond the scope of automatic reasoning tools such as model checking. It is therefore important that the user should be able to decompose the problem into manageable pieces. Theorem proving affords such control and provides a clean way of composing proofs of individual pieces into a complete proof of correctness of the system. As an example of the efficacy of deductive approaches, consider our predicate abstraction tool. Many researchers have been surprised by the fact that our tool can handle abstract systems with a large number of predicates. For example, the proofs of German protocol and the Y86 processor model generated 46 and 77 predicates respectively. The abstract system, however, had a sufficiently small number of reachable states that it could be model-checked within minutes. The reason for this is that the predicates are not arbitrary but are generated using libraries of lemmas that encode the user's understanding of how the functions involved in the definition of the systems interact. If we were to design a stand-alone automatic predicate generation tool then it would have been unlikely to generate predicates that result in such carefully crafted abstract system. Our approach does require the user to sometimes add new rules when the existing libraries are insufficient. However, in analogous situations, an automatic tool would also have required some form

of manual abstraction of the implementation or its properties.

The comments above are not intended to imply that algorithmic decision procedures are not effective, nor that theorem proving is the cure-all of all problems in formal verification. Far from it. Decision procedures are particularly useful for systems modeled at low levels of abstraction, for example gate-level netlists. Theorem proving is in fact crippled for such systems since the insight of the user about the workings of the system is obscured by the implementation details. We believe decision procedures should be applied whenever possible to provide automation in the verification with theorem proving providing ways to compose the results of the individual verifications and decompose the problem when it is beyond the scope of the decision procedure available. As the capacity of decision procedures increases, larger pieces of verification can be "pushed" to such procedures to provide more automation in the verification.

In this dissertation we have attempted to do exactly this. For different kinds of verification problems we chose different procedures for automation. In some cases, for example in the case of sequential programs, we could make the theorem prover itself to act as a symbolic simulator allowing us to abstract the operational details of the systems considered and effectively integrating assertional methods with detailed operational program models. In other cases, we developed abstraction tools or integrated external oracles. We believe that a theorem prover should be viewed as a reasoning framework in which one can integrate different procedures in one fixed formal logic and use their combination in automating proofs of computing systems.

Chapter 19

Future Directions

We have worked on using theorem proving and algorithmic decision procedures to scale up the capacity of formal verification and increase its applicability. The work in this dissertation investigates several approaches to achieve this, but we have merely scratched the surface. There are many ways to extend the work presented here. Here we discuss some of the direct extensions that we intend to work on in the near future.

19.1 Real-time Systems and Peer-to-peer Protocols

Our use of (fair) stuttering trace containment as a notion of correspondence is effective in defining specifications of systems for which we only "care" about safety and liveness properties. With fairness constraints, we can specify constraints like "some specific action is eventually selected". Real-time systems, on the other hand, require more rigid timing requirements. For example, we want to say that some specific action is selected within a specified time. We do not know if it is possible

to integrate such rigid timing constraints in a simple way with the notion of trace containment as could be done for fairness. Of course it is possible to augment the system with an extra component that keeps track of the elapsed time. Such ideas are not new. Abadi and Lamport [AL94] considered such approaches in 1994 to integrate some form of assertional reasoning for verifying real-time systems. Recently, there has been work on model checking real-time systems with an augmented time variable [Lam05]. We believe such definitions of real-time systems as augmented reactive systems with timing variables can be effectively formalized. The question, however, is what kind of execution correspondence is appropriate. Obviously, allowing refinements that are insensitive to finite stuttering destroys the real-time constraints. We are currently working on formalizing a notion of "real-time stutter" which we believe can be effectively applied to define specifications of such systems.

A related issue arises for certain systems that involve "conditional liveness". This arises in reasoning about peer-to-peer protocols. In that context, processes join and leave the protocol at different times and the system must maintain a specific pre-defined topology in the face of such joining and leaving. Protocols for maintaining topology are interesting to reason about in their own right. One might think that to do so one can define a specification in which joins and leaves are atomic and done in a way that the topology is obviously maintained, and then show that the implementation is a refinement of such a specification up to stuttering. Unfortunately, this is not possible. Most protocols cannot guarantee that the topology is maintained in the face of arbitrary leaves and joins, but rather that if eventually such activities subside then the topology is stabilized. The condition specifying that eventually leaves and joins subside has to be an environmental constraint in

the verification. It will be interesting to investigate how such constraints can be integrated with the general theory of trace containment, and what the effective proof rules will be for decomposing the verification of such systems. An answer to such questions, of course, requires us to actually *do* some verification of such systems. We are currently looking at some topology maintenance protocols due to Li, Misra, and Plaxton [LMP04] and trying to investigate how one can formally verify such systems.

19.2 Counterexamples with Predicate Abstraction

Our predicate abstraction methods suffer from one important deficiency, namely the generation of effective counterexamples. Model checking is performed on the abstract system generated by our procedure, and counterexamples are produced for this system. Thus the counterexamples are a sequence of predicates on the concrete implementation. Since the abstraction is a conservative approximation, it is also possible that the counterexample is spurious. While these counterexamples are useful to determine additional rewrite rules for better normalization of terms, it is better for the purpose of finding bugs in the concrete system if we can produce concrete counterexamples. Unfortunately the expressiveness of the predicates we handle makes it impossible for us to concretize the abstract counterexample algorithmically. Nevertheless, it might be possible to find heuristics for better counterexample generation.

One promising approach suggested by Matthews [private communication] is to use rewriting for counterexample generation. The rewrite rules for this purpose have to be very different from the ones that we are using to generate predicates.

But one can imagine building libraries of "counterexample rules" and defining a procedure that provides counterexample paths using such rules. In the work done so far we have not experimented with designing rules for that purpose, nor are we certain whether it would be possible to design generic reusable libraries as we did for the purpose of predicate discovery. We are applying our abstraction tool on more systems in an effort to understand what rules will be effective.

19.3 Integrating GSTE with Theorem Proving

We have defined and proved characterization theorems for a compositional model checker. Model checking is by far the most commonly used decision procedure for reasoning about reactive systems, but it is by no means the only one. It would be interesting to explore how and whether other decision procedures can be effectively formalized in ACL2. One of the decision procedures that we are currently exploring is Generalized Symbolic Trajectory Evaluation (GSTE) [YS02]. In this procedure, the properties of a system are written not as a temporal logic formula but in terms of a graph called the *assertion graph*. GSTE is a very efficient symbolic algorithm for reasoning about finite-state reactive systems. Several different notions are used to characterize when a system satisfies an assertion graph. The important ones are *strong satisfiability*, *terminal satisfiability*, and *fair satisfiability*. Strong satisfiability allows the user to characterize "forward properties" of a system. Thus one can write a property for an adder circuit of the form: "If the circuit is given inputs `A` and `B` at time t then at time $(t+3)$ the output has the value $(\mathtt{A}+\mathtt{B})$." Terminal satisfiability allows one to write backwards properties as well, for example saying that if `A` is one of the inputs at time t and `C` is the output at time $(t+3)$ then the other input at

time t is $(\texttt{C} - \texttt{A})$. Fair satisfiability allows one to write liveness properties as well.

GSTE is important and interesting to formalize. In particular, while the algorithms are efficient, they have one serious deficiency in that they are difficult to compose. Thus if we prove that a circuit $\mathcal{C}_1$ satisfies an assertion graph $\mathcal{A}_1$ and $\mathcal{C}_2$ satisfies $\mathcal{A}_2$ then it is difficult to specify what the composition of $\mathcal{C}_1$ and $\mathcal{C}_2$ satisfies. Thus algorithmic composition of GSTE proofs is cumbersome. We believe it would be more effective to use GSTE with theorem proving, where theorem proving is applied to do the composition and GSTE is applied to handle the low-level proof work.

In a recent work, we have formalized one simple GSTE algorithm based on strong satisfiability and proved its characterization theorem in ACL2. We believe that we can do so for terminal satisfiability as well. A more interesting question is to try to formalize fair satisfiability. This is difficult to do in ACL2, since the notion involves characterization of fair paths through the assertion graph. We believe that it might be possible to characterize them as eventually periodic paths, but we have not worked out the details. However, the proposal for function objects, once it is implemented, ought to make such reasoning easier and more natural.

19.4 Certifying Model Checkers

Defining and proving characterization theorems is certainly one way of integrating decision procedures with a theorem prover. Another way, which might involve less "overhead", is for the decision procedure itself to output a formal proof when the verification succeeds. By doing so in a case by case basis we can avoid the burden of formally verifying such procedures, especially since the procedures are optimized

for efficiency and hence naturally complex. The contrast can be likened to the difference between implementing and verifying a VCG for a sequential program and our approach in Chapter 6 which performs the same "job" as a VCG does but on a case by case basis. A decision procedure that outputs a proof together with the decision is often called a *certifying* decision procedure.

Unfortunately, most decision procedures like model checkers are not certifying. While they return a counterexample when the verification fails (which might be considered a proof of "non-satisfiability" of the temporal property by the system), they do not provide a proof when the model checking actually succeeds. However, very recently there has been work on designing certifying model checkers [Nam01]. Nevertheless the proofs produced by such checkers are very different from the formal proofs we have seen in this dissertation; the proofs take the form of infinite games. It would be interesting to see if such proofs can be algorithmically translated into proofs understandable by a theorem prover, thus obviating the needs for characterization theorems.

Bibliography

[ABP01] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling in Multiprogramming Multiprocessors. *Theory of Computing Systems*, 34:115–144, 2001.

[ACDJ01] M. Aagaard, B. Cook, N. Day, and R. B. Jones. A Framework for Microprocessor Correctness Statements. In T. Margaria and T. F. Melham, editors, *Proceedings of the 11th International Conference on Correct Hardware Design and Verification Methods (CHARME 2001)*, volume 2144 of *LNCS*, pages 443–448, Scotland, UK, 2001. Springer-Verlag.

[ACJTH04] M. Aagaard, V. C. Ciubotariu, and F. Khalvati J. T. Higgins. Combining Equivalence Verification and Completion Functions. In A. J. Hu and A. K. Martin, editors, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *LNCS*, pages 98–112, Austin, TX, November 2004. Springer-Verlag.

[ADJ04] M. Aagaard, N. Day, and R. B. Jones. Synchronization-at-Retirement for Pipeline Verification. In A. J. Hu and A. K. Martin, editors, *Pro-*

ceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004), volume 3312 of *LNCS*, pages 113–127, Austin, TX, November 2004. Springer-Verlag.

[AJK+00] M. D. Aagard, R. B. Jones, R. Kaivola, K. R. Kohatsu, and C. H. Seger. Formal verification of iterative algorithms in microprocessors. In *Proceedings of the 37th ACM/IEEE Design Automation Conference (DAC 2000)*, pages 201–206, Los Angeles, CA, 2000. ACM Press.

[AK86] K. R. Apt and D. Kozen. Limits for Automatic Verification of Finite-state Concurrent Systems. *Information Processing Letters*, 15:307–307, 1986.

[AL91] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[AL94] M. Abadi and L. Lamport. An Old-Fashioned Recipe for Real Time. *Communications of the ACM*, 16(5):1543–1571, September 1994.

[Aro04] T. Arons. Verification of an Advanced mips-Type Out-of-Order Execution Algorithm. In R. Alur and D. A. Peled, editors, *Proceedings of the 16th International Conference on Computer-Aided Verification (CAV 2004)*, volume 3117 of *LNCS*, pages 414–426. Springer-Verlag, July 2004.

[Att99] P. Attie. Liveness-preserving Simulation Relations. In J. Welch, editor, *Proceedings of 18th ACM Symposium on Principles of Distributed Computing (PODC 1999)*, pages 63–72. ACM Press, May 1999.

[Bau02] J. Baumgartner. *Automatic Structural Abstraction Techniques for Enhanced Verification.* PhD thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, 2002.

[BCG88] M. Browne, E. M. Clarke, and O. Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59, 1988.

[BD94] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In D. L. Dill, editor, *Proceedings of the 6th International Conference on Computer-Aided Verification (CAV 1994)*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[Ber03] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models.* Kluwer Academic Publishers, 2nd edition, 2003.

[Bev87] W. R. Bevier. *A Verified Operating System Kernel.* PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1987.

[BF00] D. Basin and S. Friedrich. Combining WS1S and HOL. In D. M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, pages 39–56. Research Studies Press/Wiley, Baldock, Herts, UK, February 2000.

[BGG+92] R. J. Boulton, A. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with Embedding Hardware Description Languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Con-*

ference on Theorem Provers in Circuit Design: Theory, Practice and Experience (TPCD 1992), volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland, 1992.

[BGV99] R. E. Bryant, S. German, and M. N. Velev. Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions. In N. Halbwachs and D. Peled, editors, *Proceedings of the* 11*th International Conference on Computer-Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 470–482. Springer-Verlag, 1999.

[BH97a] B. Brock and W. A. Hunt, Jr. Formally Specifying and Mechanically Verifying Programs for the Motorola Complex Arithmetic Processor DSP. In *Proceedings of the* 1997 *International Conference on Computer Design: VLSI in Computers & Processors (ICCD 1997)*, pages 31–36, Austin, TX, 1997. IEEE Computer Society Press.

[BH97b] B. Brock and W. A. Hunt, Jr. The Dual-Eval Hardware Description Language. *Formal Methods in Systems Design*, 11(1):71–104, 1997.

[BH99] B. Brock and W. A. Hunt, Jr. Formal Analysis of the Motorola CAP DSP. In *Industrial-Strength Formal Methods*. Springer-Verlag, 1999.

[Bha92] J. Bhasker. *A VHDL Primer*. Prentice-Hall, 1992.

[BHMY89] W. R. Bevier, W. A. Hunt, Jr., J S. Moore, and W. D. Young. An Approach to System Verification. *Journal of Automated Reasoning*, 5(4):409–530, December 1989.

[BHSV+96] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. A. Edwards, S. P. 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 R. Alur and T. Henzinger, editors, *Proceedings of the* 8*th International Conference on Computer-Aided Verification (CAV 1996)*, volume 1102 of *LNCS*, pages 428–432. Springer-Verlag, July 1996.

[BJNT00] A. Boujjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the* 12*th International Conference on Computer-Aided Verification (CAV 2000)*, volume 1855 of *LNCS*. Springer-Verlag, July 2000.

[BKD+04] D. Burger, S. W. Keckler, K. S. McKinley M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[BKM95] R. S. Boyer, M. Kaufmann, and J S. Moore. The Boyer-Moore Theorem Prover and Its Interactive Enhancements. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[BKM96] B. Brock, M. Kaufmann, and J S. Moore. ACL2 Theorems about Commercial Microprocessors. In M. Srivas and A. Camilleri, editors, *Proceedings of the* 1*st International Conference on Formal Methods in Computer-Aided Design (FMCAD 1996)*, volume 1166 of *LNCS*, pages 275–293. Springer-Verlag, 1996.

[Ble77] W. W. Bledsoe. Non-Resolution Theorem Proving. *Artificial Intelligence*, 9(1):1–35, 1977.

[BLS02] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the* 14*th International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, pages 78–92. Springer-Verlag, July 2002.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[BM81] R. S. Boyer and J S. Moore. Metafunctions: Proving them Correct and Using Them Efficiently as New Proof Procedure. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, London, UK, 1981.

[BM88a] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988.

[BM88b] R. S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study for Linear Arithmetic. In *Machine Intelligence*, volume 11, pages 83–124. Oxford University Press, 1988.

[BM97] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, London, UK, 1997.

[BM02] R. S. Boyer and J S. Moore. Single-threaded Objects in ACL2. In S. Krishnamurthy and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages (PADL)*, volume 2257 of *LNCS*, pages 9–27. Springer-Verlag, 2002.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*, pages 201–213, Snowbird, UT, 2001. ACM Press.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BO03] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, 2003.

[BPR99] R. D. Blumofe, C. G. Plaxton, and S. Ray. Verification of a Concurrent Deque Implementation. Technical Report TR-99-11, Department of Computer Sciences, The University of Texas at Austin, June 1999.

[BR01] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In M. B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 103–122. Springer-Verlag, 2001.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BSW69] K. A. Barlett, R. A. Scantlebury, and P. C. Wilkinson. A Note on Reliable Full Duplex Transmission over Half Duplex Links. *Communications of the ACM*, 12, 1969.

[BT90] A. Bronstein and T. L. Talcott. Formal Verification of Pipelines based on String-functional Semantics. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification, VLSI Design Methods II*, pages 349–366, 1990.

[BY96] R. S. Boyer and Y. Yu. Automated Proofs of Object Code for a Widely Used Microprocessor. *Journal of the ACM*, 43(1), January 1996.

[Can95] G. Cantor. Beitrãge zur Begrũndung der transfiniten Mengenlehre. *Mathematische Annalen*, xlvi:481–512, 1895.

[Can97] G. Cantor. Beitrãge zur Begrũndung der transfiniten Mengenlehre. *Mathematische Annalen*, xlix:207–246, 1897.

[Can52] G. Cantor. *Contributions to the Founding of the Theory of Transfinite Numbers*. Dover Publications Inc., 1952. Translated by P. E. B. Jourdain.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Approximation or Analysis of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles*

of Programming Languages (POPL 1977), pages 238–252, Los Angeles, CA, 1977. ACM Press.

[CCGR99] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. In N. Halbwacha and D. Peled, editors, *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 495–499. Springer-Verlag, 1999.

[CDE+99] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, 1999.

[CE81] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In D. C. Kozen, editor, *Logic of Programs, Workshop*, volume 131 of *LNCS*, pages 52–71, Yorktown Heights, NY, May 1981. Springer-Verlag.

[CEJS98] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry Reductions in Model Checking. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV 1998)*, volume 1427 of *LNCS*, pages 147–158. Springer-Verlag, 1998.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 8(2):244–263, April 1986.

[CGJ^{+}00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the* 12*th International Conference on Computer-Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 154–169. Springer-Verlag, 2000.

[CGP00] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model-Checking*. The MIT Press, Cambridge, MA, January 2000.

[Cho99] C. Chou. The mathematical foundation of symbolic trajectory evaluation. In N. Halbwacha and D. Peled, editors, *Proceedings of the* 11*th International Conference on Computer-Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 196–207. Springer-Verlag, 1999.

[CK37] A. Church and S. C. Kleene. Formal Definitions in the Theory of Ordinal Numbers. *Fundamenta Mathematicae*, 28:11–21, 1937.

[CM90] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Cambridge, MA, 1990.

[Coh87] A. Cohn. A Proof of Correctness of the VIPER Microprocessor. Technical Report 104, University of Cambridge, Computer Laboratory, January 1987.

[CP99] C. Chou and D. Peled. Formal Verification of a Partial-Order Reduction Technique for Model Checking. *Journal of Automated Reasoning*, 23(3-4):265–298, 1999.

[DCN+00] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. F. Melham. The PROSPER toolkit. In S. Graf and M. Schwartbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for Constructing Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 78–92, Berlin, Germany, 2000. Springer-Verlag.

[DD02] S. Das and D. L. Dill. Counter-example Based Predicate Discovery in Predicate Abstraction. In M. Aagaard and J. W. O'Leary, editors, *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2002)*, volume 2517 of *LNCS*, pages 19–32, Portland, OR, 2002. Springer-Verlag.

[DDP99] S. Das, D. Dill, and S. Park. Experience with Predicate Abstraction. In N. Halbwacha and D. Peled, editors, *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 160–171. Springer-Verlag, 1999.

[DFH+91] G. Dowek, A. Felty, G. Huet, C. Paulin, and B. Werner. The Coq Proof Assistant User Guide Version 5.6. Technical Report TR 134, INRIA, December 1991.

[Dij75] E. W. Dijkstra. Guarded Commands, Non-determinacy and a Calculus for Derivation of Programs. *Language Hierarchies and Interfaces*, pages 111–124, 1975.

[Dil96] D. L. Dill. The Murϕ Verification System. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-*

Aided Verification (CAV 1996), volume 1102 of *LNCS*, pages 390–393. Springer-Verlag, July 1996.

[DLNS98] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking for Java. Technical Report 159, Compaq Systems Research Center, December 1998.

[EdR93] K. Engelhardt and W. P. de Roever. Generalizing Abadi & Lamport's Method to Solve a Problem Posed by Pnueli. In J. Woodcock and P. G. Larsen, editors, *Industrial-strength Formal Methods, 1st International Symposium of Formal Methods Europe*, volume 670 of *LNCS*, pages 294–313, Odense, Denmark, April 1993. Springer-Verlag.

[EK00] E. A. Emerson and V. Kahlon. Reducing Model Checking of the Many to the Few. In D. A. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE 2000)*, volume 1831 of *LNCS*, pages 236–254, Pittsburg, PA, July 2000. Springer-Verlag.

[EK03] E. A. Emerson and V. Kahlon. Exact and Efficient Verification of Parameterized Cache Coherence Protocols. In D. Geist, editor, *Proceedings of the 12th International Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, volume 2860 of *LNCS*, pages 247–262. Springer-Verlag, July 2003.

[FKR+02] A. Flatau, M. Kaufmann, D. F. Reed, D. Russinoff, E. W. Smith, and R. Sumners. Formal Verification of Microprocessors at AMD. In M. Sheeran and T. F. Melham, editors, *4th International Workshop on Designing Correct Circuits (DCC 2002)*, Grenoble, France, April 2002.

[Fla92] A. D. Flatau. *A Verified Language Implementation of an Applicative Language with Dynamic Storage Allocation.* PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1992.

[Flo67] R. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematcs*, volume XIX, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[FQ02] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Proceedings of the 29th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 191–202. ACM Press, 2002.

[Gam96] R. Gamboa. Square Roots in ACL2: A Study in Sonata Form. Technical Report TR-96-34, Department of Computer Sciences, The University of Texas at Austin, 1996.

[Gam99] R. Gamboa. *Mechanically Verifying Real-valued Algorithms in ACL2.* PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1999.

[GHH+02] B. Greer, J. Harrison, G. Henry, W. Lei, and P. Tang. Scientific Computing on the Itanium® Processor. *Scientific Computing*, 10(4):329–337, 2002.

[Glo99] P. Y. Gloess. Imperative Program Verification in PVS. Technical report, École Nationale Supérieure Électronique, Informatique et Radiocom-

munications de bordeaux, 1999. See URL `http://dept-info.labri.-u.bordeaux.fr/imperative/index.html`.

[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.

[Göd31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematic und Physik*, 38:173–198, 1931.

[Göd92] K. Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover Publications, February 1992.

[Gol90] D. M. Goldshlag. Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover. *IEEE Transactions on Software Engineering*, 16(9):1005–1023, 1990.

[Gor95] M. J. C. Gordon. The Semantic Challenges of Verilog HDL. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS 1995)*, pages 136–145. IEEE Computer Society Press, 1995.

[Gor02] M. J. C. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, 2002.

[Gre98] D. A. Greve. Symbolic Simulation of the JEM1 Microprocessor. In G. Gopalakrishnan and P. Windley, editors, *Proceedings of the 2nd In-*

ternational Conference on Formal Methods in Computer-Aided Design (FMCAD 1998), volume 1522 of *LNCS*. Springer-Verlag, 1998.

[GS97] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.

[Gup92] A. Gupta. Formal Hardware Verification Methods: A Survey. *Formal Methods in Systems Design*, 2(3):151–238, October 1992.

[GvN61] H. H. Goldstein and J. von Neumann. Planning and Coding Problems for an Electronic Computing Instrument. In *John von Neumann, Collected Works, Volume V*. Pergamon Press, Oxford, 1961.

[GWH00] D. Greve, M. Wilding, and D. Hardin. High-Speed, Analyzable Simulators. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 89–106, Boston, MA, June 2000. Kluwer Academic Publishers.

[Har95] J. Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Center, 1995.

[Har00] J. Harrison. The HOL Light Manual Version 1.1. Technical report, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3Qg, England, April 2000.

[HB92] W. A. Hunt, Jr. and B. Brock. A Formal HDL and Its Use in the FM9001 Verification. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, Prentice-Hall International Series in Computer Science, pages 35–48, Englewood Cliffs, NJ, 1992. Prentice-Hall.

[Hei99] S. T. Heilmann. *Proof Support for Duration Calculus*. PhD thesis, Department of Information Technology, Technical University of Denmark, 1999.

[Hes02] W. Hesselink. Eternity Variables to Simulate Specification. In *Proceedings of Mathematics of Program Construction*, volume 2386 of *LNCS*, pages 117–130, Dagstuhl, Germany, 2002. Springer-Verlag.

[HGS00] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying Advanced Microarchitectures that Support Speculation and Exceptions. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV 2000)*, volume 1855 of *LNCS*. Springer-Verlag, July 2000.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM-SIGPLAN Conference on Principles of Programming Languages (POPL 2002)*, pages 58–70. ACM Press, 2002.

[HKM03] W. A. Hunt, Jr., R. B. Krug, and J S. Moore. Linear and Nonlinear Arithmetic in ACL2. In D. Geist, editor, *Proceedings of the 12th*

International Conference on Correct Hardware Design and Verification Methods (CHARME 2003), volume 2860 of *LNCS*, pages 319–333. Springer-Verlag, July 2003.

[HM95] P. Homeier and D. Martin. A Mechanically Verified Verification Condition Generator. *The Computer Journal*, 38(2):131–141, July 1995.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, 1969.

[Hol03] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, November 2003.

[HP02] J. L. Hennessey and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann, San Francisco, CA, 3rd edition, 2002.

[HR04] G. Hamon and J. Rushby. An Operational Semantics for Stateflow. In M. Wermelinger and T. Margaria, editors, *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE 2004)*, volume 2984 of *LNCS*, pages 229–243. Springer-Verlag, 2004.

[HR05] W. A. Hunt, Jr. and E. Reeber. Formalization of the DE2 Language. In W. Paul, editor, *Proceedings of the 13th Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, LNCS, Saarbrücken, Germany, 2005. Springer-Verlag.

[Hun94] W. A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *LNAI*. Springer-Verlag, 1994.

[Hun00] W. A. Hunt, Jr. The DE Language. In P. Manlolios, M. Kaufmann, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 119–131, Boston, MA, June 2000. Kluwer Academic Publishers.

[Jac98] P. B. Jackson. Verifying A Garbage Collector Algorithm. In J. Grundy and M. Newer, editors, *Proceedings of the* 11*th International Conference on Theorem Proving in Higher Order Logics (TPHOLS 1998)*, volume 1479 of *LNCS*, pages 225–244. Springer-Verlag, 1998.

[JM01] R. Jhala and K. McMillan. Microarchitecture Verification by Compositional Model Checking. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of* 12*th International Conference on Computer-aided Verification (CAV)*, volume 2102 of *LNCS*. Springer-Verlag, 2001.

[Jon02] R. B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Kluwer Academic Publishers, June 2002.

[JPR99] B. Jonsson, A. Pnueli, and C. Rump. Proving Refinement Using Transduction. *Distributed Computing*, 12(2-3):129–149, 1999.

[KG99] C. Kern and M. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.

[Kin69] J. C. King. *A Program Verifier*. PhD thesis, Carnegie-Melon University, 1969.

[KMa] M Kaufmann and J S. Moore. ACL2 documentation: `O-P`. See URL `http://www.cs.utexas.edu/users/moore/acl2/v2-9/O-P.html`.

[KMb] M Kaufmann and J S. Moore. ACL2 home page. See URL `http://-www.cs.utexas.edu/users/moore/acl2`.

[KMc] M. Kaufmann and J S. Moore. How to Prove Theorems Formally. See URL: `http://www.cs.utexas.edu/users/moore/-publications/how-to-prove-thms/main.ps`.

[KM94] M. Kaufmann and J S. Moore. Design Goals of ACL2. Technical Report 101, Computational Logic Incorporated (CLI), 1717 West Sixth Street, Suite 290, Austin, TX 78703, 1994.

[KM97] M. Kaufmann and J S. Moore. A Precise Description of the ACL2 Logic. See URL `http://www.cs.utexas.edu/users/moore/publications/-km97.ps.gz`, 1997.

[KM01] M. Kaufmann and J S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[KMM00a] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, Boston, MA, June 2000.

[KMM00b] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, June 2000.

[KP88] S. Katz and D. Peled. An Efficient Verification Method for Parallel and Distributed Programs. In J. W. de Bakker and W. P. de Roever, editors, *Workshop on Linear time, Branching time and Partial Order Logics and Models of Concurrency*, volume 354 of *LNCS*, pages 489–507. Springer-Verlag, 1988.

[Kr] HOL 4, Kananaskis 1 release. http://hol.sf.net/.

[KS02] M. Kaufmann and R. Sumners. Efficient Rewriting of Data Structures in ACL2. In D. Borrione, M. Kaufmann, and J S. Moore, editors, *Proceedings of 3rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2002)*, pages 141–150, Grenoble, France, April 2002.

[Kun95] K. Kunen. A Ramsey Theorem in Boyer-Moore Logic. *Journal of Automated Reasoning*, 15(2), October 1995.

[Lam74] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[Lam77] L. Lamport. Proving the Correctness of Multiprocessor Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[Lam83a] L. Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)* , 5(2):190–222, April 1983.

[Lam83b] L. Lamport. What Good is Temporal Logic? *Information Processing*, 83:657–688, 1983.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 16(3):827–923, May 1994.

[Lam05] L. Lamport. Real Time Model Checking is Really Simple. In W. Paul, editor, *Proceedings of the* 13*th Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, LNCS, Saarbrücken, Germany, 2005. Springer-Verlag.

[LB03] S. K. Lahiri and R. E. Bryant. Deductive Verification of Advanced Out-of-Order Microprocessors. In W. A. Hunt, Jr. and F. Somenzi, editors, *Proceedings of the* 15*th International Conference on Computer-Aided Verification (CAV 2003)*, volume 2275 of *LNCS*, pages 341–354. Springer-Verlag, July 2003.

[LB04a] S. K. Lahiri and R. E. Bryant. Constructing Quantified Invariants via Predicate Abstraction. In B. Stefen and G. Levi, editors, *Proceedings of the* 5*th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *LNCS*, pages 267–281. Springer-Verlag, 2004.

[LB04b] S. K. Lahiri and R. E. Bryant. Indexed Predicate Discovery for Unbounded System Verification. In R. Alur and D. A. Peled, editors, *Proceedings of the* 16*th International Conference on Computer-Aided Ver-*

ification (CAV 2004), volume 3117 of *LNCS*, pages 135–147. Springer-Verlag, July 2004.

[LBBO01] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental Verification by Abstraction. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 98–112. Springer-Verlag, April 2001.

[LBC03] S. K. Lahiri, R. E. Bryant, and B. Cook. A Symbolic Approach to Predicate Abstraction. In W. A. Hunt, Jr. and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer-Aided Verification*, volume 2275 of *LNCS*, pages 141–153. Springer-Verlag, 2003.

[LM03] H. Liu and J S. Moore. Executable JVM Model for Analytical Reasoning: A Study. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines, and Emulators*, San Diego, CA, June 2003.

[LM04] H. Liu and J S. Moore. Java Program Verification via a JVM Deep Embedding in ACL2. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3233 of *LNCS*, pages 184–200, Park City, Utah, 2004. Springer-Verlag.

[LMP04] X. Li, J. Misra, and C. G. Plaxton. Active and Concurrent Topology Maintenance. In R. Guerraoui, editor, *Proceedings of the 18th Annual*

Conference on Distributed Computing (DISC 2004), volume 3274 of *LNCS*, pages 320–334. Springer-Verlag, October 2004.

[LMTY02] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu. Specifying and Verifying Systems with TLA+. In E. Jul, editor, *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 45–48, Copenhagen, Denmark, 2002.

[Man00a] P. Manolios. Correctness of Pipelined Machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *LNCS*, pages 161–178, Austin, TX, 2000. Springer-Verlag.

[Man00b] P. Manolios. Mu-Calculus Model Checking in ACL2. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 73–88. Kluwer Academic Publishers, Boston, MA, June 2000.

[Man01] P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2001.

[Man03] P. Manolios. A Compositional Theory of Refinement for Branching Time. In D. Geist, editor, *Proceedings of the 12th Working Conference on Correct Hardware Design and Verification Methods*, volume 2860 of *LNCS*, pages 304–218. Springer-Verlag, 2003.

[Max04] C. Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools, and Flows*. Elsevier, 2004.

[McC62] J. McCarthy. Towards a Mathematical Science of Computation. In *Proceedings of the Information Processing Congress*, volume 62, pages 21–28. North-Holland, August 1962.

[McC97a] W. McCune. 33 Basic Test Problems: A practical evaluation of some paramodulation strategies. In R. Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos, Chapter* 5, pages 71–114. MIT Press, 1997.

[McC97b] W. McCune. Solution to the Robbins Problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McM98] K. McMillan. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of the* 10*th International Conference on Computer-Aided Verification (CAV 1998)*, volume 1427 of *LNCS*, pages 110–121. Springer-Verlag, 1998.

[Mil90] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1990.

[MLK98] J S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Kernel of the AMD5K86 Floating-point Division Algo-

rithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.

[MM03] P. Manolios and J S. Moore. Partial Functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.

[MM04] P. Molitor and J. Mohnke. *Equivalence Checking of Digital Circuits: Fundamentals, Principles, Methods*. Springer-Verlag, 2004.

[MMRV05] J. Matthews, J S. Moore, S. Ray, and D. Vroon. A Symbolic Simulation Approach to Assertional Program Verification. Draft, January 2005. See URL `http://www.cs.utexas.edu/users/sandip/-publications/symbolic/main.html`.

[MN95] O. Müller and T. Nipkow. Combining Model Checking and Deduction of I/O-Automata. In E. Brinksma, editor, *Proceedings of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1995)*, volume 1019 of *LNCS*, Aarhus, Denmark, May 1995. Springer-Verlag.

[MN03] F. Mehta and T. Nipkow. Proving Pointer Programs in Higher Order Logic. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE 2003)*, volume 2741 of *LNAI*, pages 121–135, Miami, FL, 2003. Springer-Verlag.

[MNS99] P. Manolios, K. Namjoshi, and R. Sumners. Linking Model-checking and Theorem-proving with Well-founded Bisimulations. In N. Halbwacha and D. Peled, editors, *Proceedings of the 11th International Con-*

ference on Computer-Aided Verification (CAV 1999), volume 1633 of *LNCS*, pages 369–379. Springer-Verlag, 1999.

[Moo96] J S. Moore. *Piton: A Mechanically Verified Assembly Language*. Kluwer Academic Publishers, 1996.

[Moo99a] J S. Moore. A Mechanically Checked Proof of a Multiprocessor Result via a Uniprocessor View. *Formal Methods in Systems Design*, 14(2):213–228, March 1999.

[Moo99b] J S. Moore. Proving Theorems about Java-like Byte Code. In E. R. Olderog and B. Stefen, editors, *Correct System Design — Recent Insights and Advances*, volume 1710 of *LNCS*, pages 139–162, 1999.

[Moo01] J S. Moore. Rewriting for Symbolic Execution of State Machine Models. In G. Berry, H. Comon, and J. Finkel, editors, *Proceedings of the 13th International Conference on Computer-Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 411–422. Springer-Verlag, September 2001.

[Moo03a] J S. Moore. A Grand Challenge Proposal for Formal Methods: A Verified Stack. In B. K. Aichernig and T. Maibaum, editors, *Formal Methods at the Crossroads: from Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University*. Springer-Verlag, 2003.

[Moo03b] J S. Moore. Inductive Assertions and Operational Semantics. In D. Geist, editor, *Proceedings of the 12th International Conference on*

Correct Hardware Design and Verification Methods, volume 2860 of *LNCS*, pages 289–303. Springer-Verlag, October 2003.

[Moo03c] J S. Moore. Proving Theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras, and Logic of Engineering Software*, pages 227–290. IOS Press, 2003.

[MP02] J S. Moore and G. Porter. The Apprentice Challenge. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 24(3):1–24, May 2002.

[MQS00] K. McMillan, S. Qadeer, and J. Saxe. Induction in Compositional Model Checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV 2000)*, volume 1855 of *LNCS*. Springer-Verlag, July 2000.

[MS00] W. McCune and O. Shumsky. Ivy: A Preprocessor and Proof Checker for First-Order Logic. In P. Manlolios, M. Kaufmann, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 217–230. Kluwer Academic Publishers, Boston, MA, June 2000.

[MS04] P. Manolios and S. Srinivasan. Automatic Verification of Safety and Liveness of XScale-Like Processor Models Using WEB Refinements. In *Design, Automation and Test in Europe (DATE 2004)*, pages 168–175, Paris, France, 2004. IEEE Computer Society Press.

[MS05] P. Manolios and S. Srinivasan. Refinement Maps for Efficient Verification of Processor Models. In *Design, Automation and Test in Europe*

(DATE 2005), pages 1304–1309, Munich, Germany, 2005. IEEE Computer Society Press.

[MV03] P. Manolios and D. Vroon. Algorithms for Ordinal Arithmetic. In F. Baader, editor, *Proceedings of the* 19*th International Conference on Automated Deduction (CADE 2003)*, volume 2741 of *LNAI*, pages 243–257, Miami, FL, July 2003. Springer-Verlag.

[MV04a] P. Manolios and D. Vroon. Integrating Reasoning about Ordinal Arithmetic into ACL2. In A. J. Hu and A. K. Martin, editors, *Proceedings of the* 5*th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *LNCS*, pages 82–97. Springer-Verlag, November 2004.

[MV04b] J. Matthews and D. Vroon. Partial Clock Functions in ACL2. In M. Kaufmann and J S. Moore, editors, 5*th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, Austin, TX, November 2004.

[Nam97] K. Namjoshi. A Simple Characterization of Stuttering Bisimulation. In S. Ramesh and G. Sivakumar, editors, *Proceedings of the* 17*th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1997)*, volume 1346 of *LNCS*, pages 284–296. Springer-Verlag, 1997.

[Nam01] K. S. Namjoshi. Certifying Model Checkers. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the* 13*th International Conference*

on Computer Aided Verification (CAV 2001), volume 2102 of *LNCS*. Springer-Verlag, 2001.

[Nec98] G. Necula. *Compiling with Proofs*. PhD thesis, Carnegie-Melon University, September 1998.

[NK00] K. S. Namjoshi and R. P. Kurshan. Syntactic Program Transformations for Automatic Abstraction. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 435–449. Springer-Verlag, July 2000.

[NO79] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2), October 1979.

[Nor98] M. Norrish. *C Formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher Order Logics*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[NSS59] A. Newell, J. C. Shaw, and H. A. Simon. Report on a General Problemn-solving Program. In *IFIP Congress*, pages 256–264, 1959.

[OG76] S. S. Owicki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–285, 1976.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapoor, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752. Springer-Verlag, June 1992.

[OZGS99] J. O'Leary, X. Zhao, R. Gerth, and C. H. Seger. Formally Verifying IEEE Compliance of Floating-point Hardware. *Intel Technology Journal*, Q1-1999, 1999.

[Pal03] S. Palnitkar. *Verilog HDL*. Prentice-Hall, 2nd edition, 2003.

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the* 5*th GI-Conference on Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer-Verlag, 1981.

[Pau] L. Paulson. The Isabelle Reference Manual. See URL `http://www.-cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle2003/doc/-ref.pdf`.

[Pau93] L. Paulson. Set Theory for Verification: I. From Foundations to Functions. *Journal of Automated Reasoning*, 11:353–389, 1993.

[Pau95] L. Paulson. Set Theory for Verification: II. Induction and Recursion. *Journal of Automated Reasoning*, 15:167–215, 1995.

[Pau00] L. Paulson. Mechanizing UNITY in Isabelle. 1(1):3–32, 2000.

[Pau01] L. Paulson. A Simple Formalization and Proof for the Mutilated Chess Board. *Logic Journal of the IGPL*, 9(3):499–509, 2001.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the* 18*th Annual IEEE Symposium of Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, October 1977.

[Pnu84] A. Pnueli. In Transition for Global to Modular Temporal Reasoning about Programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag, 1984.

[Pnu85] A. Pnueli. Linear and Branching Structures in the Semantics and Logics of Reactive Systems. In W. Brauer, editor, *Proceedings of the* 12*th International Colloquium on Automata, Languages, and Programming (ICALP 1985)*, volume 194 of *LNCS*, pages 15–32. Springer-Verlag, 1985.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic Deductive Verification with Invisible Invariants. In T. Margaria and W. Yi, editors, *Proceedings of the* 7*th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 82–97. Springer-Verlag, 2001.

[QS82] J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the* 5*th International symposimum on Programming*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, 1982.

[Ray] S. Ray. Homepage of Sandip Ray. See URL `http://www.cs.-utexas.edu/users/sandip`.

[Rey00] J. C. Reynolds. Intuitionist Reasoning about Shared Mutable Data Structures. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

[RF00] D. Russinoff and A. Flatau. RTL Verification: A Floating Point Multiplier. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 201–232, Boston, MA, June 2000. Kluwer Academic Publishers.

[RH04] S. Ray and W. A. Hunt, Jr. Deductive Verification of Pipelined Machines Using First-Order Quantification. In R. Alur and D. A. Peled, editors, *Proceedings of the* 16*th International Conference on Computer-Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 31–43, Boston, MA, July 2004. Springer-Verlag.

[RH05] E. Reeber and W. A. Hunt, Jr. Integrating External SAT Solvers with ACL2. See URL `http://www.cs.utexas.edu/users/reeber/sat.ps`, 2005.

[RM04] S. Ray and J S. Moore. Proof Styles in Operational Semantics. In A. J. Hu and A. K. Martin, editors, *Proceedings of the* 5*th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *LNCS*, pages 67–81, Austin, TX, November 2004. Springer-Verlag.

[RMT03] S. Ray, J. Matthews, and M. Tuttle. Certifying Compositional Model Checking Algorithms in ACL2. In W. A. Hunt, Jr., M. Kaufmann, and

J S. Moore, editors, *4th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2003)*, Boulder, CO, July 2003.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Rog87] H Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1987.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An Integration of Model Checking with Automated Proof Checking. In P. Wolper, editor, *Proceedings of the 7th International Conference on Computer-Aided Verification (CAV 1995)*, volume 939 of *LNCS*, pages 84–97. Springer-Verlag, 1995.

[Rus92] D. Russinoff. A Mechanical Proof of Quadratic Reciprocity. *Journal of Automated Reasoning*, 8:3–21, 1992.

[Rus94] D. Russinoff. A Mechanically Verified Incremental Garbage Collector. *Formal Aspects of Computing*, 6:359–390, 1994.

[Rus95] D. Russinoff. A Formalization of a Subset of VHDL in the Boyer-Moore Logic. *Formal Methods in Systems Design*, 7(1/2):7–25, 1995.

[Rus98] D. 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. *LMS Journal of Computation and Mathematics*, 1:148–200, December 1998.

[Rus00] D. Russinoff. A Case Study in Formal Verification of Register-Transfer Logic with ACL2: The Floating Point Adder of the AMD Athlon Processor. In W. A. Hunt, Jr. and S. Johnson, editors, *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *LNCS*, pages 3–36. Springer-Verlag, 2000.

[SA98] S. W. Smith and V. Austel. Trusting Trusted Hardware: Towards a Formal Model of Programmable Secure Coprocessors. In *Proceedings of the 3rd USENIX Workshop on Electronic Commerce*, September 1998.

[Saw00] J. Sawada. Verification of a Simple Pipelined Machine Model. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 35–53, Boston, MA, June 2000. Kluwer Academic Publishers.

[Saw04] J. Sawada. ACL2VHDL Translator: A Simple Approach to Fill the Semantic Gap. In M. Kaufmann and J S. Moore, editors, *5th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, Austin, TX, November 2004.

[SB90] M. Srivas and M. Bickford. Formal Verification of a Pipelined Microprocessor. *IEEE Software*, 7(5):52–64, September 1990.

[SG02] J. Sawada and R. Gamboa. Mechanical Verification of a Square Root Algorithm Using Taylor's Theorem. In M. Aagaard and J. W. O'Leary, editors, *Proceedings of the 4th International Conference on Formal Meth-*

ods in Computer-Aided Design (FMCAD 2002), volume 2517 of *LNCS*, pages 274–291, Portland, OR, 2002. Springer-Verlag.

[SH97] J. Sawada and W. A. Hunt, Jr. Trace Table Based Approach for Pipelined Microprocessor Verification. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 364–375. Springer-Verlag, 1997.

[SH98] J. Sawada and W. A. Hunt, Jr. Processor Verification with Precise Exceptions and Speculative Execution. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV 1998)*, volume 1427 of *LNCS*, pages 135–146. Springer-Verlag, 1998.

[SH99a] J. Sawada and W. A. Hunt, Jr. Decomposing the Verification of Pipelined Microprocessors with Invariant Conditions. See URL `http://www.cs.utexas.edu/users/sawada/publication/-decomp.ps`, 1999.

[SH99b] K. Schneider and D. W. Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to ω-Automata. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLS 1999)*, volume 1690 of *LNCS*, pages 255–272. Springer-Verlag, 1999.

[SH02] J. Sawada and W. A. Hunt, Jr. Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability. *Formal Methods in Systems Design*, 20(2):187–222, 2002.

[Sha94] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge University Press, 1994.

[Sho67] J. R. Shoenfield. *Mathematical Logic*. Adison-Wesley, Reading, MA, 1967.

[Sho79] R. E. Shostak. A Practical Decision Procedure for Arithmetic with Function Symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[SR04] R. Sumners and S. Ray. Reducing Invariant Proofs to Finite Search via Rewriting. In M. Kaufmann and J S. Moore, editors, 5*th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, Austin, TX, November 2004.

[SR05] R. Sumners and S. Ray. Proving Invariants via Rewriting and Abstraction. Technical Report TR-05-35, Department of Computer Sciences, University of Texas at Austin, July 2005.

[SS98] B. Shriver and B. Smith. *The Anatomy of a High-Performance Microprocessor*. IEEE Computer Society Press, 1998.

[SS99] H. Saidi and N. Shankar. Abstract and model check while you prove. In N. Halbwacha and D. Peled, editors, *Proceedings of the* 11*th Interna-*

tional Conference on Computer-Aided Verification (CAV 1999), volume 1633 of *LNCS*, pages 443–453. Springer-Verlag, 1999.

[SSTV04] R. Sebastini, E. Singerman, S. Tonetta, and M. Y. Vardi. GSTE Is Partitioned Model Checking. In R. Alur and D. A. Peled, editors, *Proceedings of the* 16*th International Conference on Computer-Aided Verification (CAV 2004)*, volume 3117 of *LNCS*, pages 229–241. Springer-Verlag, July 2004.

[Ste90] G. L Steele, Jr. *Common Lisp the Language*. Digital Press, 30 North Avenue, Burlington, MA 01803, 2nd edition, 1990.

[Str02] M. Strecker. Formal Verification of a Java Compiler in Isabelle. In A. Voronkov, editor, *Proeedings of the* 18*th International Conference on Automated Deduction (CADE 2002)*, volume 2392 of *LNCS*, pages 63–77. Springer-Verlag, 2002.

[Sum00] R. Sumners. An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2. In M. Kaufmann and J S. Moore, editors, 2*nd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2000)*, Austin, TX, October 2000.

[Sum03] R. Sumners. Fair Environment Assumptions in ACL2. In W. A. Hunt, Jr., M. Kaufmann, and J S. Moore, editors, 4*th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2003)*, Boulder, CO, July 2003.

[Sum05] R. Sumners. *Deductive Mechanical Verification of Concurrent Systems.* PhD thesis, Department of Electrical and Computer Engineering, The Unversity of Texas at Austin, 2005.

[TM96] D. E. Thomas and P. R. Moorby. *The Verilog® Hardware Description Language.* Kluwer Academic Publishers, Boston, MA, 3rd edition, 1996.

[Tur37] A. M. Turing. On computable Numbers, with an Application to the *Entscheidungsproblem. Proceedings of the London Mathematical Society*, 2(42):230–265, 1937.

[Tur49] A. M. Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machine*, pages 67–69, University Mathematical Laboratory, Cambridge, England, June 1949.

[VHB+03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2):203–232, April 2003.

[vW91] J. von Wright. Mechanizing the Temporal Logic of Actions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 4th International Workshop on the HOL Theorem Proving System and its Applications*, pages 155–161, Davis, CA, August 1991. IEEE Computer Society Press.

[Wan63] H. Wang. Mechanical Mathematics and Inferential Analysis. In P. Braffort and D. Hershberg, editors, *Computer Programming and Formal Systems.* North-Holland, 1963.

[Wey80] R. Weyhrauch. Prolegomena to a Theory of Mechanized Formal Reasoning. *Artificial Intelligence Journal*, 13(1):133–170, 1980.

[Wil93] M. Wilding. A Mechanically Verified Application for a Mechanically Verified Environment. In C. Courcoubetis, editor, *Proceedings of the* 5*th International Conference on Computer-Aided Verification (CAV 1993)*, volume 697 of *LNCS*, pages 268–279. Springer-Verlag, 1993.

[Wil97] M. Wilding. Robust Computer System Proofs in PVS. In C. M. Holloway and K. J. Hayhurst, editors, 4*th NASA Langley Formal Methods Workshop*, number 3356 in NASA Conference Publication, 1997.

[You88] W. D. Young. A Verified Code Generator for a Subset of Gypsy. Technical Report 33, Computational Logic Inc., 1988.

[YS02] J. Yang and C. H. Seger. Generalized Symbolic Trajectory Evaluation — Abstraction in Action. In M. Aagaard and J. W. O'Leary, editors, *Proceedings of the* 4*th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2002)*, volume 2517 of *LNCS*, pages 70–87, Portland, OR, 2002. Springer-Verlag.

[Yu92] Y. Yu. *Automated Proofs of Object Code for a Widely Used Microprocessor*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1992.

Vita

Sandip Ray was born in Calcutta, India, the only child of Swasti and Shyamapada Ray. He spent the first twenty two idyllic years of his life in this beautiful city, completing his high school degree from South Point High School (widely rumored as the largest high school in Asia with about 14000 students), and subsequently, his Bachelor's degree from Jadavpur University. He then moved to Bangalore, India, where he completed his Masters from the Indian Institute of Science. After a brief stint at Texas Instruments (India) Ltd. working as a DSP Engineer, Sandip moved to Austin, TX, USA, and joined the University of Texas at Austin as a Doctoral student. His long student life finally ends in 2005 with his Ph.D.

Permanent Address: FD 82, Sector 3, Salt Lake City,
Calcutta 700091. India.

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.